The LaTeX3 Interfaces

The LaTeX Project*

Released 2025-07-11

Abstract

This is the reference documentation for the expl3 programming environment; see the matching source3 PDF for the typeset sources. The expl3 modules set up a naming scheme for LaTeX commands, which allow the LaTeX programmer to systematically name functions and variables, and specify the argument types of functions.

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

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

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

Contents

Part I
Introduction

Chapter 1

Introduction to expl3 and this document

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

1.1 Naming functions and variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

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

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

clist Comma separated list.

dim "Rigid" lengths.

fp Floating-point values;

int Integer-valued count register.

muskip "Rubber" lengths for use in mathematics.

skip "Rubber" lengths.

str String variables: contain character data.

tl Token list variables: placeholder for a token list.

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

bool Either true or false.

box Box register.

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

flag Non-negative integer that can be incremented expandably.

fparray Fixed-size array of floating point values.

intarray Fixed-size array of integers.

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

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

regex Regular expression.

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

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

1.1.1 Behavior of c-type arguments when the N-type token resulting from expansion is undefined

When c-type expansion is applied, it will produce an N-type token to be consumed by the underlying function. If the result of this process is a token which is undefined, TEX's behavior is to make it equal to \scan_stop: (\relax).

This will likely lead to low-level errors if it occurs in contexts where expl3 expects a "variable", e.g. a prop, seq, etc. Therefore, the programmer should ensure that c-type expansion is only applied when the resulting N-type token will definitely exist, i.e., when it is either defined prior to the application of the c-type expansion or will be by the underlying N-type function.

1.1.2 Scratch variables

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

There are two more special types of constants:

q Quark constants.

s Scan mark constants.

Similarly, each quark or scan mark name starts with the module name, but doesn't end with a variable type, because the type is already marked by the prefix q or s. Some general quarks and scan marks provided by LATEX3 don't start with a module name, for example \s_stop. See documentation of quarks and scan marks in Chapter 19 for more info.

1.1.3 Terminological inexactitude

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

1.2 Documentation conventions

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

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

`\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` ⟨*seq var*⟩

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

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

`\cs_to_str:N` ⋆

`\cs_to_str:N` ⟨*cs*⟩

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

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

`\seq_map_function:NN` ☆

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

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

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

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

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

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

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

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

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

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

The normal description text.

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

Addition dates For functions added to expl3 after 2020-02-02 (the point at which is was integrated into the LaTeX kernel), the date of addition is included in the documentation as "New".

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

1.3 Formal language conventions which apply generally

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

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

1.4 TeX concepts not supported by LaTeX3

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

Part II
Bootstrapping

Chapter 2

The l3bootstrap module Bootstrap code

2.1 Using the LaTeX3 modules

The modules documented in this file (and `source3` for documented sources) are designed to be used on top of LaTeX 2ε and are already pre-loaded since LaTeX 2ε 2020-02-02. To support older formats, the `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions are still available to load them all as one.

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

`\ExplSyntaxOn` ⟨*code*⟩ `\ExplSyntaxOff`

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

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

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

Updated: 2023-08-03

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

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

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

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

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

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

Chapter 3

The l3names module Namespace for primitives

3.1 Setting up the LaTeX3 programming language

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

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

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

Part III
Programming Flow

Chapter 4

The l3basics module Basic definitions

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

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

4.1 No operation functions

`\prg_do_nothing:` ⋆

`\prg_do_nothing:`

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

`\scan_stop:`

`\scan_stop:`

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

4.2 Grouping material

`\group_begin:`
`\group_end:`

`\group_begin:`
`\group_end:`

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

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

\group_insert_after:N

\group_insert_after:N ⟨*token*⟩

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

TEXhackers note: This is the TEX primitive \aftergroup.

\group_show_list:
\group_log_list:

New: 2021-05-11

\group_show_list:
\group_log_list:

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

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

4.3 Control sequences and functions

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

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

4.3.1 Defining functions

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

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

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

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

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

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

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

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

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

`N` **and** `n` No manipulation.

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

`p` **and** `w` These are special cases.

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

4.3.2 Defining new functions using parameter text

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

4.3.3 Defining new functions using the signature

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

Updated: 2023-09-27

\cs_new:Nn ⟨*function*⟩ {⟨*code*⟩}

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

\cs_set:Nn ⟨*function*⟩ {⟨*code*⟩}

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

\cs_gset:Nn ⟨*function*⟩ {⟨*code*⟩}

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

Updated: 2023-09-27

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

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

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

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

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

4.3.4 Copying control sequences

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

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

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

`\cs_new_eq:NN` ⟨cs_1⟩ ⟨cs_2⟩
`\cs_new_eq:NN` ⟨cs_1⟩ ⟨*token*⟩

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

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

`\cs_set_eq:NN` ⟨cs_1⟩ ⟨cs_2⟩
`\cs_set_eq:NN` ⟨cs_1⟩ ⟨*token*⟩

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

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

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

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

4.3.5 Deleting control sequences

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

\cs_undefine:N
\cs_undefine:c

\cs_undefine:N ⟨*control sequence*⟩

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

4.3.6 Showing control sequences

\cs_meaning:N ⋆
\cs_meaning:c ⋆

\cs_meaning:N ⟨*control sequence*⟩

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

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

\cs_show:N
\cs_show:c

\cs_show:N ⟨*control sequence*⟩

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

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

\cs_log:N
\cs_log:c

\cs_log:N ⟨*control sequence*⟩

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

4.3.7 Converting to and from control sequences

\use:c ⋆

\use:c {⟨*control sequence name*⟩}

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

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

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

and

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

would be equivalent to

```
\abc
```

after two expansions of `\use:c`.

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

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

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

`\cs:w` ⋆
`\cs_end:` ⋆

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

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

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

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

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

and

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

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

`\cs_to_str:N` ⋆

`\cs_to_str:N` ⟨*control sequence*⟩

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

4.4 Analyzing control sequences

\cs_split_function:N ⋆

\cs_split_function:N ⟨*function*⟩

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

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

\cs_prefix_spec:N ⋆

\cs_prefix_spec:N ⟨*token*⟩

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

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

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

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

\cs_parameter_spec:N ⋆

New: 2022-06-24

\cs_parameter_spec:N ⟨*token*⟩

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

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

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

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

\cs_replacement_spec:N ⋆
\cs_replacement_spec:c ⋆

`\cs_replacement_spec:N` ⟨*token*⟩

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

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

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

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

4.5 Using or removing tokens and arguments

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

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

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

results in the input stream containing

```
abc { def }
```

i.e. only the outer braces are removed.

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

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

\use_i:nn {⟨arg_1⟩} {⟨arg_2⟩}
\use_i:nnn {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩}
\use_i:nnnn {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩} {⟨arg_4⟩}
\use_i:nnnnn {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩} {⟨arg_4⟩} {⟨arg_5⟩}
\use_i:nnnnnn {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩} {⟨arg_4⟩} {⟨arg_5⟩} {⟨arg_6⟩}
\use_i:nnnnnnn {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩} {⟨arg_4⟩} {⟨arg_5⟩} {⟨arg_6⟩} {⟨arg_7⟩}
\use_i:nnnnnnnn {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩} {⟨arg_4⟩} {⟨arg_5⟩} {⟨arg_6⟩} {⟨arg_7⟩} {⟨arg_8⟩}
\use_i:nnnnnnnnn {⟨arg_1⟩} {⟨arg_2⟩} {⟨arg_3⟩} {⟨arg_4⟩} {⟨arg_5⟩} {⟨arg_6⟩} {⟨arg_7⟩} {⟨arg_8⟩} {⟨arg_9⟩}

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

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

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

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

results in the input stream containing

```
abc { def }
```

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

`\use_ii_i:nn` ⋆ `\use_ii_i:nn {⟨arg1⟩} {⟨arg2⟩}`

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

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

`\use_none:n {⟨group1⟩}`

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

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

`\use:e` ⋆ `\use:e {⟨expandable tokens⟩}`

Updated: 2023-07-05

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

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

4.5.1 Selecting tokens from delimited arguments

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

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

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

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

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

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

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

4.6 Predicates and conditionals

LaTeX3 has three concepts for conditional flow processing:

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

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

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

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

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

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

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

```
\cs_if_free_p:N
```

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

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

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

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

4.6.1 Tests on control sequences

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

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

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

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

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

Tests whether the ⟨*control sequence*⟩ is currently defined (whether as a function or another control sequence type), and its meaning is not the primitive \relax token. This is different from \if_cs_exist:N, which evaluates to true if passed the token \relax as an argument.

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

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

This test is the negation of the above \cs_if_exist:NTF.

4.6.2 Primitive conditionals

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

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

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

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

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

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

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

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

TEXhackers note: This is the TEX primitive \ifx.

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

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

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

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

TEXhackers note: \if:w and \if_charcode:w are both the TEX primitive \if. \if_-catcode:w is the TEX primitive \ifcat.

\if_cs_exist:N ⋆
\if_cs_exist:w ⋆

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

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

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

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

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

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

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

4.7 Starting a paragraph

\mode_leave_vertical:

`\mode_leave_vertical:`

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

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

4.8 Debugging support

\debug_on:n
\debug_off:n

Updated: 2023-05-23

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

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

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

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

\debug_suspend:
\debug_resume:

`\debug_suspend: ... \debug_resume:`

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

Chapter 5

The **l3expan** module Argument expansion

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

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

5.1 Defining new variants

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

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

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

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

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

results in the definition of `\seq_gpush:No`

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

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

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

5.2 Methods for defining variants

We recall the set of available argument specifiers.

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

\cs_generate_variant:Nn
\cs_generate_variant:cn

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3 Introducing the variants

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

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

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

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

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

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

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

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

results in the call

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

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

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

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

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

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

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

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

and

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

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

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

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

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

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

5.4 Manipulating the first argument

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

`\exp_args:Nc` ⋆
`\exp_args:cc` ⋆

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

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

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

`\exp_args:No` ⋆

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

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

`\exp_args:NV` ⋆

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

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

`\exp_args:Nv` ⋆

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

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

`\exp_args:Ne` ⋆

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

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

`\exp_args:Nf` ⋆

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

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

`\exp_args:Nx`

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

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

5.5 Manipulating two arguments

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

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

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

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

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

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

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

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

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

5.6 Manipulating three arguments

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

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

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

\exp_args:NNno ⋆
\exp_args:NNnV ⋆
\exp_args:NNnv ⋆
\exp_args:NNne ⋆
\exp_args:NNcc ⋆
\exp_args:NNcf ⋆
\exp_args:NNoo ⋆
\exp_args:NNVV ⋆
\exp_args:NNVv ⋆
\exp_args:NNVe ⋆
\exp_args:NNvV ⋆
\exp_args:NNvv ⋆
\exp_args:NNve ⋆
\exp_args:NNeV ⋆
\exp_args:NNev ⋆
\exp_args:NNee ⋆
\exp_args:NnNV ⋆
\exp_args:Nnnc ⋆
\exp_args:Nnno ⋆
\exp_args:Nnnf ⋆
\exp_args:NnnV ⋆
\exp_args:Nnnv ⋆
\exp_args:Nnne ⋆
\exp_args:Nnff ⋆
\exp_args:Nnee ⋆
\exp_args:Ncnc ⋆
\exp_args:Ncno ⋆
\exp_args:NcnV ⋆
\exp_args:Ncnv ⋆
\exp_args:Ncne ⋆
\exp_args:Ncoo ⋆
\exp_args:NcVV ⋆
\exp_args:NcVv ⋆
\exp_args:NcVe ⋆
\exp_args:NcvV ⋆
\exp_args:Ncvv ⋆
\exp_args:Ncve ⋆
\exp_args:NceV ⋆
\exp_args:Ncev ⋆
\exp_args:Ncee ⋆
\exp_args:Nooo ⋆
\exp_args:Noof ⋆
\exp_args:Nffo ⋆
\exp_args:NVNV ⋆
\exp_args:Neee ⋆

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

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

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

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

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

5.7 Unbraced expansion

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

\exp_last_unbraced:No ⟨*token*⟩ {⟨*tokens*$_1$⟩}

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

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

\exp_last_unbraced:Nx

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

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

\exp_last_two_unbraced:Noo ⋆

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

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

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

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

TeXhackers note: This is the TeX primitive \expandafter.

5.8 Preventing expansion

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

\exp_not:N ⋆ \exp_not:N ⟨*token*⟩

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

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

\exp_not:c ⋆ \exp_not:c {⟨*tokens*⟩}

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

\exp_not:n ⋆ \exp_not:n {⟨*tokens*⟩}

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

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

\exp_not:o ⋆ \exp_not:o {⟨*tokens*⟩}

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

`\exp_not:V` ⋆ `\exp_not:V` ⟨*variable*⟩

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

`\exp_not:v` ⋆ `\exp_not:v {`⟨*tokens*⟩`}`

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

`\exp_not:e` ⋆ `\exp_not:e {`⟨*tokens*⟩`}`

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

`\exp_not:f` ⋆ `\exp_not:f {`⟨*tokens*⟩`}`

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

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

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

5.9 Controlled expansion

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

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

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

`\exp:w` ⋆
`\exp_end:` ⋆

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

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

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

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

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

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

`\exp:w` ⋆
`\exp_end_continue_f:w` ⋆

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

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

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

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

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

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

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

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

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

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

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

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

`\exp:w` ⋆
`\exp_end_continue_f:nw` ⋆

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

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

5.10 Internal functions

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

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

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

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

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

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

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

Chapter 6

The l3sort module
Sorting functions

6.1 Controlling sorting

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

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

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

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

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

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

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

`\sort_return_same:`
`\sort_return_swapped:`

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

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

Chapter 7

The l3tl-analysis module Analyzing token lists

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

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

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

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

New: 2021-05-11

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

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

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

Updated: 2022-03-26

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

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

Chapter 8

The l3regex module
Regular expressions in TEX

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

Let us give a few examples. After

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

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

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

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

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

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

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

8.1 Syntax of regular expressions

8.1.1 Regular expression examples

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

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

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

8.1.2 Characters in regular expressions

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

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

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

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

Characters.

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

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

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

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

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

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

`\t` Horizontal tab (hex 09).

8.1.3 Characters classes

Character properties.

`.` A single period matches any token.

`\d` Any decimal digit.

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

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

\v Any vertical space character, equivalent to [\^^J\^^K\^^L\^^M]. Note that \^^K is a vertical space, but not a space, for compatibility with Perl.

\w Any word character, i.e., alphanumerics and underscore, equivalent to the explicit class [A-Za-z0-9_].

\D Any token not matched by \d.

\H Any token not matched by \h.

\N Any token other than the \n character (hex 0A).

\S Any token not matched by \s.

\V Any token not matched by \v.

\W Any token not matched by \w.

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

Character classes match exactly one token in the subject.

[...] Positive character class. Matches any of the specified tokens.

[^...] Negative character class. Matches any token other than the specified characters.

[x-y] Within a character class, this denotes a range (can be used with escaped characters).

[:⟨*name*⟩:] Within a character class (one more set of brackets), this denotes the POSIX character class ⟨*name*⟩, which can be alnum, alpha, ascii, blank, cntrl, digit, graph, lower, print, punct, space, upper, word, or xdigit.

[:^⟨*name*⟩:] Negative POSIX character class.

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

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

8.1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

? 0 or 1, greedy.

?? 0 or 1, lazy.

* 0 or more, greedy.

*? 0 or more, lazy.

+ 1 or more, greedy.

+? 1 or more, lazy.

{n} Exactly n.

{n,} n or more, greedy.

{n,}? n or more, lazy.

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

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

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

Alternation and capturing groups.

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

`(...)` Capturing group.

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

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

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

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

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

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

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

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

8.1.5 Matching exact tokens

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.1.6 Miscellaneous

Anchors and simple assertions.

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

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

^or \A Start of the subject token list.

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

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

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

8.2 Syntax of the replacement text

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

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

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

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

For instance,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.3 Pre-compiling regular expressions

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

\regex_new:N

\regex_new:N ⟨*regex var*⟩

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

\regex_set:Nn
\regex_gset:Nn

\regex_set:Nn ⟨*regex var*⟩ {⟨*regex*⟩}

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

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

\regex_const:Nn

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

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

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

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

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

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

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

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

8.4 Matching

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

`\regex_if_match:nnTF`
`\regex_if_match:nVTF`
`\regex_if_match:NnTF`
`\regex_if_match:NVTF`

New: 2025-05-14

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

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

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

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

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

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

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

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

results in `\l_foo_int` taking the value 5.

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

New: 2022-01-10

```
\regex_match_case:nnTF
  {
    {⟨regex1⟩} {⟨code case1⟩}
    {⟨regex2⟩} {⟨code case2⟩}
    ...
    {⟨regexn⟩} {⟨code casen⟩}
  } {⟨token list⟩}
  {⟨true code⟩} {⟨false code⟩}
```

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

In detail, for each starting position in the ⟨*token list*⟩, each of the ⟨*regex*⟩ is searched in turn. If one of them matches then the corresponding ⟨*code*⟩ is used and everything else is discarded, while if none of the ⟨*regex*⟩ match at a given position then the next starting position is attempted. If none of the ⟨*regex*⟩ match anywhere in the ⟨*token list*⟩ then nothing is left in the input stream. Note that this differs from nested `\regex_if_match:nnTF` statements since all ⟨*regex*⟩ are attempted at each position rather than attempting to match ⟨$regex_1$⟩ at every position before moving on to ⟨$regex_2$⟩.

8.5 Submatch extraction

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

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

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

For instance, assume that you type

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

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

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

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

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

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

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

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

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

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

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

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

8.6 Replacement

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

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

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

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

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

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

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

New: 2022-01-10

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

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

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

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

New: 2022-01-10

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

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

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

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

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

8.7 Scratch regular expressions

`\l_tmpa_regex`
`\l_tmpb_regex`

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

`\g_tmpa_regex`
`\g_tmpb_regex`

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

8.8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

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

Additional error-checking to come.

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

Code improvements to come.

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

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

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

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

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

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

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

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

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

Chapter 9

The l3prg module Control structures

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

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

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

9.1 Defining a set of conditional functions

```
\prg_new_conditional:Npnn
\prg_set_conditional:Npnn
\prg_gset_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_gset_protected_conditional:Npnn
\prg_new_conditional:Nnn
\prg_set_conditional:Nnn
\prg_gset_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_gset_protected_conditional:Nnn
```

Updated: 2022-11-01

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

These functions create a family of conditionals using the same ⟨*code*⟩ to perform the test created. Those non-protected conditionals are expandable if ⟨*code*⟩ is. The `new` versions check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` versions do no check and perform assignments locally (*cf.* `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `T`, `F` and `TF`, and for non-protected conditionals `p`. For public conditionals, a full set of forms should be provided: this contrasts with strictly internal conditionals, where only the required subset need be defined.

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

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

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

An example can easily clarify matters here:

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

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

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

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

Updated: 2023-05-26

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

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

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

`\prg_return_true:`
`\prg_return_false:`

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

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

`\prg_generate_conditional_variant:Nnn`

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

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

9.2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, etc. which can then be used on both the boolean type and predicate functions.

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

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

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

`\bool_new:N ⟨boolean⟩`

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

`\bool_const:Nn`
`\bool_const:cn`

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

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

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

`\bool_set_false:N ⟨boolean⟩`

Sets ⟨*boolean*⟩ logically `false`.

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

`\bool_set_true:N ⟨boolean⟩`

Sets ⟨*boolean*⟩ logically `true`.

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

`\bool_set_eq:NN ⟨boolean1⟩ ⟨boolean2⟩`

Sets ⟨*boolean1*⟩ to the current value of ⟨*boolean2*⟩.

`\bool_set:Nn`
`\bool_set:cn`
`\bool_gset:Nn`
`\bool_gset:cn`

`\bool_set:Nn ⟨boolean⟩ {⟨boolexpr⟩}`

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

\bool_set_inverse:N
\bool_set_inverse:c
\bool_gset_inverse:N
\bool_gset_inverse:c

`\bool_set_inverse:N ⟨boolean⟩`

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

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

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

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

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

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

`\bool_to_str:N ⟨boolean⟩`
`\bool_to_str:n {⟨boolean expression⟩}`

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

\bool_show:N
\bool_show:c

Updated: 2021-04-29

`\bool_show:N ⟨boolean⟩`

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

\bool_show:n

`\bool_show:n {⟨boolean expression⟩}`

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

\bool_log:N
\bool_log:c

Updated: 2021-04-29

`\bool_log:N ⟨boolean⟩`

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

\bool_log:n

`\bool_log:n {⟨boolean expression⟩}`

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

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

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

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

9.2.1 Constant and scratch booleans

\c_true_bool
\c_false_bool

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

\l_tmpa_bool
\l_tmpb_bool

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

\g_tmpa_bool
\g_tmpb_bool

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

9.3 Boolean expressions

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

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

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

is a valid boolean expression.

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

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

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

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

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

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

\bool_if_p:n ⋆
\bool_if:nTF ⋆

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

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

\bool_lazy_all_p:n ⋆
\bool_lazy_all:nTF ⋆

\bool_lazy_all_p:n { {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} ··· {⟨boolexpr$_N$⟩} }
\bool_lazy_all:nTF { {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} ··· {⟨boolexpr$_N$⟩} } {⟨true code⟩} {⟨false code⟩}

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

\bool_lazy_and_p:nn ⋆
\bool_lazy_and:nnTF ⋆

\bool_lazy_and_p:nn {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩}
\bool_lazy_and:nnTF {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} {⟨true code⟩} {⟨false code⟩}

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

\bool_lazy_any_p:n ⋆
\bool_lazy_any:nTF ⋆

\bool_lazy_any_p:n { {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} ··· {⟨boolexpr$_N$⟩} }
\bool_lazy_any:nTF { {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} ··· {⟨boolexpr$_N$⟩} } {⟨true code⟩} {⟨false code⟩}

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

\bool_lazy_or_p:nn ⋆
\bool_lazy_or:nnTF ⋆

\bool_lazy_or_p:nn {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩}
\bool_lazy_or:nnTF {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} {⟨true code⟩} {⟨false code⟩}

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

\bool_not_p:n ⋆

\bool_not_p:n {⟨boolean expression⟩}

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

`\bool_xor_p:nn` ⋆ `\bool_xor:nnTF` ⋆

`\bool_xor_p:nn {⟨boolexpr₁⟩} {⟨boolexpr₂⟩}`
`\bool_xor:nnTF {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} {⟨true code⟩} {⟨false code⟩}`

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

9.4 Logical loops

Loops using either boolean expressions or stored boolean values.

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

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

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

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

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

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

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

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

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

`\bool_while_do:Nn` ☆ `\bool_while_do:cn` ☆

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

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

`\bool_do_until:nn` ☆

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

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

`\bool_do_while:nn` ☆

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

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

`\bool_until_do:nn` ☆

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

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

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

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

`\bool_case:n` ⋆
`\bool_case:nTF` ⋆

New: 2023-05-03

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

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

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

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

9.5 Producing multiple copies

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

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

9.6 Detecting TeX's mode

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

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

Detects if TeX is currently in horizontal mode.

\mode_if_inner_p: ★
\mode_if_inner:TF ★

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

Detects if TeX is currently in inner mode.

\mode_if_math_p: ★
\mode_if_math:TF ★

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

Detects if TeX is currently in maths mode.

\mode_if_vertical_p: ★
\mode_if_vertical:TF ★

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

Detects if TeX is currently in vertical mode.

9.7 Primitive conditionals

\if_predicate:w ★

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

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

\if_bool:N ★

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

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

9.8 Nestable recursions and mappings

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

\prg_break_point:Nn ★

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

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

\prg_map_break:Nn ★

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

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

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

9.8.1 Simple mappings

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

`\prg_break_point:` ⋆

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

`\prg_break:` ⋆
`\prg_break:n` ⋆

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

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

9.9 Internal programming functions

`\group_align_safe_begin:` ⋆
`\group_align_safe_end:` ⋆

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

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

Chapter 10

The l3sys module System/runtime functions

10.1 The name of the job

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

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

10.2 Date and time

`\c_sys_minute_int`
`\c_sys_hour_int`
`\c_sys_day_int`
`\c_sys_month_int`
`\c_sys_year_int`

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

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

`\c_sys_timestamp_str`
New: 2023-08-27

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

10.3 Engine

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

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

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

\sys_if_engine_opentype_p: ⋆
\sys_if_engine_opentype:TF ⋆

New: 2024-11-05

`\sys_if_engine_opentype_p:`
`\sys_if_engine_opentype:TF {⟨true code⟩} {⟨false code⟩}`

Conditional which allows functionality-specific code to be used. The test is true for engines which can use OpenType fonts and thus full Unicode typesetting. This tests for features not engine name, but currently is equivalent to requiring either XeTEX or LuaTEX.

TEXhackers note: The underlying test here checks for `\Umathcode`, which is used to implement OpenType math font typesetting. Any engine which should give a `true` result here needs to provide general Unicode support (accepting the full UTF-8 range for character codes), a mechanism to load system fonts and a suitable interface for math mode typesetting.

\c_sys_engine_str

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

\c_sys_engine_exec_str

New: 2020-08-20

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

\c_sys_engine_format_str

New: 2020-08-20

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

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

⟨*major*⟩.⟨*minor*⟩.⟨*revision*⟩

For XƎTEX, the form is

⟨*major*⟩.⟨*minor*⟩

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

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

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

`\sys_timer:` ⋆

New: 2021-05-12

`\sys_timer:`

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

10.4 Output format

In LATEX, the output format may be set in the preamble: as such, expl3 delays setting the information here until either

- `\sys_ensure_backend:` or `\sys_load_backend:n` are used
- `\begin{document}` is reached

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

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

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

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

10.5 Platform

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

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

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

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

10.6 Random numbers

`\sys_rand_seed:` ⋆ `\sys_rand_seed:`

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

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

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

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

10.7 Access to the shell

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

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

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

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

`\c_sys_shell_escape_int`

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

0 Shell escape is disabled

1 Unrestricted shell escape is enabled

2 Restricted shell escape is enabled

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

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

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

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

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

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

`\sys_if_shell_restricted_p: ⋆`
`\sys_if_shell_restricted:TF ⋆`

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

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

`\sys_shell_now:n`
`\sys_shell_now:e`

`\sys_shell_now:n {⟨tokens⟩}`

Execute ⟨*tokens*⟩ through shell escape immediately.

`\sys_shell_shipout:n`
`\sys_shell_shipout:e`

`\sys_shell_shipout:n {⟨tokens⟩}`

Execute ⟨*tokens*⟩ through shell escape at shipout.

10.8 System queries

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

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

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

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

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

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

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

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

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

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

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

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

New: 2024-03-08

Works as described for `\sys_split_query:nnnN`, but sets the ⟨*seq var*⟩ to contain one entry for each line returned by `l3sys-query`. This function should therefore be preferred where multi-line return is expected, e.g. for the `ls` command.

10.9 Loading configuration data

\sys_load_backend:n

`\sys_load_backend:n {⟨backend⟩}`

Loads the additional configuration file needed for backend support. If the ⟨*backend*⟩ is empty, the standard backend for the engine in use will be loaded. This command may only be used once.

\sys_ensure_backend:

`\sys_ensure_backend:`

New: 2022-07-29

Ensures that a backend has been loaded by calling `\sys_load_backend:n` if required.

`\c_sys_backend_str` Set to the name of the backend in use by `\sys_load_backend:n` when issued. Possible values are

- `pdftex`
- `luatex`
- `xetex`
- `dvips`
- `dvipdfmx`
- `dvisvgm`

`\sys_load_debug:` `\sys_load_debug:`

Load the additional configuration file for debugging support.

10.9.1 Final settings

`\sys_finalize:` `\sys_finalize:`

New: 2025-05-25 Finalizes all system-dependent functionality: required before loading a backend.

Chapter 11

The l3msg module Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The l3msg module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by l3msg to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behavior can be altered and messages can be entirely suppressed.

11.1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\\` may be used to force a new line and `\␣` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the LaTeX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

Some authors may find the need to include spaces as `~` characters tedious. This can be avoided by locally resetting the category code of `␣`.

```
\char_set_catcode_space:n { `\ }
\msg_new:nnn { foo } { bar }
  {Some message text using '#1' and usual message shorthands \{ \ \ \}.}
\char_set_catcode_ignore:n { `\ }
```

although in general this may be confusing; simply writing the messages using ~ characters is the method favored by the team.

`\msg_new:nnnn` `\msg_new:nnee` `\msg_new:nnn` `\msg_new:nne`

`\msg_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates a ⟨*message*⟩ for a given ⟨*module*⟩. The message is defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (#1 to #4) can be used: these will be supplied at the time the message is used. An error is raised if the ⟨*message*⟩ already exists.

`\msg_set:nnnn` `\msg_set:nnn`

`\msg_set:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Sets up the text for a ⟨*message*⟩ for a given ⟨*module*⟩. The message is defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (#1 to #4) can be used: these will be supplied at the time the message is used.

`\msg_if_exist_p:nn` ⋆ `\msg_if_exist:nnTF` ⋆

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

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

11.2 Customizable information for message modules

`\msg_module_name:n` ⋆

`\msg_module_name:n {⟨module⟩}`

Expands to the public name of the ⟨*module*⟩ as defined by `\g_msg_module_name_prop` (or otherwise leaves the ⟨*module*⟩ unchanged).

`\msg_module_type:n` ⋆

`\msg_module_type:n {⟨module⟩}`

Expands to the description which applies to the ⟨*module*⟩, for example a `Package` or `Class`. The information here is defined in `\g_msg_module_type_prop`, and will default to `Package` if an entry is not present.

`\g_msg_module_name_prop`

Provides a mapping between the module name used for messages, and that for documentation.

`\g_msg_module_type_prop`

Provides a mapping between the module name used for messages, and that type of module. For example, for LaTeX3 core messages, an empty entry is set here meaning that they are not described using the standard `Package` text.

11.3 Contextual information for messages

\msg_line_context: ☆

`\msg_line_context:`

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text `on line`.

\msg_line_number: ⋆

`\msg_line_number:`

Prints the current line number when a message is given.

\msg_fatal_text:n ⋆

`\msg_fatal_text:n {⟨module⟩}`

Produces the standard text

`Fatal Package ⟨module⟩ Error`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨module⟩ to be included. Any redefinition *must* produce output containing the ⟨module⟩ name, and will affect all messages using the expl3 mechanism.

\msg_critical_text:n ⋆

`\msg_critical_text:n {⟨module⟩}`

Produces the standard text

`Critical Package ⟨module⟩ Error`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨module⟩ to be included. Any redefinition *must* produce output containing the ⟨module⟩ name, and will affect all messages using the expl3 mechanism.

\msg_error_text:n ⋆

`\msg_error_text:n {⟨module⟩}`

Produces the standard text

`Package ⟨module⟩ Error`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨module⟩ to be included. Any redefinition *must* produce output containing the ⟨module⟩ name, and will affect all messages using the expl3 mechanism.

\msg_warning_text:n ⋆

`\msg_warning_text:n {⟨module⟩}`

Produces the standard text

`Package ⟨module⟩ Warning`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨module⟩ to be included. The ⟨type⟩ of ⟨module⟩ may be adjusted: `Package` is the standard outcome: see `\msg_module_type:n`. Any redefinition *must* produce output containing the ⟨module⟩ name, and will affect all messages using the expl3 mechanism.

`\msg_info_text:n` ⋆ `\msg_info_text:n {⟨module⟩}`

Produces the standard text:

`Package ⟨module⟩ Info`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included. The ⟨*type*⟩ of ⟨*module*⟩ may be adjusted: `Package` is the standard outcome: see `\msg_module_type:n`. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the expl3 mechanism.

`\msg_see_documentation_text:n` ⋆ `\msg_see_documentation_text:n {⟨module⟩}`

Produces the standard text

`See the ⟨module⟩ documentation for further information.`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included. The name of the ⟨*module*⟩ is produced using `\msg_module_name:n`.

11.4 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the `e`-type variants should be used to expand material. Note that this expansion takes place with the standard definitions in effect, which means that shorthands such as `\~` or `\\` are *not* available; instead one should use `\iow_char:N \~` and `\iow_newline:`, respectively. The following message classes exist:

- `fatal`, ending the TeX run;
- `critical`, ending the file being input;
- `error`, interrupting the TeX run without ending it;
- `warning`, written to terminal and log file, for important messages that may require corrections by the user;
- `note` (less common than `info`) for important information messages written to the terminal and log file;
- `info` for normal information messages written to the log file only;
- `term` and `log` for un-decorated messages written to the terminal and log file, or to the log file only;
- `none` for suppressed messages.

```
\msg_fatal:nnnnnn
\msg_fatal:nneeee
\msg_fatal:nnnnn
\msg_fatal:(nneee|nnnee)
\msg_fatal:nnnn
\msg_fatal:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_fatal:nnn
\msg_fatal:(nnV|nne)
\msg_fatal:nn
```

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

Issues ⟨module⟩ error ⟨message⟩, passing ⟨arg one⟩ to ⟨arg four⟩ to the text-creating functions. After issuing a fatal error the TEX run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).

```
\msg_critical:nnnnnn
\msg_critical:nneeee
\msg_critical:nnnnn
\msg_critical:(nneee|nnnee)
\msg_critical:nnnn
\msg_critical:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_critical:nnn
\msg_critical:(nnV|nne)
\msg_critical:nn
```

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

Issues ⟨module⟩ error ⟨message⟩, passing ⟨arg one⟩ to ⟨arg four⟩ to the text-creating functions. After issuing a critical error, TEX stops reading the current input file. This may halt the TEX run (if the current file is the main file) or may abort reading a sub-file.

TEXhackers note: The TEX \endinput primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```
\msg_error:nnnnnn
\msg_error:nneeee
\msg_error:nnnnn
\msg_error:(nneee|nnnee)
\msg_error:nnnn
\msg_error:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_error:nnn
\msg_error:(nnV|nne)
\msg_error:nn
```

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

Issues ⟨module⟩ error ⟨message⟩, passing ⟨arg one⟩ to ⟨arg four⟩ to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

\msg_warning:nnnnnn
\msg_warning:nneeee
\msg_warning:nnnnn
\msg_warning:(nneee|nnnee)
\msg_warning:nnnn
\msg_warning:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_warning:nnn
\msg_warning:(nnV|nne)
\msg_warning:nn

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

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

\msg_note:nnnnnn
\msg_note:nneeee
\msg_note:nnnnn
\msg_note:(nneee|nnnee)
\msg_note:nnnn
\msg_note:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_note:nnn
\msg_note:(nnV|nne)
\msg_note:nn
\msg_info:nnnnnn
\msg_info:nneeee
\msg_info:nnnnn
\msg_info:(nneee|nnnee)
\msg_info:nnnn
\msg_info:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_info:nnn
\msg_info:(nnV|nne)
\msg_info:nn

New: 2021-05-18

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

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. For the more common \msg_info:nnnnnn, the information text is added to the log file only, while \msg_note:nnnnnn adds the info text to both the log file and the terminal. The TeX run is not interrupted.

\msg_term:nnnnnn
\msg_term:nneeee
\msg_term:nnnnn
\msg_term:(nneee|nnnee)
\msg_term:nnnn
\msg_term:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_term:nnn
\msg_term:(nnV|nne)
\msg_term:nn
\msg_log:nnnnnn
\msg_log:nneeee
\msg_log:nnnnn
\msg_log:(nneee|nnnee)
\msg_log:nnnn
\msg_log:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_log:nnn
\msg_log:(nnV|nne)
\msg_log:nn

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

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The output is briefer than \msg_info:nnnnnn, omitting for instance the module name. It is added to the log file by \msg_log:nnnnnn while \msg_term:nnnnnn also prints it on the terminal.

\msg_none:nnnnnn
\msg_none:nneeee
\msg_none:nnnnn
\msg_none:(nneee|nnnee)
\msg_none:nnnn
\msg_none:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_none:nnn
\msg_none:(nnV|nne)
\msg_none:nn

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

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

11.4.1 Messages for showing material

```
\msg_show:nnnnnn
\msg_show:nneeee
\msg_show:nnnnn
\msg_show:(nneee|nnnee)
\msg_show:nnnn
\msg_show:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_show:nnn
\msg_show:(nnV|nne)
\msg_show:nn
```

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

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text is shown on the terminal and the TeX run is interrupted in a manner similar to `\tl_show:n`. This is used in conjunction with `\msg_show_item:n` and similar functions to print complex variable contents completely. If the formatted text does not contain `>~` at the start of a line, an additional line `>~.` will be put at the end. In addition, a final period is added if not present.

```
\msg_show_item:n ⋆
\msg_show_item_unbraced:n ⋆
\msg_show_item:nn ⋆
\msg_show_item_unbraced:nn ⋆
```

`\seq_map_function:NN ⟨seq var⟩ \msg_show_item:n`
`\prop_map_function:NN ⟨property list⟩ \msg_show_item:nn`

Used in the text of messages for `\msg_show:nnnnnn` to show or log a list of items or key–value pairs. The output of `\msg_show_item:n` produces a newline, the prefix `>`, two spaces, then the braced string representation of its argument. The two-argument versions separates the key and value using `␣␣=>␣␣`, and the `unbraced` versions don't print the surrounding braces.

These functions are suitable for usage with iterator functions like `\seq_map_function:NN`, `\prop_map_function:NN`, etc. For example, with a sequence `\l_tmpa_seq` containing `a`, `{b}` and `\c`,

```
\seq_map_function:NN \l_tmpa_seq \msg_show_item:n
```

would expand to three lines:

```
>␣␣{a}
>␣␣{{b}}
>␣␣{\c␣}
```

11.4.2 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using `\msg_error:nnnnnn` or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, short-hands such as `\{` or `\\` do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

\msg_expandable_error:nnnnnn ⋆
\msg_expandable_error:nnffff ⋆
\msg_expandable_error:nnnnn ⋆
\msg_expandable_error:nnfff ⋆
\msg_expandable_error:nnnn ⋆
\msg_expandable_error:nnff ⋆
\msg_expandable_error:nnn ⋆
\msg_expandable_error:nnf ⋆
\msg_expandable_error:nn ⋆

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

Issues an "Undefined error" message from TeX itself using the undefined control sequence \??? then prints "! ⟨*module*⟩: ⟨*error message*⟩", which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

11.5 Redirecting messages

Each message has a "name", which can be used to alter the behavior of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some~text } { Some~more~text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behavior with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \to B$, $B \to C$ and $C \to A$ in this order, then the $A \to B$ redirection is cancelled.

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

Changes the behavior of messages of ⟨*class one*⟩ so that they are processed using the code for those of ⟨*class two*⟩. Each ⟨*class*⟩ can be one of `fatal`, `critical`, `error`, `warning`, `note`, `info`, `term`, `log`, `none`.

`\msg_redirect_module:nnn` `\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` `\msg_redirect_name:nnn {⟨module⟩} {⟨message⟩} {⟨class⟩}`

Redirects a specific ⟨*message*⟩ from a specific ⟨*module*⟩ to act as a member of ⟨*class*⟩ of messages. No further redirection is performed. This function can be used to make a selected message "silent" without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

Chapter 12

The l3file module File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix \file_..., while others are used to work with files on a line by line basis and have prefix \ior_... (reading) or \iow_... (writing).

It is important to remember that when reading external files TeX attempts to locate them using both the operating system path and entries in the TeX file database (most TeX systems use such a database). Thus the "current path" for TeX is somewhat broader than that for other programs.

For functions which expect a ⟨*file name*⟩ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in \l_char_active_seq) are *not* expanded, allowing the direct use of these in file names. Quote tokens (") are not permitted in file names as they are reserved for internal use by some TeX primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

12.1 Input–output stream management

As TeX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in LaTeX3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

\ior_new:N
\ior_new:c
\iow_new:N
\iow_new:c

\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

\ior_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until an \ior_close:N instruction is given or the TeX run ends. If the file is not found, an error is raised.

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

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

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

\iow_open:Nn
\iow_open:(NV|cn|cV)

\iow_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

Opens ⟨*file name*⟩ for writing using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a \iow_close:N instruction is given or the TeX run ends. Opening a file for writing clears any existing content in the file (i.e., writing is *not* additive).

\ior_shell_open:Nn

\ior_shell_open:Nn ⟨*stream*⟩ {⟨*shell command*⟩}

Opens the *pseudo*-file created by the output of the ⟨*shell command*⟩ for reading using ⟨*stream*⟩ as the control sequence for access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*shell command*⟩ until a \ior_close:N instruction is given or the TeX run ends. If piped system calls are disabled an error is raised.

For details of handling of the ⟨*shell command*⟩, see \sys_get_shell:nnNTF.

\iow_shell_open:Nn

New: 2023-05-25

\iow_shell_open:Nn ⟨*stream*⟩ {⟨*shell command*⟩}

Opens the *pseudo*-file created by the output of the ⟨*shell command*⟩ for writing using ⟨*stream*⟩ as the control sequence for access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*shell command*⟩ until an \iow_close:N instruction is given or the TeX run ends. If piped system calls are disabled an error is raised.

For details of handling of the ⟨*shell command*⟩, see \sys_get_shell:nnNTF.

\ior_close:N
\ior_close:c
\iow_close:N
\iow_close:c

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

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

\ior_show:N
\ior_show:c
\ior_log:N
\ior_log:c
\iow_show:N
\iow_show:c
\iow_log:N
\iow_log:c

New: 2021-05-11

\ior_show:N ⟨*stream*⟩
\ior_log:N ⟨*stream*⟩
\iow_show:N ⟨*stream*⟩
\iow_log:N ⟨*stream*⟩

Display (to the terminal or log file) the file name associated to the (read or write) ⟨*stream*⟩.

\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:

\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:

Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

12.1.1 Reading from files

Reading from files and reading from the terminal are separate processes in expl3. The functions \ior_get:NN and \ior_str_get:NN, and their branching equivalents, are designed to work with *files*.

\ior_get:NN
\ior_get:NNTF

\ior_get:NN ⟨*stream*⟩ ⟨*tl var*⟩
\ior_get:NNTF ⟨*stream*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. The material read from the ⟨*stream*⟩ is tokenized by TEX according to the category codes and `\endlinechar` in force when the function is used. Assuming normal settings, any lines which do not end in a comment character `%` have the line ending converted to a space, so for example input

```
a b  c
```

results in a token list `a␣b␣c␣`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_ior \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens. In the non-branching version, where the ⟨*stream*⟩ is not open the ⟨*tl var*⟩ is set to `\q_no_value`.

TEXhackers note: This protected macro is a wrapper around the TEX primitive `\read`. Regardless of settings, TEX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code `\endlinechar`, omitted if `\endlinechar` is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

\ior_str_get:NN
\ior_str_get:NNTF

\ior_str_get:NN ⟨*stream*⟩ ⟨*tl var*⟩
\ior_str_get:NNTF ⟨*stream*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Function that reads one line from the file input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the ⟨*tl var*⟩ being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

```
a b  c
```

results in a token list `a b  c` with the letters `a`, `b`, and `c` having category code 12. In the non-branching version, where the⟨*stream*⟩ is not open the ⟨*tl var*⟩ is set to `\q_no_value`.

TEXhackers note: This protected macro is a wrapper around the ε-TEX primitive `\readline`. Regardless of settings, TEX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

All mappings are done at the current group level, i.e., any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

`\ior_map_inline:Nn` `\ior_map_inline:Nn` ⟨*stream*⟩ {⟨*inline function*⟩}

Applies the ⟨*inline function*⟩ to each set of ⟨*lines*⟩ obtained by calling `\ior_get:NN` until reaching the end of the file. TEX ignores any trailing new-line marker from the file it reads. The ⟨*inline function*⟩ should consist of code which receives the ⟨*line*⟩ as `#1`.

`\ior_str_map_inline:Nn` `\ior_str_map_inline:Nn` ⟨*stream*⟩ {⟨*inline function*⟩}

Applies the ⟨*inline function*⟩ to every ⟨*line*⟩ in the ⟨*stream*⟩. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨*inline function*⟩ should consist of code which receives the ⟨*line*⟩ as `#1`. Note that TEX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TEX also ignores any trailing new-line marker from the file it reads.

`\ior_map_variable:NNn` `\ior_map_variable:NNn` ⟨*stream*⟩ ⟨*tl var*⟩ {⟨*code*⟩}

For each set of ⟨*lines*⟩ obtained by calling `\ior_get:NN` until reaching the end of the file, stores the ⟨*lines*⟩ in the ⟨*tl var*⟩ then applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last set of ⟨*lines*⟩, or its original value if the ⟨*stream*⟩ is empty. TEX ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_map_inline:Nn`.

`\ior_str_map_variable:NNn` `\ior_str_map_variable:NNn` ⟨*stream*⟩ ⟨*variable*⟩ {⟨*code*⟩}

For each ⟨*line*⟩ in the ⟨*stream*⟩, stores the ⟨*line*⟩ in the ⟨*variable*⟩ then applies the ⟨*code*⟩. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*line*⟩, or its original value if the ⟨*stream*⟩ is empty. Note that TEX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TEX also ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_str_map_inline:Nn`.

`\ior_map_break:` `\ior_map_break:`

Used to terminate a `\ior_map_...` function before all lines from the ⟨*stream*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a `\ior_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\ior_map_break:n` `\ior_map_break:n {⟨code⟩}`

Used to terminate a `\ior_map_...` function before all lines in the ⟨*stream*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\ior_map_...` scenario leads to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

`\ior_if_eof_p:N` ⋆ `\ior_if_eof_p:N ⟨stream⟩`
`\ior_if_eof:NTF` ⋆ `\ior_if_eof:NTF ⟨stream⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the end of a file ⟨*stream*⟩ has been reached during a reading operation. The test also returns a `true` value if the ⟨*stream*⟩ is not open.

12.1.2 Reading from the terminal

\ior_get_term:nN
\ior_str_get_term:nN

\ior_get_term:nN {⟨*prompt*⟩} ⟨*tl var*⟩

Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the ⟨*token list*⟩ variable. Tokenization occurs as described for \ior_get:NN or \ior_str_get:NN, respectively. When the ⟨*prompt*⟩ is empty, TEX will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. \iow_term:n. Where the ⟨*prompt*⟩ is given, it will appear in the terminal followed by an =, e.g.

```
prompt=
```

12.1.3 Writing to files

\iow_now:Nn
\iow_now:(NV|Ne|cn|cV|ce)

\iow_now:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ immediately (i.e., the write operation is called on expansion of \iow_now:Nn).

\iow_log:n
\iow_log:e

\iow_log:n {⟨*tokens*⟩}

This function writes the given ⟨*tokens*⟩ to the log (transcript) file immediately: it is a dedicated version of \iow_now:Nn.

\iow_show:n
\iow_show:e

New: 2025-05-19

\iow_show:n {⟨*tokens*⟩}

This function writes the given ⟨*tokens*⟩ immediately to the same output as used by \show and \showtokens. At the start of a TEX run this will be the terminal, but may be redirected to a file if the primitive \showsteam has been set.

TEXhackers note: At present, there is no expl3 interface to set \showstream, but use of the \iow_show:n function is encouraged in places where direct writing to an I/O stream is intermixed with show functions.

\iow_term:n
\iow_term:e

\iow_term:n {⟨*tokens*⟩}

This function writes the given ⟨*tokens*⟩ to the terminal file immediately: it is a dedicated version of \iow_now:Nn.

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

\iow_shipout:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalized (i.e., at shipout). The e-type variants expand the ⟨*tokens*⟩ at the point where the function is used but *not* when the resulting tokens are written to the ⟨*stream*⟩ (*cf.* \iow_shipout_e:Nn).

TEXhackers note: When using expl3 with a format other than LATEX, new line characters inserted using \iow_newline: or using the line-wrapping code \iow_wrap:nnnN are not recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additional unwanted line-breaks.

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

Updated: 2023-09-17

`\iow_shipout_e:Nn` ⟨*stream*⟩ {⟨*tokens*⟩}

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalized (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 finalized during the page building process (such as the page number integer).

TEXhackers note: This is a wrapper around the TEX primitive `\write`. When using expl3 with a format other than LaTeX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

`\iow_char:N` ⋆

`\iow_char:N \`⟨*char*⟩

Inserts ⟨*char*⟩ into the output stream. Useful when trying to write difficult characters such as `%`, `{`, `}`, etc. in messages, for example:

```
\iow_now:Ne \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

`\iow_newline:` ⋆

`\iow_newline:`

Function to add a new line within the ⟨*tokens*⟩ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

TEXhackers note: When using expl3 with a format other than LaTeX, the character inserted by `\iow_newline:` is not recognized by TEX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_e:Nn` and direct uses of primitive operations.

12.1.4 Wrapping lines in output

\iow_wrap:nnnN
\iow_wrap:nenN

\iow_wrap:nnnN {⟨*text*⟩} {⟨*run-on text*⟩} {⟨*set up*⟩} ⟨*function*⟩

This function wraps the ⟨*text*⟩ to a fixed number of characters per line. At the start of each line which is wrapped, the ⟨*run-on text*⟩ is inserted. The line character count targeted is the value of \l_iow_line_count_int minus the number of characters in the ⟨*run-on text*⟩ for all lines except the first, for which the target number of characters is simply \l_iow_line_count_int since there is no run-on text. The ⟨*text*⟩ and ⟨*run-on text*⟩ are exhaustively expanded by the function, with the following substitutions:

- \\ or \iow_newline: may be used to force a new line,
- \␣ may be used to represent a forced space (for example after a control sequence),
- \#, \%, \{, \}, \~ may be used to represent the corresponding character,
- \iow_wrap_allow_break: may be used to allow a line-break without inserting a space,
- \iow_indent:n may be used to indent a part of the ⟨*text*⟩ (not the ⟨*run-on text*⟩).

Additional functions may be added to the wrapping by using the ⟨*set up*⟩, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the ⟨*text*⟩ which is not to be expanded on wrapping should be converted to a string using \token_to_str:N, \tl_to_str:n, \tl_to_str:N, etc.

The result of the wrapping operation is passed as a braced argument to the ⟨*function*⟩, which is typically a wrapper around a write operation. The output of \iow_wrap:nnnN (i.e., the argument passed to the ⟨*function*⟩) consists of characters of category "other" (category code 12), with the exception of spaces which have category "space" (category code 10). This means that the output does *not* expand further when written to a file.

TEXhackers note: Internally, \iow_wrap:nnnN carries out an e-type expansion on the ⟨*text*⟩ to expand it. This is done in such a way that \exp_not:N or \exp_not:n *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the ⟨*text*⟩.

\iow_wrap_allow_break:

New: 2023-04-25

\iow_wrap_allow_break:

In the first argument of \iow_wrap:nnnN (for instance in messages), inserts a break-point that allows a line break. If no break occurs, this function adds nothing to the output.

\iow_indent:n

\iow_indent:n {⟨*text*⟩}

In the first argument of \iow_wrap:nnnN (for instance in messages), indents ⟨*text*⟩ by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the ⟨*text*⟩. In case the indented ⟨*text*⟩ should appear on separate lines from the surrounding text, use \\ to force line breaks.

`\l_iow_line_count_int` The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the TeX system in use: the standard value is 78, which is typically correct for unmodified TeX Live and MiKTeX systems.

12.1.5 Constant input–output streams, and variables

`\g_tmpa_ior`
`\g_tmpb_ior`

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

`\c_log_iow`
`\c_term_iow`

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

`\g_tmpa_iow`
`\g_tmpb_iow`

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

12.1.6 Primitive conditionals

`\if_eof:w` ⋆

```
\if_eof:w ⟨stream⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests if the ⟨*stream*⟩ returns "end of file", which is true for non-existent files. The `\else:` branch is optional.

> **TeXhackers note:** This is the TeX primitive `\ifeof`.

12.2 File operations

12.2.1 Basic file operations

`\g_file_curr_dir_str`
`\g_file_curr_name_str`
`\g_file_curr_ext_str`

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (i.e., if it is in the TeX search path), and does *not* end in `/` other than the case that it is exactly equal to the root directory. The ⟨*name*⟩ and ⟨*ext*⟩ parts together make up the file name, thus the ⟨*name*⟩ part may be thought of as the "job name" for the current file.

Note that TeX does not provide information on the ⟨*dir*⟩ and ⟨*ext*⟩ part for the main (top level) file and that this file always has empty ⟨*dir*⟩ and ⟨*ext*⟩ components. Also, the ⟨*name*⟩ here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

\l_file_search_path_seq

Updated: 2023-06-15

Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and need not include the trailing slash. Spaces need not be quoted.

TeXhackers note: When working as a package in LaTeX 2ε, expl3 will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

`\file_if_exist_p:n` ★
`\file_if_exist_p:V` ★
`\file_if_exist:nTF` ★
`\file_if_exist:VTF` ★

Updated: 2023-09-18

`\file_if_exist_p:n {⟨file name⟩}`
`\file_if_exist:nTF {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

Expands the argument of the ⟨*file name*⟩ to give a string, then searches for this string using the current TeX search path and the additional paths controlled by `\l_file_search_path_seq`.

Since TeX cannot remove files, only write to them, once a file has been found during a TeX run, it will exist until the end of the run unless a non-TeX process intervenes. Since file operations are relatively slow, expl3 therefore internally tracks when a file is seen, and uses this information to avoid multiple filesystem checks. See `\file_forget:n` for how to indicate to expl3 that a file may have been deleted *during* a TeX run, so that its presence in the filesystem can be reasserted with `\file_if_exist:nTF` and similar commands.

`\file_forget:n`

New: 2024-12-09

`\file_forget:n {⟨file name⟩}`

Resets the internal tracker for files such that a subsequent use of `\file_if_exist:nTF`, `\file_size:n`, etc., for the ⟨*file name*⟩ will re-query the filesystem rather than use any cached information. This can be used whether or not the file has previously been seen. This function is intended to be used where non-TeX processes may result in file deletion, for example if LuaTeX is in use, `os.remove()` may be used to delete a file part-way through a run.

12.2.2 Information about files and file contents

Functions in this section return information about files as expl3 `str` data, *except* that the non-expandable functions set their return *token list* to `\q_no_value` if the file requested is not found. As such, comparison of file names, hashes, sizes, etc., should use `\str_if_eq:nnTF` rather than `\tl_if_eq:nnTF` and so on.

`\file_hex_dump:n` ☆
`\file_hex_dump:V` ☆
`\file_hex_dump:nnn` ☆
`\file_hex_dump:Vnn` ☆

`\file_hex_dump:n {⟨file name⟩}`
`\file_hex_dump:nnn {⟨file name⟩} {⟨start index⟩} {⟨end index⟩}`

Searches for ⟨*file name*⟩ using the current TeX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most TeX behavior there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The {⟨*start index*⟩} and {⟨*end index*⟩} values work as described for `\str_range:nnn`.

\file_get_hex_dump:nN
\file_get_hex_dump:VN
\file_get_hex_dump:nNTF
\file_get_hex_dump:VNTF
\file_get_hex_dump:nnnN
\file_get_hex_dump:VnnN
\file_get_hex_dump:nnnNTF
\file_get_hex_dump:VnnNTF

\file_get_hex_dump:nN {⟨file name⟩} ⟨tl var⟩
\file_get_hex_dump:nnnN {⟨file name⟩} {⟨start index⟩} {⟨end index⟩} ⟨tl var⟩

Sets the ⟨tl var⟩ to the result of applying \file_hex_dump:n/\file_hex_dump:nnn to the ⟨file⟩. If the file is not found, the ⟨tl var⟩ will be set to \q_no_value.

\file_mdfive_hash:n ☆
\file_mdfive_hash:V ☆

\file_mdfive_hash:n {⟨file name⟩}

Searches for ⟨file name⟩ using the current TeX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most TeX behavior there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.

\file_get_mdfive_hash:nN
\file_get_mdfive_hash:VN
\file_get_mdfive_hash:nNTF
\file_get_mdfive_hash:VNTF

\file_get_mdfive_hash:nN {⟨file name⟩} ⟨tl var⟩

Sets the ⟨tl var⟩ to the result of applying \file_mdfive_hash:n to the ⟨file⟩. If the file is not found, the ⟨tl var⟩ will be set to \q_no_value.

\file_size:n ☆
\file_size:V ☆

\file_size:n {⟨file name⟩}

Searches for ⟨file name⟩ using the current TeX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.

\file_get_size:nN
\file_get_size:VN
\file_get_size:nNTF
\file_get_size:VNTF

\file_get_size:nN {⟨file name⟩} ⟨tl var⟩

Sets the ⟨tl var⟩ to the result of applying \file_size:n to the ⟨file⟩. If the file is not found, the ⟨tl var⟩ will be set to \q_no_value.

\file_timestamp:n ☆
\file_timestamp:V ☆

\file_timestamp:n {⟨file name⟩}

Searches for ⟨file name⟩ using the current TeX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form D:⟨year⟩⟨month⟩⟨day⟩⟨hour⟩⟨minute⟩⟨second⟩⟨offset⟩, where the latter may be Z (UTC) or ⟨plus-minus⟩⟨hours⟩'⟨minutes⟩'. When the file is not found, the result of expansion is empty.

\file_get_timestamp:nN
\file_get_timestamp:VN
\file_get_timestamp:nNTF
\file_get_timestamp:VNTF

\file_get_timestamp:nN {⟨file name⟩} ⟨tl var⟩

Sets the ⟨tl var⟩ to the result of applying \file_timestamp:n to the ⟨file⟩. If the file is not found, the ⟨tl var⟩ will be set to \q_no_value.

\file_compare_timestamp_p:nNn ★
\file_compare_timestamp_p:(nNV|VNn|VNV) ★
\file_compare_timestamp:nNnTF ★
\file_compare_timestamp:(nNV|VNn|VNV)TF ★

\file_compare_timestamp_p:nNn {⟨file-1⟩} ⟨relation⟩ {⟨file-2⟩}
\file_compare_timestamp:nNnTF {⟨file-1⟩} ⟨relation⟩ {⟨file-2⟩} {⟨true code⟩} {⟨false code⟩}

Compares the file stamps on the two ⟨files⟩ as indicated by the ⟨relation⟩, and inserts either the ⟨true code⟩ or ⟨false case⟩ as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
  {
    % Code to regenerate derived file
  }
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different.

\file_get_full_name:nN
\file_get_full_name:VN
\file_get_full_name:nNTF
\file_get_full_name:VNTF

\file_get_full_name:nN {⟨file name⟩} ⟨tl var⟩
\file_get_full_name:nNTF {⟨file name⟩} ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

Searches for ⟨file name⟩ in the path as detailed for \file_if_exist:nTF, and if found sets the ⟨tl var⟩ the fully-qualified name of the file, i.e., the path and file name. This includes an extension .tex when the given ⟨file name⟩ has no extension but the file found has that extension. In the non-branching version, the ⟨tl var⟩ will be set to \q_no_value in the case that the file does not exist.

\file_full_name:n ☆
\file_full_name:V ☆

\file_full_name:n {⟨file name⟩}

Searches for ⟨file name⟩ in the path as detailed for \file_if_exist:nTF, and if found leaves the fully-qualified name of the file, i.e., the path and file name, in the input stream. This includes an extension .tex when the given ⟨file name⟩ has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.

\file_parse_full_name:nNNN
\file_parse_full_name:VNNN

Updated: 2020-06-24

\file_parse_full_name:nNNN {⟨full name⟩} ⟨dir⟩ ⟨name⟩ ⟨ext⟩

Parses the ⟨full name⟩ and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The ⟨dir⟩: everything up to the last / (path separator) in the ⟨file path⟩. As with system PATH variables and related functions, the ⟨dir⟩ does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), ⟨dir⟩ is empty.
- The ⟨name⟩: everything after the last / up to the last ., where both of those characters are optional. The ⟨name⟩ may contain multiple . characters. It is empty if ⟨full name⟩ consists only of a directory name.
- The ⟨ext⟩: everything after the last . (including the dot). The ⟨ext⟩ is empty if there is no . after the last /.

Before parsing, the ⟨full name⟩ is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (") are invalid in file names and are discarded from the input.

\file_parse_full_name:n ⋆
\file_parse_full_name:V ⋆

New: 2020-06-24

\file_parse_full_name:n {⟨*full name*⟩}

Parses the ⟨*full name*⟩ as described for \file_parse_full_name:nNNN, and leaves ⟨*dir*⟩, ⟨*name*⟩, and ⟨*ext*⟩ in the input stream, each inside a pair of braces.

\file_parse_full_name_apply:nN ⋆
\file_parse_full_name_apply:VN ⋆

New: 2020-06-24

\file_parse_full_name_apply:nN {⟨*full name*⟩} ⟨*function*⟩

Parses the ⟨*full name*⟩ as described for \file_parse_full_name:nNNN, and passes ⟨*dir*⟩, ⟨*name*⟩, and ⟨*ext*⟩ as arguments to ⟨*function*⟩, as an n-type argument each, in this order.

12.2.3 Accessing file contents

\file_get:nnN
\file_get:VnN
\file_get:nnN*TF*
\file_get:VnN*TF*

\file_get:nnN {⟨*file name*⟩} {⟨*setup*⟩} ⟨*tl var*⟩
\file_get:nnNTF {⟨*file name*⟩} {⟨*setup*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Defines ⟨*tl var*⟩ to the contents of ⟨*file name*⟩. Category codes may need to be set appropriately via the ⟨*setup*⟩ argument. The non-branching version sets the ⟨*tl var*⟩ to \q_no_value if the file is not found. The branching version runs the ⟨*true code*⟩ after the assignment to ⟨*tl var*⟩ if the file is found, and ⟨*false code*⟩ otherwise. The file content will be tokenized using the current category code régime,

\file_input:n
\file_input:V

\file_input:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ in the path as detailed for \file_if_exist:nTF, and if found reads in the file as additional LaTeX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

\file_input_raw:n ⋆
\file_input_raw:V ⋆

New: 2023-05-18
Updated: 2025-05-26

\file_input_raw:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ in the path as detailed for \file_if_exist:nTF, and if found reads in the file as additional TeX source. No data concerning the file is tracked. If the file is not found, no action is taken.

TeXhackers note: This function *requires* the availability of the \input primitive accepting braces (LuaTeX or other engines from TeX Live 2020 onwards.)

This function is intended only for contexts where files must be read purely by expansion, for example at the start of a table cell in an \halign.

\file_if_exist_input:n
\file_if_exist_input:V
\file_if_exist_input:nF
\file_if_exist_input:VF

\file_if_exist_input:n {⟨*file name*⟩}
\file_if_exist_input:nF {⟨*file name*⟩} {⟨*false code*⟩}

Searches for ⟨*file name*⟩ using the current TeX search path and the additional paths included in \l_file_search_path_seq. If found then reads in the file as additional LaTeX source as described for \file_input:n, otherwise inserts the ⟨*false code*⟩. Note that these functions do not raise an error if the file is not found, in contrast to \file_input:n.

\file_input_stop: \file_input_stop:

Ends the reading of a file started by \file_input:n or similar before the end of the file is reached. Where the file reading is being terminated due to an error, \msg_-critical:nn(nn) should be preferred.

TeXhackers note: This function must be used on a line on its own: TeX reads files line-by-line and so any additional tokens in the "current" line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn't help as it is the line where it is used that counts!

\file_show_list: \file_show_list:
\file_log_list: \file_log_list:

These functions list all files loaded by LaTeX 2$_\varepsilon$ commands that populate \@filelist or by \file_input:n. While \file_show_list: displays the list in the terminal, \file_-log_list: outputs it to the log file only.

Chapter 13

The l3luatex module LuaTEX-specific functions

The LuaTEX engine provides access to the Lua programming language, and with it access to the "internals" of TEX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTEX, pTEX, upTEX or XeTEX these raise an error: use `\sys_if_engine_luatex:T` to avoid this. Details on using Lua with the LuaTEX engine are given in the LuaTEX manual.

13.1 Breaking out to Lua

`\lua_now:n` ⋆
`\lua_now:e` ⋆

`\lua_now:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TEX, which includes converting line ends to spaces in the usual TEX manner and which respects currently-applicable TEX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the ⟨*Lua input*⟩ immediately, and in an expandable manner.

TEXhackers note: `\lua_now:e` is a macro wrapper around `\directlua`: when LuaTEX is in use two expansions are required to yield the result of the Lua code.

`\lua_shipout_e:n`
`\lua_shipout:n`

`\lua_shipout:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TEX, which includes converting line ends to spaces in the usual TEX manner and which respects currently-applicable TEX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter when the current page is finalized (i.e., at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨*Lua input*⟩ during the page-building routine: no TEX expansion of the ⟨*Lua input*⟩ will occur at this stage.

In the case of the `\lua_shipout_e:n` version the input is fully expanded by TEX in an `e`-type manner during the shipout operation.

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

\lua_escape:n ★
\lua_escape:e ★

\lua_escape:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to \n and \r, respectively.

TeXhackers note: \lua_escape:e is a macro wrapper around \luaescapestring: when LuaTeX is in use two expansions are required to yield the result of the Lua code.

\lua_load_module:n

New: 2022-05-14

\lua_load_module:n {⟨*Lua module name*⟩}

Loads a Lua module into the Lua interpreter.

\lua_now:n passes its {⟨*token list*⟩} argument to the Lua interpreter as a single line, with characters interpreted under the current catcode régime. These two facts mean that \lua_now:n rarely behaves as expected for larger pieces of code. Therefore, package authors should **not** write significant amounts of Lua code in the arguments to \lua_now:n. Instead, it is strongly recommended that they write the majorty of their Lua code in a separate file, and then load it using \lua_load_module:n.

TeXhackers note: This is a wrapper around the Lua call require '⟨*module*⟩'.

13.2 Lua interfaces

As well as interfaces for TeX, there are a small number of Lua functions provided here.

ltx.utils

Most public interfaces provided by the module are stored within the ltx.utils table.

ltx.utils.filedump

⟨*dump*⟩ = ltx.utils.filedump(⟨*file*⟩,⟨*offset*⟩,⟨*length*⟩)

Returns the uppercase hexadecimal representation of the content of the ⟨*file*⟩ read as bytes. If the ⟨*length*⟩ is given, only this part of the file is returned; similarly, one may specify the ⟨*offset*⟩ from the start of the file. If the ⟨*length*⟩ is not given, the entire file is read starting at the ⟨*offset*⟩.

ltx.utils.filemd5sum

⟨*hash*⟩ = ltx.utils.filemd5sum(⟨*file*⟩)

Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal TeX behavior. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised.*

ltx.utils.filemoddate

⟨*date*⟩ = ltx.utils.filemoddate(⟨*file*⟩)

Returns the date/time of last modification of the ⟨*file*⟩ in the format

D:⟨*year*⟩⟨*month*⟩⟨*day*⟩⟨*hour*⟩⟨*minute*⟩⟨*second*⟩⟨*offset*⟩

where the latter may be Z (UTC) or ⟨*plus-minus*⟩⟨*hours*⟩'⟨*minutes*⟩'. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised.*

`ltx.utils.filesize` `size = ltx.utils.filesize(⟨file⟩)`

Returns the size of the ⟨*file*⟩ in bytes. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised*.

Chapter 14

The l3legacy module Interfaces to legacy concepts

There are a small number of TeX or LaTeX 2ε concepts which are not used in expl3 code but which need to be manipulated when working as a LaTeX 2ε package. To allow these to be integrated cleanly into expl3 code, a set of legacy interfaces are provided here.

`\legacy_if_p:n ⋆`
`\legacy_if:nTF ⋆`

`\legacy_if_p:n {⟨name⟩}`
`\legacy_if:nTF {⟨name⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the LaTeX 2ε/plain TeX conditional (generated by `\newif`) is `true` or `false` and branches accordingly. The ⟨*name*⟩ of the conditional should *omit* the leading `if`.

`\legacy_if_set_true:n`
`\legacy_if_set_false:n`
`\legacy_if_gset_true:n`
`\legacy_if_gset_false:n`

New: 2021-05-10

`\legacy_if_set_true:n {⟨name⟩}`
`\legacy_if_set_false:n {⟨name⟩}`

Sets the LaTeX 2ε/plain TeX conditional `\if⟨name⟩` (generated by `\newif`) to be `true` or `false`.

`\legacy_if_set:nn`
`\legacy_if_gset:nn`

New: 2021-05-10

`\legacy_if_set:nn {⟨name⟩} {⟨boolexpr⟩}`

Sets the LaTeX 2ε/plain TeX conditional `\if⟨name⟩` (generated by `\newif`) to the result of evaluating the ⟨*boolean expression*⟩.

Part IV

Data types

Chapter 15

The l3tl module Token lists

TeX works with tokens, and LaTeX3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called "tl var" (⟨*tl var*⟩), which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two "views" of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of "items", or a list of "tokens". An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or ␣, `{`, or `}` (assuming normal TeX category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (`Hello`, `w`, `o`, `r`, `l` and `d`), but thirteen tokens (`{`, `H`, `e`, `l`, `l`, `o`, `}`, ␣, `w`, `o`, `r`, `l` and `d`). Functions which act on items are often faster than their analogue acting directly on tokens.

15.1 Creating and initializing token list variables

`\tl_new:N`
`\tl_new:c`

`\tl_new:N` ⟨*tl var*⟩

Creates a new ⟨*tl var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*tl var*⟩ is initially empty.

\tl_const:Nn
\tl_const:(NV|Ne|cn|cV|ce)

\tl_const:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

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

\tl_clear:N
\tl_clear:c
\tl_gclear:N
\tl_gclear:c

\tl_clear:N ⟨*tl var*⟩

Clears all entries from the ⟨*tl var*⟩.

\tl_clear_new:N
\tl_clear_new:c
\tl_gclear_new:N
\tl_gclear_new:c

\tl_clear_new:N ⟨*tl var*⟩

Ensures that the ⟨*tl var*⟩ exists globally by applying \tl_new:N if necessary, then applies \tl_(g)clear:N to leave the ⟨*tl var*⟩ empty.

\tl_set_eq:NN
\tl_set_eq:(cN|Nc|cc)
\tl_gset_eq:NN
\tl_gset_eq:(cN|Nc|cc)

\tl_set_eq:NN ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩

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

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

\tl_concat:NNN ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩ ⟨*tl var*$_3$⟩

Concatenates the content of ⟨*tl var*$_2$⟩ and ⟨*tl var*$_3$⟩ together and saves the result in ⟨*tl var*$_1$⟩. The ⟨*tl var*$_2$⟩ is placed at the left side of the new token list.

\tl_if_exist_p:N ⋆
\tl_if_exist_p:c ⋆
\tl_if_exist:N*TF* ⋆
\tl_if_exist:c*TF* ⋆

\tl_if_exist_p:N ⟨*tl var*⟩
\tl_if_exist:NTF ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*tl var*⟩ is currently defined. This does not check that the ⟨*tl var*⟩ really is a token list variable.

15.2 Adding data to token list variables

\tl_set:Nn
\tl_set:(NV|Nv|No|Ne|Nf|cn|cV|cv|co|ce|cf)
\tl_gset:Nn
\tl_gset:(NV|Nv|No|Ne|Nf|cn|cV|cv|co|ce|cf)

\tl_set:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, removing any previous content from the variable.

\tl_put_left:Nn
\tl_put_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\tl_gput_left:Nn
\tl_gput_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\tl_put_left:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

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

\tl_put_right:Nn
\tl_put_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\tl_gput_right:Nn
\tl_gput_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\tl_put_right:Nn ⟨tl var⟩ {⟨tokens⟩}

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

15.3 Token list conditionals

\tl_if_blank_p:n ⋆
\tl_if_blank_p:(e|V|o) ⋆
\tl_if_blank:nTF ⋆
\tl_if_blank:(e|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:NTF ⋆
\tl_if_empty:cTF ⋆

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

Tests if the ⟨tl var⟩ is entirely empty (i.e., contains no tokens at all).

\tl_if_empty_p:n ⋆
\tl_if_empty_p:(V|o|e) ⋆
\tl_if_empty:nTF ⋆
\tl_if_empty:(V|o|e)TF ⋆

\tl_if_empty_p:n {⟨token list⟩}
\tl_if_empty:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

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

\tl_if_eq_p:NN ⋆
\tl_if_eq_p:(Nc|cN|cc) ⋆
\tl_if_eq:NNTF ⋆
\tl_if_eq:(Nc|cN|cc)TF ⋆

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

Compares the content of ⟨tl var₁⟩ and ⟨tl var₂⟩ and is logically `true` if the two contain the same list of tokens (i.e., identical in both the list of characters they contain and the category codes of those characters). Thus for example

```
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Ne \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields `false`. See also `\str_if_eq:nnTF` for a comparison that ignores category codes.

\tl_if_eq:NnTF
\tl_if_eq:cnTF
New: 2020-07-14

\tl_if_eq:NnTF ⟨tl var₁⟩ {⟨token list₂⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨tl var₁⟩ and the ⟨token list₂⟩ contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see `\tl_if_eq:NNTF` for an expandable version when both token lists are stored in variables, or `\str_if_eq:nnTF` if category codes are not important.

\tl_if_eq:nnTF
\tl_if_eq:(nV|ne|Vn|en|ee)TF

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

Tests if ⟨token list₁⟩ and ⟨token list₂⟩ contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see `\tl_if_eq:NNTF` for an expandable version when token lists are stored in variables, or `\str_if_eq:nnTF` if category codes are not important.

\tl_if_in:NnTF
\tl_if_in:(NV|No|cn|cV|co)TF

\tl_if_in:NnTF ⟨*tl var*⟩ {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

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

\tl_if_in:nnTF
\tl_if_in:(Vn|VV|on|oo|nV|no)TF

\tl_if_in:nnTF {⟨*token 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). The search does *not* enter brace (category code 1/2) groups.

\tl_if_novalue_p:n ⋆
\tl_if_novalue:nTF ⋆

\tl_if_novalue_p:n {⟨*token list*⟩}
\tl_if_novalue:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ and the special \c_novalue_tl marker contain the same list of tokens, both in respect of character codes and category codes. This means that \exp_args:No \tl_if_novalue:nTF { \c_novalue_tl } is logically true but \tl_if_novalue:nTF { \c_novalue_tl } is logically false. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

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

\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 ⋆

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

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

\tl_if_single_token_p:n ⋆
\tl_if_single_token:nTF ⋆

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

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

\tl_if_regex_match:nnTF
\tl_if_regex_match:VnTF
\tl_if_regex_match:nNTF
\tl_if_regex_match:VNTF

New: 2024-12-08

\tl_if_regex_match:nnTF {⟨*token list*⟩} {⟨*regex*⟩} {⟨*true code*⟩} {⟨*false code*⟩}
\tl_if_regex_match:nNTF {⟨*token list*⟩} ⟨*regex var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

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

leaves TRUE then FALSE in the input stream. Theses are alternative names for \regex_if_match:nnTF and friends, with arguments re-ordered for ⟨*token list*⟩ testing; see l3regex chapter for more details of the ⟨*regex*⟩ format.

15.3.1 Testing the first token

\tl_if_head_eq_catcode_p:nN ⋆
\tl_if_head_eq_catcode_p:(VN|eN|oN) ⋆
\tl_if_head_eq_catcode:nNTF ⋆
\tl_if_head_eq_catcode:(VN|eN|oN)TF ⋆

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

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

\tl_if_head_eq_charcode_p:nN ⋆
\tl_if_head_eq_charcode_p:(VN|eN|fN) ⋆
\tl_if_head_eq_charcode:nNTF ⋆
\tl_if_head_eq_charcode:(VN|eN|fN)TF ⋆

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

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

\tl_if_head_eq_meaning_p:nN ⋆
\tl_if_head_eq_meaning_p:(VN|eN) ⋆
\tl_if_head_eq_meaning:nNTF ⋆
\tl_if_head_eq_meaning:(VN|eN)TF ⋆

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

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

\tl_if_head_is_group_p:n ⋆
\tl_if_head_is_group:nTF ⋆

\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_p:n {⟨token list⟩}
\tl_if_head_is_N_type:nTF ⋆ \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_p:n {⟨token list⟩}
\tl_if_head_is_space:nTF ⋆ \tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the first ⟨token⟩ in the ⟨token list⟩ is an explicit space character (explicit token with character code 32 and category code 10). In particular, the test is false if the ⟨token list⟩ starts with an implicit token such as \c_space_token, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

15.4 Working with token lists as a whole

15.4.1 Using token lists

\tl_to_str:n ⋆ \tl_to_str:n {⟨token list⟩}
\tl_to_str:(o|V|v|e) ⋆

Converts the ⟨token list⟩ to a ⟨string⟩, leaving the resulting character tokens in the input stream. A ⟨string⟩ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). The base function requires only a single expansion. Its argument *must* be braced.

TEXhackers note: This is the ε-TEX primitive \detokenize. Converting a ⟨token list⟩ to a ⟨string⟩ yields a concatenation of the string representations of every token in the ⟨token list⟩. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter \escapechar, absent if the \escapechar is negative or greater than the largest character code;
- the control sequence name, as defined by \cs_to_str:N;
- a space, unless the control sequence name is a single character whose category at the time of expansion of \tl_to_str:n is not "letter".

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally #). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the \escapechar: for instance \tl_to_str:n {\a} normally produces the three character "backslash", "lower-case a", "space", but it may also produce a single "lower-case a" if the escape character is negative and a is currently not a letter.

\tl_to_str:N ⋆
\tl_to_str:c ⋆

\tl_to_str:N ⟨*tl var*⟩

Converts the content of the ⟨*tl var*⟩ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This ⟨*string*⟩ is then left in the input stream. For low-level details, see the notes given for \tl_to_str:n.

\tl_use:N ⋆
\tl_use:c ⋆

\tl_use:N ⟨*tl var*⟩

Recovers the content of a ⟨*tl var*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a ⟨*tl var*⟩ directly without an accessor function.

15.4.2 Counting and reversing token lists

\tl_count:n ⋆
\tl_count:(V|v|e|o) ⋆

\tl_count:n {⟨*token list*⟩}

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

\tl_count:N ⋆
\tl_count:c ⋆

\tl_count:N ⟨*tl var*⟩

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

\tl_count_tokens:n ⋆

\tl_count_tokens:n {⟨*token list*⟩}

Counts the number of TeX tokens in the ⟨*token list*⟩ and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of a~{bc} is 6.

\tl_reverse:n ⋆
\tl_reverse:(V|o|f|e) ⋆

\tl_reverse:n {⟨*token list*⟩}

Reverses the order of the ⟨*items*⟩ in the ⟨*token list*⟩, so that ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ ...⟨$item_n$⟩ becomes ⟨$item_n$⟩...⟨$item_3$⟩⟨$item_2$⟩⟨$item_1$⟩. This process preserves unprotected space within the ⟨*token list*⟩. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider \tl_reverse_items:n. See also \tl_reverse:N.

TeXhackers note: The result is returned within \unexpanded, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

\tl_reverse:N
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c

\tl_reverse:N ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to contain the result of reversing the order of its ⟨*items*⟩, so that ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ … ⟨$item_n$⟩ becomes ⟨$item_n$⟩… ⟨$item_3$⟩⟨$item_2$⟩⟨$item_1$⟩. This process preserves unprotected spaces within the ⟨*tl var*⟩. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. This is equivalent to a combination of an assignment and \tl_reverse:V. See also \tl_reverse_items:n for improved performance.

\tl_reverse_items:n ⋆

\tl_reverse_items:n {⟨*token list*⟩}

Reverses the order of the ⟨*items*⟩ in the ⟨*token list*⟩, so that ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ … ⟨$item_n$⟩ becomes {⟨$item_n$⟩} … {⟨$item_3$⟩}{⟨$item_2$⟩}{⟨$item_1$⟩}. This process removes any unprotected space within the ⟨*token list*⟩. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function \tl_reverse:n.

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

\tl_trim_spaces:n ⋆
\tl_trim_spaces:(V|v|e|o) ⋆

\tl_trim_spaces:n {⟨*token list*⟩}

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and leaves the result in the input stream.

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

\tl_trim_left_spaces:n ⋆
\tl_trim_left_spaces:(V|v|e|o) ⋆
\tl_trim_right_spaces:n ⋆
\tl_trim_right_spaces:(V|v|e|o) ⋆

New: 2025-02-02

\tl_trim_left_spaces:n {⟨*token list*⟩}

Analogue of \tl_trim_spaces:n which removes any leading *or* trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and leaves the result in the input stream.

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

\tl_trim_spaces_apply:nN ⋆
\tl_trim_spaces_apply:oN ⋆

\tl_trim_spaces_apply:nN {⟨*token list*⟩} ⟨*function*⟩

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and passes the result to the ⟨*function*⟩ as an n-type argument.

\tl_trim_left_spaces_apply:nN ⋆
\tl_trim_left_spaces_apply:oN ⋆
\tl_trim_right_spaces_apply:nN ⋆
\tl_trim_right_spaces_apply:oN ⋆

New: 2025-02-02

\tl_trim_left_spaces_apply:nN {⟨*token list*⟩} ⟨*function*⟩

Analogue of \tl_trim_spaces_apply:nN which removes any leading *or* trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and passes the result to the ⟨*function*⟩ as an n-type argument.

\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c

\tl_trim_spaces:N ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.

\tl_trim_left_spaces:N
\tl_trim_left_spaces:c
\tl_trim_right_spaces:N
\tl_trim_right_spaces:c
\tl_gtrim_left_spaces:N
\tl_gtrim_left_spaces:c
\tl_gtrim_right_spaces:N
\tl_gtrim_right_spaces:c

New: 2025-02-02

\tl_trim_left_spaces:N ⟨*tl var*⟩

Analogue of \tl_trim_spaces:N which sets the ⟨*tl var*⟩ to contain the result of removing any leading *or* trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.

15.4.3 Viewing token lists

\tl_show:N
\tl_show:c

Updated: 2021-04-29

\tl_show:N ⟨*tl var*⟩

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

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

\tl_show:n
\tl_show:e

\tl_show:n {⟨*token list*⟩}

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

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

\tl_log:N
\tl_log:c

Updated: 2021-04-29

\tl_log:N ⟨*tl var*⟩

Writes the content of the ⟨*tl var*⟩ in the log file. See also \tl_show:N which displays the result in the terminal.

\tl_log:n
\tl_log:(e|x)

\tl_log:n {⟨*token list*⟩}

Writes the ⟨*token list*⟩ in the log file. See also \tl_show:n which displays the result in the terminal.

15.5 Manipulating items in token lists

15.5.1 Mapping over token lists

All mappings are done at the current group level, i.e., any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

`\tl_map_function:NN` ☆
`\tl_map_function:cN` ☆

`\tl_map_function:NN` ⟨*tl var*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ in the ⟨*tl var*⟩. The ⟨*function*⟩ receives one argument for each iteration. This may be a number of tokens if the ⟨*item*⟩ was stored within braces. Hence the ⟨*function*⟩ should anticipate receiving `n`-type arguments. See also `\tl_map_function:nN`.

`\tl_map_function:nN` ☆
`\tl_map_function:eN` ☆

`\tl_map_function:nN {`⟨*token list*⟩`}` ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ in the ⟨*token list*⟩, The ⟨*function*⟩ receives one argument for each iteration. This may be a number of tokens if the ⟨*item*⟩ was stored within braces. Hence the ⟨*function*⟩ should anticipate receiving `n`-type arguments. See also `\tl_map_function:NN`.

`\tl_map_inline:Nn`
`\tl_map_inline:cn`

`\tl_map_inline:Nn` ⟨*tl var*⟩ `{`⟨*inline function*⟩`}`

Applies the ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*tl var*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as `#1`. See also `\tl_map_function:NN`.

`\tl_map_inline:nn`

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

Applies the ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*token list*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as `#1`. See also `\tl_map_function:nN`.

`\tl_map_tokens:Nn` ☆
`\tl_map_tokens:cn` ☆
`\tl_map_tokens:nn` ☆

`\tl_map_tokens:Nn` ⟨*tl var*⟩ `{`⟨*code*⟩`}`
`\tl_map_tokens:nn {`⟨*token list*⟩`} {`⟨*code*⟩`}`

Analogue of `\tl_map_function:NN` which maps several tokens instead of a single function. The ⟨*code*⟩ receives each ⟨*item*⟩ in the ⟨*tl var*⟩ or in the ⟨*token list*⟩ as a trailing brace group. For instance,

```
\tl_map_tokens:Nn \l_my_tl { \prg_replicate:nn { 2 } }
```

expands to twice each ⟨*item*⟩ in the ⟨*tl var*⟩: for each ⟨*item*⟩ in `\l_my_tl` the function `\prg_replicate:nn` receives `2` and ⟨*item*⟩ as its two arguments. The function `\tl_map_inline:Nn` is typically faster but is not expandable.

`\tl_map_variable:NNn`
`\tl_map_variable:cNn`

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

Stores each ⟨*item*⟩ of the ⟨*tl var*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*tl var*⟩, or its original value if the ⟨*tl var*⟩ is blank. See also `\tl_map_inline:Nn`.

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

Stores each ⟨*item*⟩ of the ⟨*token list*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*tl var*⟩, or its original value if the ⟨*tl var*⟩ is blank. See also \tl_map_inline:nn.

\tl_map_break: ☆ \tl_map_break:

Used to terminate a \tl_map_... function before all entries in the ⟨*token list*⟩ have been processed. This normally takes place within a conditional statement, for example

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

See also \tl_map_break:n. Use outside of a \tl_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\tl_map_break:n ☆ \tl_map_break:n {⟨*code*⟩}

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

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

Use outside of a \tl_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

15.5.2 Head and tail of token lists

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

`\tl_head:N` ⋆
`\tl_head:n` ⋆
`\tl_head:(V|v|f|e)` ⋆

`\tl_head:n {⟨token list⟩}`

Leaves in the input stream the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

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

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

yields ␣`ab`. A blank ⟨*token list*⟩ (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `e`-type or `x`-type argument expansion.

`\tl_head:w` ⋆

`\tl_head:w ⟨token list⟩ { } \q_stop`

Leaves in the input stream the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank ⟨*token list*⟩ (which consists only of space characters) results in a low-level TEX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a "blank" argument. This function requires only a single expansion, and thus is suitable for use within an `o`-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

`\tl_tail:N` ⋆
`\tl_tail:n` ⋆
`\tl_tail:(V|v|f|e)` ⋆

`\tl_tail:n {⟨token list⟩}`

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

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

and

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

both leave ␣`{bc}d` in the input stream. A blank ⟨*token list*⟩ (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `e`-type or `x`-type argument expansion.

If you wish to handle token lists where the first token may be a space, and this needs to be treated as the head/tail, this can be accomplished using \tl_if_head_is_-space:nTF, for example

```
\exp_last_unbraced:NNo
  \cs_new:Npn \__mypkg_gobble_space:w \c_space_tl { }
\cs_new:Npn \mypkg_tl_head_keep_space:n #1
  {
    \tl_if_head_is_space:nTF {#1}
      { ~ }
      { \tl_head:n {#1} }
  }
\cs_new:Npn \mypkg_tl_tail_keep_space:n #1
  {
    \tl_if_head_is_space:nTF {#1}
      { \exp_not:o { \__mypkg_gobble_space:w #1 } }
      { \tl_tail:n {#1} }
  }
```

15.5.3 Items and ranges in token lists

\tl_item:nn ⋆
\tl_item:Nn ⋆
\tl_item:cn ⋆

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

Indexing items in the ⟨*token list*⟩ from 1 on the left, this function evaluates the ⟨*integer expression*⟩ and leaves the appropriate item from the ⟨*token list*⟩ in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then the function expands to nothing.

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

\tl_rand_item:N ⋆
\tl_rand_item:c ⋆
\tl_rand_item:n ⋆

\tl_rand_item:N ⟨*tl var*⟩
\tl_rand_item:n {⟨*token list*⟩}

Selects a pseudo-random item of the ⟨*token list*⟩. If the ⟨*token list*⟩ is blank, the result is empty.

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

`\tl_range:Nnn` ⋆ `\tl_range:Nnn` ⟨*tl var*⟩ {⟨*start index*⟩} {⟨*end index*⟩}
`\tl_range:nnn` ⋆ `\tl_range:nnn` {⟨*token list*⟩} {⟨*start index*⟩} {⟨*end index*⟩}

Leaves in the input stream the items from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here ⟨*start index*⟩ and ⟨*end index*⟩ should be ⟨*integer expressions*⟩. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', and a negative index means 'from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$.

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to `\tl_trim_spaces:n` to begin with.

Thus, with $l = 7$ as in the examples below, all of the following are equivalent and result in the whole token list

```
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 12 }
\tl_range:nnn { abcd~{e{}}fg } { -7 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { -12 } { 7 }
```

Here are some more interesting examples. The calls

```
\iow_term:e { \tl_range:nnn { abcd{e{}}fg } { 2 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd{e{}}fg } { 2 } { -3 } }
\iow_term:e { \tl_range:nnn { abcd{e{}}fg } { -6 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd{e{}}` on the terminal; similarly

```
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd {e{}}` on the terminal (note the space in the middle). To the contrary,

```
\tl_range:nnn { abcd~{e{}}f } { 2 } { 4 }
```

will discard the space after 'd'.

If we want to get the items from, say, the third to the last in a token list `<tl>`, the call is `\tl_range:nnn { <tl> } { 3 } { -1 }`. Similarly, for discarding the last item, we can do `\tl_range:nnn { <tl> } { 1 } { -2 }`.

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

15.5.4 Formatting token lists

\tl_format:Nn ⋆
\tl_format:cn ⋆
\tl_format:nn ⋆
New: 2025-06-09

\tl_format:Nn ⟨*tl var*⟩ {⟨*format specification*⟩}
\tl_format:nn {⟨*token list*⟩} {⟨*format specification*⟩}

Converts the ⟨*tl var*⟩ or ⟨*token list*⟩ to a string according to the ⟨*format specification*⟩. The ⟨*style*⟩, if present, must be s. If ⟨*precision*⟩ is given, all characters of the string representation of the ⟨*token list*⟩ beyond the first ⟨*precision*⟩ characters are discarded. The details of the ⟨*format specification*⟩ are described in Section 19.1.

15.5.5 Sorting token lists

\tl_sort:Nn
\tl_sort:cn
\tl_gsort:Nn
\tl_gsort:cn

\tl_sort:Nn ⟨*tl var*⟩ {⟨*comparison code*⟩}

Sorts the items in the ⟨*tl var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*tl var*⟩. The details of sorting comparison are described in Section 6.1.

\tl_sort:nN ⋆

\tl_sort:nN {⟨*token list*⟩} ⟨*conditional*⟩

Sorts the items in the ⟨*token list*⟩, using the ⟨*conditional*⟩ to compare items, and leaves the result in the input stream. The ⟨*conditional*⟩ should have signature :nnTF, and return true if the two items being compared should be left in the same order, and false if the items should be swapped. The details of sorting comparison are described in Section 6.1.

TEXhackers note: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

15.6 Manipulating tokens in token lists

15.6.1 Replacing tokens

Within token lists, replacement takes place at the top level: there is no recursion into brace groups (more precisely, within a group defined by a category code 1/2 pair).

\tl_replace_once:Nnn
\tl_replace_once:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)
\tl_greplace_once:Nnn
\tl_greplace_once:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)

\tl_replace_once:Nnn ⟨*tl var*⟩ {⟨*old tokens*⟩} {⟨*new tokens*⟩}

Replaces the first (leftmost) occurrence of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tl_replace_all:Nnn
\tl_replace_all:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)
\tl_greplace_all:Nnn
\tl_greplace_all:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)

\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_regex_replace_once:Nnn
\tl_regex_replace_once:cnn
\tl_regex_replace_once:NNn
\tl_regex_replace_once:cNn
\tl_regex_greplace_once:Nnn
\tl_regex_greplace_once:cnn
\tl_regex_greplace_once:NNn
\tl_regex_greplace_once:cNn

New: 2024-12-08

\tl_regex_replace_once:Nnn ⟨*tl var*⟩ {⟨*regex*⟩} {⟨*replacement*⟩}
\tl_regex_replace_once:NNn ⟨*tl var*⟩ ⟨*regex var*⟩ {⟨*replacement*⟩}

Searches for the ⟨*regular expression*⟩ in the contents of the ⟨*tl var*⟩ and replaces the first match with the ⟨*replacement*⟩. In the ⟨*replacement*⟩, \0 represents the full match, \1 represents the contents of the first capturing group, \2 of the second, etc. Theses are alternative names for \regex_replace_once:nnN and friends, with arguments re-ordered for ⟨*tl var*⟩ setting; See l3regex chapter for more details of the ⟨*regex*⟩ format.

\tl_regex_replace_all:Nnn
\tl_regex_replace_all:cnn
\tl_regex_replace_all:NNn
\tl_regex_replace_all:cNn
\tl_regex_greplace_all:Nnn
\tl_regex_greplace_all:cnn
\tl_regex_greplace_all:NNn
\tl_regex_greplace_all:cNn

New: 2024-12-08

\tl_regex_replace_all:Nnn ⟨*tl var*⟩ {⟨*regex*⟩} {⟨*replacement*⟩}
\tl_regex_replace_all:NNn ⟨*tl var*⟩ ⟨*regex var*⟩ {⟨*replacement*⟩}

Replaces all occurrences of the ⟨*regular expression*⟩ in the contents of the ⟨*tl var*⟩ by the ⟨*replacement*⟩, where \0 represents the full match, \1 represent the contents of the first capturing group, \2 of the second, etc. Every match is treated independently, and matches cannot overlap. Theses are alternative names for \regex_replace_all:nnN and friends, with arguments re-ordered for ⟨*tl var*⟩ setting; see l3regex chapter for more details of the ⟨*regex*⟩ format.

\tl_remove_once:Nn
\tl_remove_once:(NV|Ne|cn|cV|ce)
\tl_gremove_once:Nn
\tl_gremove_once:(NV|Ne|cn|cV|ce)

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

Removes the first (leftmost) occurrence of ⟨*tokens*⟩ from the ⟨*tl var*⟩. The ⟨*tokens*⟩ cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_remove_all:Nn
\tl_remove_all:(NV|Ne|cn|cV|ce)
\tl_gremove_all:Nn
\tl_gremove_all:(NV|Ne|cn|cV|ce)
```

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

Removes all occurrences of ⟨*tokens*⟩ from the ⟨*tl var*⟩. The ⟨*tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern ⟨*tokens*⟩ may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing `abcd`.

15.6.2 Reassigning category codes

These functions allow the rescanning of tokens: re-apply TEX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using `\char_generate:nn`).

```
\tl_set_rescan:Nnn
\tl_set_rescan:(NnV|Nne|Nno|cnn|cnV|cne|cno)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(NnV|Nne|Nno|cnn|cnV|cne|cno)
```

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

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, applying the category code régime specified in the ⟨*setup*⟩ before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the ⟨*setup*⟩ are those in force at the point of use of `\tl_set_rescan:Nnn`.) This allows the ⟨*tl var*⟩ to contain material with category codes other than those that apply when ⟨*tokens*⟩ are absorbed. The ⟨*setup*⟩ is run within a group and may contain any valid input, although only changes in category codes, such as uses of `\cctab_select:N`, are relevant. See also `\tl_rescan:nn`.

TEXhackers note: The ⟨*tokens*⟩ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user ⟨*setup*⟩), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

\tl_rescan:nn
\tl_rescan:nV

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

Rescans ⟨*tokens*⟩ applying the category code régime specified in the ⟨*setup*⟩, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the ⟨*setup*⟩ are those in force at the point of use of \tl_rescan:nn.) The ⟨*setup*⟩ is run within a group and may contain any valid input, although only changes in category codes, such as uses of \cctab_select:N, are relevant. See also \tl_set_-rescan:Nnn, which is more robust than using \tl_set:Nn in the ⟨*tokens*⟩ argument of \tl_rescan:nn.

TEXhackers note: The ⟨*tokens*⟩ are first turned into a string (using \tl_to_str:n). If the string contains one or more characters with character code \newlinechar (set equal to \endlinechar unless that is equal to 32, before the user ⟨*setup*⟩), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

Contrarily to the \scantokens ε-TEX primitive, \tl_rescan:nn tokenizes the whole string in the same category code régime rather than one token at a time, so that directives such as \verb that rely on changing category codes will not function properly.

15.7 Constant token lists

\c_empty_tl

Constant that is always empty.

\c_novalue_tl

A marker for the absence of an argument. This constant tl can safely be typeset (*cf.* \q_-nil), with the result being -NoValue-. It is important to note that \c_novalue_tl is constructed such that it will *not* match the simple text input -NoValue-, i.e. that

```
\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }
```

is logically false. The \c_novalue_tl marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty tl.

\c_space_tl

An explicit space character contained in a token list (compare this with \c_space_token). For use where an explicit space is required.

15.8 Scratch token lists

\l_tmpa_tl
\l_tmpb_tl

Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_tl`
`\g_tmpb_tl`

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

Chapter 16

The l3tl-build module Piecewise `tl` constructions

16.1 Constructing ⟨*tl var*⟩ by accumulation

When creating a ⟨*tl var*⟩ by accumulation of many tokens, the performance available using a combination of `\tl_set:Nn` and `\tl_put_right:Nn` or similar begins to become an issue. To address this, a set of functions are available to "build" a ⟨*tl var*⟩. The performance of this approach is much more efficient than the standard `\tl_put_right:Nn`, but the constructed token list cannot be accessed during construction other than by methods provided in this section.

Whilst the exact performance difference is dependent on the size of each added block of tokens and the total number of blocks, in general, the `\tl_build_(g)put...` functions will out-perform the basic `\tl_(g)put...` equivalent if more than 100 non-empty addition operations occur. See https://github.com/latex3/latex3/issues/1393#issuecomment-1880164756 for a more detailed analysis.

`\tl_build_begin:N`
`\tl_build_gbegin:N`

`\tl_build_begin:N` ⟨*tl var*⟩

Clears the ⟨*tl var*⟩ and sets it up to support other `\tl_build_...` functions. Until `\tl_build_end:N` ⟨*tl var*⟩ or `\tl_build_gend:N` ⟨*tl var*⟩ is called, applying any function from l3tl other than `\tl_build_...` will lead to incorrect results. The `begin` and `gbegin` functions must be used for local and global ⟨*tl var*⟩ respectively.

`\tl_build_put_left:Nn`
`\tl_build_put_left:Ne`
`\tl_build_gput_left:Nn`
`\tl_build_gput_left:Ne`
`\tl_build_put_right:Nn`
`\tl_build_put_right:Ne`
`\tl_build_gput_right:Nn`
`\tl_build_gput_right:Ne`

`\tl_build_put_left:Nn` ⟨*tl var*⟩ `{`⟨*tokens*⟩`}`
`\tl_build_put_right:Nn` ⟨*tl var*⟩ `{`⟨*tokens*⟩`}`

Adds ⟨*tokens*⟩ to the left or right side of the current contents of ⟨*tl var*⟩. The ⟨*tl var*⟩ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global ⟨*tl var*⟩ respectively. The `right` functions are about twice faster than the `left` functions.

`\tl_build_end:N`
`\tl_build_gend:N`

`\tl_build_end:N` ⟨*tl var*⟩

Gets the contents of ⟨*tl var*⟩ and stores that into the ⟨*tl var*⟩ using `\tl_set:Nn` or `\tl_gset:Nn`. The ⟨*tl var*⟩ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `end` and `gend` functions must be used for local and global ⟨*tl var*⟩ respectively. These functions completely remove the setup code that enabled ⟨*tl var*⟩ to be used for other `\tl_build_...` functions. After the action of `end`/`gend`, the ⟨*tl var*⟩ may be manipulated using standard `tl` functions.

`\tl_build_get_intermediate:NN`

New: 2023-12-14

`\tl_build_get_intermediate:NN` ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩

Stores the contents of the ⟨*tl var*$_1$⟩ in the ⟨*tl var*$_2$⟩. The ⟨*tl var*$_1$⟩ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The ⟨*tl var*$_2$⟩ is a "normal" token list variable, assigned locally using `\tl_set:Nn`.

Chapter 17

The l3str module Strings

TeX associates each character with a category code: as such, there is no concept of a "string" as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense "ignoring" category codes: this is done by treating token lists as strings in a TeX sense.

A TeX string (and thus an expl3 string) is a series of characters which have category code 12 ("other") with the exception of space characters which have category code 10 ("space"). Thus at a technical level, a TeX string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialized token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn't primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in l3basics, l3tl and l3token, respectively.

Most expandable functions in this module come in three flavors:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

17.1 Creating and initializing string variables

\str_new:N
\str_new:c

\str_new:N ⟨*str var*⟩

Creates a new ⟨*str var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*str var*⟩ is initially empty.

\str_const:Nn
\str_const:(NV|Ne|cn|cV|ce)

\str_const:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Creates a new constant ⟨*str var*⟩ or raises an error if the name is already taken. The value of the ⟨*str var*⟩ is set globally to the ⟨*token list*⟩, converted to a string.

\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c

\str_clear:N ⟨*str var*⟩

Clears the content of the ⟨*str var*⟩.

\str_clear_new:N
\str_clear_new:c
\str_gclear_new:N
\str_gclear_new:c

\str_clear_new:N ⟨*str var*⟩

Ensures that the ⟨*str var*⟩ exists globally by applying \str_new:N if necessary, then applies \str_(g)clear:N to leave the ⟨*str var*⟩ empty.

\str_set_eq:NN
\str_set_eq:(cN|Nc|cc)
\str_gset_eq:NN
\str_gset_eq:(cN|Nc|cc)

\str_set_eq:NN ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩

Sets the content of ⟨*str var*$_1$⟩ equal to that of ⟨*str var*$_2$⟩.

\str_concat:NNN
\str_concat:ccc
\str_gconcat:NNN
\str_gconcat:ccc

\str_concat:NNN ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩ ⟨*str var*$_3$⟩

Concatenates the content of ⟨*str var*$_2$⟩ and ⟨*str var*$_3$⟩ together and saves the result in ⟨*str var*$_1$⟩. The ⟨*str var*$_2$⟩ is placed at the left side of the new string variable. The ⟨*str var*$_2$⟩ and ⟨*str var*$_3$⟩ must indeed be strings, as this function does not convert their contents to a string.

\str_if_exist_p:N ⋆
\str_if_exist_p:c ⋆
\str_if_exist:N*TF* ⋆
\str_if_exist:c*TF* ⋆

\str_if_exist_p:N ⟨*str var*⟩
\str_if_exist:NTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*str var*⟩ is currently defined. This does not check that the ⟨*str var*⟩ really is a string.

17.2 Adding data to string variables

\str_set:Nn
\str_set:(NV|Ne|cn|cV|ce)
\str_gset:Nn
\str_gset:(NV|Ne|cn|cV|ce)

\str_set:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and stores the result in ⟨*str var*⟩.

\str_put_left:Nn
\str_put_left:(NV|Ne|cn|cV|ce)
\str_gput_left:Nn
\str_gput_left:(NV|Ne|cn|cV|ce)

\str_put_left:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and prepends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

\str_put_right:Nn
\str_put_right:(NV|Ne|cn|cV|ce)
\str_gput_right:Nn
\str_gput_right:(NV|Ne|cn|cV|ce)

\str_put_right:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and appends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

17.3 String conditionals

\str_if_empty_p:N ⋆
\str_if_empty_p:c ⋆
\str_if_empty:N*TF* ⋆
\str_if_empty:c*TF* ⋆
\str_if_empty_p:n ⋆
\str_if_empty:n*TF* ⋆

Updated: 2022-03-21

\str_if_empty_p:N ⟨*str var*⟩
\str_if_empty:NTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*string variable*⟩ is entirely empty (i.e., contains no characters at all).

\str_if_eq_p:NN ⋆
\str_if_eq_p:(Nc|cN|cc) ⋆
\str_if_eq:NN*TF* ⋆
\str_if_eq:(Nc|cN|cc)*TF* ⋆

\str_if_eq_p:NN ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩
\str_if_eq:NNTF ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*str variables*⟩ and is logically true if the two contain the same characters in the same order. See \tl_if_eq:NNTF to compare tokens (including their category codes) rather than characters.

\str_if_eq_p:nn ⋆
\str_if_eq_p:(Vn|on|no|nV|VV|vn|nv|ee) ⋆
\str_if_eq:nn*TF* ⋆
\str_if_eq:(Vn|on|no|nV|VV|vn|nv|ee)*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 (namely after converting them to strings), and is true if the two ⟨*strings*⟩ contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically true. See \tl_if_eq:nnTF to compare tokens (including their category codes) rather than characters.

\str_if_in:Nn*TF*
\str_if_in:cn*TF*

\str_if_in:NnTF ⟨*str var*⟩ {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ and tests if that ⟨*string*⟩ is found in the content of the ⟨*str var*⟩.

`\str_if_in:nnTF` `\str_if_in:nnTF {⟨tl1⟩} {⟨tl2⟩} {⟨true code⟩} {⟨false code⟩}`

Converts both ⟨*token lists*⟩ to ⟨*strings*⟩ and tests whether ⟨*string*$_2$⟩ is found inside ⟨*string*$_1$⟩.

```
\str_case:nn                         ★
\str_case:(Vn|on|en|nV|nv|ne)        ★
\str_case:nnTF                       ★
\str_case:(Vn|on|en|nV|nv|ne)TF      ★
\str_case:Nn                         ★
\str_case:NnTF                       ★
```

Updated: 2022-03-21

```
\str_case:nnTF {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

Compares the ⟨*test string*⟩ in turn with each of the ⟨*string case*⟩s until a match is found (all token lists are converted to strings). If the two are equal (as described for `\str_if_eq:nnTF`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

This set of functions performs no expansion on each ⟨*string case*⟩ argument, so any variable in there will be compared as a string. If expansion is needed in the ⟨*string case*⟩s, then `\str_case_e:nn(TF)` should be used instead.

```
\str_case_e:nn    ★
\str_case_e:en    ★
\str_case_e:nnTF  ★
\str_case_e:enTF  ★
```

```
\str_case_e:nnTF {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

Compares the full expansion of the ⟨*test string*⟩ in turn with the full expansion of the ⟨*string case*⟩s (all token lists are converted to strings). If the two full expansions are equal (as described for `\str_if_eq:eeTF`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case_e:nn`, which does nothing if there is no match, is also available. In `\str_case_e:nn(TF)`, the ⟨*test string*⟩ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

```
\str_compare_p:nNn ⋆
\str_compare_p:eNe ⋆
\str_compare:nNnTF ⋆
\str_compare:eNeTF ⋆
```

New: 2021-05-17

```
\str_compare_p:nNn {⟨tl1⟩} ⟨relation⟩ {⟨tl2⟩}
\str_compare:nNnTF {⟨tl1⟩} ⟨relation⟩ {⟨tl2⟩} {⟨true code⟩} {⟨false code⟩}
```

Compares the two ⟨*token lists*⟩ on a character by character basis (namely after converting them to strings) in a lexicographic order according to the character codes of the characters. The ⟨*relation*⟩ can be `<`, `=`, or `>` and the test is `true` under the following conditions:

- for `<`, if the first string is earlier than the second in lexicographic order;
- for `=`, if the two strings have exactly the same characters;
- for `>`, if the first string is later than the second in lexicographic order.

Thus for example the following is logically `true`:

```
\str_compare_p:nNn { ab } < { abc }
```

TEXhackers note: This is a wrapper around the TEX primitive `\(pdf)strcmp`. It is meant for programming and not for sorting textual contents, as it simply considers character codes and not more elaborate considerations of grapheme clusters, locale, etc.

17.4 Mapping over strings

All mappings are done at the current group level, i.e., any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

```
\str_map_function:nN ☆
\str_map_function:NN ☆
\str_map_function:cN ☆
```

```
\str_map_function:nN {⟨token list⟩} ⟨function⟩
\str_map_function:NN ⟨str var⟩ ⟨function⟩
```

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then applies ⟨*function*⟩ to every ⟨*character*⟩ in the ⟨*string*⟩ including spaces.

```
\str_map_inline:nn
\str_map_inline:Nn
\str_map_inline:cn
```

```
\str_map_inline:nn {⟨token list⟩} {⟨inline function⟩}
\str_map_inline:Nn ⟨str var⟩ {⟨inline function⟩}
```

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then applies the ⟨*inline function*⟩ to every ⟨*character*⟩ in the ⟨*str var*⟩ including spaces. The ⟨*inline function*⟩ should consist of code which receives the ⟨*character*⟩ as `#1`.

```
\str_map_tokens:nn ☆
\str_map_tokens:Nn ☆
\str_map_tokens:cn ☆
```

New: 2021-05-05

```
\str_map_tokens:nn {⟨token list⟩} {⟨code⟩}
\str_map_tokens:Nn ⟨str var⟩ {⟨code⟩}
```

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then applies ⟨*code*⟩ to every ⟨*character*⟩ in the ⟨*string*⟩ including spaces. The ⟨*code*⟩ receives each character as a trailing brace group. This is equivalent to `\str_map_function:nN` if the ⟨*code*⟩ consists of a single function.

`\str_map_variable:nNn`
`\str_map_variable:NNn`
`\str_map_variable:cNn`

`\str_map_variable:nNn {⟨token list⟩} ⟨variable⟩ {⟨code⟩}`
`\str_map_variable:NNn ⟨str var⟩ ⟨variable⟩ {⟨code⟩}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then stores each ⟨*character*⟩ in the ⟨*string*⟩ (including spaces) in turn in the (string or token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*character*⟩ in the ⟨*string*⟩, or its original value if the ⟨*string*⟩ is empty. See also `\str_map_inline:Nn`.

`\str_map_break:` ☆

`\str_map_break:`

Used to terminate a `\str_map_...` function before all characters in the ⟨*string*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
  {
    \str_if_eq:nnT { #1 } { bingo } { \str_map_break: }
    % Do something useful
  }
```

See also `\str_map_break:n`. Use outside of a `\str_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.

`\str_map_break:n` ☆

`\str_map_break:n {⟨code⟩}`

Used to terminate a `\str_map_...` function before all characters in the ⟨*string*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
  {
    \str_if_eq:nnT { #1 } { bingo }
      { \str_map_break:n { <code> } }
    % Do something useful
  }
```

Use outside of a `\str_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

17.5 Working with the content of strings

\str_use:N ⋆
\str_use:c ⋆

\str_use:N ⟨*str var*⟩

Recovers the content of a ⟨*str var*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a ⟨*str*⟩ directly without an accessor function.

\str_count:N ⋆
\str_count:c ⋆
\str_count:n ⋆
\str_count_ignore_spaces:n ⋆

\str_count:n {⟨*token list*⟩}

Leaves in the input stream the number of characters in the string representation of ⟨*token list*⟩, as an integer denotation. The functions differ in their treatment of spaces. In the case of \str_count:N and \str_count:n, all characters including spaces are counted. The \str_count_ignore_spaces:n function leaves the number of non-space characters in the input stream.

\str_count_spaces:N ⋆
\str_count_spaces:c ⋆
\str_count_spaces:n ⋆

\str_count_spaces:n {⟨*token list*⟩}

Leaves in the input stream the number of space characters in the string representation of ⟨*token list*⟩, as an integer denotation. Of course, this function has no _ignore_spaces variant.

\str_head:N ⋆
\str_head:c ⋆
\str_head:n ⋆
\str_head_ignore_spaces:n ⋆

\str_head:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ into a ⟨*string*⟩. The first character in the ⟨*string*⟩ is then left in the input stream, with category code "other". The functions differ if the first character is a space: \str_head:N and \str_head:n return a space token with category code 10 (blank space), while the \str_head_ignore_spaces:n function ignores this space character and leaves the first non-space character in the input stream. If the ⟨*string*⟩ is empty (or only contains spaces in the case of the _ignore_spaces function), then nothing is left on the input stream.

\str_tail:N ⋆
\str_tail:c ⋆
\str_tail:n ⋆
\str_tail_ignore_spaces:n ⋆

\str_tail:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: \str_tail:N and \str_tail:n only trim that space, while \str_tail_ignore_spaces:n removes the first non-space character and any space before it. If the ⟨*token list*⟩ is empty (or blank in the case of the _ignore_spaces variant), then nothing is left on the input stream.

```
\str_item:Nn               ⋆ \str_item:nn {⟨token list⟩} {⟨integer expression⟩}
\str_item:cn               ⋆
\str_item:nn               ⋆
\str_item_ignore_spaces:nn ⋆
```

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the character in position ⟨*integer expression*⟩ of the ⟨*string*⟩, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the ⟨*integer expression*⟩ is negative, characters are counted from the end of the ⟨*string*⟩. Hence, -1 is the right-most character, etc.

```
\str_range:Nn                ⋆ \str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}
\str_range:cnn               ⋆
\str_range:nnn               ⋆
\str_range_ignore_spaces:nnn ⋆
```

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the characters from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Spaces are preserved and counted as items (contrast this with `\tl_range:nnn` where spaces are not counted as items and are possibly discarded from the output).

Here ⟨*start index*⟩ and ⟨*end index*⟩ should be integer denotations. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', a negative index means 'start counting from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$. For instance,

```
\iow_term:e { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:e { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:e { \str_range:nnn { abcdef } { 0 } { -1 } }
```

prints `bcde`, `cdef`, `ef`, and an empty line to the terminal. The ⟨*start index*⟩ must always be smaller than or equal to the ⟨*end index*⟩: if this is not the case then no output is generated. Thus

```
\iow_term:e { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:e { \str_range:nnn { abcdef } { -1 } { -4 } }
```

both yield empty strings.

The behavior of `\str_range_ignore_spaces:nnn` is similar, but spaces are removed before starting the job. The input

```
\iow_term:e { \str_range:nnn { abcdefg } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abcdefg } { 2 } { -3 } }
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { 5 } }
```

```
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { -3 } }

\iow_term:e { \str_range:nnn { abc~efg } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abc~efg } { 2 } { -3 } }
\iow_term:e { \str_range:nnn { abc~efg } { -6 } { 5 } }
\iow_term:e { \str_range:nnn { abc~efg } { -6 } { -3 } }

\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } }

\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { -3 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { -3 } }
```

will print four instances of `bcde`, four instances of `bc e` and eight instances of `bcde`.

17.6 Modifying string variables

`\str_replace_once:Nnn`
`\str_replace_once:cnn`
`\str_greplace_once:Nnn`
`\str_greplace_once:cnn`

`\str_replace_once:Nnn` ⟨*str var*⟩ `{`⟨*old*⟩`}` `{`⟨*new*⟩`}`

Converts the ⟨*old*⟩ and ⟨*new*⟩ token lists to strings, then replaces the first (leftmost) occurrence of ⟨*old string*⟩ in the ⟨*str var*⟩ with ⟨*new string*⟩.

`\str_replace_all:Nnn`
`\str_replace_all:cnn`
`\str_greplace_all:Nnn`
`\str_greplace_all:cnn`

`\str_replace_all:Nnn` ⟨*str var*⟩ `{`⟨*old*⟩`}` `{`⟨*new*⟩`}`

Converts the ⟨*old*⟩ and ⟨*new*⟩ token lists to strings, then replaces all occurrences of ⟨*old string*⟩ in the ⟨*str var*⟩ with ⟨*new string*⟩. As this function operates from left to right, the pattern ⟨*old string*⟩ may remain after the replacement (see `\str_remove_all:Nn` for an example).

`\str_remove_once:Nn`
`\str_remove_once:cn`
`\str_gremove_once:Nn`
`\str_gremove_once:cn`

`\str_remove_once:Nn` ⟨*str var*⟩ `{`⟨*token list*⟩`}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then removes the first (leftmost) occurrence of ⟨*string*⟩ from the ⟨*str var*⟩.

`\str_remove_all:Nn`
`\str_remove_all:cn`
`\str_gremove_all:Nn`
`\str_gremove_all:cn`

`\str_remove_all:Nn` ⟨*str var*⟩ `{`⟨*token list*⟩`}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then removes all occurrences of ⟨*string*⟩ from the ⟨*str var*⟩. As this function operates from left to right, the pattern ⟨*string*⟩ may remain after the removal, for instance,

```
\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str
{bc}
```

results in `\l_tmpa_str` containing `abcd`.

17.7 String manipulation

\str_lowercase:n ⋆
\str_lowercase:f ⋆
\str_uppercase:n ⋆
\str_uppercase:f ⋆

\str_lowercase:n {⟨*tokens*⟩}
\str_uppercase:n {⟨*tokens*⟩}

Converts the input ⟨*tokens*⟩ to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
  {
    \cs_set_protected:cpn
      {
        user
        \str_uppercase:f { \tl_head:n {#1} }
        \str_lowercase:f { \tl_tail:n {#1} }
      }
      { #2 }
  }
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_casefold:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\text_lowercase:n(n)`, `\text_uppercase:n(n)` and `\text_titlecase_(all|first):n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

\str_casefold:n ⋆
\str_casefold:V ⋆

New: 2022-10-16

\str_casefold:n {⟨*tokens*⟩}

Converts the input ⟨*tokens*⟩ to their string representation, as described for \tl_to_-str:n, and then folds the case of the resulting ⟨*string*⟩ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for "text". The folding provided by \str_casefold:n follows the mappings provided by the Unicode Consortium, who state:

> Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by \str_casefold:n follows the "full" scheme defined by the Unicode Consortium (*e.g.* SS folds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (i.e., I always folds to i and not to ı).

\str_mdfive_hash:n ⋆
\str_mdfive_hash:e ⋆

New: 2023-05-19

\str_mdfive_hash:n {⟨*tokens*⟩}

Expands to the MD5 sum generated from the ⟨*tokens*⟩, which is converted to a ⟨*string*⟩ as described for \tl_to_str:n.

17.8 Viewing strings

\str_show:N
\str_show:c
\str_show:n

Updated: 2021-04-29

\str_show:N ⟨*str var*⟩

Displays the content of the ⟨*str var*⟩ on the terminal.

\str_log:N
\str_log:c
\str_log:n

Updated: 2021-04-29

\str_log:N ⟨*str var*⟩

Writes the content of the ⟨*str var*⟩ in the log file.

17.9 Constant strings

`\c_ampersand_str`
`\c_atsign_str`
`\c_backslash_str`
`\c_left_brace_str`
`\c_right_brace_str`
`\c_circumflex_str`
`\c_colon_str`
`\c_dollar_str`
`\c_hash_str`
`\c_percent_str`
`\c_tilde_str`
`\c_underscore_str`
`\c_zero_str`

Updated: 2020-12-22

Constant strings, containing a single character token, with category code 12.

`\c_empty_str`

New: 2023-12-07

Constant that is always empty.

17.10 Scratch strings

`\l_tmpa_str`
`\l_tmpb_str`

Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_str`
`\g_tmpb_str`

Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 18

The l3str-convert module String encoding conversions

18.1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points ("character codes") are expressed as bytes following a given "encoding". This can be UTF-16, ISO 8859-1, etc. See Table 1 for a list of supported encodings.[6]
- Bytes are translated to TeX tokens through a given "escaping". Those are defined for the most part by the `pdf` file format. See Table 2 for a list of escaping methods supported.[6]

[6]Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

⟨*Encoding*⟩	description
`utf8`	UTF-8
`utf16`	UTF-16, with byte-order mark
`utf16be`	UTF-16, big-endian
`utf16le`	UTF-16, little-endian
`utf32`	UTF-32, with byte-order mark
`utf32be`	UTF-32, big-endian
`utf32le`	UTF-32, little-endian
`iso88591`, `latin1`	ISO 8859-1
`iso88592`, `latin2`	ISO 8859-2
`iso88593`, `latin3`	ISO 8859-3
`iso88594`, `latin4`	ISO 8859-4
`iso88595`	ISO 8859-5
`iso88596`	ISO 8859-6
`iso88597`	ISO 8859-7
`iso88598`	ISO 8859-8
`iso88599`, `latin5`	ISO 8859-9
`iso885910`, `latin6`	ISO 8859-10
`iso885911`	ISO 8859-11
`iso885913`, `latin7`	ISO 8859-13
`iso885914`, `latin8`	ISO 8859-14
`iso885915`, `latin9`	ISO 8859-15
`iso885916`, `latin10`	ISO 8859-16
`clist`	comma-list of integers
⟨*empty*⟩	native (Unicode) string
`default`	like `utf8` with 8-bit engines, and like native with unicode-engines

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

⟨*Escaping*⟩	description
`bytes`, or empty	arbitrary bytes
`hex`, `hexadecimal`	byte = two hexadecimal digits
`name`	see `\pdfescapename`
`string`	see `\pdfescapestring`
`url`	encoding used in URLs

18.2 Conversion functions

\str_set_convert:Nnnn
\str_gset_convert:Nnnn

\str_set_convert:Nnnn ⟨*str var*⟩ {⟨*string*⟩} {⟨*name*$_1$⟩} {⟨*name*$_2$⟩}

This function converts the ⟨*string*⟩ from the encoding given by ⟨*name*$_1$⟩ to the encoding given by ⟨*name*$_2$⟩, and stores the result in the ⟨*str var*⟩. Each ⟨*name*⟩ can have the form ⟨*encoding*⟩ or ⟨*encoding*⟩/⟨*escaping*⟩, where the possible values of ⟨*encoding*⟩ and ⟨*escaping*⟩ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty ⟨*name*⟩ indicates the use of "native" strings, 8-bit for pdfTEX, and Unicode strings for the other two engines.

For example,

```
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
```

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark "FEFF, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the ⟨*string*⟩ is not valid according to the ⟨*escaping 1*⟩ and ⟨*encoding 1*⟩, or if it cannot be reencoded in the ⟨*encoding 2*⟩ and ⟨*escaping 2*⟩ (for instance, if a character does not exist in the ⟨*encoding 2*⟩). Erroneous input is replaced by the Unicode replacement character "FFFD, and characters which cannot be reencoded are replaced by either the replacement character "FFFD if it exists in the ⟨*encoding 2*⟩, or an encoding-specific replacement character, or the question mark character.

\str_set_convert:Nnnn*TF*
\str_gset_convert:Nnnn*TF*

\str_set_convert:NnnnTF ⟨*str var*⟩ {⟨*string*⟩} {⟨*name*$_1$⟩} {⟨*name*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

As `\str_set_convert:Nnnn`, converts the ⟨*string*⟩ from the encoding given by ⟨*name*$_1$⟩ to the encoding given by ⟨*name*$_2$⟩, and assigns the result to ⟨*str var*⟩. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the ⟨*string*⟩ is not valid according to the ⟨*name*$_1$⟩ encoding, or cannot be expressed in the ⟨*name*$_2$⟩ encoding. Instead, the ⟨*false code*⟩ is performed.

18.3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These *assume UTF-8* and *no escaping* in the input.

\str_convert_pdfname:n ⋆

\str_convert_pdfname:n {⟨*string*⟩}

As `\str_set_convert:Nnnn`, converts the ⟨*string*⟩ on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

18.4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In XeTeX/LuaTeX, would it be better to use the `^^^^....` approach to build a string from a given list of character codes? Namely, within a group, assign `0-9a-f` and all characters we want to category "other", then assign `^` the category superscript, and use `\scantokens`.
- Change `\str_set_convert:Nnnn` to expand its last two arguments.
- Describe the internal format in the code comments. Refuse code points in ["D800, "DFFF] in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps the characters `!'()*-./0123456789_` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko's stringenc). CESU?
- More escapings: ASCII85, shell escapes, lua escapes, etc.?

Chapter 19

The **l3str-format** package Formatting strings of characters

19.1 Format specifications

LaTeX3 has the notion of a string ⟨*format*⟩. The syntax follows that of Python's `format` built-in function. A ⟨*format specification*⟩ is a string of the form

⟨*format specification*⟩ = [[⟨*fill*⟩]⟨*alignment*⟩][⟨*sign*⟩][⟨*width*⟩][.⟨*precision*⟩][⟨*style*⟩]

where each [. . .] denotes an independent optional part.

- ⟨*fill*⟩ can be any character: it is assumed to be present whenever the second character of the ⟨*format specification*⟩ is a valid ⟨*alignment*⟩ character.
- ⟨*alignment*⟩ can be `<` (left alignment), `>` (right alignment), `^` (centering), or `=` (for numeric types only).
- ⟨*sign*⟩ is allowed for numeric types; it can be `+` (show a sign for positive and negative numbers), `-` (only put a sign for negative numbers), or a space (show a space or a `-`).
- ⟨*width*⟩ is the minimum number of characters of the result: if the result is naturally shorter than this ⟨*width*⟩, then it is padded with copies of the character ⟨*fill*⟩, with a position depending on the choice of ⟨*alignment*⟩. If the result is naturally longer, it is not truncated.
- ⟨*precision*⟩, whose presence is indicated by a period, can have different meanings depending on the type.
- ⟨*style*⟩ is one character, which controls how the given data should be formatted. The list of allowed ⟨*styles*⟩ depends on the type.

The choice of ⟨*alignment*⟩ `=` is only valid for numeric types: in this case the padding is inserted between the sign and the rest of the number.

Chapter 20

The l3quark module Quarks and scan marks

Two special types of constants in LaTeX3 are "quarks" and "scan marks". By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

20.1 Quarks

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the 'stop token' (i.e., `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
  { <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster.

20.2 Defining quarks

`\quark_new:N` \quark_new:N ⟨*quark*⟩

Creates a new ⟨*quark*⟩ which expands only to ⟨*quark*⟩. The ⟨*quark*⟩ is defined globally, and an error message is raised if the name was already taken.

`\q_stop` Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

`\q_mark` Used as a marker for delimited arguments when \q_stop is already in use.

`\q_nil` Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to \q_stop, which is only ever used as a delimiter).

`\q_no_value` A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a "return" value by functions such as \prop_get:NnN if there is no data to return.

20.3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

`\quark_if_nil_p:N` ⋆
`\quark_if_nil:NTF` ⋆

\quark_if_nil_p:N ⟨*token*⟩
\quark_if_nil:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is equal to \q_nil.

`\quark_if_nil_p:n` ⋆
`\quark_if_nil_p:(o|V)` ⋆
`\quark_if_nil:nTF` ⋆
`\quark_if_nil:(o|V)TF` ⋆

\quark_if_nil_p:n {⟨*token list*⟩}
\quark_if_nil:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ contains only \q_nil (distinct from ⟨*token list*⟩ being empty or containing \q_nil plus one or more other tokens).

`\quark_if_no_value_p:N` ⋆
`\quark_if_no_value_p:c` ⋆
`\quark_if_no_value:NTF` ⋆
`\quark_if_no_value:cTF` ⋆

\quark_if_no_value_p:N ⟨*token*⟩
\quark_if_no_value:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is equal to \q_no_value.

`\quark_if_no_value_p:n` ⋆
`\quark_if_no_value:nTF` ⋆

\quark_if_no_value_p:n {⟨*token list*⟩}
\quark_if_no_value:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ contains only \q_no_value (distinct from ⟨*token list*⟩ being empty or containing \q_no_value plus one or more other tokens).

20.4 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 20.4.1.

`\q_recursion_tail`

This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

`\q_recursion_stop`

This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

`\quark_if_recursion_tail_stop:N` ⋆

`\quark_if_recursion_tail_stop:N` ⟨*token*⟩

Tests if ⟨*token*⟩ contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop:n` ⋆
`\quark_if_recursion_tail_stop:o` ⋆

`\quark_if_recursion_tail_stop:n` {⟨*token list*⟩}

Tests if the ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop_do:Nn` ⋆

`\quark_if_recursion_tail_stop_do:Nn` ⟨*token*⟩ {⟨*insertion*⟩}

Tests if ⟨*token*⟩ contains only the marker `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The ⟨*insertion*⟩ code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_stop_do:nn` ⋆
`\quark_if_recursion_tail_stop_do:on` ⋆

`\quark_if_recursion_tail_stop_do:nn` {⟨*token list*⟩} {⟨*insertion*⟩}

Tests if the ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The ⟨*insertion*⟩ code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_break:NN` ⋆
`\quark_if_recursion_tail_break:nN` ⋆

`\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:`.

20.4.1 An example of recursion with quarks

Quarks are mainly used internally in the expl3 code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce "`[-a-b-] [-c-d-] `". Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here's the definition of `\my_map_dbl:nn`. First of all, define the function that does the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map_dbl:nn #1#2
 {
   \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
   \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
   \q_recursion_stop
 }
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map_dbl:nn
 {
   \quark_if_recursion_tail_stop:n {#1}
   \quark_if_recursion_tail_stop:n {#2}
   \__my_map_dbl_fn:nn {#1} {#2}
```

Finally, recurse:

```
   \__my_map_dbl:nn
 }
```

Note that contrarily to LaTeX3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `\__my_map_dbl_fn:nn`.

20.5 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by TeX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see l3regex).

`\scan_new:N` `\scan_new:N` ⟨*scan mark*⟩

Creates a new ⟨*scan mark*⟩ which is set equal to `\scan_stop:`. The ⟨*scan mark*⟩ is defined globally, and an error message is raised if the name was already taken by another scan mark.

`\s_stop` 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.

Chapter 21

The l3seq module Sequences and stacks

LaTeX3 implements a "sequence" data type, which contain an ordered list of entries which may contain any ⟨*balanced text*⟩. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in LaTeX3. This is achieved using a number of dedicated stack functions.

21.1 Creating and initializing sequences

\seq_new:N
\seq_new:c

\seq_new:N ⟨*seq var*⟩

Creates a new ⟨*seq var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*seq var*⟩ initially contains no items.

\seq_clear:N
\seq_clear:c
\seq_gclear:N
\seq_gclear:c

\seq_clear:N ⟨*seq var*⟩

Clears all items from the ⟨*seq var*⟩.

\seq_clear_new:N
\seq_clear_new:c
\seq_gclear_new:N
\seq_gclear_new:c

\seq_clear_new:N ⟨*seq var*⟩

Ensures that the ⟨*seq var*⟩ exists globally by applying \seq_new:N if necessary, then applies \seq_(g)clear:N to leave the ⟨*seq var*⟩ empty.

\seq_set_eq:NN
\seq_set_eq:(cN|Nc|cc)
\seq_gset_eq:NN
\seq_gset_eq:(cN|Nc|cc)

\seq_set_eq:NN ⟨*seq* var_1⟩ ⟨*seq* var_2⟩

Sets the content of ⟨*seq* var_1⟩ equal to that of ⟨*seq* var_2⟩.

\seq_set_from_clist:NN
\seq_set_from_clist:(cN|Nc|cc)
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:(cN|Nc|cc)
\seq_gset_from_clist:Nn
\seq_gset_from_clist:cn

\seq_set_from_clist:NN ⟨*seq var*⟩ ⟨*clist var*⟩

Converts the data in the ⟨*clist var*⟩ into a ⟨*seq var*⟩: the original ⟨*clist var*⟩ is unchanged.

\seq_const_from_clist:Nn
\seq_const_from_clist:cn

\seq_const_from_clist:Nn ⟨*seq var*⟩ {⟨*comma-list*⟩}

Creates a new constant ⟨*seq var*⟩ or raises an error if the name is already taken. The ⟨*seq var*⟩ is set globally to contain the items in the ⟨*comma list*⟩.

\seq_set_split:Nnn
\seq_set_split:(NVn|NnV|NVV|Nne|Nee)
\seq_gset_split:Nnn
\seq_gset_split:(NVn|NnV|NVV|Nne|Nee)

\seq_set_split:Nnn ⟨*seq var*⟩ {⟨*delimiter*⟩} {⟨*token list*⟩}

Splits the ⟨*token list*⟩ into ⟨*items*⟩ separated by ⟨*delimiter*⟩, and assigns the result to the ⟨*seq var*⟩. Spaces on both sides of each ⟨*item*⟩ are ignored, then one set of outer braces is removed (if any); this space trimming behavior is identical to that of l3clist functions. Empty ⟨*items*⟩ are preserved by \seq_set_split:Nnn, and can be removed afterwards using \seq_remove_all:Nn ⟨*seq var*⟩ {}. The ⟨*delimiter*⟩ may not contain {, } or # (assuming TEX's normal category code régime). If the ⟨*delimiter*⟩ is empty, the ⟨*token list*⟩ is split into ⟨*items*⟩ as described for \tl_map_function:nN. See also \seq_set_split_keep_spaces:Nnn, which omits space stripping.

\seq_set_split_keep_spaces:Nnn
\seq_set_split_keep_spaces:NnV
\seq_gset_split_keep_spaces:Nnn
\seq_gset_split_keep_spaces:NnV

New: 2021-03-24

\seq_set_split_keep_spaces:Nnn ⟨*seq var*⟩ {⟨*delimiter*⟩} {⟨*token list*⟩}

Splits the ⟨*token list*⟩ into ⟨*items*⟩ separated by ⟨*delimiter*⟩, and assigns the result to the ⟨*seq var*⟩. One set of outer braces is removed (if any) but any surrounding spaces are retained: any braces *inside* one or more spaces are therefore kept. Empty ⟨*items*⟩ are preserved by \seq_set_split_keep_spaces:Nnn, and can be removed afterwards using \seq_remove_all:Nn ⟨*seq var*⟩ {}. The ⟨*delimiter*⟩ may not contain {, } or # (assuming TEX's normal category code régime). If the ⟨*delimiter*⟩ is empty, the ⟨*token list*⟩ is split into ⟨*items*⟩ as described for \tl_map_function:nN; note in this case spaces will *not* be preserved. See also \seq_set_split:Nnn, which removes spaces around the delimiters.

\seq_set_filter:NNn
\seq_gset_filter:NNn

\seq_set_filter:NNn ⟨*seq var*$_1$⟩ ⟨*seq var*$_2$⟩ {⟨*inline boolexpr*⟩}

Evaluates the ⟨*inline boolexpr*⟩ for every ⟨*item*⟩ stored within the ⟨*seq var*$_2$⟩. The ⟨*inline boolexpr*⟩ receives the ⟨*item*⟩ as #1. The sequence of all ⟨*items*⟩ for which the ⟨*inline boolexpr*⟩ evaluated to true is assigned to ⟨*seq var*$_1$⟩.

TEXhackers note: Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level TEX errors.

\seq_set_regex_extract_once:Nnn
\seq_set_regex_extract_once:cnn
\seq_set_regex_extract_once:NNn
\seq_set_regex_extract_once:cNn
\seq_gset_regex_extract_once:Nnn
\seq_gset_regex_extract_once:cnn
\seq_gset_regex_extract_once:NNn
\seq_gset_regex_extract_once:cNn

New: 2024-12-08

\seq_set_regex_extract_once:Nnn ⟨*seq var*⟩ {⟨*regex*⟩} {⟨*token list*⟩}
\seq_set_regex_extract_once:NNn ⟨*seq var*⟩ ⟨*regex var*⟩ {⟨*token list*⟩}

Finds the first match of the ⟨*regex*⟩ in the ⟨*token list*⟩. If it exists, the match is stored as the first item of the ⟨*seq var*⟩, and further items are the contents of capturing groups, in the order of their opening parenthesis. If there is no match, the ⟨*seq var*⟩ is cleared. Theses are alternative names for \regex_extract_once:nnN and friends, with arguments re-ordered for ⟨*seq var*⟩ setting; see l3regex chapter for more details of the ⟨*regex*⟩ format.

\seq_set_regex_extract_all:Nnn
\seq_set_regex_extract_all:cnn
\seq_set_regex_extract_all:NNn
\seq_set_regex_extract_all:cNn
\seq_gset_regex_extract_all:Nnn
\seq_gset_regex_extract_all:cnn
\seq_gset_regex_extract_all:NNn
\seq_gset_regex_extract_all:cNn

New: 2024-12-08

\seq_set_regex_extract_all:Nnn ⟨*seq var*⟩ {⟨*regex*⟩} {⟨*token list*⟩}
\seq_set_regex_extract_all:NNn ⟨*seq var*⟩ ⟨*regex var*⟩ {⟨*token list*⟩}

Finds all matches of the ⟨*regex*⟩ in the ⟨*token list*⟩, and stores all the submatch information in a single sequence (concatenating the results of multiple \seq_set_regex_extract_all:Nnn calls). If there is no match, the ⟨*seq var*⟩ is cleared. Theses are alternative names for \regex_extract_all:nnN and friends, with arguments re-ordered for ⟨*seq var*⟩ setting; see l3regex chapter for more details of the ⟨*regex*⟩ format.

\seq_set_regex_split:Nnn
\seq_set_regex_split:cnn
\seq_set_regex_split:NNn
\seq_set_regex_split:cNn
\seq_gset_regex_split:Nnn
\seq_gset_regex_split:cnn
\seq_gset_regex_split:NNn
\seq_gset_regex_split:cNn

New: 2024-12-08

\seq_set_regex_split:Nnn ⟨*seq var*⟩ {⟨*regex*⟩} {⟨*token list*⟩}
\seq_set_regex_split:NNn ⟨*seq var*⟩ ⟨*regex var*⟩ {⟨*token list*⟩}

Splits the ⟨*token list*⟩ into a sequence of parts, delimited by matches of the ⟨*regular expression*⟩. If the ⟨*regular expression*⟩ has capturing groups, then the token lists that they match are stored as items of the sequence as well. If no match is found the resulting ⟨*seq var*⟩ has the ⟨*token list*⟩ as its sole item. If the ⟨*regular expression*⟩ matches the empty token list, then the ⟨*token list*⟩ is split into single tokens. For example, after

```
\seq_set_regex_split:Nnn \l_path_seq { / } { the/path/for/this/file.tex }
```

the sequence \l_path_seq contains the items {the}, {path}, {for}, {this}, and {file.tex}. Theses are alternative names for \regex_split:nnN and friends, with arguments re-ordered for ⟨*seq var*⟩ setting; see l3regex chapter for more details of the ⟨*regex*⟩ format.

\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc

\seq_concat:NNN ⟨*seq var*$_1$⟩ ⟨*seq var*$_2$⟩ ⟨*seq var*$_3$⟩

Concatenates the content of ⟨*seq var*$_2$⟩ and ⟨*seq var*$_3$⟩ together and saves the result in ⟨*seq var*$_1$⟩. The items in ⟨*seq var*$_2$⟩ are placed at the left side of the new sequence.

\seq_if_exist_p:N ⋆
\seq_if_exist_p:c ⋆
\seq_if_exist:N*TF* ⋆
\seq_if_exist:c*TF* ⋆

\seq_if_exist_p:N ⟨*seq var*⟩
\seq_if_exist:NTF ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*seq var*⟩ is currently defined. This does not check that the ⟨*seq var*⟩ really is a sequence variable.

21.2 Appending data to sequences

\seq_put_left:Nn
\seq_put_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\seq_gput_left:Nn
\seq_gput_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\seq_put_left:Nn ⟨*seq var*⟩ {⟨*item*⟩}

Appends the ⟨*item*⟩ to the left of the ⟨*seq var*⟩.

\seq_put_right:Nn
\seq_put_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\seq_gput_right:Nn
\seq_gput_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\seq_put_right:Nn ⟨*seq var*⟩ {⟨*item*⟩}

Appends the ⟨*item*⟩ to the right of the ⟨*seq var*⟩.

21.3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, i.e., setting the ⟨*tl var*⟩ used with \tl_set:Nn and *never* \tl_gset:Nn.

\seq_get_left:NN
\seq_get_left:cN

\seq_get_left:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Stores the left-most item from a ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩. The ⟨*tl var*⟩ is assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_get_right:NN
\seq_get_right:cN

\seq_get_right:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Stores the right-most item from a ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩. The ⟨*tl var*⟩ is assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_pop_left:NN
\seq_pop_left:cN

\seq_pop_left:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the left-most item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩, i.e., removes the item from the sequence and stores it in the ⟨*tl var*⟩. Both of the variables are assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_gpop_left:NN
\seq_gpop_left:cN

\seq_gpop_left:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the left-most item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩, i.e., removes the item from the sequence and stores it in the ⟨*tl var*⟩. The ⟨*seq var*⟩ is modified globally, while the assignment of the ⟨*tl var*⟩ is local. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_pop_right:NN
\seq_pop_right:cN

\seq_pop_right:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the right-most item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩, i.e., removes the item from the sequence and stores it in the ⟨*tl var*⟩. Both of the variables are assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_gpop_right:NN
\seq_gpop_right:cN

\seq_gpop_right:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the right-most item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩, i.e., removes the item from the sequence and stores it in the ⟨*tl var*⟩. The ⟨*seq var*⟩ is modified globally, while the assignment of the ⟨*tl var*⟩ is local. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_item:Nn ⋆
\seq_item:(NV|Ne|cn|cV|ce) ⋆

\seq_item:Nn ⟨*seq var*⟩ {⟨*integer expression*⟩}

Indexing items in the ⟨*seq var*⟩ from 1 at the top (left), this function evaluates the ⟨*integer expression*⟩ and leaves the appropriate item from the sequence in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the sequence. If the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*seq var*⟩ (as calculated by \seq_count:N) then the function expands to nothing.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*item*⟩ does not expand further when appearing in an e-type or x-type argument expansion.

\seq_rand_item:N ⋆
\seq_rand_item:c ⋆

\seq_rand_item:N ⟨*seq var*⟩

Selects a pseudo-random item of the ⟨*seq var*⟩. If the ⟨*seq var*⟩ is empty the result is empty.

TEXhackers note: The result is returned within the **\unexpanded** primitive (**\exp_not:n**), which means that the ⟨*item*⟩ does not expand further when appearing in an e-type or x-type argument expansion.

21.4 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*

\seq_get_left:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, stores the left-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*tl var*⟩ is assigned locally.

\seq_get_right:NN*TF*
\seq_get_right:cN*TF*

\seq_get_right:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, stores the right-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*tl var*⟩ is assigned locally.

\seq_pop_left:NN*TF*
\seq_pop_left:cN*TF*

\seq_pop_left:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the left-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, i.e., removes the item from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. Both the ⟨*seq var*⟩ and the ⟨*tl var*⟩ are assigned locally.

\seq_gpop_left:NN*TF*
\seq_gpop_left:cN*TF*

\seq_gpop_left:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the left-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, i.e., removes the item from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*seq var*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally.

\seq_pop_right:NNTF
\seq_pop_right:cNTF

\seq_pop_right:NNTF ⟨seq var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨seq var⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨tl var⟩ is not defined in this case and should not be relied upon. If the ⟨seq var⟩ is non-empty, pops the right-most item from the ⟨seq var⟩ in the ⟨tl var⟩, i.e., removes the item from the ⟨seq var⟩, then leaves the ⟨true code⟩ in the input stream. Both the ⟨seq var⟩ and the ⟨tl var⟩ are assigned locally.

\seq_gpop_right:NNTF
\seq_gpop_right:cNTF

\seq_gpop_right:NNTF ⟨seq var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨seq var⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨tl var⟩ is not defined in this case and should not be relied upon. If the ⟨seq var⟩ is non-empty, pops the right-most item from the ⟨seq var⟩ in the ⟨tl var⟩, i.e., removes the item from the ⟨seq var⟩, then leaves the ⟨true code⟩ in the input stream. The ⟨seq var⟩ is modified globally, while the ⟨tl var⟩ is assigned locally.

21.5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c

\seq_remove_duplicates:N ⟨seq var⟩

Removes duplicate items from the ⟨seq var⟩, leaving the left most copy of each item in the ⟨seq var⟩. The ⟨item⟩ comparison takes place on a token basis, as for \tl_if_eq:nnTF.

TeXhackers note: This function iterates through every item in the ⟨seq var⟩ and does a comparison with the ⟨items⟩ already checked. It is therefore relatively slow with large sequences.

\seq_remove_all:Nn
\seq_remove_all:(NV|Ne|cn|cV|ce)
\seq_gremove_all:Nn
\seq_gremove_all:(NV|Ne|cn|cV|ce)

\seq_remove_all:Nn ⟨seq var⟩ {⟨item⟩}

Removes every occurrence of ⟨item⟩ from the ⟨seq var⟩. The ⟨item⟩ comparison takes place on a token basis, as for \tl_if_eq:nnTF.

\seq_set_item:Nnn
\seq_set_item:cnn
\seq_set_item:NnnTF
\seq_set_item:cnnTF
\seq_gset_item:Nnn
\seq_gset_item:cnn
\seq_gset_item:NnnTF
\seq_gset_item:cnnTF

New: 2021-04-29

\seq_set_item:Nnn ⟨seq var⟩ {⟨int expr⟩} {⟨item⟩}
\seq_set_item:NnnTF ⟨seq var⟩ {⟨int expr⟩} {⟨item⟩} {⟨true code⟩} {⟨false code⟩}

Removes the item of ⟨seq var⟩ at the position given by evaluating the ⟨int expr⟩ and replaces it by ⟨item⟩. Items are indexed from 1 on the left/top of the ⟨seq var⟩, or from −1 on the right/bottom. If the ⟨int expr⟩ is zero or is larger (in absolute value) than the number of items in the sequence, the ⟨seq var⟩ is not modified. In these cases, \seq_set_item:Nnn raises an error while \seq_set_item:NnnTF runs the ⟨false code⟩. In cases where the assignment was successful, ⟨true code⟩ is run afterwards.

\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c

\seq_reverse:N ⟨*seq var*⟩

Reverses the order of the items stored in the ⟨*seq var*⟩.

\seq_sort:Nn
\seq_sort:cn
\seq_gsort:Nn
\seq_gsort:cn

\seq_sort:Nn ⟨*seq var*⟩ {⟨*comparison code*⟩}

Sorts the items in the ⟨*seq var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*seq var*⟩. The details of sorting comparison are described in Section 6.1.

\seq_shuffle:N
\seq_shuffle:c
\seq_gshuffle:N
\seq_gshuffle:c

\seq_shuffle:N ⟨*seq var*⟩

Sets the ⟨*seq var*⟩ to the result of placing the items of the ⟨*seq var*⟩ in a random order. Each item is (roughly) as likely to end up in any given position.

TeXhackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed \sys_rand_seed: only has 28-bits. The use of \toks internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

21.6 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 ⟨*seq var*⟩
\seq_if_empty:NTF ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*seq var*⟩ is empty (containing no items).

\seq_if_in:NnTF
\seq_if_in:(NV|Nv|Ne|No|cn|cV|cv|ce|co)TF

\seq_if_in:NnTF ⟨*seq var*⟩ {⟨*item*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*item*⟩ is present in the ⟨*seq var*⟩.

21.7 Mapping over sequences

All mappings are done at the current group level, i.e., any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

\seq_map_function:NN ☆
\seq_map_function:cN ☆

\seq_map_function:NN ⟨*seq var*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ stored in the ⟨*seq var*⟩. The ⟨*function*⟩ will receive one argument for each iteration. The ⟨*items*⟩ are returned from left to right. To pass further arguments to the ⟨*function*⟩, see \seq_map_tokens:Nn. The function \seq_map_inline:Nn is faster than \seq_map_function:NN for sequences with more than about 10 items.

\seq_map_inline:Nn
\seq_map_inline:cn

\seq_map_inline:Nn ⟨*seq var*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*seq var*⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as #1. The ⟨*items*⟩ are returned from left to right.

`\seq_map_tokens:Nn` ☆
`\seq_map_tokens:cn` ☆

`\seq_map_tokens:Nn` ⟨*seq var*⟩ {⟨*code*⟩}

Analogue of `\seq_map_function:NN` which maps several tokens instead of a single function. The ⟨*code*⟩ receives each item in the ⟨*seq var*⟩ as a trailing brace group. For instance,

```
\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }
```

expands to twice each item in the ⟨*seq var*⟩: for each item in `\l_my_seq` the function `\prg_replicate:nn` receives `2` and ⟨*item*⟩ as its two arguments. The function `\seq_map_inline:Nn` is typically faster but it is not expandable.

`\seq_map_variable:NNn`
`\seq_map_variable:(Ncn|cNn|ccn)`

`\seq_map_variable:NNn` ⟨*seq var*⟩ ⟨*variable*⟩ {⟨*code*⟩}

Stores each ⟨*item*⟩ of the ⟨*seq var*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*seq var*⟩, or its original value if the ⟨*seq var*⟩ is empty. The ⟨*items*⟩ are returned from left to right.

`\seq_map_indexed_function:NN` ☆

`\seq_map_indexed_function:NN` ⟨*seq var*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every entry in the ⟨*seq var*⟩. The ⟨*function*⟩ should have signature `:nn`. It receives two arguments for each iteration: the ⟨*index*⟩ (namely `1` for the first entry, then `2` and so on) and the ⟨*item*⟩.

`\seq_map_indexed_inline:Nn`

`\seq_map_indexed_inline:Nn` ⟨*seq var*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every entry in the ⟨*seq var*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*index*⟩ (namely `1` for the first entry, then `2` and so on) as `#1` and the ⟨*item*⟩ as `#2`.

`\seq_map_pairwise_function:NNN` ☆
`\seq_map_pairwise_function:(NcN|cNN|ccN)` ☆

New: 2023-05-10

`\seq_map_pairwise_function:NNN` ⟨*seq* var_1⟩ ⟨*seq* var_2⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every pair of items ⟨*seq* var_1*-item*⟩–⟨*seq* var_2*-item*⟩ from the two sequences, returning items from both sequences from left to right. The ⟨*function*⟩ receives two `n`-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (i.e., whichever sequence has fewer items determines how many iterations occur).

`\seq_map_break:` ☆ `\seq_map_break:`

Used to terminate a `\seq_map_...` function before all entries in the ⟨*seq var*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \seq_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a `\seq_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\seq_map_break:n` ☆ `\seq_map_break:n {⟨code⟩}`

Used to terminate a `\seq_map_...` function before all entries in the ⟨*seq var*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \seq_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\seq_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`
Updated: 2020-07-16

`\seq_set_map:NNn ⟨seq var₁⟩ ⟨seq var₂⟩ {⟨inline function⟩}`

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*seq var*$_2$⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as `#1`. The sequence resulting from applying ⟨*inline function*⟩ to each ⟨*item*⟩ is assigned to ⟨*seq var*$_1$⟩.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

\seq_set_map_e:NNn
\seq_gset_map_e:NNn

New: 2020-07-16
Updated: 2023-10-26

\seq_set_map_e:NNn ⟨*seq var*$_1$⟩ ⟨*seq var*$_2$⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*seq var*$_2$⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as #1. The sequence resulting from e-expanding ⟨*inline function*⟩ applied to each ⟨*item*⟩ is assigned to ⟨*seq var*$_1$⟩. As such, the code in ⟨*inline function*⟩ should be expandable.

TEXhackers note: Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level TEX errors.

\seq_count:N ⋆
\seq_count:c ⋆

\seq_count:N ⟨*seq var*⟩

Leaves the number of items in the ⟨*seq var*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*seq var*⟩ includes those which are empty and duplicates, i.e., every item in a ⟨*seq var*⟩ is unique.

21.8 Using the content of sequences directly

\seq_use:Nnnn ⋆
\seq_use:cnnn ⋆

\seq_use:Nnnn ⟨*seq var*⟩ {⟨*separator between two*⟩}
{⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Places the contents of the ⟨*seq var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the sequence has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts "a, b, c, de, and f" in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*items*⟩ do not expand further when appearing in an e-type or x-type argument expansion.

\seq_use:Nn ⋆
\seq_use:cn ⋆

\seq_use:Nn ⟨*seq var*⟩ {⟨*separator*⟩}

Places the contents of the ⟨*seq var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the sequence has a single item, it is placed in the input stream with no ⟨*separator*⟩, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts "a and b and c and de and f" in the input stream.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*items*⟩ do not expand further when appearing in an e-type or x-type argument expansion.

\seq_format:Nn ⋆
\seq_format:cn ⋆
New: 2025-06-09

\seq_format:Nn ⟨*seq var*⟩ {⟨*format specification*⟩}

Converts each item in the ⟨*seq var*⟩ as a token list to a string according to the ⟨*format specification*⟩, and concatenates the results. The details of the ⟨*format specification*⟩ are described in Section 19.1.

21.9 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

\seq_get:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Reads the top item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩. The ⟨*tl var*⟩ is assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_pop:NN
\seq_pop:cN

\seq_pop:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the top item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩. Both of the variables are assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_gpop:NN
\seq_gpop:cN

\seq_gpop:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the top item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩. The ⟨*seq var*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_get:NN*TF*
\seq_get:cN*TF*

\seq_get:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, stores the top item from a ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩. The ⟨*tl var*⟩ is assigned locally.

\seq_pop:NNTF
\seq_pop:cNTF

\seq_pop:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the top item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, i.e., removes the item from the ⟨*seq var*⟩. Both the ⟨*seq var*⟩ and the ⟨*tl var*⟩ are assigned locally.

\seq_gpop:NNTF
\seq_gpop:cNTF

\seq_gpop:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the top item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, i.e., removes the item from the ⟨*seq var*⟩. The ⟨*seq var*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally.

\seq_push:Nn
\seq_push:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\seq_gpush:Nn
\seq_gpush:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\seq_push:Nn ⟨*seq var*⟩ {⟨*item*⟩}

Adds the {⟨*item*⟩} to the top of the ⟨*seq var*⟩.

21.10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a ⟨*seq var*⟩ only has distinct items, use \seq_remove_duplicates:N ⟨*seq var*⟩. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set ⟨*seq var*⟩ are straightforward. For instance, \seq_count:N ⟨*seq var*⟩ expands to the number of items, while \seq_if_in:NnTF ⟨*seq var*⟩ {⟨*item*⟩} tests if the ⟨*item*⟩ is in the set.

Adding an ⟨*item*⟩ to a set ⟨*seq var*⟩ can be done by appending it to the ⟨*seq var*⟩ if it is not already in the ⟨*seq var*⟩:

```
\seq_if_in:NnF ⟨seq var⟩ {⟨item⟩}
  { \seq_put_right:Nn ⟨seq var⟩ {⟨item⟩} }
```

Removing an ⟨*item*⟩ from a set ⟨*seq var*⟩ can be done using \seq_remove_all:Nn,

```
\seq_remove_all:Nn ⟨seq var⟩ {⟨item⟩}
```

The intersection of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by collecting items of ⟨*seq var*$_1$⟩ which are in ⟨*seq var*$_2$⟩.

```
\seq_clear:N ⟨seq var3⟩
\seq_map_inline:Nn ⟨seq var1⟩
  {
    \seq_if_in:NnT ⟨seq var2⟩ {#1}
      { \seq_put_right:Nn ⟨seq var3⟩ {#1} }
  }
```

The code as written here only works if ⟨*seq var*$_3$⟩ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence `\l__`⟨*pkg*⟩`_tmp_seq`, then ⟨*seq var*$_3$⟩ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ through

```
\seq_concat:NNN ⟨seq var3⟩ ⟨seq var1⟩ ⟨seq var2⟩
\seq_remove_duplicates:N ⟨seq var3⟩
```

or by adding items to (a copy of) ⟨*seq var*$_1$⟩ one by one

```
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
  {
    \seq_if_in:NnF ⟨seq var3⟩ {#1}
      { \seq_put_right:Nn ⟨seq var3⟩ {#1} }
  }
```

The second approach is faster than the first when the ⟨*seq var*$_2$⟩ is short compared to ⟨*seq var*$_1$⟩.

The difference of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by removing items of the ⟨*seq var*$_2$⟩ from (a copy of) the ⟨*seq var*$_1$⟩ one by one.

```
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
  { \seq_remove_all:Nn ⟨seq var3⟩ {#1} }
```

The symmetric difference of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by computing the difference between ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ and storing the result as `\l__`⟨*pkg*⟩`_tmp_seq`, then the difference between ⟨*seq var*$_2$⟩ and ⟨*seq var*$_1$⟩, and finally concatenating the two differences to get the symmetric differences.

```
\seq_set_eq:NN \l__⟨pkg⟩_tmp_seq ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
  { \seq_remove_all:Nn \l__⟨pkg⟩_tmp_seq {#1} }
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var2⟩
\seq_map_inline:Nn ⟨seq var1⟩
  { \seq_remove_all:Nn ⟨seq var3⟩ {#1} }
\seq_concat:NNN ⟨seq var3⟩ ⟨seq var3⟩ \l__⟨pkg⟩_tmp_seq
```

21.11 Constant and scratch sequences

`\c_empty_seq` Constant that is always empty.

\l_tmpa_seq
\l_tmpb_seq

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

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.

21.12 Viewing sequences

\seq_show:N
\seq_show:c

Updated: 2021-04-29

`\seq_show:N ⟨seq var⟩`

Displays the entries in the ⟨*seq var*⟩ in the terminal.

\seq_log:N
\seq_log:c

Updated: 2021-04-29

`\seq_log:N ⟨seq var⟩`

Writes the entries in the ⟨*seq var*⟩ in the log file.

Chapter 22

The l3int module Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* ("⟨*int expr*⟩").

22.1 Integer expressions

Throughout this module, (almost) all `n`-type argument allow for an ⟨*intexpr*⟩ argument with the following syntax. The ⟨*integer expression*⟩ should consist, after expansion, of `+`, `-`, `*`, `/`, `(`, `)` and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- `/` denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31}-1$, except in the case of scaling operations a*b/c, for which a*b may be arbitrarily large (but the operands a, b, c are still constrained to an absolute value at most $2^{31}-1$);
- parentheses may not appear after unary `+` or `-`, namely placing `+(` or `-(` at the start of an expression or after `+`, `-`, `*`, `/` or `(` leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_show:n { 5 +  4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N  \l_my_tl
\tl_set:Nn \l_my_tl { 5 }
\int_new:N  \l_my_int
\int_set:Nn \l_my_int { 4 }
\int_show:n { \l_my_tl +  \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

show the same result −6 because `\l_my_tl` expands to the integer denotation `5` while the integer variable `\l_my_int` takes the value 4. As the ⟨*integer expression*⟩ is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and `\exp_not:n` and its variants have no effect while `\exp_-not:N` may incorrectly interrupt the expression.

`\int_eval:n` ⋆ `\int_eval:n {`⟨*int expr*⟩`}`

Evaluates the ⟨*int expr*⟩ and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with `0`, for negative results `-` followed by such a sequence, and `0` for zero.

TEXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an ⟨*internal integer*⟩, and therefore requires suitable termination if used in a TEX-style integer assignment.

As all TEX integers, integer operands can also be dimension or skip variables, converted to integers in `sp`, or octal numbers given as `'` followed by digits other than `8` and `9`, or hexadecimal numbers given as `"` followed by digits or upper case letters from `A` to `F`, or the character code of some character or one-character control sequence, given as `` ` ``⟨*char*⟩.

`\int_eval:w` ⋆ `\int_eval:w` ⟨*int expr*⟩

Evaluates the ⟨*int expr*⟩ as described for `\int_eval:n`. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is `\scan_stop:` it is removed, otherwise not. Spaces do *not* terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, `\int_eval:w 1␣+␣1␣9` (with explicit space tokens inserted using `~` in a code setting) expands to `29` since the digit `9` is not part of the expression. Expansion details, etc., are as given for `\int_eval:n`.

`\int_sign:n` ⋆ `\int_sign:n {`⟨*int expr*⟩`}`

Evaluates the ⟨*int expr*⟩ then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

`\int_abs:n` ⋆ `\int_abs:n {`⟨*int expr*⟩`}`

Evaluates the ⟨*int expr*⟩ as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_div_round:nn` ⋆ `\int_div_round:nn {`⟨*int* $expr_1$⟩`} {`⟨*int* $expr_2$⟩`}`

Evaluates the two ⟨*int expr*⟩s as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using `/` directly in an ⟨*int expr*⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_div_truncate:nn` ⋆ `\int_div_truncate:nn {`⟨*int* $expr_1$⟩`} {`⟨*int* $expr_2$⟩`}`

Evaluates the two ⟨*int expr*⟩s as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using `/` rounds to the closest integer instead. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

```
\int_max:nn ★
\int_min:nn ★
```

\int_max:nn {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}
\int_min:nn {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}

Evaluates the ⟨*int expr*⟩s as described for \int_eval:n and leaves either the larger or smaller value in the input stream as an ⟨*integer denotation*⟩ after two expansions.

```
\int_mod:nn ★
```

\int_mod:nn {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}

Evaluates the two ⟨*int expr*⟩s as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting \int_div_-truncate:nn {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩} times ⟨*int expr*$_2$⟩ from ⟨*int expr*$_1$⟩. Thus, the result has the same sign as ⟨*int expr*$_1$⟩ and its absolute value is strictly less than that of ⟨*int expr*$_2$⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

22.2 Creating and initializing integers

```
\int_new:N
\int_new:c
```

\int_new:N ⟨*integer*⟩

Creates a new ⟨*integer*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*integer*⟩ is initially equal to 0.

```
\int_const:Nn
\int_const:cn
```

\int_const:Nn ⟨*integer*⟩ {⟨*int expr*⟩}

Creates a new constant ⟨*integer*⟩ or raises an error if the name is already taken. The value of the ⟨*integer*⟩ is set globally to the ⟨*int expr*⟩.

```
\int_zero:N
\int_zero:c
\int_gzero:N
\int_gzero:c
```

\int_zero:N ⟨*integer*⟩

Sets ⟨*integer*⟩ to 0.

```
\int_zero_new:N
\int_zero_new:c
\int_gzero_new:N
\int_gzero_new:c
```

\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 ★
```

\int_if_exist_p:N ⟨*integer*⟩
\int_if_exist:NTF ⟨*integer*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*integer*⟩ is currently defined. This does not check that the ⟨*integer*⟩ really is an integer variable.

22.3 Setting and incrementing integers

\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn

\int_add:Nn ⟨*integer*⟩ {⟨*int expr*⟩}

Adds the result of the ⟨*int expr*⟩ to the current content of the ⟨*integer*⟩.

\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c

\int_decr:N ⟨*integer*⟩

Decreases the value stored in ⟨*integer*⟩ by 1.

\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c

\int_incr:N ⟨*integer*⟩

Increases the value stored in ⟨*integer*⟩ by 1.

\int_set:Nn
\int_set:(cn|NV|cV)
\int_gset:Nn
\int_gset:(cn|NV|cV)

\int_set:Nn ⟨*integer*⟩ {⟨*int expr*⟩}

Sets ⟨*integer*⟩ to the value of ⟨*int expr*⟩, which must evaluate to an integer (as described for \int_eval:n).

\int_set_regex_count:Nnn
\int_set_regex_count:cnn
\int_set_regex_count:NNn
\int_set_regex_count:cNn
\int_gset_regex_count:Nnn
\int_gset_regex_count:cnn
\int_gset_regex_count:NNn
\int_gset_regex_count:cNn

New: 2024-12-08

\int_set_regex_count:Nnn ⟨*integer*⟩ {⟨*regex*⟩} {⟨*token list*⟩}
\int_set_regex_count:NNn ⟨*integer*⟩ ⟨*regex var*⟩ {⟨*token list*⟩}

Sets ⟨*integer*⟩ equal to the number of times ⟨*regular expression*⟩ appears in ⟨*token list*⟩. The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_set_regex_count:Nnn \l_foo_int { (b+|c) } { abbababcbb }
```

results in \l_foo_int taking the value 5. Theses are alternative names for \regex_-count:nnN and friends, with arguments re-ordered for ⟨*integer*⟩ setting; see l3regex chapter for more details of the ⟨*regex*⟩ format.

\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

\int_sub:Nn ⟨*integer*⟩ {⟨*int expr*⟩}

Subtracts the result of the ⟨*int expr*⟩ from the current content of the ⟨*integer*⟩.

22.4 Using integers

\int_use:N ⋆
\int_use:c ⋆

\int_use:N ⟨*integer*⟩

Recovers the content of an ⟨*integer*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an ⟨*integer*⟩ is required (such as in the first and third arguments of \int_compare:nNnTF).

TEXhackers note: \int_use:N is the TEX primitive \the: this is one of several LATEX3 names for this primitive.

22.5 Integer expression conditionals

\int_compare_p:nNn ⋆
\int_compare:nNn*TF* ⋆

\int_compare_p:nNn {⟨*int expr*$_1$⟩} ⟨*relation*⟩ {⟨*int expr*$_2$⟩}
\int_compare:nNnTF
  {⟨*int expr*$_1$⟩} ⟨*relation*⟩ {⟨*int expr*$_2$⟩}
  {⟨*true code*⟩} {⟨*false code*⟩}

This function first evaluates each of the ⟨*int expr*⟩s as described for \int_eval:n. The two results are then compared using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

This function is less flexible than \int_compare:nTF but around 5 times faster.

`\int_compare_p:n` ⋆
`\int_compare:nTF` ⋆

```
\int_compare_p:n
  {
    ⟨int expr1⟩ ⟨relation1⟩
    ...
    ⟨int exprN⟩ ⟨relationN⟩
    ⟨int exprN+1⟩
  }
\int_compare:nTF
  {
    ⟨int expr1⟩ ⟨relation1⟩
    ...
    ⟨int exprN⟩ ⟨relationN⟩
    ⟨int exprN+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*int expr*⟩s as described for `\int_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*int expr*$_1$⟩ and ⟨*int expr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*int expr*$_2$⟩ and ⟨*int expr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*int expr*$_N$⟩ and ⟨*int expr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*int expr*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*integer expression*⟩ is evaluated and no other comparison is performed. The ⟨*relations*⟩ can be any of the following:

Equal	`=` or `==`
Greater than or equal to	`>=`
Greater than	`>`
Less than or equal to	`<=`
Less than	`<`
Not equal	`!=`

This function is more flexible than `\int_compare:nNnTF` but around 5 times slower.

`\int_case:nn` ⋆
`\int_case:nnTF` ⋆

```
\int_case:nnTF {⟨test int expr⟩}
  {
    {⟨int expr case1⟩} {⟨code case1⟩}
    {⟨int expr case2⟩} {⟨code case2⟩}
    ...
    {⟨int expr casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test int expr*⟩ and compares this in turn to each of the ⟨*int expr case*⟩s until a match is found. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

```
\int_case:nnF
  { 2 * 5 }
  {
    { 5 }       { Small }
    { 4 + 6 }   { Medium }
    { -2 * 10 } { Negative }
  }
  { No idea! }
```

leaves "`Medium`" in the input stream. Since evaluation of the test expressions stops at the first successful case, the order of possible matches should normally be that the most likely are earlier: this will reduce the average steps required to complete expansion.

`\int_if_even_p:n` ⋆
`\int_if_even:nTF` ⋆
`\int_if_odd_p:n` ⋆
`\int_if_odd:nTF` ⋆

```
\int_if_odd_p:n {⟨int expr⟩}
\int_if_odd:nTF {⟨int expr⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates the ⟨*int expr*⟩ as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

`\int_if_zero_p:n` ⋆
`\int_if_zero:nTF` ⋆

New: 2023-05-17

```
\int_if_zero_p:n {⟨int expr⟩}
\int_if_zero:nTF {⟨int expr⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates the ⟨*int expr*⟩ as described for `\int_eval:n`. It then evaluates if this is zero or not.

22.6 Integer expression loops

`\int_do_until:nNnn` ☆

```
\int_do_until:nNnn {⟨int expr1⟩} ⟨relation⟩ {⟨int expr2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*int expr*⟩s as described for `\int_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\int_do_while:nNnn` ☆ `\int_do_while:nNnn {`⟨*int expr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*int expr*$_2$⟩`}` `{`⟨*code*⟩`}`

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*int expr*⟩s as described for `\int_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\int_until_do:nNnn` ☆ `\int_until_do:nNnn {`⟨*int expr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*int expr*$_2$⟩`}` `{`⟨*code*⟩`}`

Evaluates the relationship between the two ⟨*int expr*⟩s as described for `\int_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `true`.

`\int_while_do:nNnn` ☆ `\int_while_do:nNnn {`⟨*int expr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*int expr*$_2$⟩`}` `{`⟨*code*⟩`}`

Evaluates the relationship between the two ⟨*int expr*⟩s as described for `\int_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `false`.

`\int_do_until:nn` ☆ `\int_do_until:nn {`⟨*integer relation*⟩`}` `{`⟨*code*⟩`}`

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\int_do_while:nn` ☆ `\int_do_while:nn {`⟨*integer relation*⟩`}` `{`⟨*code*⟩`}`

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\int_until_do:nn` ☆ `\int_until_do:nn {`⟨*integer relation*⟩`}` `{`⟨*code*⟩`}`

Evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `true`.

`\int_while_do:nn` ☆ `\int_while_do:nn {`⟨*integer relation*⟩`}` `{`⟨*code*⟩`}`

Evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `false`.

22.7 Integer step functions

\int_step_function:nN ☆
\int_step_function:nnN ☆
\int_step_function:nnnN ☆

\int_step_function:nN {⟨*final value*⟩} ⟨*function*⟩
\int_step_function:nnN {⟨*initial value*⟩} {⟨*final value*⟩} ⟨*function*⟩
\int_step_function:nnnN {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*function*⟩

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. The ⟨*function*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]

The functions \int_step_function:nN and \int_step_function:nnN both use a fixed ⟨*step*⟩ of 1, and in the case of \int_step_function:nN the ⟨*initial value*⟩ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\int_step_tokens:nn ☆
\int_step_tokens:nnn ☆
\int_step_tokens:nnnn ☆

New: 2025-01-13

\int_step_tokens:nn {⟨*final value*⟩} {⟨*code*⟩}
\int_step_tokens:nnn {⟨*initial value*⟩} {⟨*final value*⟩} {⟨*code*⟩}
\int_step_tokens:nnnn {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*code*⟩}

This function works just like \int_step_function:nnnN but instead of mapping a single function to each stepped ⟨*value*⟩ between ⟨*initial value*⟩ and ⟨*final value*⟩ this maps the multiple tokens in ⟨*code*⟩, so that it gets the current ⟨*value*⟩ as a braced argument following it. For instance

```
\cs_set:Npn \my_product:nn #1#2
  { $#1 \times #2 = \int_eval:n { #1 * #2 }$ \quad }
\int_step_tokens:nnnn { 1 } { 1 } { 4 } { \my_product:nn { 2 } }
```

would print

$2 \times 1 = 2$ $2 \times 2 = 4$ $2 \times 3 = 6$ $2 \times 4 = 8$

\int_step_inline:nn
\int_step_inline:nnn
\int_step_inline:nnnn

\int_step_inline:nn {⟨*final value*⟩} {⟨*code*⟩}
\int_step_inline:nnn {⟨*initial value*⟩} {⟨*final value*⟩} {⟨*code*⟩}
\int_step_inline:nnnn {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with #1 replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (#1).

The functions \int_step_inline:nn and \int_step_inline:nnn both use a fixed ⟨*step*⟩ of 1, and in the case of \int_step_inline:nn the ⟨*initial value*⟩ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\int_step_variable:nNn
\int_step_variable:nnNn
\int_step_variable:nnnNn

\int_step_variable:nNn {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}
\int_step_variable:nnNn {⟨*initial value*⟩} {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}
\int_step_variable:nnnNn {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

The functions \int_step_variable:nNn and \int_step_variable:nnNn both use a fixed ⟨*step*⟩ of 1, and in the case of \int_step_variable:nNn the ⟨*initial value*⟩ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

22.8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

\int_to_arabic:n ⋆
\int_to_arabic:v ⋆

\int_to_arabic:n {⟨*int expr*⟩}

Places the value of the ⟨*int expr*⟩ in the input stream as digits, with category code 12 (other).

\int_to_alph:n ⋆
\int_to_Alph:n ⋆

\int_to_alph:n {⟨*int expr*⟩}

Evaluates the ⟨*int expr*⟩ and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alph:n { 1 }
```

places a in the input stream,

```
\int_to_alph:n { 26 }
```

is represented as z and

```
\int_to_alph:n { 27 }
```

is converted to aa. For conversions using other alphabets, use \int_to_symbols:nnn to define an alphabet-specific function. The basic \int_to_alph:n and \int_to_Alph:n functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_symbols:nnn` ⋆

`\int_to_symbols:nnn`
  `{⟨int expr⟩} {⟨total symbols⟩}`
  `{⟨value to symbol mapping⟩}`

This is the low-level function for conversion of an ⟨`int expr`⟩ into a symbolic form (often letters). The ⟨`total symbols`⟩ available should be given as an integer expression. Values are actually converted to symbols according to the ⟨`value to symbol mapping`⟩. This should be given as ⟨`total symbols`⟩ pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alph:n` function is defined as

```
\cs_new:Npn \int_to_alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        ...
        { 26 } { z }
      }
  }
```

`\int_to_bin:n` ⋆

`\int_to_bin:n {⟨int expr⟩}`

Calculates the value of the ⟨`int expr`⟩ and places the binary representation of the result in the input stream.

`\int_to_hex:n` ⋆
`\int_to_Hex:n` ⋆

`\int_to_hex:n {⟨int expr⟩}`

Calculates the value of the ⟨`int expr`⟩ and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for `\int_to_hex:n` and upper case ones for `\int_to_Hex:n`. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_oct:n` ⋆

`\int_to_oct:n {⟨int expr⟩}`

Calculates the value of the ⟨`int expr`⟩ and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_base:nn` ⋆
`\int_to_Base:nn` ⋆

`\int_to_base:nn {⟨int expr⟩} {⟨base⟩}`

Calculates the value of the ⟨`int expr`⟩ and converts it into the appropriate representation in the ⟨`base`⟩; the later may be given as an integer expression. For bases greater than 10 the higher "digits" are represented by letters from the English alphabet: lower case letters for `\int_to_base:n` and upper case ones for `\int_to_Base:n`. The maximum ⟨`base`⟩ value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

TeXhackers note: This is a generic version of `\int_to_bin:n`, etc.

`\int_to_roman:n` ☆
`\int_to_Roman:n` ☆

`\int_to_roman:n {⟨int expr⟩}`

Places the value of the ⟨*int expr*⟩ in the input stream as Roman numerals, either lower case (`\int_to_roman:n`) or upper case (`\int_to_Roman:n`). If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are `mdclxvi`, repeated as needed: the notation with bars (such as $\bar{\text{v}}$ for 5000) is *not* used. For instance `\int_to_roman:n { 8249 }` expands to `mmmmmmmmccxlix`.

`\int_format:nn` ⋆

New: 2025-06-09

`\int_format:nn {⟨int expr⟩} {⟨format specification⟩}`

Evaluates the ⟨*int expr*⟩ and converts the result to a string according to the ⟨*format specification*⟩. The ⟨*precision*⟩ argument is not allowed. The ⟨*style*⟩ can be `b` for binary output, `d` for decimal output (this is the default), `o` for octal output, `X` for hexadecimal output (using capital letters). The details of the ⟨*format specification*⟩ are described in Section 19.1.

22.9 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 first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with "a" equal to 1 through to "z" equal to 26. The function also accepts a leading sign, made of `+` and `-`. This is the inverse function of `\int_to_alph:n` and `\int_to_Alph:n`.

`\int_from_bin:n` ⋆

`\int_from_bin:n {⟨binary number⟩}`

Converts the ⟨*binary number*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*binary number*⟩ is first converted to a string, with no expansion. The function accepts a leading sign, made of `+` and `-`, followed by binary digits. This is the inverse function of `\int_to_bin:n`.

`\int_from_hex:n` ⋆

`\int_from_hex:n {⟨hexadecimal number⟩}`

Converts the ⟨*hexadecimal number*⟩ into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the ⟨*hexadecimal number*⟩ by upper or lower case letters. The ⟨*hexadecimal number*⟩ is first converted to a string, with no expansion. The function also accepts a leading sign, made of `+` and `-`. This is the inverse function of `\int_to_hex:n` and `\int_to_Hex:n`.

`\int_from_oct:n` ⋆

`\int_from_oct:n {⟨octal number⟩}`

Converts the ⟨*octal number*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*octal number*⟩ is first converted to a string, with no expansion. The function accepts a leading sign, made of `+` and `-`, followed by octal digits. This is the inverse function of `\int_to_oct:n`.

`\int_from_roman:n` ⋆ `\int_from_roman:n {⟨roman numeral⟩}`

Converts the ⟨*roman numeral*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*roman numeral*⟩ is first converted to a string, with no expansion. The ⟨*roman numeral*⟩ may be in upper or lower case; if the numeral contains characters besides `mdclxvi` or `MDCLXVI` then the resulting value is -1. This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

`\int_from_base:nn` ⋆ `\int_from_base:nn {⟨number⟩} {⟨base⟩}`

Converts the ⟨*number*⟩ expressed in ⟨*base*⟩ into the appropriate value in base 10. The ⟨*number*⟩ is first converted to a string, with no expansion. The ⟨*number*⟩ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum ⟨*base*⟩ value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

22.10 Random integers

`\int_rand:nn` ⋆ `\int_rand:nn {⟨int expr₁⟩} {⟨int expr₂⟩}`

Evaluates the two ⟨*int expr*⟩s and produces a pseudo-random number between the two (with bounds included).

`\int_rand:n` ⋆ `\int_rand:n {⟨int expr⟩}`

Evaluates the ⟨*int expr*⟩ then produces a pseudo-random number between 1 and the ⟨*int expr*⟩ (included).

22.11 Viewing integers

`\int_show:N`
`\int_show:c`
`\int_show:N ⟨integer⟩`

Displays the value of the ⟨*integer*⟩ on the terminal.

`\int_show:n` `\int_show:n {⟨int expr⟩}`

Displays the result of evaluating the ⟨*int expr*⟩ on the terminal.

`\int_log:N`
`\int_log:c`
`\int_log:N ⟨integer⟩`

Writes the value of the ⟨*integer*⟩ in the log file.

`\int_log:n` `\int_log:n {⟨int expr⟩}`

Writes the result of evaluating the ⟨*int expr*⟩ in the log file.

22.12 Constant integers

\c_zero_int
\c_one_int

Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.

\c_max_int

The maximum value that can be stored as an integer.

\c_max_register_int

Maximum number of registers.

\c_max_char_int

Maximum character code completely supported by the engine.

22.13 Scratch integers

\l_tmpa_int
\l_tmpb_int

Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_int
\g_tmpb_int

Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

22.14 Direct number expansion

`\int_value:w` ⋆

```
\int_value:w ⟨integer⟩
\int_value:w ⟨integer denotation⟩ ⟨optional space⟩
```

Expands the following tokens until an ⟨*integer*⟩ is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit `0` except for zero) in the input stream as category code 12 (other) characters. The ⟨*integer*⟩ can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any TEX register except `\toks`) or
- explicit digits (or by `'`⟨*octal digits*⟩ or `"`⟨*hexadecimal digits*⟩ or `` ` ``⟨*character*⟩).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in `f`-expansion, and so `\exp_stop_f:` may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required "directly". In general, `\int_eval:n` is the preferred approach to generating numbers.

TEXhackers note: This is the TEX primitive `\number`.

22.15 Primitive conditionals

`\if_int_compare:w` ⋆

```
\if_int_compare:w ⟨integer1⟩ ⟨relation⟩ ⟨integer2⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Compare two integers using ⟨*relation*⟩, which must be one of `=`, `<` or `>` with category code 12. The `\else:` branch is optional.

TEXhackers note: This is the TEX primitive `\ifnum`.

`\if_case:w` ⋆
`\or:` ⋆

```
\if_case:w ⟨integer⟩ ⟨case0⟩
  \or: ⟨case1⟩
  \or: ...
  \else: ⟨default⟩
\fi:
```

Selects a case to execute based on the value of the ⟨*integer*⟩. The first case (⟨$case_0$⟩) is executed if ⟨*integer*⟩ is 0, the second (⟨$case_1$⟩) if the ⟨*integer*⟩ is 1, etc. The ⟨*integer*⟩ may be a literal, a constant or an integer expression (*e.g.* using `\int_eval:n`).

TEXhackers note: These are the TEX primitives `\ifcase` and `\or`.

`\if_int_odd:w` ⋆

```
\if_int_odd:w ⟨tokens⟩ ⟨optional space⟩
  ⟨true code⟩
\else:
  ⟨true code⟩
\fi:
```

Expands ⟨*tokens*⟩ until a non-numeric token or a space is found, and tests whether the resulting ⟨*integer*⟩ is odd. If so, ⟨*true code*⟩ is executed. The `\else:` branch is optional.

> **TEXhackers note:** This is the TEX primitive `\ifodd`.

Chapter 23

The l3flag module Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any (small) non-negative value, which we call its ⟨*height*⟩. In expansion-only contexts, a flag can only be "raised": this increases the ⟨*height*⟩ by 1. The ⟨*height*⟩ can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by l3str-convert, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height and that the memory cannot be reclaimed even if the flag is cleared. Flags should not be used unless it is unavoidable.

In earlier versions, flags were referenced by an `n`-type ⟨*flag name*⟩ such as `fp_overflow`, used as part of `\use:c` constructions. All of the commands described below have `n`-type analogues that can still appear in old code, but the `N`-type commands are to be preferred moving forward. The `n`-type ⟨*flag name*⟩ is simply mapped to `\l_`⟨*flag name*⟩`_flag`, which makes it easier for packages using public flags (such as l3fp) to retain backwards compatibility.

23.1 Setting up flags

`\flag_new:N`
`\flag_new:c`
New: 2024-01-12

`\flag_new:N` ⟨*flag var*⟩

Creates a new ⟨*flag var*⟩, or raises an error if the name is already taken. The declaration is global, but flags are always local variables. The ⟨*flag var*⟩ initially has zero height.

\flag_clear:N
\flag_clear:c

New: 2024-01-12

\flag_clear:N ⟨*flag var*⟩

Sets the height of the ⟨*flag var*⟩ to zero. The assignment is local.

\flag_clear_new:N
\flag_clear_new:c

New: 2024-01-12

\flag_clear_new:N ⟨*flag var*⟩

Ensures that the ⟨*flag var*⟩ exists globally by applying \flag_new:N if necessary, then applies \flag_clear:N, setting the height to zero locally.

\flag_show:N
\flag_show:c

New: 2024-01-12

\flag_show:N ⟨*flag var*⟩

Displays the height of the ⟨*flag var*⟩ in the terminal.

\flag_log:N
\flag_log:c

New: 2024-01-12

\flag_log:N ⟨*flag var*⟩

Writes the height of the ⟨*flag var*⟩ in the log file.

23.2 Expandable flag commands

\flag_if_exist_p:N ⋆
\flag_if_exist_p:c ⋆
\flag_if_exist:N*TF* ⋆
\flag_if_exist:c*TF* ⋆

New: 2024-01-12

\flag_if_exist_p:N ⟨*flag var*⟩
\flag_if_exist:NTF ⟨*flag var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

This function returns true if the ⟨*flag var*⟩ is currently defined, and false otherwise. This does not check that the ⟨*flag var*⟩ really is a flag variable.

\flag_if_raised_p:N ⋆
\flag_if_raised_p:c ⋆
\flag_if_raised:N*TF* ⋆
\flag_if_raised:c*TF* ⋆

New: 2024-01-12

\flag_if_raised_p:N ⟨*flag var*⟩
\flag_if_raised:NTF ⟨*flag var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

This function returns true if the ⟨*flag var*⟩ has non-zero height, and false if the ⟨*flag var*⟩ has zero height.

\flag_height:N ⋆
\flag_height:c ⋆

New: 2024-01-12

\flag_height:N ⟨*flag var*⟩

Expands to the height of the ⟨*flag var*⟩ as an integer denotation.

\flag_raise:N ⋆
\flag_raise:c ⋆

New: 2024-01-12

\flag_raise:N ⟨*flag var*⟩

The height of ⟨*flag var*⟩ is increased by 1 locally.

\flag_ensure_raised:N ⋆
\flag_ensure_raised:c ⋆

New: 2024-01-12

\flag_ensure_raised:N ⟨*flag var*⟩

Ensures the ⟨*flag var*⟩ is raised by making its height at least 1, locally.

`\l_tmpa_flag`
`\l_tmpb_flag`

New: 2024-01-12

Scratch flag for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 24

The l3clist module Comma separated lists

Comma lists (in short, `clist`) contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in l3seq) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with LaTeX 2ε or other code that expects or provides items separated by commas.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn` ... `\ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists "by hand"). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e~} , , {{f}} , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{{f}}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty or blank items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_set:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an "unsafe" item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Any `n`-type token list is a valid comma list input for l3clist functions, which will split the token list at every comma and process the items as described above. On the other hand, `N`-type functions expect comma list variables, which are particular token list variables in which this processing of items (and removal of blank items) has already occurred. Because comma list variables are token list variables, expanding them once yields their items separated by commas, and l3tl functions such as `\tl_show:N` can be applied to them. (These functions often have l3clist analogues, which should be preferred.)

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using `\seq_set_from_clist:Nn` (see l3seq) may be advisable if speed is important. The exception is that `\clist_if_in:NnTF` and `\clist_remove_duplicates:N` may be faster than their sequence analogues for large lists. However, these functions work slowly for "unsafe" items that must be braced, and may produce errors when their argument contains `{`, `}` or `#` (assuming the usual TeX category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

24.1 Creating and initializing comma lists

`\clist_new:N`
`\clist_new:c`

`\clist_new:N` ⟨*clist var*⟩

Creates a new ⟨*clist var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*clist var*⟩ initially contains no items.

`\clist_const:Nn`
`\clist_const:(Ne|cn|ce)`

`\clist_const:Nn` ⟨*clist var*⟩ `{`⟨*comma list*⟩`}`

Creates a new constant ⟨*clist var*⟩ or raises an error if the name is already taken. The value of the ⟨*clist var*⟩ is set globally to the ⟨*comma list*⟩.

`\clist_clear:N`
`\clist_clear:c`
`\clist_gclear:N`
`\clist_gclear:c`

`\clist_clear:N` ⟨*clist var*⟩

Clears all items from the ⟨*clist var*⟩.

`\clist_clear_new:N`
`\clist_clear_new:c`
`\clist_gclear_new:N`
`\clist_gclear_new:c`

`\clist_clear_new:N` ⟨*clist var*⟩

Ensures that the ⟨*clist var*⟩ exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

`\clist_set_eq:NN`
`\clist_set_eq:(cN|Nc|cc)`
`\clist_gset_eq:NN`
`\clist_gset_eq:(cN|Nc|cc)`

`\clist_set_eq:NN` ⟨*clist var*$_1$⟩ ⟨*clist var*$_2$⟩

Sets the content of ⟨*clist var*$_1$⟩ equal to that of ⟨*clist var*$_2$⟩. To set a token list variable equal to a comma list variable, use `\tl_set_eq:NN`. Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.

`\clist_set_from_seq:NN`
`\clist_set_from_seq:(cN|Nc|cc)`
`\clist_gset_from_seq:NN`
`\clist_gset_from_seq:(cN|Nc|cc)`

`\clist_set_from_seq:NN` ⟨*clist var*⟩ ⟨*seq var*⟩

Converts the data in the ⟨*seq var*⟩ into a ⟨*clist var*⟩: the original ⟨*seq var*⟩ is unchanged. Items which contain either spaces or commas are surrounded by braces.

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

\clist_concat:NNN ⟨*clist var*$_1$⟩ ⟨*clist var*$_2$⟩ ⟨*clist var*$_3$⟩

Concatenates the content of ⟨*clist var*$_2$⟩ and ⟨*clist var*$_3$⟩ together and saves the result in ⟨*clist var*$_1$⟩. The items in ⟨*clist var*$_2$⟩ are placed at the left side of the new comma list.

\clist_if_exist_p:N ⋆
\clist_if_exist_p:c ⋆
\clist_if_exist:N*TF* ⋆
\clist_if_exist:c*TF* ⋆

\clist_if_exist_p:N ⟨*clist var*⟩
\clist_if_exist:NTF ⟨*clist var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*clist var*⟩ is currently defined. This does not check that the ⟨*clist var*⟩ really is a comma list.

24.2 Adding data to comma lists

\clist_set:Nn
\clist_set:(NV|Ne|No|cn|cV|ce|co)
\clist_gset:Nn
\clist_gset:(NV|Ne|No|cn|cV|ce|co)

\clist_set:Nn ⟨*clist var*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Sets ⟨*clist var*⟩ to contain the ⟨*items*⟩, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: \clist_set:Nn ⟨*clist var*⟩ { {⟨*tokens*⟩} }.

\clist_put_left:Nn
\clist_put_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\clist_gput_left:Nn
\clist_gput_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\clist_put_left:Nn ⟨*clist var*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Appends the ⟨*items*⟩ to the left of the ⟨*clist var*⟩. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: \clist_put_left:Nn ⟨*clist var*⟩ { {⟨*tokens*⟩} }.

\clist_put_right:Nn
\clist_put_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\clist_gput_right:Nn
\clist_gput_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\clist_put_right:Nn ⟨*clist var*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Appends the ⟨*items*⟩ to the right of the ⟨*clist var*⟩. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: \clist_put_right:Nn ⟨*clist var*⟩ { {⟨*tokens*⟩} }.

24.3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

\clist_remove_duplicates:N
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c

\clist_remove_duplicates:N ⟨*clist var*⟩

Removes duplicate items from the ⟨*clist var*⟩, leaving the left most copy of each item in the ⟨*clist var*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for \tl_if_eq:nnTF.

TeXhackers note: This function iterates through every item in the ⟨*clist var*⟩ and does a comparison with the ⟨*items*⟩ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the ⟨*clist var*⟩ contains {, }, or # (assuming the usual TeX category codes apply).

\clist_remove_all:Nn
\clist_remove_all:(cn|NV|cV|Ne|ce)
\clist_gremove_all:Nn
\clist_gremove_all:(cn|NV|cV|Ne|ce)

\clist_remove_all:Nn ⟨*clist var*⟩ {⟨*item*⟩}

Removes every occurrence of ⟨*item*⟩ from the ⟨*clist var*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for \tl_if_eq:nnTF.

TeXhackers note: The function may fail if the ⟨*item*⟩ contains {, }, or # (assuming the usual TeX category codes apply).

\clist_reverse:N
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c

\clist_reverse:N ⟨*clist var*⟩

Reverses the order of items stored in the ⟨*clist var*⟩.

\clist_reverse:n ⋆

\clist_reverse:n {⟨*comma list*⟩}

Leaves the items in the ⟨*comma list*⟩ in the input stream in reverse order. Contrarily to other what is done for other n-type ⟨*comma list*⟩ arguments, braces and spaces are preserved by this process.

TeXhackers note: The result is returned within \unexpanded, which means that the comma list does not expand further when appearing in an e-type or x-type argument expansion.

\clist_sort:Nn
\clist_sort:cn
\clist_gsort:Nn
\clist_gsort:cn

\clist_sort:Nn ⟨*clist var*⟩ {⟨*comparison code*⟩}

Sorts the items in the ⟨*clist var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*clist var*⟩. The details of sorting comparison are described in Section 6.1.

24.4 Comma list conditionals

\clist_if_empty_p:N ⋆
\clist_if_empty_p:c ⋆
\clist_if_empty:NTF ⋆
\clist_if_empty:cTF ⋆

`\clist_if_empty_p:N ⟨clist var⟩`
`\clist_if_empty:NTF ⟨clist var⟩ {⟨true code⟩} {⟨false code⟩}`

Tests if the ⟨*clist var*⟩ is empty (containing no items).

\clist_if_empty_p:n ⋆
\clist_if_empty: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.

\clist_if_in:NnTF
\clist_if_in:(NV|No|cn|cV|co)TF
\clist_if_in:nnTF
\clist_if_in:(nV|no)TF

`\clist_if_in:NnTF ⟨clist var⟩ {⟨item⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the ⟨*item*⟩ is present in the ⟨*clist var*⟩. In the case of an `n`-type ⟨*comma list*⟩, the usual rules of space trimming and brace stripping apply. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields `true`.

TEXhackers note: The function may fail if the ⟨*item*⟩ contains `{`, `}`, or `#` (assuming the usual TEX category codes apply).

24.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, i.e., any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

When the comma list is given explicitly, as an `n`-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is `{a␣,␣{{b}␣},␣,{},␣{c},}` then the arguments passed to the mapped function are '`a`', '`{b}␣`', an empty argument, and '`c`'.

When the comma list is given as an `N`-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using `n`-type comma lists.

\clist_map_function:NN ☆
\clist_map_function:cN ☆
\clist_map_function:nN ☆
\clist_map_function:eN ☆

`\clist_map_function:NN ⟨clist var⟩ ⟨function⟩`

Applies ⟨*function*⟩ to every ⟨*item*⟩ stored in the ⟨*clist var*⟩. The ⟨*function*⟩ receives one argument for each iteration. The ⟨*items*⟩ are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`.

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn

\clist_map_inline:Nn ⟨*clist var*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*clist var*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as #1. The ⟨*items*⟩ are returned from left to right.

\clist_map_variable:NNn
\clist_map_variable:cNn
\clist_map_variable:nNn

\clist_map_variable:NNn ⟨*clist var*⟩ ⟨*variable*⟩ {⟨*code*⟩}

Stores each ⟨*item*⟩ of the ⟨*clist var*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*clist var*⟩, or its original value if there were no ⟨*item*⟩. The ⟨*items*⟩ are returned from left to right.

\clist_map_tokens:Nn ☆
\clist_map_tokens:cn ☆
\clist_map_tokens:nn ☆

New: 2021-05-05

\clist_map_tokens:Nn ⟨*clist var*⟩ {⟨*code*⟩}
\clist_map_tokens:nn {⟨*comma list*⟩} {⟨*code*⟩}

Calls ⟨*code*⟩ {⟨*item*⟩} for every ⟨*item*⟩ stored in the ⟨*clist var*⟩. The ⟨*code*⟩ receives each ⟨*item*⟩ as a trailing brace group. If the ⟨*code*⟩ consists of a single function this is equivalent to \clist_map_function:nN.

\clist_map_break: ☆

\clist_map_break:

Used to terminate a \clist_map_... function before all entries in the ⟨*comma list*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \clist_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\clist_map_break:n ☆ \clist_map_break:n {⟨*code*⟩}

Used to terminate a \clist_map_... function before all entries in the ⟨*comma list*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \clist_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

\clist_count:N ⋆
\clist_count:c ⋆
\clist_count:n ⋆
\clist_count:e ⋆

\clist_count:N ⟨*clist var*⟩

Leaves the number of items in the ⟨*clist var*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*clist var*⟩ includes those which are duplicates, i.e., every item in a ⟨*clist var*⟩ is counted.

24.6 Using the content of comma lists directly

\clist_use:Nnnn ⋆
\clist_use:cnnn ⋆

\clist_use:Nnnn ⟨*clist var*⟩ {⟨*separator between two*⟩}
{⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the comma list has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts "a, b, c, de, and f" in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

TeXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*items*⟩ do not expand further when appearing in an e-type or x-type argument expansion.

\clist_use:Nn ⋆
\clist_use:cn ⋆

\clist_use:Nn ⟨*clist var*⟩ {⟨*separator*⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts "a and b and c and de and f" in the input stream.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*items*⟩ do not expand further when appearing in an e-type or x-type argument expansion.

\clist_use:N ⋆
\clist_use:c ⋆

New: 2024-11-12

\clist_use:N ⟨*clist var*⟩

Places the contents of the ⟨*clist var*⟩ in the input stream, with a comma between each item. The result is exactly the stored ⟨*clist*⟩, which will include braces around (for example) entries with retained spaces at the ends.

TEXhackers note: The result is returned as-is, in the same way as \tl_use:N and *without* protection from expansion, cf. \clist_use:Nnnnn, etc. It is equivalent to V-type expansion of a clist.

\clist_use:nnnn ⋆
\clist_use:nn ⋆

New: 2021-05-10

\clist_use:nnnn {⟨*comma list*⟩} {⟨*separator between two*⟩}
{⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}
\clist_use:nn {⟨*comma list*⟩} {⟨*separator*⟩}

Places the contents of the ⟨*comma list*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. As for \clist_set:Nn, blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. The ⟨*separators*⟩ are then inserted in the same way as for \clist_use:Nnnn and \clist_use:Nn, respectively.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*items*⟩ do not expand further when appearing in an e-type or x-type argument expansion.

24.7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

\clist_get:NN
\clist_get:cN
\clist_get:NNTF
\clist_get:cNTF

\clist_get:NN ⟨clist var⟩ ⟨tl var⟩

Stores the left-most item from a ⟨clist var⟩ in the ⟨tl var⟩ without removing it from the ⟨clist var⟩. The ⟨tl var⟩ is assigned locally. In the non-branching version, if the ⟨clist var⟩ is empty the ⟨tl var⟩ is set to the marker value \q_no_value.

\clist_pop:NN
\clist_pop:cN

\clist_pop:NN ⟨clist var⟩ ⟨tl var⟩

Pops the left-most item from a ⟨clist var⟩ into the ⟨tl var⟩, i.e., removes the item from the comma list and stores it in the ⟨tl var⟩. Both of the variables are assigned locally.

\clist_gpop:NN
\clist_gpop:cN

\clist_gpop:NN ⟨clist var⟩ ⟨tl var⟩

Pops the left-most item from a ⟨clist var⟩ into the ⟨tl var⟩, i.e., removes the item from the comma list and stores it in the ⟨tl var⟩. The ⟨clist var⟩ is modified globally, while the assignment of the ⟨tl var⟩ is local.

\clist_pop:NNTF
\clist_pop:cNTF

\clist_pop:NNTF ⟨clist var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨clist var⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨tl var⟩ is not defined in this case and should not be relied upon. If the ⟨clist var⟩ is non-empty, pops the top item from the ⟨clist var⟩ in the ⟨tl var⟩, i.e., removes the item from the ⟨clist var⟩. Both the ⟨clist var⟩ and the ⟨tl var⟩ are assigned locally.

\clist_gpop:NNTF
\clist_gpop:cNTF

\clist_gpop:NNTF ⟨clist var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨clist var⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨tl var⟩ is not defined in this case and should not be relied upon. If the ⟨clist var⟩ is non-empty, pops the top item from the ⟨clist var⟩ in the ⟨tl var⟩, i.e., removes the item from the ⟨clist var⟩. The ⟨clist var⟩ is modified globally, while the ⟨tl var⟩ is assigned locally.

\clist_push:Nn
\clist_push:(NV|No|cn|cV|co)
\clist_gpush:Nn
\clist_gpush:(NV|No|cn|cV|co)

\clist_push:Nn ⟨clist var⟩ {⟨items⟩}

Adds the {⟨items⟩} to the top of the ⟨clist var⟩. Spaces are removed from both sides of each item as for any n-type comma list.

24.8 Using a single item

\clist_item:Nn ⋆
\clist_item:cn ⋆
\clist_item:nn ⋆
\clist_item:en ⋆

\clist_item:Nn ⟨*clist var*⟩ {⟨*int expr*⟩}

Indexing items in the ⟨*clist var*⟩ from 1 at the top (left), this function evaluates the ⟨*int expr*⟩ and leaves the appropriate item from the comma list in the input stream. If the ⟨*int expr*⟩ is negative, indexing occurs from the bottom (right) of the comma list. When the ⟨*int expr*⟩ is larger than the number of items in the ⟨*clist var*⟩ (as calculated by \clist_count:N) then the function expands to nothing.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*item*⟩ does not expand further when appearing in an e-type or x-type argument expansion.

\clist_rand_item:N ⋆
\clist_rand_item:c ⋆
\clist_rand_item:n ⋆

\clist_rand_item:N ⟨*clist var*⟩
\clist_rand_item:n {⟨*comma list*⟩}

Selects a pseudo-random item of the ⟨*clist var*⟩/⟨*comma list*⟩. If the ⟨*comma list*⟩ has no item, the result is empty.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*item*⟩ does not expand further when appearing in an e-type or x-type argument expansion.

24.9 Viewing comma lists

\clist_show:N
\clist_show:c

Updated: 2021-04-29

\clist_show:N ⟨*clist var*⟩

Displays the entries in the ⟨*clist var*⟩ in the terminal.

\clist_show:n

\clist_show:n {⟨*tokens*⟩}

Displays the entries in the comma list in the terminal.

\clist_log:N
\clist_log:c

Updated: 2021-04-29

\clist_log:N ⟨*clist var*⟩

Writes the entries in the ⟨*clist var*⟩ in the log file. See also \clist_show:N which displays the result in the terminal.

\clist_log:n

\clist_log:n {⟨*tokens*⟩}

Writes the entries in the comma list in the log file. See also \clist_show:n which displays the result in the terminal.

24.10 Constant and scratch comma lists

\c_empty_clist Constant that is always empty.

\l_tmpa_clist
\l_tmpb_clist

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

Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 25

The l3token module
Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in TeX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its "shape" (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its "meaning", which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of TeX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, TeX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section 25.7.

25.1 Creating character tokens

\char_set_active_eq:NN
\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc

\char_set_active_eq:NN ⟨*char*⟩ ⟨*function*⟩

Sets the behavior of the ⟨*char*⟩ in situations where it is active (category code 13) to be equivalent to that of the definition of the ⟨*function*⟩ at the time \char_set_active_-eq:NN is used. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc

\char_set_active_eq:nN {⟨*integer expression*⟩} ⟨*function*⟩

Sets the behavior of the ⟨*char*⟩ which has character code as given by the ⟨*integer expression*⟩ in situations where it is active (category code 13) to be equivalent to that of the ⟨*function*⟩ at the time \char_set_active_eq:nN is used. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

\char_generate:nn ⋆

\char_generate:nn {⟨*charcode*⟩} {⟨*catcode*⟩}

Generates a character token of the given ⟨*charcode*⟩ and ⟨*catcode*⟩ (both of which may be integer expressions). The ⟨*catcode*⟩ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The ⟨*charcode*⟩ may be any one valid for the engine in use, except that for ⟨*catcode*⟩ 10, ⟨*charcode*⟩ 0 is not allowed. Active characters cannot be generated in older versions of XeTeX. Another way to build token lists with unusual category codes is \regex_replace_all:nnN {.*} {⟨*replacement*⟩} ⟨*tl var*⟩.

TeXhackers note: Exactly two expansions are needed to produce the character.

\c_catcode_active_space_tl

Token list containing one character with category code 13, (“active”), and character code 32 (space).

`\c_catcode_other_space_tl` Token list containing one character with category code 12, ("other"), and character code 32 (space).

25.2 Manipulating and interrogating character tokens

```
\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
```

`\char_set_catcode_letter:N ⟨character⟩`

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 {⟨int expr_1⟩} {⟨int expr_2⟩}`

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_lccode:nn {⟨int expr_1⟩} {⟨int expr_2⟩}`

Sets up the behavior of the ⟨*character*⟩ when found inside `\text_lowercase:n`, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TeX `‘`⟨*character*⟩ method for converting a single character into its character code:

```
\char_set_lccode:nn { ‘\A } { ‘\a } % Standard behavior
\char_set_lccode:nn { ‘\A } { ‘\A + 32 }
\char_set_lccode:nn { 50 } { 60 }
```

The setting applies within the current TeX group.

`\char_value_lccode:n` ⋆ `\char_value_lccode:n {⟨integer expression⟩}`

Expands to the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_lccode:n` `\char_show_value_lccode:n {⟨integer expression⟩}`

Displays the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_uccode:nn` `\char_set_uccode:nn {⟨int expr_1⟩} {⟨int expr_2⟩}`

Sets up the behavior of the ⟨*character*⟩ when found inside `\text_uppercase:n`, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TeX `‘`⟨*character*⟩ method for converting a single character into its character code:

```
\char_set_uccode:nn { ‘\a } { ‘\A } % Standard behavior
\char_set_uccode:nn { ‘\A } { ‘\A - 32 }
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current TeX group.

`\char_value_uccode:n` ⋆ `\char_value_uccode:n {⟨integer expression⟩}`

Expands to the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_uccode:n` `\char_show_value_uccode:n {⟨integer expression⟩}`

Displays the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_mathcode:nn` `\char_set_mathcode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}`

This function sets up the math code of ⟨*character*⟩. The ⟨*character*⟩ is specified as an ⟨*integer expression*⟩ which will be used as the character code of the relevant character. The setting applies within the current TeX group.

`\char_value_mathcode:n` ⋆ `\char_value_mathcode:n {⟨integer expression⟩}`

Expands to the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_mathcode:n` `\char_show_value_mathcode:n {⟨integer expression⟩}`

Displays the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_sfcode:nn` `\char_set_sfcode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}`

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` Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category ⟨*active*⟩ (catcode 13). Each entry in the sequence consists of a single escaped token, for example `\~`. Active tokens should be added to the sequence when they are defined for general document use.

`\l_char_special_seq` 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.

25.3 Generic tokens

`\c_group_begin_token`
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`
`\c_math_superscript_token`
`\c_math_subscript_token`
`\c_space_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

TEXhackers note: The tokens `\c_group_begin_token`, `\c_group_end_token`, and `\c_space_token` are expl3 counterparts of LaTeX 2ε's `\bgroup`, `\egroup`, and `\@sptoken`.

`\c_catcode_letter_token`
`\c_catcode_other_token`

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

25.4 Converting tokens

`\token_to_meaning:N` ⋆
`\token_to_meaning:c` ⋆

`\token_to_meaning:N` ⟨*token*⟩

Inserts the current meaning of the ⟨*token*⟩ into the input stream as a series of characters of category code 12 (other). This is the primitive TEX description of the ⟨*token*⟩, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as `macros`.

TEXhackers note: This is the TEX primitive `\meaning`. The ⟨*token*⟩ can thus be an explicit space token or an explicit begin-group or end-group character token (`{` or `}` when normal TEX category codes apply) even though these are not valid `N`-type arguments.

`\token_to_str:N` ⋆
`\token_to_str:c` ⋆

`\token_to_str:N` ⟨*token*⟩

Converts the given ⟨*token*⟩ into a series of characters with category code 12 (other). If the ⟨*token*⟩ is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the ⟨*token*⟩). This function requires only a single expansion.

TEXhackers note: `\token_to_str:N` is the TEX primitive `\string`. The ⟨*token*⟩ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal TEX category codes apply) even though these are not valid `N`-type arguments.

`\token_to_catcode:N` ⋆

New: 2023-10-15

`\token_to_catcode:N` ⟨*token*⟩

Converts the given ⟨*token*⟩ into a number describing its category code. If ⟨*token*⟩ is a control sequence this expands to 16. This can't detect the categories 0 (escape character), 5 (end of line), 9 (ignored character), 14 (comment character), or 15 (invalid character). Control sequences or active characters let to a token of one of the detectable category codes will yield that category.

25.5 Token conditionals

\token_if_group_begin_p:N ⋆
\token_if_group_begin:NTF ⋆

\token_if_group_begin_p:N ⟨token⟩
\token_if_group_begin:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of a begin group token ({ when normal TEX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

\token_if_group_end_p:N ⋆
\token_if_group_end:NTF ⋆

\token_if_group_end_p:N ⟨token⟩
\token_if_group_end:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of an end group token (} when normal TEX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

\token_if_math_toggle_p:N ⋆
\token_if_math_toggle:NTF ⋆

\token_if_math_toggle_p:N ⟨token⟩
\token_if_math_toggle:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of a math shift token ($ when normal TEX category codes are in force).

\token_if_alignment_p:N ⋆
\token_if_alignment:NTF ⋆

\token_if_alignment_p:N ⟨token⟩
\token_if_alignment:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of an alignment token (& when normal TEX category codes are in force).

\token_if_parameter_p:N ⋆
\token_if_parameter:NTF ⋆

\token_if_parameter_p:N ⟨token⟩
\token_if_parameter:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of a macro parameter token (# when normal TEX category codes are in force).

\token_if_math_superscript_p:N ⋆
\token_if_math_superscript:NTF ⋆

\token_if_math_superscript_p:N ⟨token⟩
\token_if_math_superscript:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of a superscript token (^ when normal TEX category codes are in force).

\token_if_math_subscript_p:N ⋆
\token_if_math_subscript:NTF ⋆

\token_if_math_subscript_p:N ⟨token⟩
\token_if_math_subscript:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of a subscript token (_ when normal TEX category codes are in force).

\token_if_space_p:N ⋆
\token_if_space:NTF ⋆

\token_if_space_p:N ⟨token⟩
\token_if_space:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

`\token_if_letter_p:N` ⋆
`\token_if_letter:NTF` ⋆

`\token_if_letter_p:N` ⟨*token*⟩
`\token_if_letter:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a letter token.

`\token_if_other_p:N` ⋆
`\token_if_other:NTF` ⋆

`\token_if_other_p:N` ⟨*token*⟩
`\token_if_other:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an "other" token.

`\token_if_active_p:N` ⋆
`\token_if_active:NTF` ⋆

`\token_if_active_p:N` ⟨*token*⟩
`\token_if_active:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an active character.

`\token_if_eq_catcode_p:NN` ⋆
`\token_if_eq_catcode:NNTF` ⋆

`\token_if_eq_catcode_p:NN` ⟨$token_1$⟩ ⟨$token_2$⟩
`\token_if_eq_catcode:NNTF` ⟨$token_1$⟩ ⟨$token_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the two ⟨*tokens*⟩ have the same category code.

`\token_if_eq_charcode_p:NN` ⋆
`\token_if_eq_charcode:NNTF` ⋆

`\token_if_eq_charcode_p:NN` ⟨$token_1$⟩ ⟨$token_2$⟩
`\token_if_eq_charcode:NNTF` ⟨$token_1$⟩ ⟨$token_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the two ⟨*tokens*⟩ have the same character code.

`\token_if_eq_meaning_p:NN` ⋆
`\token_if_eq_meaning:NNTF` ⋆

`\token_if_eq_meaning_p:NN` ⟨$token_1$⟩ ⟨$token_2$⟩
`\token_if_eq_meaning:NNTF` ⟨$token_1$⟩ ⟨$token_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the two ⟨*tokens*⟩ have the same meaning when expanded.

`\token_if_macro_p:N` ⋆
`\token_if_macro:NTF` ⋆

`\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_control_symbol_p:N` ⋆ `\token_if_control_symbol_p:N` ⟨*token*⟩
`\token_if_control_symbol:NTF` ⋆ `\token_if_control_symbol:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

New: 2025-05-12

Tests whether the ⟨*token*⟩ is a control sequence with a name comprised of exactly one non-letter character (called a "control symbol"). Specifically, only the following tokens leave ⟨*false code*⟩:

- explicit characters, such as `a` or `"`
- the escape character followed by one or more characters of category code 11 (letter), such as `\foo`

Any other token will leave ⟨*true code*⟩. The category codes which apply are those at the point the test is used, not those used when the ⟨*token*⟩ is defined.

`\token_if_control_word_p:N` ⋆ `\token_if_control_word_p:N` ⟨*token*⟩
`\token_if_control_word:NTF` ⋆ `\token_if_control_word:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

New: 2025-05-12

Tests whether the ⟨*token*⟩ is a control sequence with a name comprised of one or more letters (called a "control word"). Specifically, only these tokens leave ⟨*false code*⟩:

- explicit characters, such as `a` or `"`
- the escape character followed by exactly one character whose category code is not 11 (letter) when used (not tokenized), such as `\ `, `\&`

Any other token will leave ⟨*true code*⟩. The category codes which apply are those at the point the test is used, not those used when the ⟨*token*⟩ is defined.

`\token_if_long_macro_p:N` ⋆ `\token_if_long_macro_p:N` ⟨*token*⟩
`\token_if_long_macro:NTF` ⋆ `\token_if_long_macro:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is a long macro with no other prefix; to test for a macro that is both long and protected, use `\token_if_protected_long_macro:N(TF)`.

`\token_if_protected_macro_p:N` ⋆ `\token_if_protected_macro_p:N` ⟨*token*⟩
`\token_if_protected_macro:NTF` ⋆ `\token_if_protected_macro:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is a protected macro with no other prefix; to test for a macro that is both protected and long, use `\token_if_protected_long_macro:N(TF)`.

`\token_if_protected_long_macro_p:N` ⋆ `\token_if_protected_long_macro_p:N` ⟨*token*⟩
`\token_if_protected_long_macro:NTF` ⋆ `\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_p:N` ⟨*token*⟩
`\token_if_chardef:NTF` ⋆ `\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 `\chardef`s.

\token_if_mathchardef_p:N ⋆ \token_if_mathchardef_p:N ⟨*token*⟩
\token_if_mathchardef:N*TF* ⋆ \token_if_mathchardef:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a mathchardef.

\token_if_font_selection_p:N ⋆ \token_if_font_selection_p:N ⟨*token*⟩
\token_if_font_selection:N*TF* ⋆ \token_if_font_selection:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

New: 2020-10-27

Tests if the ⟨*token*⟩ is defined to be a font selection command.

\token_if_dim_register_p:N ⋆ \token_if_dim_register_p:N ⟨*token*⟩
\token_if_dim_register:N*TF* ⋆ \token_if_dim_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a dimension register.

\token_if_int_register_p:N ⋆ \token_if_int_register_p:N ⟨*token*⟩
\token_if_int_register:N*TF* ⋆ \token_if_int_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a integer register.

TeXhackers note: Constant integers may be implemented as integer registers, \chardefs, or \mathchardefs depending on their value.

\token_if_muskip_register_p:N ⋆ \token_if_muskip_register_p:N ⟨*token*⟩
\token_if_muskip_register:N*TF* ⋆ \token_if_muskip_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a muskip register.

\token_if_skip_register_p:N ⋆ \token_if_skip_register_p:N ⟨*token*⟩
\token_if_skip_register:N*TF* ⋆ \token_if_skip_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a skip register.

\token_if_toks_register_p:N ⋆ \token_if_toks_register_p:N ⟨*token*⟩
\token_if_toks_register:N*TF* ⋆ \token_if_toks_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a toks register (not used by LaTeX3).

\token_if_primitive_p:N ⋆ \token_if_primitive_p:N ⟨*token*⟩
\token_if_primitive:N*TF* ⋆ \token_if_primitive:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2020-09-11

Tests if the ⟨*token*⟩ is an engine primitive. In LuaTeX this includes primitive-like commands defined using token.set_lua.

\token_case_catcode:Nn ⋆
\token_case_catcode:Nn*TF* ⋆
\token_case_charcode:Nn ⋆
\token_case_charcode:Nn*TF* ⋆
\token_case_meaning:Nn ⋆
\token_case_meaning:Nn*TF* ⋆

New: 2020-12-03

```
\token_case_meaning:NnTF ⟨test token⟩
  {
    ⟨token case1⟩ {⟨code case1⟩}
    ⟨token case2⟩ {⟨code case2⟩}
    ...
    ⟨token casen⟩ {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the ⟨*test token*⟩ in turn with each of the ⟨*token case*⟩s. If the two are equal (as described for \token_if_eq_catcode:NNTF, \token_if_eq_-charcode:NNTF and \token_if_eq_meaning:NNTF, respectively) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The functions \token_case_catcode:Nn, \token_case_charcode:Nn, and \token_case_meaning:Nn, which do nothing if there is no match, are also available.

25.6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the "peek" functions. The generic \peek_after:Nw is provided along with a family of predefined tests for common cases. Peeking ahead does *not* skip spaces: rather, \peek_remove_spaces:n should be used. In addition, using \peek_analysis_map_inline:n, one can map through the following tokens in the input stream and repeatedly perform some tests.

\peek_after:Nw

\peek_after:Nw ⟨*function*⟩ ⟨*token*⟩

Locally sets the test variable \l_peek_token equal to ⟨*token*⟩ (as an implicit token, *not* as a token list), and then expands the ⟨*function*⟩. The ⟨*token*⟩ remains in the input stream as the next item after the ⟨*function*⟩. The ⟨*token*⟩ here may be ␣, { or } (assuming normal TeX category codes), i.e., it is not necessarily the next argument which would be grabbed by a normal function.

\peek_gafter:Nw

\peek_gafter:Nw ⟨*function*⟩ ⟨*token*⟩

Globally sets the test variable \g_peek_token equal to ⟨*token*⟩ (as an implicit token, *not* as a token list), and then expands the ⟨*function*⟩. The ⟨*token*⟩ remains in the input stream as the next item after the ⟨*function*⟩. The ⟨*token*⟩ here may be ␣, { or } (assuming normal TeX category codes), i.e., it is not necessarily the next argument which would be grabbed by a normal function.

\l_peek_token

Token set by \peek_after:Nw and available for testing as described above.

\g_peek_token

Token set by \peek_gafter:Nw and available for testing as described above.

\peek_catcode:N*TF* \peek_catcode:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_catcode_remove:N*TF* \peek_catcode_remove:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_charcode:N*TF* \peek_charcode:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_charcode_remove:N*TF* \peek_charcode_remove:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_meaning:N*TF* \peek_meaning:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_meaning_remove:N*TF* \peek_meaning_remove:NTF ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_remove_spaces:n \peek_remove_spaces:n {⟨*code*⟩}

Peeks ahead and detect if the following token is a space (category code 10 and character code 32). If so, removes the token and checks the next token. Once a non-space token is found, the ⟨*code*⟩ will be inserted into the input stream. Typically this will contain a peek operation, but this is not required.

`\peek_remove_filler:n` `\peek_remove_filler:n {⟨code⟩}`

New: 2022-01-10

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to `\scan_stop:`. If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using `f`-type expansion, then checks the resulting leading token in the same way. If after expansion the next token is neither of the two test cases, the ⟨*code*⟩ will be inserted into the input stream. Typically this will contain a `peek` operation, but this is not required.

TEXhackers note: This is essentially a macro-based implementation of how TEX handles the search for a left brace after for example `\everypar`, except that any non-expandable token cleanly ends the ⟨*filler*⟩ (i.e., it does not lead to a TEX error).

In contrast to TEX's filler removal, a construct `\exp_not:N \foo` will be treated in the same way as `\foo`.

`\peek_N_type:TF` `\peek_N_type:TF {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream can be safely grabbed as an `N`-type argument. The test is ⟨*false*⟩ if the next ⟨*token*⟩ is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in LATEX3) and ⟨*true*⟩ in all other cases. Note that a ⟨*true*⟩ result ensures that the next ⟨*token*⟩ is a valid `N`-type argument. However, if the next ⟨*token*⟩ is for instance `\c_space_token`, the test takes the ⟨*false*⟩ branch, even though the next ⟨*token*⟩ is in fact a valid `N`-type argument. The ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_analysis_map_inline:n` `\peek_analysis_map_inline:n {⟨inline function⟩}`

New: 2020-12-03
Updated: 2024-02-07

Repeatedly removes one ⟨*token*⟩ from the input stream and applies the ⟨*inline function*⟩ to it, until `\peek_analysis_map_break:` is called. The ⟨*inline function*⟩ receives three arguments for each ⟨*token*⟩ in the input stream:

- ⟨*tokens*⟩, which both `o`-expand and `e`/`x`-expand to the ⟨*token*⟩. The detailed form of ⟨*tokens*⟩ may change in later releases.
- ⟨*char code*⟩, a decimal representation of the character code of the ⟨*token*⟩, −1 if it is a control sequence.
- ⟨*catcode*⟩, a capital hexadecimal digit which denotes the category code of the ⟨*token*⟩ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing `"`⟨*catcode*⟩.

These arguments are the same as for `\tl_analysis_map_inline:nn` defined in l3tl-analysis. The ⟨*char code*⟩ and ⟨*catcode*⟩ do not take the meaning of a control sequence or active character into account: for instance, upon encountering the token `\c_group_begin_token` in the input stream, `\peek_analysis_map_inline:n` calls the ⟨*inline function*⟩ with `#1` being `\exp_not:n { \c_group_begin_token }` (with the current implementation), `#2` being −1, and `#3` being 0, as for any other control sequence. In contrast, upon encountering an explicit begin-group token `{`, the ⟨*inline function*⟩ is called with arguments `\exp_after:wN { \if_false: } \fi:`, 123 and 1.

The mapping is done at the current group level, i.e., any local assignments made by the ⟨*inline function*⟩ remain in effect after the loop. Within the code, `\l_peek_token` is set equal (as a token, not a token list) to the token under consideration.

Peek functions cannot be used within this mapping function (nor other mapping functions) since the input stream contains trailing material necessary for the functioning of the loop.

TeXhackers note: In case the input stream has not yet been tokenized (converted from characters to tokens), characters are tokenized one by one as needed by `\peek_analysis_map_inline:n` using the current category code régime.

`\peek_analysis_map_break:`
`\peek_analysis_map_break:n`

```
\peek_analysis_map_inline:n
{ ...  \peek_analysis_map_break:n {⟨code⟩} }
```

New: 2020-12-03

Stops the `\peek_analysis_map_inline:n` loop from seeking more tokens, and inserts ⟨*code*⟩ in the input stream (empty for `\peek_analysis_map_break:`).

\peek_regex:n*TF*
\peek_regex:N*TF*

New: 2020-12-03

\peek_regex:nTF {⟨*regex*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*tokens*⟩ that follow in the input stream match the ⟨*regular expression*⟩. Any ⟨*tokens*⟩ that have been read are left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test). See l3regex for documentation of the syntax of regular expressions. The ⟨*regular expression*⟩ is implicitly anchored at the start, so for instance \peek_regex:nTF { a } is essentially equivalent to \peek_-charcode:NTF a.

TEXhackers note: Implicit character tokens are correctly considered by \peek_regex:nTF as control sequences, while functions that inspect individual tokens (for instance \peek_-charcode:NTF) only take into account their meaning.

The \peek_regex:nTF function only inspects as few tokens as necessary to determine whether the regular expression matches. For instance \peek_regex:nTF { abc | [a-z] } { } { } abc will only inspect the first token a even though the first branch abc of the alternative is preferred in functions such as \peek_regex_remove_once:nTF. This may have an effect on tokenization if the input stream has not yet been tokenized and category codes are changed.

\peek_regex_remove_once:n*TF*
\peek_regex_remove_once:N*TF*

New: 2020-12-03

\peek_regex_remove_once:nTF {⟨*regex*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*tokens*⟩ that follow in the input stream match the ⟨*regex*⟩. If the test is true, the ⟨*tokens*⟩ are removed from the input stream and the ⟨*true code*⟩ is inserted, while if the test is false, the ⟨*false code*⟩ is inserted followed by the ⟨*tokens*⟩ that were originally in the input stream. See l3regex for documentation of the syntax of regular expressions. The ⟨*regular expression*⟩ is implicitly anchored at the start, so for instance \peek_regex_remove_once:nTF { a } is essentially equivalent to \peek_-charcode_remove:NTF a.

TEXhackers note: Implicit character tokens are correctly considered by \peek_regex_-remove_once:nTF as control sequences, while functions that inspect individual tokens (for instance \peek_charcode:NTF) only take into account their meaning.

`\peek_regex_replace_once:nn`
`\peek_regex_replace_once:nnTF`
`\peek_regex_replace_once:Nn`
`\peek_regex_replace_once:NnTF`

New: 2020-12-03

`\peek_regex_replace_once:nnTF {⟨regex⟩} {⟨replacement⟩} {⟨true code⟩} {⟨false code⟩}`

If the ⟨*tokens*⟩ that follow in the input stream match the ⟨*regex*⟩, replaces them according to the ⟨*replacement*⟩ as for `\regex_replace_once:nnN`, and leaves the result in the input stream, after the ⟨*true code*⟩. Otherwise, leaves ⟨*false code*⟩ followed by the ⟨*tokens*⟩ that were originally in the input stream, with no modifications. See l3regex for documentation of the syntax of regular expressions and of the ⟨*replacement*⟩: for instance `\0` in the ⟨*replacement*⟩ is replaced by the tokens that were matched in the input stream. The ⟨*regular expression*⟩ is implicitly anchored at the start. In contrast to `\regex_replace_once:nnN`, no error arises if the ⟨*replacement*⟩ leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

TEXhackers note: Implicit character tokens are correctly considered by `\peek_regex_replace_once:nnTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

25.7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on TeX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for LuaTeX and XeTeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A "frozen" `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand` ⟨*token*⟩ (when the ⟨*token*⟩ is expandable) results in an internal token, displayed (temporarily) as `\notexpanded:` ⟨*token*⟩, whose shape coincides with the ⟨*token*⟩ and whose meaning differs from `\relax`.

- An `\outer endtemplate:` can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.
- In LuaTeX, there is also the strange case of "bytes" `^^^^^^1100`xy where x, y are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from "11 0000 = 1 114 112 to "110 0ff = 1 114 367. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (*not characters*) whose `\meaning` is `the␣character␣` followed by the given byte. If this byte is in the range `80`–`ff` this gives an "invalid utf-8 sequence" error: applying `\token_to_str:N` or `\token_to_meaning:N` to these tokens is unsafe. Unfortunately, they don't seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their LaTeX3 names and most common example:

1 begin-group character (`group_begin`, often `{`),

2 end-group character (`group_end`, often `}`),

3 math shift character (`math_toggle`, often `$`),

4 alignment tab character (`alignment`, often `&`),

6 macro parameter character (`parameter`, often `#`),

7 superscript character (`math_superscript`, often `^`),

8 subscript character (`math_subscript`, often `_`),

10 blank space (`space`, often character code 32),

11 the letter (`letter`, such as `A`),

12 the character (`other`, such as `0`).

Category code 13 (`active`) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in LaTeX3 for most functions and some variables (`tl`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in LaTeX3 for some functions,
- a register such as `\count123`, used in LaTeX3 for the implementation of some variables (`int`, `dim`, ...),

- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros can be `\protected` or not, `\long` or not (the opposite of what LaTeX3 calls `nopar`), and `\outer` or not (unused in LaTeX3). Their `\meaning` takes the form

⟨*prefix*⟩ `macro:`⟨*argument*⟩`->`⟨*replacement*⟩

where ⟨*prefix*⟩ is among `\protected\long\outer`, ⟨*argument*⟩ describes parameters that the macro expects, such as `#1#2#3`, and ⟨*replacement*⟩ describes how the parameters are manipulated, such as `\int_eval:n{#2+#1*#3}`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then TeX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens "N-type", as they are suitable to be used as an argument for a function with the signature `:N`.

When a macro takes a delimited argument TeX scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are *not* ignored in this case (and they prevent brace-stripping).

Chapter 26

The l3prop module Property lists

expl3 implements a "property list" data type, which contains an unordered list of entries each of which consists of a ⟨*key*⟩ (string) and an associated ⟨*value*⟩ (token list). The ⟨*key*⟩ and ⟨*value*⟩ may both be given as any balanced text, and the ⟨*key*⟩ is processed using `\tl_to_str:n`, meaning that category codes are ignored. Entries can be manipulated individually, as well as collectively by applying a function to every key–value pair within the list.

Each entry in a property list must have a unique ⟨*key*⟩: if an entry is added to a property list which already contains the ⟨*key*⟩ then the new entry overwrites the existing one. The ⟨*keys*⟩ are compared on a string basis, using the same method as `\str_if_eq:nnTF`.

Property lists are intended for storing key-based information for use within code. They can be converted from and to key–value lists, which are a form of *input* parsed by the l3keys module. If a key–value list contains a ⟨*key*⟩ multiple times, only the last ⟨*value*⟩ associated to it will be kept in the conversion to a property list.

Internally, property lists can use two distinct implementations with different data storage, which are decided when declaring the property list variable using `\prop_new:N` ("flat" storage) or `\prop_new_linked:N` ("linked" storage). After a property list is declared with `\prop_new:N` or `\prop_new_linked:N`, the type of internal data storage can be changed by `\prop_make_flat:N` or `\prop_make_linked:N`, but only at the outermost group level. All other l3prop functions transparently manipulate either storage method and convert as needed.

- The (default) "flat" storage method is suited for a relatively small number of entries, or when the property list is likely to be manipulated (copied, mapped) as a whole rather than entry-wise. It is significantly faster for `\prop_set_eq:NN`, and only slightly faster for `\prop_clear:N`, `\prop_concat:NNN`, and mapping functions `\prop_map_`....
- The "linked" storage method is meant for property lists with a large numbers of entries. It takes up more of TeX's memory during a run, but is significantly faster (for long lists) when accessing or modifying individual entries using functions such as `\prop_if_in:Nn`, `\prop_item:Nn`, `\prop_put:Nnn`, `\prop_get:NnN`, `\prop_pop:NnN`, `\prop_remove:Nn`, as it takes a constant time for these operations (rather

than the number of items for a "flat" property list). A technical drawback is that memory is permanently used[7] by ⟨*keys*⟩ stored in a "linked" property list, even after they are removed and the property list is deleted.

26.1 Creating and initializing property lists

\prop_new:N
\prop_new:c

\prop_new:N ⟨*property list*⟩

Creates a new "flat" ⟨*property list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*property list*⟩ initially contains no entries. See also \prop_new_linked:N.

\prop_new_linked:N
\prop_new_linked:c

New: 2024-02-12

\prop_new_linked:N ⟨*property list*⟩

Creates a new "linked" ⟨*property list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*property list*⟩ initially contains no entries. The internal data storage differs from that produced by \prop_new:N and it is optimized for property lists with a large number of entries.

\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c

\prop_clear:N ⟨*property list*⟩

Clears all entries from the ⟨*property list*⟩.

\prop_clear_new:N
\prop_clear_new:c
\prop_gclear_new:N
\prop_gclear_new:c

\prop_clear_new:N ⟨*property list*⟩

Ensures that the ⟨*property list*⟩ exists globally by applying \prop_new:N if necessary, then applies \prop_(g)clear:N to leave the list empty.

TeXhackers note: If the property list exists and is of "linked" type, it is cleared but not made into a flat property list.

\prop_clear_new_linked:N
\prop_clear_new_linked:c
\prop_gclear_new_linked:N
\prop_gclear_new_linked:c

New: 2024-02-12

\prop_clear_new_linked:N ⟨*property list*⟩

Ensures that the ⟨*property list*⟩ exists globally by applying \prop_new_linked:N if necessary, then applies \prop_(g)clear:N to leave the list empty.

TeXhackers note: If the property list exists and is of "flat" type, it is cleared but not made into a linked property list.

\prop_set_eq:NN
\prop_set_eq:(cN|Nc|cc)
\prop_gset_eq:NN
\prop_gset_eq:(cN|Nc|cc)

\prop_set_eq:NN ⟨*property list*$_1$⟩ ⟨*property list*$_2$⟩

Sets the content of ⟨*property list*$_1$⟩ equal to that of ⟨*property list*$_2$⟩. This converts as needed between the two storage types.

[7]Until the end of the run, that is.

\prop_set_from_keyval:Nn
\prop_set_from_keyval:cn
\prop_gset_from_keyval:Nn
\prop_gset_from_keyval:cn

Updated: 2021-11-07

```
\prop_set_from_keyval:Nn ⟨property list⟩
  {
    ⟨key1⟩ = ⟨value1⟩ ,
    ⟨key2⟩ = ⟨value2⟩ , ...
  }
```

Sets ⟨property list⟩ to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept. In contrast to most keyval lists (*e.g.* those in l3keys), each key here *must* be followed with an = sign even to specify an empty ⟨value⟩.

Spaces are trimmed around every ⟨key⟩ and every ⟨value⟩, and if the result of trimming spaces consists of a single brace group then a set of outer braces is removed. This enables both the ⟨key⟩ and the ⟨value⟩ to contain spaces, commas or equal signs. The ⟨key⟩ is then processed by \tl_to_str:n. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

\prop_const_from_keyval:Nn
\prop_const_from_keyval:cn

Updated: 2021-11-07

```
\prop_const_from_keyval:Nn ⟨property list⟩
  {
    ⟨key1⟩ = ⟨value1⟩ ,
    ⟨key2⟩ = ⟨value2⟩ , ...
  }
```

Creates a new constant "flat" ⟨property list⟩ or raises an error if the name is already taken. The ⟨property list⟩ is set globally to contain key–value pairs given in the second argument, processed in the way described for \prop_set_from_keyval:Nn. If duplicate keys appear only the last of the values is kept. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

\prop_const_linked_from_keyval:Nn
\prop_const_linked_from_keyval:cn

New: 2024-02-12

```
\prop_const_linked_from_keyval:Nn ⟨property list⟩
  {
    ⟨key1⟩ = ⟨value1⟩ ,
    ⟨key2⟩ = ⟨value2⟩ , ...
  }
```

Creates a new constant "linked" ⟨property list⟩ or raises an error if the name is already taken. The ⟨property list⟩ is set globally to contain key–value pairs given in the second argument, processed in the way described for \prop_set_from_keyval:Nn. If duplicate keys appear only the last of the values is kept. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

\prop_make_flat:N
\prop_make_flat:c

New: 2024-02-12

\prop_make_flat:N ⟨property list⟩

Changes the internal storage type of the ⟨property list⟩ to be the same "flat" storage as \prop_new:N. The key–value pairs of the ⟨property list⟩ are preserved by the change. If the property list was already flat then nothing is done. This function can only be used at the outermost group level.

\prop_make_linked:N
\prop_make_linked:c

New: 2024-02-12

\prop_make_linked:N ⟨property list⟩

Changes the internal storage type of the ⟨property list⟩ to be the same "linked" storage as \prop_new_linked:N. The key–value pairs of the ⟨property list⟩ are preserved by the change. If the property list was already linked then nothing is done. This function can only be used at the outermost group level.

26.2 Adding and updating property list entries

\prop_put:Nnn
\prop_put:(NnV|Nnv|Nne|NVn|NVV|NVv|NVe|Nvn|NvV|Nvv|Nve|Nen|NeV|Nev|Nee|Nno|Non|Noo|cnn|cnV|cnv|cne|cVn|cVV|cVv|cVe|cvn|cvV|cvv|cve|cen|ceV|cev|cee|cno|con|coo)
\prop_gput:Nnn
\prop_gput:(NnV|Nnv|Nne|NVn|NVV|NVv|NVe|Nvn|NvV|Nvv|Nve|Nen|NeV|Nev|Nee|Nno|Non|Noo|cnn|cnV|cnv|cne|cVn|cVV|cVv|cVe|cvn|cvV|cvv|cve|cen|ceV|cev|cee|cno|con|coo)

\prop_put:Nnn ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

Adds an entry to the ⟨*property list*⟩ which may be accessed using the ⟨*key*⟩ and which has ⟨*value*⟩. If the ⟨*key*⟩ is already present in the ⟨*property list*⟩, the existing entry is overwritten by the new ⟨*value*⟩. Both the ⟨*key*⟩ and ⟨*value*⟩ may contain any ⟨*balanced text*⟩. The ⟨*key*⟩ is stored after processing with \tl_to_str:n, meaning that category codes are ignored.

\prop_put_if_not_in:Nnn
\prop_put_if_not_in:(NnV|Nnv|Nne|NVn|NVV|NVv|NVe|Nvn|NvV|Nvv|Nve|Nen|NeV|Nev|Nee|cnn|cnV|cnv|cne|cVn|cVV|cVv|cVe|cvn|cvV|cvv|cve|cen|ceV|cev|cee)
\prop_gput_if_not_in:Nnn
\prop_gput_if_not_in:(NnV|Nnv|Nne|NVn|NVV|NVv|NVe|Nvn|NvV|Nvv|Nve|Nen|NeV|Nev|Nee|cnn|cnV|cnv|cne|cVn|cVV|cVv|cVe|cvn|cvV|cvv|cve|cen|ceV|cev|cee)

New: 2024-03-30
Updated: 2024-05-07

\prop_put_if_not_in:Nnn ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

If the ⟨*key*⟩ is present in the ⟨*property list*⟩ then no action is taken. Otherwise, a new entry is added as described for \prop_put:Nnn.

\prop_concat:NNN
\prop_concat:ccc
\prop_gconcat:NNN
\prop_gconcat:ccc

New: 2021-05-16

\prop_concat:NNN ⟨*property list*$_1$⟩ ⟨*property list*$_2$⟩ ⟨*property list*$_3$⟩

Combines the key–value pairs of ⟨*property list*$_2$⟩ and ⟨*property list*$_3$⟩, and saves the result in ⟨*property list*$_1$⟩. If a key appears in both ⟨*property list*$_2$⟩ and ⟨*property list*$_3$⟩ then the last value, namely the value in ⟨*property list*$_3$⟩ is kept. This converts as needed between the two storage types.

\prop_put_from_keyval:Nn
\prop_put_from_keyval:cn
\prop_gput_from_keyval:Nn
\prop_gput_from_keyval:cn

New: 2021-05-16
Updated: 2021-11-07

```
\prop_put_from_keyval:Nn ⟨property list⟩
  {
    ⟨key1⟩ = ⟨value1⟩ ,
    ⟨key2⟩ = ⟨value2⟩ , ...
  }
```

Updates the ⟨*property list*⟩ by adding entries for each key–value pair given in the second argument. The addition is done through \prop_put:Nnn, hence if the ⟨*property list*⟩ already contains some of the keys, the corresponding values are discarded and replaced by those given in the key–value list. If duplicate keys appear in the key–value list then only the last of the values is kept.

The function is equivalent to storing the key–value pairs in a temporary property list using \prop_set_from_keyval:Nn, then combining ⟨*property list*⟩ with the temporary variable using \prop_concat:NNN. In particular, the ⟨*keys*⟩ and ⟨*values*⟩ are space-trimmed and unbraced as described in \prop_set_from_keyval:Nn. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

26.3 Recovering values from property lists

\prop_get:NnN
\prop_get:(NVN|NvN|NeN|NoN|cnN|cVN|cvN|ceN|coN|cnc)

\prop_get:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*tl var*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*tl var*⟩ is set to the special marker \q_no_value. The ⟨*tl var*⟩ is set within the current TeX group. See also \prop_get:NnNTF.

\prop_pop:NnN
\prop_pop:(NVN|NoN|cnN|cVN|coN)

\prop_pop:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*tl var*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*tl var*⟩ is set to the special marker \q_no_value. The ⟨*key*⟩ and ⟨*value*⟩ are then deleted from the property list. Both assignments are local. See also \prop_pop:NnNTF.

\prop_gpop:NnN
\prop_gpop:(NVN|NoN|cnN|cVN|coN)

\prop_gpop:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*tl var*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*tl var*⟩ is set to the special marker \q_no_value. The ⟨*key*⟩ and ⟨*value*⟩ are then deleted from the property list. The ⟨*property list*⟩ is modified globally, while the assignment of the ⟨*tl var*⟩ is local. See also \prop_gpop:NnNTF.

\prop_item:Nn ⋆
\prop_item:(NV|Ne|No|cn|cV|ce|co) ⋆

\prop_item:Nn ⟨property list⟩ {⟨key⟩}

Expands to the ⟨value⟩ corresponding to the ⟨key⟩ in the ⟨property list⟩. If the ⟨key⟩ is missing, this has an empty expansion.

TEXhackers note: For "flat" property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on \prop_get:NnN. (For "linked" property lists both functions are fast.)

The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨value⟩ does not expand further when appearing in an e-type or x-type argument expansion.

\prop_count:N ⋆
\prop_count:c ⋆

\prop_count:N ⟨property list⟩

Leaves the number of key–value pairs in the ⟨property list⟩ in the input stream as an ⟨integer denotation⟩.

\prop_to_keyval:N ⋆

\prop_to_keyval:N ⟨property list⟩

Expands to the ⟨property list⟩ in a key–value notation. Keep in mind that a ⟨property list⟩ is *unordered*, while key–value interfaces are not necessarily, so this cannot be used for arbitrary interfaces.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the key–value list does not expand further when appearing in an e-type or x-type argument expansion. It also needs exactly two steps of expansion.

26.4 Modifying property lists

\prop_remove:Nn
\prop_remove:(NV|Ne|cn|cV|ce)
\prop_gremove:Nn
\prop_gremove:(NV|Ne|cn|cV|ce)

\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.

26.5 Property list conditionals

\prop_if_exist_p:N ⋆
\prop_if_exist_p:c ⋆
\prop_if_exist:N*TF* ⋆
\prop_if_exist:c*TF* ⋆

\prop_if_exist_p:N ⟨property list⟩
\prop_if_exist:NTF ⟨property list⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨property list⟩ is currently defined. This does not check that the ⟨property list⟩ really is a property list variable.

\prop_if_empty_p:N ⋆
\prop_if_empty_p:c ⋆
\prop_if_empty:NTF ⋆
\prop_if_empty:cTF ⋆

\prop_if_empty_p:N ⟨*property list*⟩
\prop_if_empty:NTF ⟨*property list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*property list*⟩ is empty (containing no entries).

\prop_if_in_p:Nn ⋆
\prop_if_in_p:(NV|Ne|No|cn|cV|ce|co) ⋆
\prop_if_in:NnTF ⋆
\prop_if_in:(NV|Ne|No|cn|cV|ce|co)TF ⋆

\prop_if_in_p:Nn ⟨*property list*⟩ {⟨*key*⟩}
\prop_if_in:NnTF ⟨*property list*⟩ {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ is present in the ⟨*property list*⟩, making the comparison using the method described by \str_if_eq:nnTF.

TEXhackers note: For "flat" property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on \prop_-get:NnNTF. (For "linked" property lists both functions are fast.)

26.6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different code follows depending on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

\prop_get:NnNTF
\prop_get:(NVN|NvN|NeN|NoN|cnN|cVN|cvN|ceN|coN|cnc)TF

\prop_get:NnNTF ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩
{⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, stores the corresponding ⟨*value*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*property list*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*tl var*⟩ is assigned locally.

\prop_pop:NnNTF
\prop_pop:(NVN|NoN|cnN|cVN|coN)TF

\prop_pop:NnNTF ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩
{⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, pops the corresponding ⟨*value*⟩ in the ⟨*tl var*⟩, i.e., removes the item from the ⟨*property list*⟩. Both the ⟨*property list*⟩ and the ⟨*tl var*⟩ are assigned locally.

\prop_gpop:NnNTF
\prop_gpop:(NVN|NoN|cnN|cVN|coN)TF

\prop_gpop:NnNTF ⟨property list⟩ {⟨key⟩} ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨key⟩ is not present in the ⟨property list⟩, leaves the ⟨false code⟩ in the input stream. The value of the ⟨tl var⟩ is not defined in this case and should not be relied upon. If the ⟨key⟩ is present in the ⟨property list⟩, pops the corresponding ⟨value⟩ in the ⟨tl var⟩, i.e., removes the item from the ⟨property list⟩. The ⟨property list⟩ is modified globally, while the ⟨tl var⟩ is assigned locally.

26.7 Mapping over property lists

All mappings are done at the current group level, i.e., any local assignments made by the ⟨function⟩ or ⟨code⟩ discussed below remain in effect after the loop.

\prop_map_function:NN ☆
\prop_map_function:cN ☆

\prop_map_function:NN ⟨property list⟩ ⟨function⟩

Applies ⟨function⟩ to every ⟨entry⟩ stored in the ⟨property list⟩. The ⟨function⟩ receives two arguments for each iteration: the ⟨key⟩ and associated ⟨value⟩. The order in which ⟨entries⟩ are returned is not defined and should not be relied upon. To pass further arguments to the ⟨function⟩, see \prop_map_inline:Nn (non-expandable) or \prop_map_tokens:Nn.

\prop_map_inline:Nn
\prop_map_inline:cn

\prop_map_inline:Nn ⟨property list⟩ {⟨inline function⟩}

Applies ⟨inline function⟩ to every ⟨entry⟩ stored within the ⟨property list⟩. The ⟨inline function⟩ should consist of code which receives the ⟨key⟩ as #1 and the ⟨value⟩ as #2. The order in which ⟨entries⟩ are returned is not defined and should not be relied upon.

\prop_map_tokens:Nn ☆
\prop_map_tokens:cn ☆

\prop_map_tokens:Nn ⟨property list⟩ {⟨code⟩}

Analogue of \prop_map_function:NN which maps several tokens instead of a single function. The ⟨code⟩ receives each key–value pair in the ⟨property list⟩ as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

expands to the value corresponding to mykey: for each pair in \l_my_prop the function \str_if_eq:nnT receives mykey, the ⟨key⟩ and the ⟨value⟩ as its three arguments. For that specific task, \prop_item:Nn is faster.

\prop_map_break: ☆ \prop_map_break:

Used to terminate a \prop_map_... function before all entries in the ⟨*property list*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \prop_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\prop_map_break:n ☆ \prop_map_break:n {⟨*code*⟩}

Used to terminate a \prop_map_... function before all entries in the ⟨*property list*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \prop_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

26.8 Viewing property lists

\prop_show:N
\prop_show:c

Updated: 2021-04-29

\prop_show:N ⟨*property list*⟩

Displays the entries in the ⟨*property list*⟩ in the terminal, and specifies its storage type.

\prop_log:N
\prop_log:c

Updated: 2021-04-29

\prop_log:N ⟨*property list*⟩

Writes the entries in the ⟨*property list*⟩ in the log file, and specifies its storage type.

26.9 Scratch property lists

There is no need to include both flat and linked property lists as scratch variables. We arbitrarily pick the older implementation.

\l_tmpa_prop
\l_tmpb_prop

Scratch "flat" property lists for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_prop
\g_tmpb_prop

Scratch "flat" property lists for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

26.10 Constants

\c_empty_prop

A permanently-empty property list used for internal comparisons.

Chapter 27

The l3skip module Dimensions and skips

LaTeX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in `mu`). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

Many functions take *dimension expressions* ("⟨*dim expr*⟩") or *skip expressions* ("⟨*skip expr*⟩") as arguments.

27.1 Creating and initializing `dim` variables

`\dim_new:N`
`\dim_new:c`

`\dim_new:N ⟨dimension⟩`

Creates a new ⟨*dimension*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*dimension*⟩ is initially equal to 0 pt.

`\dim_const:Nn`
`\dim_const:cn`

`\dim_const:Nn ⟨dimension⟩ {⟨dim expr⟩}`

Creates a new constant ⟨*dimension*⟩ or raises an error if the name is already taken. The value of the ⟨*dimension*⟩ is set globally to the ⟨*dim expr*⟩.

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

`\dim_zero:N ⟨dimension⟩`

Sets ⟨*dimension*⟩ to 0 pt.

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`

`\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 ⋆

\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.

27.2 Setting dim variables

\dim_add:Nn
\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn

\dim_add:Nn ⟨*dimension*⟩ {⟨*dim expr*⟩}

Adds the result of the ⟨*dim expr*⟩ to the current content of the ⟨*dimension*⟩.

\dim_set:Nn
\dim_set:(cn|NV|cV)
\dim_gset:Nn
\dim_gset:(cn|NV|cV)

\dim_set:Nn ⟨*dimension*⟩ {⟨*dim expr*⟩}

Sets ⟨*dimension*⟩ to the value of ⟨*dim expr*⟩, which must evaluate to a length with units.

\dim_set_eq:NN
\dim_set_eq:(cN|Nc|cc)
\dim_gset_eq:NN
\dim_gset_eq:(cN|Nc|cc)

\dim_set_eq:NN ⟨*dimension*$_1$⟩ ⟨*dimension*$_2$⟩

Sets the content of ⟨*dimension*$_1$⟩ equal to that of ⟨*dimension*$_2$⟩.

\dim_sub:Nn
\dim_sub:cn
\dim_gsub:Nn
\dim_gsub:cn

\dim_sub:Nn ⟨*dimension*⟩ {⟨*dim expr*⟩}

Subtracts the result of the ⟨*dim expr*⟩ from the current content of the ⟨*dimension*⟩.

27.3 Utilities for dimension calculations

\dim_abs:n ⋆

\dim_abs:n {⟨*dim expr*⟩}

Converts the ⟨*dim expr*⟩ to its absolute value, leaving the result in the input stream as a ⟨*dimension denotation*⟩.

\dim_max:nn ⋆
\dim_min:nn ⋆

\dim_max:nn {⟨*dim expr*$_1$⟩} {⟨*dim expr*$_2$⟩}
\dim_min:nn {⟨*dim expr*$_1$⟩} {⟨*dim expr*$_2$⟩}

Evaluates the two ⟨*dim exprs*⟩ and leaves either the maximum or minimum value in the input stream as appropriate, as a ⟨*dimension denotation*⟩.

`\dim_ratio:nn` ☆ `\dim_ratio:nn {`⟨*dim expr*$_1$⟩`} {`⟨*dim expr*$_2$⟩`}`

Parses the two ⟨*dim exprs*⟩ and converts the ratio of the two to a form suitable for use inside a ⟨*dim expr*⟩. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
  { 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Ne \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays `327680/655360` on the terminal.

27.4 Dimension expression conditionals

`\dim_compare_p:nNn` ⋆
`\dim_compare:nNnTF` ⋆

`\dim_compare_p:nNn {`⟨*dim expr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*dim expr*$_2$⟩`}`
`\dim_compare:nNnTF`
`{`⟨*dim expr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*dim expr*$_2$⟩`}`
`{`⟨*true code*⟩`} {`⟨*false code*⟩`}`

This function first evaluates each of the ⟨*dim exprs*⟩ as described for `\dim_eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	`=`
Greater than	`>`
Less than	`<`

This function is less flexible than `\dim_compare:nTF` but around 5 times faster.

`\dim_compare_p:n` ⋆
`\dim_compare:nTF` ⋆

```
\dim_compare_p:n
  {
    ⟨dim expr1⟩ ⟨relation1⟩
    ...
    ⟨dim exprN⟩ ⟨relationN⟩
    ⟨dim exprN+1⟩
  }
\dim_compare:nTF
  {
    ⟨dim expr1⟩ ⟨relation1⟩
    ...
    ⟨dim exprN⟩ ⟨relationN⟩
    ⟨dim exprN+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*dim exprs*⟩ as described for `\dim_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*dim expr*$_1$⟩ and ⟨*dim expr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*dim expr*$_2$⟩ and ⟨*dim expr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*dim expr*$_N$⟩ and ⟨*dim expr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*dim expr*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*dim expr*⟩ is evaluated and no other comparison is performed. The ⟨*relations*⟩ can be any of the following:

Equal	`=` or `==`
Greater than or equal to	`>=`
Greater than	`>`
Less than or equal to	`<=`
Less than	`<`
Not equal	`!=`

This function is more flexible than `\dim_compare:nNnTF` but around 5 times slower.

`\dim_case:nn` ⋆
`\dim_case:nnTF` ⋆

```
\dim_case:nnTF {⟨test dim expr⟩}
  {
    {⟨dim expr case_1⟩} {⟨code case_1⟩}
    {⟨dim expr case_2⟩} {⟨code case_2⟩}
    ...
    {⟨dim expr case_n⟩} {⟨code case_n⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test dim expr*⟩ and compares this in turn to each of the ⟨*dim expr case*⟩s until a match is found. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
  { 2 \l_tmpa_dim }
  {
    { 5 pt }          { Small }
    { 4 pt + 6 pt } { Medium }
    { - 10 pt }     { Negative }
  }
  { No idea! }
```

leaves "`Medium`" in the input stream. Since evaluation of the test expressions stops at the first successful case, the order of possible matches should normally be that the most likely are earlier: this will reduce the average steps required to complete expansion.

27.5 Dimension expression loops

`\dim_do_until:nNnn` ☆

`\dim_do_until:nNnn {⟨dim expr_1⟩} ⟨relation⟩ {⟨dim expr_2⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dim exprs*⟩ as described for `\dim_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\dim_do_while:nNnn` ☆

`\dim_do_while:nNnn {⟨dim expr_1⟩} ⟨relation⟩ {⟨dim expr_2⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dim exprs*⟩ as described for `\dim_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

\dim_until_do:nNnn ☆ \dim_until_do:nNnn {⟨*dim expr*$_1$⟩} ⟨*relation*⟩ {⟨*dim expr*$_2$⟩} {⟨*code*⟩}

Evaluates the relationship between the two ⟨*dim exprs*⟩ as described for \dim_compare:nNnTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is false. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is true.

\dim_while_do:nNnn ☆ \dim_while_do:nNnn {⟨*dim expr*$_1$⟩} ⟨*relation*⟩ {⟨*dim expr*$_2$⟩} {⟨*code*⟩}

Evaluates the relationship between the two ⟨*dim exprs*⟩ as described for \dim_compare:nNnTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is true. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is false.

\dim_do_until:nn ☆ \dim_do_until:nn {⟨*dimension relation*⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the ⟨*dimension relation*⟩ as described for \dim_compare:nTF. If the test is false then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is true.

\dim_do_while:nn ☆ \dim_do_while:nn {⟨*dimension relation*⟩} {⟨*code*⟩}

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the ⟨*dimension relation*⟩ as described for \dim_compare:nTF. If the test is true then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is false.

\dim_until_do:nn ☆ \dim_until_do:nn {⟨*dimension relation*⟩} {⟨*code*⟩}

Evaluates the ⟨*dimension relation*⟩ as described for \dim_compare:nTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is false. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is true.

\dim_while_do:nn ☆ \dim_while_do:nn {⟨*dimension relation*⟩} {⟨*code*⟩}

Evaluates the ⟨*dimension relation*⟩ as described for \dim_compare:nTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is true. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is false.

27.6 Dimension step functions

\dim_step_function:nnnN ☆ \dim_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 dimension expressions. The ⟨*function*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one argument.

\dim_step_inline:nnnn

`\dim_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}`

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be dimension expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with #1 replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (#1).

\dim_step_variable:nnnNn

```
\dim_step_variable:nnnNn
  {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}
```

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be dimension expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

27.7 Using dim expressions and variables

\dim_eval:n ⋆

`\dim_eval:n {⟨dim expr⟩}`

Evaluates the ⟨*dim expr*⟩, expanding any dimensions and token list variables within the ⟨*expression*⟩ to their content (without requiring `\dim_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*dimension denotation*⟩ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal dimension*⟩.

\dim_sign:n ⋆

`\dim_sign:n {⟨dim expr⟩}`

Evaluates the ⟨*dim expr*⟩ then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

\dim_use:N ⋆
\dim_use:c ⋆

`\dim_use:N ⟨dimension⟩`

Recovers the content of a ⟨*dimension*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of `\dim_eval:n`).

TeXhackers note: `\dim_use:N` is the TeX primitive `\the`: this is one of several LaTeX3 names for this primitive.

\dim_to_decimal:n ⋆

`\dim_to_decimal:n {⟨dim expr⟩}`

Evaluates the ⟨*dim expr*⟩, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by TeX to at most five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal:n { 1bp }
```

leaves `1.00374` in the input stream, i.e., the magnitude of one "big point" when converted to (TeX) points.

`\dim_to_decimal_in_bp:n` ⋆ `\dim_to_decimal_in_bp:n {⟨dim expr⟩}`

Updated: 2023-05-20

Evaluates the ⟨*dim expr*⟩, and leaves the result, expressed in big points (`bp`) in the input stream, with *no units*. The result is rounded by TEX to at most five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves `0.99628` in the input stream, i.e., the magnitude of one (TEX) point when converted to big points.

TEXhackers note: The implementation of this function is re-entrant: the result of

```
\dim_compare:nNnTF
  { <x>bp } =
    { \dim_to_decimal_in_bp:n { <x>bp } bp }
```

will be logically `true`. The decimal representations may differ provided they produce the same TEX dimension.

`\dim_to_decimal_in_cc:n` ⋆
`\dim_to_decimal_in_cm:n` ⋆
`\dim_to_decimal_in_dd:n` ⋆
`\dim_to_decimal_in_in:n` ⋆
`\dim_to_decimal_in_mm:n` ⋆
`\dim_to_decimal_in_pc:n` ⋆

New: 2023-05-20

`\dim_to_decimal_in_cm:n {⟨dim expr⟩}`

Evaluates the ⟨*dim expr*⟩, and leaves the result, expressed with the appropriate scaling in the input stream, with *no units*. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

The maximum TEX allowable dimension value (available as `\maxdimen` in plain TEX and LATEX and `\c_max_dim` in expl3) can only be expressed exactly in the units `pt`, `bp` and `sp`. The maximum allowable input values to five decimal places are

$$\begin{aligned} 1276.00215\,\mathrm{cc} \\ 575.83174\,\mathrm{cm} \\ 15312.02584\,\mathrm{dd} \\ 226.70540\,\mathrm{in} \\ 5758.31742\,\mathrm{mm} \\ 1365.33333\,\mathrm{pc} \end{aligned}$$

(Note that these are not all equal, but rather any larger value will overflow due to the way TEX converts to `sp`.) Values given to five decimal places larger that these will result in TEX errors; the behavior if additional decimal places are given depends on the TEX internals and thus larger values are *not* supported by expl3.

TEXhackers note: The implementation of these functions is re-entrant: the result of

```
\dim_compare:nNnTF
  { <x><unit> } =
    { \dim_to_decimal_in_<unit>:n { <x><unit> } <unit> }
```

will be logically `true`. The decimal representations may differ provided they produce the same TEX dimension.

\dim_to_decimal_in_sp:n ⋆ \dim_to_decimal_in_sp:n {⟨*dim expr*⟩}

Evaluates the ⟨*dim expr*⟩, and leaves the result, expressed in scaled points (sp) in the input stream, with *no units*. The result is necessarily an integer.

\dim_to_decimal_in_unit:nn ⋆ \dim_to_decimal_in_unit:nn {⟨*dim expr*$_1$⟩} {⟨*dim expr*$_2$⟩}

Updated: 2023-05-20

Evaluates the ⟨*dim exprs*⟩, and leaves the value of ⟨*dim expr*$_1$⟩, expressed in a unit given by ⟨*dim expr*$_2$⟩, in the input stream. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves 0.35278 in the input stream, i.e., the magnitude of one big point when expressed in millimetres. The conversions do *not* guarantee that TeX would yield identical results for the direct input in an equality test, thus for instance

```
\dim_compare:nNnTF
  { 1bp } =
    { \dim_to_decimal_in_unit:nn { 1bp } { 1mm } mm }
```

will take the false branch.

\dim_to_fp:n ⋆ \dim_to_fp:n {⟨*dim expr*⟩}

Expands to an internal floating point number equal to the value of the ⟨*dim expr*⟩ in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, \dim_to_fp:n can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

27.8 Viewing dim variables

\dim_show:N
\dim_show:c
\dim_show:N ⟨*dimension*⟩

Displays the value of the ⟨*dimension*⟩ on the terminal.

\dim_show:n \dim_show:n {⟨*dim expr*⟩}

Displays the result of evaluating the ⟨*dim expr*⟩ on the terminal.

\dim_log:N
\dim_log:c
\dim_log:N ⟨*dimension*⟩

Writes the value of the ⟨*dimension*⟩ in the log file.

\dim_log:n \dim_log:n {⟨*dim expr*⟩}

Writes the result of evaluating the ⟨*dim expr*⟩ in the log file.

27.9 Constant dimensions

\c_max_dim The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

\c_zero_dim A zero length as a dimension. This can also be used as a component of a skip.

27.10 Scratch dimensions

\l_tmpa_dim
\l_tmpb_dim
Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_dim
\g_tmpb_dim
Scratch dimension for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

27.11 Creating and initializing skip variables

\skip_new:N
\skip_new:c

`\skip_new:N ⟨skip⟩`

Creates a new ⟨*skip*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*skip*⟩ is initially equal to 0 pt.

\skip_const:Nn
\skip_const:cn

`\skip_const:Nn ⟨skip⟩ {⟨skip expr⟩}`

Creates a new constant ⟨*skip*⟩ or raises an error if the name is already taken. The value of the ⟨*skip*⟩ is set globally to the ⟨*skip expr*⟩.

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

`\skip_zero:N ⟨skip⟩`

Sets ⟨*skip*⟩ to 0 pt.

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

`\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 ⋆

`\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.

27.12 Setting skip variables

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn

\skip_add:Nn ⟨*skip*⟩ {⟨*skip expr*⟩}

Adds the result of the ⟨*skip expr*⟩ to the current content of the ⟨*skip*⟩.

\skip_set:Nn
\skip_set:(cn|NV|cV)
\skip_gset:Nn
\skip_gset:(cn|NV|cV)

\skip_set:Nn ⟨*skip*⟩ {⟨*skip expr*⟩}

Sets ⟨*skip*⟩ to the value of ⟨*skip expr*⟩, which must evaluate to a length with units and may include a rubber component (for example `1 cm plus 0.5 cm`).

\skip_set_eq:NN
\skip_set_eq:(cN|Nc|cc)
\skip_gset_eq:NN
\skip_gset_eq:(cN|Nc|cc)

\skip_set_eq:NN ⟨*skip*$_1$⟩ ⟨*skip*$_2$⟩

Sets the content of ⟨*skip*$_1$⟩ equal to that of ⟨*skip*$_2$⟩.

\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

\skip_sub:Nn ⟨*skip*⟩ {⟨*skip expr*⟩}

Subtracts the result of the ⟨*skip expr*⟩ from the current content of the ⟨*skip*⟩.

27.13 Skip expression conditionals

\skip_if_eq_p:nn ⋆
\skip_if_eq:nn*TF* ⋆

\skip_if_eq_p:nn {⟨*skip expr*$_1$⟩} {⟨*skip expr*$_2$⟩}
\skip_if_eq:nnTF
  {⟨*skip expr*$_1$⟩} {⟨*skip expr*$_2$⟩}
  {⟨*true code*⟩} {⟨*false code*⟩}

This function first evaluates each of the ⟨*skip exprs*⟩ as described for `\skip_eval:n`. The two results are then compared for exact equality, i.e., both the fixed and rubber components must be the same for the test to be true.

\skip_if_finite_p:n ⋆
\skip_if_finite:n*TF* ⋆

\skip_if_finite_p:n {⟨*skip expr*⟩}
\skip_if_finite:nTF {⟨*skip expr*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Evaluates the ⟨*skip expr*⟩ as described for `\skip_eval:n`, and then tests if all of its components are finite.

27.14 Using skip expressions and variables

\skip_eval:n ⋆

\skip_eval:n {⟨*skip expr*⟩}

Evaluates the ⟨*skip expr*⟩, expanding any skips and token list variables within the ⟨*expression*⟩ to their content (without requiring `\skip_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*glue denotation*⟩ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal glue*⟩.

\skip_use:N ⋆
\skip_use:c ⋆

\skip_use:N ⟨*skip*⟩

Recovers the content of a ⟨*skip*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ or ⟨*skip*⟩ is required (such as in the argument of \skip_eval:n).

TEXhackers note: \skip_use:N is the TEX primitive \the: this is one of several LATEX3 names for this primitive.

27.15 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

\skip_show:n {⟨*skip expr*⟩}

Displays the result of evaluating the ⟨*skip expr*⟩ on the terminal.

\skip_log:N
\skip_log:c

\skip_log:N ⟨*skip*⟩

Writes the value of the ⟨*skip*⟩ in the log file.

\skip_log:n

\skip_log:n {⟨*skip expr*⟩}

Writes the result of evaluating the ⟨*skip expr*⟩ in the log file.

27.16 Constant skips

\c_max_skip

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

A zero length as a skip, with no stretch nor shrink component.

27.17 Scratch skips

\l_tmpa_skip
\l_tmpb_skip

Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_skip
\g_tmpb_skip

Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

27.18 Inserting skips into the output

\skip_horizontal:N
\skip_horizontal:c
\skip_horizontal:n

\skip_horizontal:N ⟨*skip*⟩
\skip_horizontal:n {⟨*skip expr*⟩}

Inserts a horizontal ⟨*skip*⟩ into the current list. The argument can also be a ⟨*dim*⟩.

TeXhackers note: \skip_horizontal:N is the TeX primitive \hskip.

\skip_vertical:N
\skip_vertical:c
\skip_vertical:n

\skip_vertical:N ⟨*skip*⟩
\skip_vertical:n {⟨*skip expr*⟩}

Inserts a vertical ⟨*skip*⟩ into the current list. The argument can also be a ⟨*dim*⟩.

TeXhackers note: \skip_vertical:N is the TeX primitive \vskip.

27.19 Creating and initializing muskip variables

\muskip_new:N
\muskip_new:c

\muskip_new:N ⟨*muskip*⟩

Creates a new ⟨*muskip*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*muskip*⟩ is initially equal to 0 mu.

\muskip_const:Nn
\muskip_const:cn

\muskip_const:Nn ⟨*muskip*⟩ {⟨*muskip expr*⟩}

Creates a new constant ⟨*muskip*⟩ or raises an error if the name is already taken. The value of the ⟨*muskip*⟩ is set globally to the ⟨*muskip expr*⟩.

\muskip_zero:N
\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c

\skip_zero:N ⟨*muskip*⟩

Sets ⟨*muskip*⟩ to 0 mu.

\muskip_zero_new:N
\muskip_zero_new:c
\muskip_gzero_new:N
\muskip_gzero_new:c

\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* ⋆

\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.

27.20 Setting muskip variables

\muskip_add:Nn
\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn

\muskip_add:Nn ⟨*muskip*⟩ {⟨*muskip expr*⟩}

Adds the result of the ⟨*muskip expr*⟩ to the current content of the ⟨*muskip*⟩.

\muskip_set:Nn
\muskip_set:(cn|NV|cV)
\muskip_gset:Nn
\muskip_gset:(cn|NV|cV)

\muskip_set:Nn ⟨*muskip*⟩ {⟨*muskip expr*⟩}

Sets ⟨*muskip*⟩ to the value of ⟨*muskip expr*⟩, which must evaluate to a math length with units and may include a rubber component (for example `1 mu plus 0.5 mu`.

\muskip_set_eq:NN
\muskip_set_eq:(cN|Nc|cc)
\muskip_gset_eq:NN
\muskip_gset_eq:(cN|Nc|cc)

\muskip_set_eq:NN ⟨*muskip*$_1$⟩ ⟨*muskip*$_2$⟩

Sets the content of ⟨*muskip*$_1$⟩ equal to that of ⟨*muskip*$_2$⟩.

\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

\muskip_sub:Nn ⟨*muskip*⟩ {⟨*muskip expr*⟩}

Subtracts the result of the ⟨*muskip expr*⟩ from the current content of the ⟨*muskip*⟩.

27.21 Using muskip expressions and variables

\muskip_eval:n ⋆

\muskip_eval:n {⟨*muskip expr*⟩}

Evaluates the ⟨*muskip expr*⟩, expanding any skips and token list variables within the ⟨*expression*⟩ to their content (without requiring `\muskip_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*muglue denotation*⟩ after two expansions. This is expressed in `mu`, and requires suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal muglue*⟩.

\muskip_use:N ⋆
\muskip_use:c ⋆

\muskip_use:N ⟨*muskip*⟩

Recovers the content of a ⟨*skip*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of `\muskip_eval:n`).

TeXhackers note: `\muskip_use:N` is the TeX primitive `\the`: this is one of several LaTeX3 names for this primitive.

27.22 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

\muskip_show:n {⟨*muskip expr*⟩}

Displays the result of evaluating the ⟨*muskip expr*⟩ on the terminal.

\muskip_log:N
\muskip_log:c

\muskip_log:N ⟨*muskip*⟩

Writes the value of the ⟨*muskip*⟩ in the log file.

\muskip_log:n

\muskip_log:n {⟨*muskip expr*⟩}

Writes the result of evaluating the ⟨*muskip expr*⟩ in the log file.

27.23 Constant muskips

\c_max_muskip

The maximum value that can be stored as a muskip, with no stretch nor shrink component.

\c_zero_muskip

A zero length as a muskip, with no stretch nor shrink component.

27.24 Scratch muskips

\l_tmpa_muskip
\l_tmpb_muskip

Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_muskip
\g_tmpb_muskip

Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

27.25 Primitive conditional

`\if_dim:w` ⋆

```
\if_dim:w ⟨dimen1⟩ ⟨relation⟩ ⟨dimen2⟩
  ⟨true code⟩
\else:
  ⟨false⟩
\fi:
```

Compare two dimensions. The ⟨*relation*⟩ is one of `<`, `=` or `>` with category code 12.

> **TEXhackers note:** This is the TEX primitive `\ifdim`.

Chapter 28

The l3keys module
Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behavior. 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, minimizing the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
  {
    key-one .code:n   = code including parameter #1,
    key-two .tl_set:N = \l_mymodule_store_tl
  }
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
  {
    key-one = value one,
    key-two = value two
  }
```

As illustrated, keys are created inside a ⟨*module*⟩: a set of related keys, typically those for a single module/LaTeX 2ε package. See Section 28.2 for suggestions on how to divide large numbers of keys for a single module.

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
  { \keys_set:nn { mymodule } { #1 }  }
\DeclareDocumentCommand \MyModuleMacro { o m }
  {
    \group_begin:
      \keys_set:nn { mymodule } { #1 }
      % Main code for \MyModuleMacro
    \group_end:
  }
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As discussed in section 28.2, it is suggested that the character `/` is reserved for sub-division of keys into different subsets. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
  {
    \l_mymodule_tmp_tl .code:n = code
  }
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

28.1 Creating keys

`\keys_define:nn`
`\keys_define:ne`

`\keys_define:nn {⟨module⟩} {⟨keyval list⟩}`

Parses the ⟨*keyval list*⟩ and defines the keys listed there for ⟨*module*⟩. The ⟨*module*⟩ name is treated as a string. In practice the ⟨*module*⟩ should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The ⟨*keyval list*⟩ should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some~code~using~#1,
    keyname .value_required:n = true
  }
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary ⟨*key*⟩, which when used may be supplied with a ⟨*value*⟩. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define "actions", such as `.code:n`, `.tl_set:N`, etc., override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviors may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
  {
    keyname .code:n            = Some~code~using~#1,
    keyname .value_required:n = true
  }
\keys_define:nn { mymodule }
  {
    keyname .value_required:n = true,
    keyname .code:n            = Some~code~using~#1
  }
```

Note that all key properties define the key within the current TeX group, with an exception that the special `.undefine:` property *undefines* the key within the current TeX group.

`.bool_set:N`
`.bool_set:c`
`.bool_gset:N`
`.bool_gset:c`

⟨*key*⟩ `.bool_set:N =` ⟨*boolean*⟩

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to ⟨*value*⟩. If the ⟨*value*⟩ is given, it must be one either "`true`" or "`false`"); it may be omitted, which is equivalent to `true`. 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`

⟨*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 28.3.

`.choices:nn`
`.choices:(Vn|en|on)`

⟨*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 28.3.

`.clist_set:N`
`.clist_set:c`
`.clist_gset:N`
`.clist_gset:c`

⟨*key*⟩ `.clist_set:N =` ⟨*comma list variable*⟩

Defines ⟨*key*⟩ to set ⟨*comma list variable*⟩ to ⟨*value*⟩. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

.code:n ⟨*key*⟩ .code:n = {⟨*code*⟩}

Stores the ⟨*code*⟩ for execution when ⟨*key*⟩ is used. The ⟨*code*⟩ can include one parameter (#1), which will be the ⟨*value*⟩ given for the ⟨*key*⟩.

.cs_set:Np
.cs_set:cp
.cs_set_protected:Np
.cs_set_protected:cp
.cs_gset:Np
.cs_gset:cp
.cs_gset_protected:Np
.cs_gset_protected:cp

⟨*key*⟩ .cs_set:Np = ⟨*control sequence*⟩ ⟨*arg. spec.*⟩

Defines ⟨*key*⟩ to set ⟨*control sequence*⟩ to have ⟨*arg. spec.*⟩ and replacement text ⟨*value*⟩.

.default:n
.default:(V|e|o)

⟨*key*⟩ .default:n = {⟨*default*⟩}

Creates a ⟨*default*⟩ value for ⟨*key*⟩, which is used if no value is given. This will be used if only the key name is given, but not if a blank ⟨*value*⟩ is given:

```
\keys_define:nn { mymodule }
  {
    key .code:n    = Hello~#1,
    key .default:n = World
  }
\keys_set:nn { mymodule }
  {
    key = Fred, % Prints 'Hello Fred'
    key,        % Prints 'Hello World'
    key = ,     % Prints 'Hello '
  }
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

When no value is given for a key as part of \keys_set:nn, the .default:n value provides the value before key properties are considered. The only exception is when the .value_required:n property is active: a required value cannot be supplied by the default, and must be explicitly given as part of \keys_set:nn.

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c

⟨*key*⟩ .dim_set:N = ⟨*dimension*⟩

Defines ⟨*key*⟩ to set ⟨*dimension*⟩ to ⟨*value*⟩ (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c

⟨*key*⟩ .fp_set:N = ⟨*fp var*⟩

Defines ⟨*key*⟩ to set ⟨*fp var*⟩ to ⟨*value*⟩ (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

`.groups:n` ⟨*key*⟩ `.groups:n = {`⟨*groups*⟩`}`

Defines ⟨*key*⟩ as belonging to the ⟨*groups*⟩ (a comma-separated list). Groups provide a "secondary axis" for selectively setting keys, and are described in Section 28.7.

TeXhackers note: The ⟨*groups*⟩ argument is turned into a string then interpreted as a comma-separated list, so group names cannot contain commas nor start or end with a space character.

`.inherit:n` ⟨*key*⟩ `.inherit:n = {`⟨*parents*⟩`}`

Specifies that the ⟨*key*⟩ path should inherit the keys listed as any of the ⟨*parents*⟩ (a comma list), which can be a module or a sub-division thereof. For example, after setting

```
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
```

setting

```
\keys_set:nn { bar } { test = a }
```

will be equivalent to

```
\keys_set:nn { foo } { test = a }
```

Inheritance applies at point of use, not at definition, thus keys may be added to the ⟨*parent*⟩ after the use of `.inherit:n` and will be active. If more than one ⟨*parent*⟩ is specified, the presence of the ⟨*key*⟩ will be tested for each in turn, with the first successful hit taking priority.

`.initial:n`
`.initial:(V|e|o)`

⟨*key*⟩ `.initial:n = {`⟨*value*⟩`}`

Initializes 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 is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

`.legacy_if_set:n`
`.legacy_if_gset:n`
`.legacy_if_set_inverse:n`
`.legacy_if_gset_inverse:n`

Updated: 2022-01-15

⟨*key*⟩ `.legacy_if_set:n =` ⟨*switch*⟩

Defines ⟨*key*⟩ to set legacy `\if`⟨*switch*⟩ to ⟨*value*⟩ (which must be either "`true`" or "`false`"). The ⟨*switch*⟩ is the name of the switch *without the leading* `if`.

The `inverse` versions will set the ⟨*switch*⟩ to the logical opposite of the ⟨*value*⟩.

`.meta:n` ⟨*key*⟩ `.meta:n = {`⟨*keyval list*⟩`}`

Makes ⟨*key*⟩ a meta-key, which will set ⟨*keyval list*⟩ in one go. The ⟨*keyval list*⟩ can refer as `#1` to the value given at the time the ⟨*key*⟩ is used (or, if no value is given, the ⟨*key*⟩'s default value).

.meta:nn ⟨*key*⟩ .meta:nn = {⟨*path*⟩} {⟨*keyval list*⟩}

Makes ⟨*key*⟩ a meta-key, which will set ⟨*keyval list*⟩ in one go using the ⟨*path*⟩ in place of the current one. The ⟨*keyval list*⟩ can refer as #1 to the value given at the time the ⟨*key*⟩ is used (or, if no value is given, the ⟨*key*⟩'s default value).

.multichoice: ⟨*key*⟩ .multichoice:

Sets ⟨*key*⟩ to act as a multiple choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 28.3.

.multichoices:nn
.multichoices:(Vn|en|on)

⟨*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 28.3.

.muskip_set:N
.muskip_set:c
.muskip_gset:N
.muskip_gset:c

⟨*key*⟩ .muskip_set:N = ⟨*muskip*⟩

Defines ⟨*key*⟩ to set ⟨*muskip*⟩ to ⟨*value*⟩ (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.prop_put:N
.prop_put:c
.prop_gput:N
.prop_gput:c

⟨*key*⟩ .prop_put:N = ⟨*property list*⟩

Defines ⟨*key*⟩ to put the ⟨*value*⟩ onto the ⟨*property list*⟩ stored under the ⟨*key*⟩. If the variable does not exist, it is created globally at the point that the key is set up.

.skip_set:N
.skip_set:c
.skip_gset:N
.skip_gset:c

⟨*key*⟩ .skip_set:N = ⟨*skip*⟩

Defines ⟨*key*⟩ to set ⟨*skip*⟩ to ⟨*value*⟩ (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.str_set:N
.str_set:c
.str_gset:N
.str_gset:c
New: 2021-10-30

⟨*key*⟩ .str_set:N = ⟨*string variable*⟩

Defines ⟨*key*⟩ to set ⟨*string variable*⟩ to ⟨*value*⟩. If the variable does not exist, it is created globally at the point that the key is set up.

.str_set_e:N
.str_set_e:c
.str_gset_e:N
.str_gset_e:c
New: 2023-09-18

⟨*key*⟩ .str_set_e:N = ⟨*string variable*⟩

Defines ⟨*key*⟩ to set ⟨*string variable*⟩ to ⟨*value*⟩, which will be subjected to an e-type expansion (i.e., using \str_set:Ne). If the variable does not exist, it is created globally at the point that the key is set up.

.tl_set:N
.tl_set:c
.tl_gset:N
.tl_gset:c

⟨*key*⟩ .tl_set:N = ⟨*tl var*⟩

Defines ⟨*key*⟩ to set ⟨*tl var*⟩ to ⟨*value*⟩. If the variable does not exist, it is created globally at the point that the key is set up.

.tl_set_e:N
.tl_set_e:c
.tl_gset_e:N
.tl_gset_e:c

New: 2023-09-18

⟨*key*⟩ .tl_set_e:N = ⟨*tl var*⟩

Defines ⟨*key*⟩ to set ⟨*tl var*⟩ to ⟨*value*⟩, which will be subjected to an e-type expansion (i.e., using \tl_set:Ne). If the variable does not exist, it is created globally at the point that the key is set up.

.undefine:

⟨*key*⟩ .undefine:

Removes the definition of the ⟨*key*⟩ within the current TeX group.

.value_forbidden:n

⟨*key*⟩ .value_forbidden:n = true|false

Specifies that ⟨*key*⟩ cannot receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is given then an error will be issued. Setting the property "false" cancels the restriction.

.value_required:n

⟨*key*⟩ .value_required:n = true|false

Specifies that ⟨*key*⟩ must receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is not given then an error will be issued. Setting the property "false" cancels the restriction.

28.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several subsets for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { mymodule / subset }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subset / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is /. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name mymodule/subset/key.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

28.3 Choice and multiple choice keys

The l3keys system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. "Multiple" choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the .choice: property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependent only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
  {
    key .choices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e., the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined behavior 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 28.6. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
```

```
      key / unknown  .code:n =
        \msg_error:nneee { mymodule } { unknown-choice }
          { key }                              % Name of choice key
          { choice-a , choice-b ,  choice-c }  % Valid choices
          { \exp_not:n {#1} }                  % Invalid choice given
    }
```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are defined as sub-keys. Thus both

```
  \keys_define:nn { mymodule }
    {
      key .multichoices:nn =
        { choice-a, choice-b, choice-c }
        {
          You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
          which~is~in~position~
          \int_use:N \l_keys_choice_int \c_space_tl
          in~the~list.
        }
    }
```

and

```
  \keys_define:nn { mymodule }
    {
      key .multichoice:,
      key / choice-a .code:n = code-a,
      key / choice-b .code:n = code-b,
      key / choice-c .code:n = code-c,
    }
```

are valid.

When a multiple choice key is set

```
  \keys_set:nn { mymodule }
    {
      key = { a , b , c } % 'key' defined as a multiple choice
    }
```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```
  \keys_set:nn { mymodule }
    {
      key = a ,
      key = b ,
      key = c ,
    }
```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

28.4 Key usage scope

Some keys will be used as settings which have a strictly limited scope of usage. Some will be only available once, others will only be valid until typesetting begins. To allow formats to support this in a structured way, l3keys allows this information to be specified using the `.usage:n` property.

`.usage:n`

New: 2022-01-10

⟨*key*⟩ `.usage:n =` ⟨*scope*⟩

Defines the ⟨*key*⟩ to have usage within the ⟨*scope*⟩, which should be one of `general`, `preamble` or `load`.

`\l_keys_usage_load_prop`
`\l_keys_usage_preamble_prop`

New: 2022-01-10

l3keys itself does *not* attempt to redefine keys based on the usage scope. Rather, this information is made available with these two property lists. These hold an entry for each module (prefix); the value of each entry is a comma-separated list of the usage-restricted key(s).

28.5 Setting keys

`\keys_set:nn`
`\keys_set:(nV|nv|ne|no)`

`\keys_set:nn {`⟨*module*⟩`} {`⟨*keyval list*⟩`}`

Parses the ⟨*keyval list*⟩, and sets those keys which are defined for ⟨*module*⟩. The behavior on finding an unknown key can be set by defining a special `unknown` key: this is illustrated later.

`\l_keys_path_str`
`\l_keys_key_str`
`\l_keys_value_tl`

Updated: 2020-02-08

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within two string and one token list variables. These may be used within the code of the key.

The *path* of the key is a "full" description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset  / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_str`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_str`.

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

28.6 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts. The `unknown` key also supports the `.default:n` property.

```
\keys_define:nn { mymodule }
  {
    unknown .code:n =
      You~tried~to~set~key~'\l_keys_key_str'~to~'#1'. ,
    unknown .default:V = \c_novalue_tl
  }
```

Key inheritance does *not* include looking for the special `unknown` key. Handling of `unknown` keys should be set up explicitly for each path where it applies.

28.7 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys_define:nn { mymodule }
  {
    key-one   .code:n   = { \my_func:n {#1} } ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
  }
```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys_define:nn { mymodule }
  {
    key-one   .code:n   = { \my_func:n {#1} } ,
    key-one   .groups:n = { first }           ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-two   .groups:n = { first }           ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-three .groups:n = { second }          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
  }
```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made "active", or by marking one or more groups to be ignored in key setting.

\keys_set_known:nn
\keys_set_known:(nV|nv|ne|no)
\keys_set_known:nnN
\keys_set_known:(nVN|nvN|neN|noN)
\keys_set_known:nnnN
\keys_set_known:(nVnN|nvnN|nenN|nonN)

\keys_set_known:nn {⟨module⟩} {⟨keyval list⟩}
\keys_set_known:nnN {⟨module⟩} {⟨keyval list⟩} ⟨tl var⟩
\keys_set_known:nnnN {⟨module⟩} {⟨keyval list⟩} {⟨root⟩} ⟨tl var⟩

These functions set keys which are known for the ⟨module⟩, and simply ignore other keys. The \keys_set_known:nn function parses the ⟨keyval list⟩, and sets those keys which are defined for ⟨module⟩. Any keys which are unknown are not processed further by the parser.

In addition, \keys_set_known:nnN and \keys_set_known:nnnN store the key–value pairs for unknown keys in the ⟨tl var⟩ in comma-separated form (i.e., an edited version of the ⟨keyval list⟩). When a ⟨root⟩ is given (\keys_set_known:nnnN), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

\keys_set_groups:nnn
\keys_set_groups:(nnV|nnv|nno)
\keys_set_groups:nnnN
\keys_set_groups:(nnVN|nnvN|nnoN)
\keys_set_groups:nnnnN
\keys_set_groups:(nnVnN|nnvnN|nnonN)

Updated: 2024-05-08

\keys_set_groups:nnn {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩}
\keys_set_groups:nnnN {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩} ⟨tl var⟩
\keys_set_groups:nnnnN {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩} {⟨root⟩} ⟨tl var⟩

These functions activate key selection in an "opt-in" sense: only keys assigned to one or more of the ⟨groups⟩ specified are set. The ⟨groups⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

In addition, \keys_set_groups:nnnN and \keys_set_groups:nnnnN store the key–value pairs for skipped keys in the ⟨tl var⟩ in comma-separated form (i.e., an edited version of the ⟨keyval list⟩). When a ⟨root⟩ is given (\keys_set_groups:nnnnN), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

\keys_set_exclude_groups:nnn
\keys_set_exclude_groups:(nnV|nnv|nno)
\keys_set_exclude_groups:nnnN
\keys_set_exclude_groups:(nnVN|nnvN|nnoN)
\keys_set_exclude_groups:nnnnN
\keys_set_exclude_groups:(nnVnN|nnvnN|nnonN)

New: 2024-01-10

\keys_set_exclude_groups:nnn {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩}
\keys_set_exclude_groups:nnnN {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩} ⟨tl var⟩
\keys_set_exclude_groups:nnnnN {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩} {⟨root⟩} ⟨tl var⟩

These functions activate key selection in an "opt-out" sense: keys assigned to one or more of the ⟨groups⟩ specified are *not* set. The ⟨groups⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set.

In addition, \keys_set_exclude_groups:nnnN and \keys_set_exclude_groups:nnnnN store the key–value pairs for skipped keys in the ⟨tl var⟩ in comma-separated form (i.e., an edited version of the ⟨keyval list⟩). When a ⟨root⟩ is given (\keys_set_exclude_groups:nnnnN), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

28.8 Precompiling keys

\keys_precompile:nnN

New: 2022-03-09

\keys_precompile:nnN {⟨*module*⟩} {⟨*keyval list*⟩} ⟨*tl var*⟩

Parses the ⟨*keyval list*⟩ as for \keys_set:nn, placing the resulting code for those which set variables or functions into the ⟨*tl var*⟩. Thus this function "precompiles" the keyval list into a set of results which can be applied rapidly.

It is important to note that when precompiling keys, no expansion of variables takes place. This means that any key setting which simply stores variable names, rather than variable values, may not work correctly. Most notably, any key setting which uses key status variables (\l_keys_key_str, etc.) will yield unpredictable outcomes. As such, keys intended to be precompiled should fully expand any values at the point of setting.

28.9 Utility functions for keys

\keys_if_exist_p:nn ⋆
\keys_if_exist_p:ne ⋆
\keys_if_exist:nn*TF* ⋆
\keys_if_exist:ne*TF* ⋆

Updated: 2022-01-10

\keys_if_exist_p:nn {⟨*module*⟩} {⟨*key*⟩}
\keys_if_exist:nnTF {⟨*module*⟩} {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ exists for ⟨*module*⟩, i.e., if any code has been defined for ⟨*key*⟩.

\keys_if_choice_exist_p:nnn ⋆
\keys_if_choice_exist:nnn*TF* ⋆

\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*⟩}

Displays in the terminal the information associated to the ⟨*key*⟩ for a ⟨*module*⟩, including the function which is used to actually implement it.

\keys_log:nn

\keys_log:nn {⟨*module*⟩} {⟨*key*⟩}

Writes in the log file the information associated to the ⟨*key*⟩ for a ⟨*module*⟩. See also \keys_show:nn which displays the result in the terminal.

28.10 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a ⟨*key-value list*⟩ into ⟨*keys*⟩ and associated ⟨*values*⟩. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double `#` tokens or expand any input. Active tokens `=` and `,` appearing at the outer level of braces are converted to category "other" (12) so that the parser does not "miss" any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:nnn` ☆
`\keyval_parse:(nnV|nnv)` ☆

New: 2020-12-19
Updated: 2021-05-10

`\keyval_parse:nnn {⟨code₁⟩} {⟨code₂⟩} {⟨key-value list⟩}`

Parses the ⟨*key-value list*⟩ into a series of ⟨*keys*⟩ and associated ⟨*values*⟩, or keys alone (if no ⟨*value*⟩ was given). ⟨$code_1$⟩ receives each ⟨*key*⟩ (with no ⟨*value*⟩) as a trailing brace group, whereas ⟨$code_2$⟩ is appended by two brace groups, the ⟨*key*⟩ and ⟨*value*⟩. The order of the ⟨*keys*⟩ in the ⟨*key-value list*⟩ is preserved. Thus

```
\keyval_parse:nnn
  { \use_none:nn  { code 1 } }
  { \use_none:nnn { code 2 } }
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\use_none:nnn { code 2 } { key1 } { value1 }
\use_none:nnn { code 2 } { key2 } { value2 }
\use_none:nnn { code 2 } { key3 } { }
\use_none:nn  { code 1 } { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the ⟨*key*⟩ and ⟨*value*⟩, then one *outer* set of braces is removed from the ⟨*key*⟩ and ⟨*value*⟩ as part of the processing. If you need exactly the output shown above, you'll need to either `e`-type or `x`-type expand the function.

TEXhackers note: The result of each list element is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an `e`-type or `x`-type argument expansion.

`\keyval_parse:NNn` ☆
`\keyval_parse:(NNV|NNv)` ☆

Updated: 2021-05-10

`\keyval_parse:NNn` ⟨*function*$_1$⟩ ⟨*function*$_2$⟩ {⟨*key-value list*⟩}

Parses the ⟨*key-value list*⟩ into a series of ⟨*keys*⟩ and associated ⟨*values*⟩, or keys alone (if no ⟨*value*⟩ was given). ⟨*function*$_1$⟩ should take one argument, while ⟨*function*$_2$⟩ should absorb two arguments. After `\keyval_parse:NNn` has parsed the ⟨*key-value list*⟩, ⟨*function*$_1$⟩ is used to process keys given with no value and ⟨*function*$_2$⟩ is used to process keys given with a value. The order of the ⟨*keys*⟩ in the ⟨*key-value list*⟩ is preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n  { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the ⟨*key*⟩ and ⟨*value*⟩, then one *outer* set of braces is removed from the ⟨*key*⟩ and ⟨*value*⟩ as part of the processing.

This shares the implementation of `\keyval_parse:nnn`, the difference is only semantically.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an `e`-type or `x`-type argument expansion.

Chapter 29

The l3intarray module
Fast global integer arrays

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialization
- The absolute value of each entry has maximum $2^{30} - 1$ (i.e., one power lower than the usual `\c_max_int` ceiling of $2^{31} - 1$)

The use of `intarray` data is therefore recommended for cases where the need for fast access is of paramount importance.

29.1 Creating and initializing integer array variables

`\intarray_new:Nn`
`\intarray_new:cn`

`\intarray_new:Nn` ⟨*intarray var*⟩ {⟨*size*⟩}

Evaluates the integer expression ⟨*size*⟩ and allocates an ⟨*integer array variable*⟩ with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

`\intarray_const_from_clist:Nn`
`\intarray_const_from_clist:cn`

`\intarray_const_from_clist:Nn` ⟨*intarray var*⟩ {⟨*int expr clist*⟩}

Creates a new constant ⟨*integer array variable*⟩ or raises an error if the name is already taken. The ⟨*integer array variable*⟩ is set (globally) to contain as its items the results of evaluating each ⟨*integer expression*⟩ in the ⟨*comma list*⟩.

`\intarray_gzero:N`
`\intarray_gzero:c`

`\intarray_gzero:N` ⟨*intarray var*⟩

Sets all entries of the ⟨*integer array variable*⟩ to zero. Assignments are always global.

29.2 Adding data to integer arrays

\intarray_gset:Nnn
\intarray_gset:cnn

\intarray_gset:Nnn ⟨*intarray var*⟩ {⟨*position*⟩} {⟨*value*⟩}

Stores the result of evaluating the integer expression ⟨*value*⟩ into the ⟨*integer array variable*⟩ at the (integer expression) ⟨*position*⟩. If the ⟨*position*⟩ is not between 1 and the \intarray_count:N, or the ⟨*value*⟩'s absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

29.3 Counting entries in integer arrays

\intarray_count:N ⋆
\intarray_count:c ⋆

\intarray_count:N ⟨*intarray var*⟩

Expands to the number of entries in the ⟨*integer array variable*⟩. Contrarily to \seq_count:N this is performed in constant time.

29.4 Using a single entry

\intarray_item:Nn ⋆
\intarray_item:cn ⋆

\intarray_item:Nn ⟨*intarray var*⟩ {⟨*position*⟩}

Expands to the integer entry stored at the (integer expression) ⟨*position*⟩ in the ⟨*integer array variable*⟩. If the ⟨*position*⟩ is not between 1 and the \intarray_count:N, an error occurs.

\intarray_rand_item:N ⋆
\intarray_rand_item:c ⋆

\intarray_rand_item:N ⟨*intarray var*⟩

Selects a pseudo-random item of the ⟨*integer array*⟩. If the ⟨*integer array*⟩ is empty, produce an error.

29.5 Integer array conditional

\intarray_if_exist_p:N ⋆
\intarray_if_exist_p:c ⋆
\intarray_if_exist:NTF ⋆
\intarray_if_exist:cTF ⋆

New: 2024-03-31

\intarray_if_exist_p:N ⟨*intarray var*⟩
\intarray_if_exist:NTF ⟨*intarray var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*intarray var*⟩ is currently defined. This does not check that the ⟨*intarray var*⟩ really is an integer array variable.

29.6 Viewing integer arrays

\intarray_show:N
\intarray_show:c
\intarray_log:N
\intarray_log:c

\intarray_show:N ⟨*intarray var*⟩
\intarray_log:N ⟨*intarray var*⟩

Displays the items in the ⟨*integer array variable*⟩ in the terminal or writes them in the log file.

29.7 Implementation notes

It is a wrapper around the \fontdimen primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to l3seq sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive \fontdimen stores dimensions but the l3intarray module transparently converts these from/to integers. Assignments are always global.

While LuaTeX's memory is extensible, other engines can "only" deal with a bit less than 4×10^6 entries in all \fontdimen arrays combined (with default TeX Live settings).

Chapter 30

The l3fp module Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. *Floating point expressions* ("⟨`fp expr`⟩") support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x/y, square root $\sqrt{x}$, and parentheses.
- Comparison operators: $x < y$, $x <= y$, $x >? y$, $x \,!= y$ etc.
- Boolean logic: sign $\operatorname{sign} x$, negation $!\,x$, conjunction $x \,\&\&\, y$, disjunction $x \,||\, y$, ternary operator $x \,?\, y : z$.
- Exponentials: $\exp x$, $\ln x$, x^y, $\operatorname{logb} x$.
- Integer factorial: $\operatorname{fact} x$.
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\operatorname{sind} x$, $\operatorname{cosd} x$, $\operatorname{tand} x$, $\operatorname{cotd} x$, $\operatorname{secd} x$, $\operatorname{cscd} x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\operatorname{asin} x$, $\operatorname{acos} x$, $\operatorname{atan} x$, $\operatorname{acot} x$, $\operatorname{asec} x$, $\operatorname{acsc} x$ giving a result in radians, and $\operatorname{asind} x$, $\operatorname{acosd} x$, $\operatorname{atand} x$, $\operatorname{acotd} x$, $\operatorname{asecd} x$, $\operatorname{acscd} x$ giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\operatorname{sech} x$, csch, and $\operatorname{asinh} x$, $\operatorname{acosh} x$, $\operatorname{atanh} x$, $\operatorname{acoth} x$, $\operatorname{asech} x$, $\operatorname{acsch} x$.

- Extrema: $\max(x_1, x_2, \ldots)$, $\min(x_1, x_2, \ldots)$, $\operatorname{abs}(x)$.
- Rounding functions, controlled by two optional values, n (number of places, 0 by default) and t (behavior on a tie, `nan` by default):
  - $\operatorname{trunc}(x, n)$ rounds towards zero,
  - $\operatorname{floor}(x, n)$ rounds towards $-\infty$,

  - ceil(x, n) rounds towards $+\infty$,
  - round(x, n, t) rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.

  And *(not yet)* modulo, and "quantize".

- Random numbers: rand(), randint(m, n).
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\⟨type⟩_use:N`) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: $(x_1, \ldots, x_n)$ that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. A "floating point" is a floating point number or a tuple thereof. See section 30.13.1 for a description of what a floating point is, section 30.13.2 for details about how an expression is parsed, and section 30.13.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 30.11.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {sin(3.5)/2 + 2e-3} $.
```

The operation `round` can be used to limit the result's precision. Adding $+0$ avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the l3fp module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnum } { m }
  { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnum { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

See the documentation of siunitx for various options of `\num`.

30.1 Creating and initializing floating point variables

\fp_new:N
\fp_new:c

\fp_new:N ⟨*fp var*⟩

Creates a new ⟨*fp var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*fp var*⟩ is initially +0.

\fp_const:Nn
\fp_const:cn

\fp_const:Nn ⟨*fp var*⟩ {⟨*fp expr*⟩}

Creates a new constant ⟨*fp var*⟩ or raises an error if the name is already taken. The ⟨*fp var*⟩ is set globally equal to the result of evaluating the ⟨*fp expr*⟩.

\fp_zero:N
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

\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

\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.

30.2 Setting floating point variables

\fp_set:Nn
\fp_set:(cn|NV|cV)
\fp_gset:Nn
\fp_gset:(cn|NV|cV)

\fp_set:Nn ⟨*fp var*⟩ {⟨*fp expr*⟩}

Sets ⟨*fp var*⟩ equal to the result of computing the ⟨*fp expr*⟩.

\fp_set_eq:NN
\fp_set_eq:(cN|Nc|cc)
\fp_gset_eq:NN
\fp_gset_eq:(cN|Nc|cc)

\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

\fp_add:Nn ⟨*fp var*⟩ {⟨*fp expr*⟩}

Adds the result of computing the ⟨*fp expr*⟩ to the ⟨*fp var*⟩. This also applies if ⟨*fp var*⟩ and ⟨*floating point expression*⟩ evaluate to tuples of the same size.

\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn

\fp_sub:Nn ⟨*fp var*⟩ {⟨*fp expr*⟩}

Subtracts the result of computing the ⟨*floating point expression*⟩ from the ⟨*fp var*⟩. This also applies if ⟨*fp var*⟩ and ⟨*floating point expression*⟩ evaluate to tuples of the same size.

30.3 Using floating points

\fp_eval:n ⋆

\fp_eval:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and nan trigger an "invalid operation" exception. For a tuple, each item is converted using \fp_eval:n and they are combined as (⟨*fp*$_1$⟩,␣⟨*fp*$_2$⟩,␣...⟨*fp*$_n$⟩) if $n > 1$ and (⟨*fp*$_1$⟩,) or () for fewer items. This function is identical to \fp_to_decimal:n.

\fp_sign:n ⋆

\fp_sign:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and leaves its sign in the input stream using \fp_eval:n {sign(⟨*result*⟩)}: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0. If the operand is a tuple or is nan, then "invalid operation" occurs and the result is 0.

\fp_to_decimal:N ⋆
\fp_to_decimal:c ⋆
\fp_to_decimal:n ⋆

\fp_to_decimal:N ⟨*fp var*⟩
\fp_to_decimal:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and nan trigger an "invalid operation" exception. For a tuple, each item is converted using \fp_to_decimal:n and they are combined as (⟨*fp*$_1$⟩,␣⟨*fp*$_2$⟩,␣...⟨*fp*$_n$⟩) if $n > 1$ and (⟨*fp*$_1$⟩,) or () for fewer items.

\fp_to_dim:N ⋆
\fp_to_dim:c ⋆
\fp_to_dim:n ⋆

\fp_to_dim:N ⟨*fp var*⟩
\fp_to_dim:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to \fp_to_decimal:n, with an additional trailing pt (both letter tokens). In particular, the result may be outside the range $[-2^{14}+2^{-17}, 2^{14}-2^{-17}]$ of valid TeX dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and nan, trigger an "invalid operation" exception.

\fp_to_int:N ⋆
\fp_to_int:c ⋆
\fp_to_int:n ⋆

\fp_to_int:N ⟨*fp var*⟩
\fp_to_int:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31}+1, 2^{31}-1]$ of valid TeX integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and nan, trigger an "invalid operation" exception.

\fp_to_scientific:N ⋆
\fp_to_scientific:c ⋆
\fp_to_scientific:n ⋆

\fp_to_scientific:N ⟨*fp var*⟩
\fp_to_scientific:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result in scientific notation:

⟨*optional -*⟩⟨*digit*⟩.⟨*15 digits*⟩e⟨*optional sign*⟩⟨*exponent*⟩

The leading ⟨*digit*⟩ is non-zero except in the case of ± 0. The values $\pm\infty$ and nan trigger an "invalid operation" exception. Normal category codes apply: thus the e is category code 11 (a letter). For a tuple, each item is converted using \fp_to_scientific:n and they are combined as (⟨fp_1⟩,␣⟨fp_2⟩,␣...⟨fp_n⟩) if $n > 1$ and (⟨fp_1⟩,) or () for fewer items.

\fp_to_tl:N ⋆
\fp_to_tl:c ⋆
\fp_to_tl:n ⋆

\fp_to_tl:N ⟨*fp var*⟩
\fp_to_tl:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from \fp_to_scientific:n). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see \fp_to_decimal:n. Negative numbers start with -. The special values ± 0, $\pm\infty$ and nan are rendered as 0, -0, inf, -inf, and nan respectively. Normal category codes apply and thus inf or nan, if produced, are made up of letters. For a tuple, each item is converted using \fp_to_tl:n and they are combined as (⟨fp_1⟩,␣⟨fp_2⟩,␣...⟨fp_n⟩) if $n > 1$ and (⟨fp_1⟩,) or () for fewer items.

\fp_use:N ⋆
\fp_use:c ⋆

\fp_use:N ⟨*fp var*⟩

Inserts the value of the ⟨*fp var*⟩ into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and nan trigger an "invalid operation" exception. For a tuple, each item is converted using \fp_to_decimal:n and they are combined as (⟨fp_1⟩,␣⟨fp_2⟩,␣...⟨fp_n⟩) if $n > 1$ and (⟨fp_1⟩,) or () for fewer items. This function is identical to \fp_to_decimal:N.

30.4 Formatting floating points

`\fp_format:nn` ⋆

New: 2025-06-09

`\fp_format:nn {⟨fp expr⟩} {⟨format specification⟩}`

Evaluates the ⟨*fp expr*⟩ and converts the result to a string according to the ⟨*format specification*⟩. The ⟨*style*⟩ can be

- `e` for scientific notation, with one digit before and ⟨*precision*⟩ digits after the decimal separator, and an integer exponent, following `e`;
- `f` for a fixed point notation, with ⟨*precision*⟩ digits after the decimal separator and no exponent;
- `g` for a general format, which uses style `f` for numbers in the range $[10^{-4}, 10^{\langle precision\rangle})$ and style `e` otherwise.

When there is no ⟨*style*⟩ specifier nor ⟨*precision*⟩ the number is displayed without rounding. Otherwise the ⟨*precision*⟩ defaults to 6. The details of the ⟨*format specification*⟩ are described in Section 19.1.

30.5 Floating point conditionals

`\fp_if_exist_p:N` ⋆
`\fp_if_exist_p:c` ⋆
`\fp_if_exist:NTF` ⋆
`\fp_if_exist:cTF` ⋆

`\fp_if_exist_p:N ⟨fp var⟩`
`\fp_if_exist:NTF ⟨fp var⟩ {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*fp var*⟩ is currently defined. This does not check that the ⟨*fp var*⟩ really is a floating point variable.

`\fp_compare_p:nNn` ⋆
`\fp_compare:nNnTF` ⋆

`\fp_compare_p:nNn {⟨fp expr₁⟩} ⟨relation⟩ {⟨fp expr₂⟩}`
`\fp_compare:nNnTF {⟨fp expr₁⟩} ⟨relation⟩ {⟨fp expr₂⟩} {⟨true code⟩} {⟨false code⟩}`

Compares the ⟨*fp expr*$_1$⟩ and the ⟨*fp expr*$_2$⟩, and returns `true` if the ⟨*relation*⟩ is obeyed. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ ("not ordered"). The last case occurs exactly if one or both operands is `nan` or is a tuple, unless they are equal tuples. Note that a `nan` is distinct from any value, even another `nan`, hence $x = x$ is not true for a `nan`. To test if a value is `nan`, compare it to an arbitrary number with the "not ordered" relation.

```
\fp_compare:nNnTF { <value> } ? { 0 }
  { } % <value> is nan
  { } % <value> is not nan
```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no `nan`). At present any other comparison with tuples yields `?` (not ordered). This is experimental.

This function is less flexible than `\fp_compare:nTF` but slightly faster. It is provided for consistency with `\int_compare:nNnTF` and `\dim_compare:nNnTF`.

\fp_compare_p:n ★
\fp_compare:nTF ★

```
\fp_compare_p:n
  {
    ⟨fp expr_1⟩ ⟨relation_1⟩
    ...
    ⟨fp expr_N⟩ ⟨relation_N⟩
    ⟨fp expr_N+1⟩
  }
\fp_compare:nTF
  {
    ⟨fp expr_1⟩ ⟨relation_1⟩
    ...
    ⟨fp expr_N⟩ ⟨relation_N⟩
    ⟨fp expr_N+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*fp exprs*⟩ as described for `\fp_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*fp expr*$_1$⟩ and ⟨*fp expr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*fp expr*$_2$⟩ and ⟨*fp expr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*fp expr*$_N$⟩ and ⟨*fp expr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*floating point expression*⟩ is evaluated only once. Contrarily to `\int_compare:nTF`, all ⟨*fp exprs*⟩ are computed, even if one comparison is `false`. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ ("not ordered"). The last case occurs exactly if one or both operands is `nan` or is a tuple, unless they are equal tuples. Each ⟨*relation*⟩ can be any (non-empty) combination of `<`, `=`, `>`, and `?`, plus an optional leading `!` (which negates the ⟨*relation*⟩), with the restriction that the ⟨*relation*⟩ may not start with `?`, as this symbol has a different meaning (in combination with `:`) within floating point expressions. The comparison x ⟨*relation*⟩ y is then `true` if the ⟨*relation*⟩ does not start with `!` and the actual relation (`<`, `=`, `>`, or `?`) between x and y appears within the ⟨*relation*⟩, or on the contrary if the ⟨*relation*⟩ starts with `!` and the relation between x and y does not appear within the ⟨*relation*⟩. Common choices of ⟨*relation*⟩ include `>=` (greater or equal), `!=` (not equal), `!?` or `<=>` (comparable).

This function is more flexible than `\fp_compare:nNnTF` and only slightly slower.

\fp_if_nan_p:n ★
\fp_if_nan:nTF ★

```
\fp_if_nan_p:n {⟨fp expr⟩}
\fp_if_nan:nTF {⟨fp expr⟩} {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*fp expr*⟩ and tests whether the result is exactly `nan`. The test returns `false` for any other result, even a tuple containing `nan`.

30.6 Floating point expression loops

\fp_do_until:nNnn ☆

```
\fp_do_until:nNnn {⟨fp expr_1⟩} ⟨relation⟩ {⟨fp expr_2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\fp_do_while:nNnn` ☆ `\fp_do_while:nNnn {⟨fp expr₁⟩} ⟨relation⟩ {⟨fp expr₂⟩} {⟨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*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\fp_until_do:nNnn` ☆ `\fp_until_do:nNnn {⟨fp expr₁⟩} ⟨relation⟩ {⟨fp expr₂⟩} {⟨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 is repeated, and a loop occurs until the test is `true`.

`\fp_while_do:nNnn` ☆ `\fp_while_do:nNnn {⟨fp expr₁⟩} ⟨relation⟩ {⟨fp expr₂⟩} {⟨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 is repeated, and a loop occurs until the test is `false`.

`\fp_do_until:nn` ☆ `\fp_do_until:nn { ⟨fp expr₁⟩ ⟨relation⟩ ⟨fp expr₂⟩ } {⟨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*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\fp_do_while:nn` ☆ `\fp_do_while:nn { ⟨fp expr₁⟩ ⟨relation⟩ ⟨fp expr₂⟩ } {⟨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*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\fp_until_do:nn` ☆ `\fp_until_do:nn { ⟨fp expr₁⟩ ⟨relation⟩ ⟨fp expr₂⟩ } {⟨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 is repeated, and a loop occurs until the test is `true`.

`\fp_while_do:nn` ☆ `\fp_while_do:nn { ⟨fp expr₁⟩ ⟨relation⟩ ⟨fp expr₂⟩ } {⟨code⟩}`

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

\fp_step_function:nnnN ☆
\fp_step_function:nnnc ☆

\fp_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩

This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The ⟨function⟩ is then placed in front of each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩). The ⟨step⟩ must be non-zero. If the ⟨step⟩ is positive, the loop stops when the ⟨value⟩ becomes larger than the ⟨final value⟩. If the ⟨step⟩ is negative, the loop stops when the ⟨value⟩ becomes smaller than the ⟨final value⟩. The ⟨function⟩ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

[I saw 1.0] [I saw 1.1] [I saw 1.2] [I saw 1.3] [I saw 1.4] [I saw 1.5]

TeXhackers note: Due to rounding, it may happen that adding the ⟨step⟩ to the ⟨value⟩ does not change the ⟨value⟩; such cases give an error, as they would otherwise lead to an infinite loop.

\fp_step_inline:nnnn

\fp_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}

This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩), the ⟨code⟩ is inserted into the input stream with #1 replaced by the current ⟨value⟩. Thus the ⟨code⟩ should define a function of one argument (#1).

\fp_step_variable:nnnNn

\fp_step_variable:nnnNn
{⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}

This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩), the ⟨code⟩ is inserted into the input stream, with the ⟨tl var⟩ defined as the current ⟨value⟩. Thus the ⟨code⟩ should make use of the ⟨tl var⟩.

30.7 Symbolic expressions

Floating point expressions support variables: these can only be set locally, so act like standard \l_... variables.

```
\fp_new_variable:n { A }
\fp_set:Nn \l_tmpb_fp { 1 * sin(A) + 3**2 }
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn { A } { pi/2 }
```

```
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn { A } { 0 }
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
```

defines `A` to be a variable, then defines `\l_tmpb_fp` to stand for `1*sin(A)+9` (note that `3**2` is evaluated, but the `1*` product is not simplified away). Until `\l_tmpb_fp` is changed, `\fp_show:N \l_tmpb_fp` will show `((1*sin(A))+9)` regardless of the value of `A`. The next step defines `A` to be equal to `pi/2`: then `\fp_show:n { \l_tmpb_fp }` will evaluate `\l_tmpb_fp` and show `10`. We then redefine `A` to be `0`: since `\l_tmpb_fp` still stands for `1*sin(A)+9`, the value shown is then `9`. Variables can be set with `\fp_set_variable:nn` to arbitrary floating point expressions including other variables.

At present, the following operations and functions are *not* supported

- infix binary comparisons like `>`, `=`, `<`
- infix ternary operator like `?:`
- prefix variable-ary functions like `round` and friends, `min`, `max`, `atan`, `acot`, `atand`, `acotd`

\fp_new_variable:n

New: 2023-10-19

`\fp_new_variable:n {⟨identifier⟩}`

Declares the ⟨*identifier*⟩ as a variable, which allows it to be used in floating point expressions. For instance,

```
\fp_new_variable:n { A }
\fp_show:n { A**2 - A + 1 }
```

shows `(((A^2)-A)+1)`. If the declaration was missing, the parser would complain about an "`Unknown fp word 'A'`". The ⟨*identifier*⟩ must consist entirely of Latin letters among `[a-zA-Z]`.

\fp_set_variable:nn

New: 2023-10-19

`\fp_set_variable:nn {⟨identifier⟩} {⟨fp expr⟩}`

Sets the ⟨*identifier*⟩ to stand in any further expression for the result of evaluating the ⟨*floating point expression*⟩ as much as possible. The ⟨*identifier*⟩ must be declared by `\fp_new_function:n` first. The result may contain other variables, which are then replaced by their values if they have any. For instance,

```
\fp_new_variable:n { A }
\fp_new_variable:n { B }
\fp_new_variable:n { C }
\fp_set_variable:nn { A } { 3 }
\fp_set_variable:nn { C } { A ** 2 + B * 1 }
\fp_show:n { C + 4 }
\fp_set_variable:nn { A } { 4 }
\fp_show:n { C + 4 }
```

shows `((9+(B*1))+4)` twice: changing the value of `A` to `4` does not alter `C` because `A` was replaced by its value `3` when evaluating `A**2+B*1`.

\fp_clear_variable:n \fp_clear_variable:n {⟨*identifier*⟩}

New: 2023-10-19 Removes any value given by \fp_set_variable:nn to the variable with this ⟨*identifier*⟩. For instance,

```
\fp_new_variable:n { A }
\fp_set_variable:nn { A } { 3 }
\fp_show:n { A ^ 2 }
\fp_clear_variable:n { A }
\fp_show:n { A ^ 2 }
```

shows 9, then (A^2).

30.8 User-defined functions

It is possible to define new user functions which can be used inside the argument to \fp_eval:n, etc. These functions may take one or more named arguments, and should be implemented using expansion methods only.

\fp_new_function:n \fp_new_function:n {⟨*identifier*⟩}

New: 2023-10-19 Declares the ⟨*identifier*⟩ as a function, which allows it to be used in floating point expressions. For instance,

```
\fp_new_function:n { foo }
\fp_show:n { foo ( 1 + 2 , foo(3), A ) ** 2 } }
```

shows (foo(3, foo(3), A))^(2). If the declaration was missing, the parser would complain about an "Unknown fp word 'foo'". The ⟨*identifier*⟩ must consist entirely of Latin letters [a-zA-Z].

\fp_set_function:nnn \fp_set_function:nnn {⟨*identifier*⟩} {⟨*vars*⟩} {⟨*fp expr*⟩}

New: 2023-10-19 Sets the ⟨*identifier*⟩ to stand in any further expression for the result of evaluating the ⟨*floating point expression*⟩, with the ⟨*identifier*⟩ accepting the ⟨*vars*⟩ (a non-empty comma-separated list). The ⟨*identifier*⟩ must be declared by \fp_new_function:n first. The result may contain other functions, which are then replaced by their results if they have any. For instance,

```
\fp_new_function:n { npow }
\fp_set_function:nnn { npow } { a,b } { a**b }
\fp_show:n { npow(16,0.25) }
```

shows 2. The names of the ⟨*vars*⟩ must consist entirely of Latin letters [a-zA-Z], but are otherwise not restricted: in particular, they are independent of any variables declared by \fp_new_variable:n.

\fp_clear_function:n \fp_clear_function:n {⟨*identifier*⟩}

New: 2023-10-19 Removes any definition given by \fp_set_function:nnn to the function with this ⟨*identifier*⟩.

30.9 Some useful constants, and scratch variables

`\c_zero_fp`
`\c_minus_zero_fp`

Zero, with either sign.

`\c_one_fp`

One as an `fp`: useful for comparisons in some places.

`\c_inf_fp`
`\c_minus_inf_fp`

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

`\c_nan_fp`

Not a number. This can be input directly in a floating point expression as `nan`.

`\c_e_fp`

The value of the base of the natural logarithm, $\mathrm{e} = \exp(1)$.

`\c_pi_fp`

The value of π. This can be input directly in a floating point expression as `pi`.

`\c_one_degree_fp`

The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

30.10 Scratch variables

`\l_tmpa_fp`
`\l_tmpb_fp`

Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_fp`
`\g_tmpb_fp`

Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

30.11 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

"Exceptions" may occur when performing some floating point operations, such as `0 / 0`, or `10 ** 1e9999`. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0.
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$ or $\sin(\infty)$, and results in a `nan`. It also occurs for conversion functions whose target type does not have the appropriate infinite or `nan` value (*e.g.*, `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, *e.g.*, $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

(not yet) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in LaTeX3.

To each exception we associate a "flag": `\l_fp_overflow_flag`, `\l_fp_underflow_flag`, `\l_fp_invalid_operation_flag` and `\l_fp_division_by_zero_flag`. The state of these flags can be tested and modified with commands from l3flag

By default, the "invalid operation" exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behavior when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

`\fp_trap:nn`

`\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}`

All occurrences of the ⟨*exception*⟩ (`overflow`, `underflow`, `invalid_operation` or `division_by_zero`) within the current group are treated as ⟨*trap type*⟩, which can be

- `none`: the ⟨*exception*⟩ will be entirely ignored, and leave no trace;
- `flag`: the ⟨*exception*⟩ will turn the corresponding flag on when it occurs;
- `error`: additionally, the ⟨*exception*⟩ will halt the TeX run and display some information about the current operation in the terminal.

`\l_fp_overflow_flag`
`\l_fp_underflow_flag`
`\l_fp_invalid_operation_flag`
`\l_fp_division_by_zero_flag`

Flags denoting the occurrence of various floating-point exceptions.

30.12 Viewing floating points

\fp_show:N
\fp_show:c
\fp_show:n

Updated: 2021-04-29

\fp_show:N ⟨*fp var*⟩
\fp_show:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and displays the result in the terminal.

\fp_log:N
\fp_log:c
\fp_log:n

Updated: 2021-04-29

\fp_log:N ⟨*fp var*⟩
\fp_log:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and writes the result in the log file.

30.13 Floating point expressions

30.13.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- ± 0, zero, with a given sign;
- $\pm\infty$, infinity, with a given sign;
- `nan`, is "not a number", and can be either quiet or signaling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- ⟨*sign*⟩: a possibly empty string of `+` and `-` characters;
- ⟨*significand*⟩: a non-empty string of digits together with zero or one dot;
- ⟨*exponent*⟩ optionally: the character `e` or `E`, followed by a possibly empty string of `+` and `-` tokens, and a non-empty string of digits.

The sign of the resulting number is `+` if ⟨*sign*⟩ contains an even number of `-`, and `-` otherwise, hence, an empty ⟨*sign*⟩ denotes a non-negative input. The stored significand is obtained from ⟨*significand*⟩ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input ⟨*significand*⟩ has at most 16 digits. The stored ⟨*exponent*⟩ is obtained by combining the input ⟨*exponent*⟩ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting ⟨*exponent*⟩ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm\infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if ⟨*significand*⟩ contains no non-zero digit (i.e., consists only in characters `0`, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

The ⟨*significand*⟩ must be non-empty, so `e1` and `e-1` are not valid floating point numbers. Note that the latter could be mistaken with the difference of "`e`" and 1. To avoid confusions, the base of natural logarithms cannot be input as `e` and should be input as `exp(1)` or `\c_e_fp` (which is faster).

Special numbers are input as follows:

- `inf` represents $+\infty$, and can be preceded by any ⟨*sign*⟩, yielding $\pm\infty$ as appropriate.
- `nan` represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a `nan`.
- Note that commands such as `\infty`, `\pi`, or `\sin` *do not* work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

30.13.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (`2pi`) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance `3(4+5)`).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc*.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$\begin{aligned}
\texttt{1/2pi} &= 1/(2\pi),\\
\texttt{1/2pi(pi + pi)} &= (2\pi)^{-1}(\pi+\pi) \simeq 1,\\
\texttt{sin2pi} &= \sin(2)\pi \neq 0,\\
\texttt{2\^{}2max(3,5)} &= 2^2 \max(3,5) = 20,\\
\texttt{1in/1cm} &= (1\mathrm{in})/(1\mathrm{cm}) = 2.54.
\end{aligned}$$

Functions are called on the value of their argument, contrarily to TeX macros.

30.13.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ± 0, and `true` otherwise, including when it is `nan` or a tuple such as $(0,0)$. Tuples are only supported to some extent by operations that work with truth values (`?:`, `||`, `&&`, `!`), by comparisons (`!<=>?`), and by `+`, `-`, `*`, `/`. Unless otherwise specified, providing a tuple as an argument of any other operation yields the "invalid operation" exception and a `nan` result.

`?:` `\fp_eval:n { ⟨operand₁⟩ ? ⟨operand₂⟩ : ⟨operand₃⟩ }`

The ternary operator `?:` results in ⟨*operand*$_2$⟩ if ⟨*operand*$_1$⟩ is true (not ± 0), and ⟨*operand*$_3$⟩ if ⟨*operand*$_1$⟩ is false (± 0). All three ⟨*operands*⟩ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
  {
    1 + 3 > 4 ? 1 :
    2 + 4 > 5 ? 2 :
    3 + 5 > 6 ? 3 : 4
  }
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following `:` is taken, and $2+4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

`||` `\fp_eval:n { ⟨operand₁⟩ || ⟨operand₂⟩ }`

If ⟨*operand*$_1$⟩ is true (not ± 0), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases; they may be tuples. In ⟨*operand*$_1$⟩ `||` ⟨*operand*$_2$⟩ `||` ... `||` ⟨*operands*$_n$⟩, the first true (nonzero) ⟨*operand*⟩ is used and if all are zero the last one (± 0) is used.

`&&` `\fp_eval:n { ⟨operand₁⟩ && ⟨operand₂⟩ }`

If ⟨*operand*$_1$⟩ is false (equal to ± 0), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases; they may be tuples. In ⟨*operand*$_1$⟩ `&&` ⟨*operand*$_2$⟩ `&&` ... `&&` ⟨*operands*$_n$⟩, the first false (± 0) ⟨*operand*⟩ is used and if none is zero the last one is used.

```
<   \fp_eval:n
=     {
>       ⟨operand1⟩ ⟨relation1⟩
?       ...
        ⟨operandN⟩ ⟨relationN⟩
        ⟨operandN+1⟩
      }
```

Each ⟨*relation*⟩ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`, and may not start with `?`. This evaluates to $+1$ if all comparisons ⟨*operand*$_i$⟩ ⟨*relation*$_i$⟩ ⟨*operand*$_{i+1}$⟩ are true, and $+0$ otherwise. All ⟨*operands*⟩ are evaluated (once) in all cases. See `\fp_compare:nTF` for details.

```
+   \fp_eval:n { ⟨operand1⟩ + ⟨operand2⟩ }
-   \fp_eval:n { ⟨operand1⟩ - ⟨operand2⟩ }
```

Computes the sum or the difference of its two ⟨*operands*⟩. The "invalid operation" exception occurs for $\infty - \infty$. "Underflow" and "overflow" occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the "invalid operation" exception occurs and the result is `nan`.

```
*   \fp_eval:n { ⟨operand1⟩ * ⟨operand2⟩ }
/   \fp_eval:n { ⟨operand1⟩ / ⟨operand2⟩ }
```

Computes the product or the ratio of its two ⟨*operands*⟩. The "invalid operation" exception occurs for ∞/∞, $0/0$, or $0 * \infty$. "Division by zero" occurs when dividing a finite non-zero number by ± 0. "Underflow" and "overflow" occur when appropriate. When ⟨*operand*$_1$⟩ is a tuple and ⟨*operand*$_2$⟩ is a floating point number, each item of ⟨*operand*$_1$⟩ is multiplied or divided by ⟨*operand*$_2$⟩. Multiplication also supports the case where ⟨*operand*$_1$⟩ is a floating point number and ⟨*operand*$_2$⟩ a tuple. Other combinations yield an "invalid operation" exception and a `nan` result.

```
+   \fp_eval:n { + ⟨operand⟩ }
-   \fp_eval:n { - ⟨operand⟩ }
!   \fp_eval:n { ! ⟨operand⟩ }
```

The unary `+` does nothing, the unary `-` changes the sign of the ⟨*operand*⟩ (for a tuple, of all its components), and `!` ⟨*operand*⟩ evaluates to 1 if ⟨*operand*⟩ is false (is ± 0) and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

```
**  \fp_eval:n { ⟨operand1⟩ ** ⟨operand2⟩ }
^   \fp_eval:n { ⟨operand1⟩ ^ ⟨operand2⟩ }
```

Raises ⟨*operand*$_1$⟩ to the power ⟨*operand*$_2$⟩. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. If ⟨*operand*$_1$⟩ is negative or -0 then: the result's sign is $+$ if the ⟨*operand*$_2$⟩ is infinite and $(-1)^p$ if the ⟨*operand*$_2$⟩ is $p/5^q$ with p, q integers; the result is $+0$ if `abs(`⟨*operand*$_1$⟩`)^`⟨*operand*$_2$⟩ evaluates to zero; in other cases the "invalid operation" exception occurs because the sign cannot be determined. "Division by zero" occurs when raising ± 0 to a finite strictly negative power. "Underflow" and "overflow" occur when appropriate. If either operand is a tuple, "invalid operation" occurs.

```
abs \fp_eval:n { abs( ⟨fp expr⟩ ) }
```

Computes the absolute value of the ⟨*fp expr*⟩. If the operand is a tuple, "invalid operation" occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.

exp `\fp_eval:n { exp( ⟨fp expr⟩ ) }`

Computes the exponential of the ⟨*fp expr*⟩. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

fact `\fp_eval:n { fact( ⟨fp expr⟩ ) }`

Computes the factorial of the ⟨*fp expr*⟩. If the ⟨*fp expr*⟩ is an integer between -0 and 3248 included, the result is finite and correctly rounded. Larger positive integers give $+\infty$ with "overflow", while `fact(`$+\infty$`)` $= +\infty$ and `fact(nan)` = `nan` with no exception. All other inputs give `nan` with the "invalid operation" exception.

ln `\fp_eval:n { ln( ⟨fp expr⟩ ) }`

Computes the natural logarithm of the ⟨*fp expr*⟩. Negative numbers have no (real) logarithm, hence the "invalid operation" is raised in that case, including for $\ln(-0)$. "Division by zero" occurs when evaluating $\ln(+0) = -\infty$. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

logb ⋆ `\fp_eval:n { logb( ⟨fp expr⟩ ) }`

Determines the exponent of the ⟨*fp expr*⟩, namely the floor of the base-10 logarithm of its absolute value. "Division by zero" occurs when evaluating $\mathrm{logb}(\pm 0) = -\infty$. Other special values are $\mathrm{logb}(\pm\infty) = +\infty$ and logb(`nan`) = `nan`. If the operand is a tuple or is `nan`, then "invalid operation" occurs and the result is `nan`.

max `\fp_eval:n { max( ⟨fp expr1⟩ , ⟨fp expr2⟩ , ... ) }`
min `\fp_eval:n { min( ⟨fp expr1⟩ , ⟨fp expr2⟩ , ... ) }`

Evaluates each ⟨*fp expr*⟩ and computes the largest (smallest) of those. If any of the ⟨*fp expr*⟩ is a `nan` or tuple, the result is `nan`. If any operand is a tuple, "invalid operation" occurs; these operations do not raise exceptions in other cases.

```
round \fp_eval:n { round ( ⟨fp expr⟩ ) }
trunc \fp_eval:n { round ( ⟨fp expr1⟩ , ⟨fp expr2⟩ ) }
ceil  \fp_eval:n { round ( ⟨fp expr1⟩ , ⟨fp expr2⟩ , ⟨fp expr3⟩ ) }
floor
```

Only `round` accepts a third argument. Evaluates ⟨*fp expr*$_1$⟩ $= x$ and ⟨*fp expr*$_2$⟩ $= n$ and ⟨*fp expr*$_3$⟩ $= t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n}; if $n = +\infty$, this always yields x; if $n = -\infty$, this yields one of ± 0, $\pm\infty$, or `nan`; if $n =$ `nan`, this yields `nan`; if n is neither $\pm\infty$ nor an integer, then an "invalid operation" exception is raised. When ⟨*fp expr*$_2$⟩ is omitted, $n = 0$, i.e., ⟨*fp expr*$_1$⟩ is rounded to an integer. The rounding direction depends on the function.

- `round` yields the multiple of 10^{-n} closest to x, with ties (x half-way between two such multiples) rounded as follows. If t is `nan` (or not given) the even multiple is chosen ("ties to even"), if $t = \pm 0$ the multiple closest to 0 is chosen ("ties to zero"), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen ("ties towards positive/negative infinity").
- `floor` yields the largest multiple of 10^{-n} smaller or equal to x ("round towards negative infinity");
- `ceil` yields the smallest multiple of 10^{-n} greater or equal to x ("round towards positive infinity");
- `trunc` yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x ("round towards zero").

"Overflow" occurs if x is finite and the result is infinite (this can only happen if ⟨*fp expr*$_2$⟩ < -9984). If any operand is a tuple, "invalid operation" occurs.

```
sign \fp_eval:n { sign( ⟨fp expr⟩ ) }
```

Evaluates the ⟨*fp expr*⟩ and determines its sign: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0, and `nan` for `nan`. If the operand is a tuple, "invalid operation" occurs. This operation does not raise exceptions in other cases.

```
sin \fp_eval:n { sin( ⟨fp expr⟩ ) }
cos \fp_eval:n { cos( ⟨fp expr⟩ ) }
tan \fp_eval:n { tan( ⟨fp expr⟩ ) }
cot \fp_eval:n { cot( ⟨fp expr⟩ ) }
csc \fp_eval:n { csc( ⟨fp expr⟩ ) }
sec \fp_eval:n { sec( ⟨fp expr⟩ ) }
```

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨*fp expr*⟩ given in radians. For arguments given in degrees, see `sind`, `cosd`, etc. Note that since π is irrational, sin(8pi) is not quite zero, while its analogue sind(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the "invalid operation" exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a "division by zero" exception. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

```
sind \fp_eval:n { sind( ⟨fp expr⟩ ) }
cosd \fp_eval:n { cosd( ⟨fp expr⟩ ) }
tand \fp_eval:n { tand( ⟨fp expr⟩ ) }
cotd \fp_eval:n { cotd( ⟨fp expr⟩ ) }
cscd \fp_eval:n { cscd( ⟨fp expr⟩ ) }
secd \fp_eval:n { secd( ⟨fp expr⟩ ) }
```

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨*fp expr*⟩ given in degrees. For arguments given in radians, see `sin`, `cos`, etc. Note that since π is irrational, sin(8pi) is not quite zero, while its analogue sind(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the "invalid operation" exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a "division by zero" exception. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

```
asin \fp_eval:n { asin( ⟨fp expr⟩ ) }
acos \fp_eval:n { acos( ⟨fp expr⟩ ) }
acsc \fp_eval:n { acsc( ⟨fp expr⟩ ) }
asec \fp_eval:n { asec( ⟨fp expr⟩ ) }
```

Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨*fp expr*⟩ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, etc. If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an "invalid operation" exception is raised. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

```
asind \fp_eval:n { asind( ⟨fp expr⟩ ) }
acosd \fp_eval:n { acosd( ⟨fp expr⟩ ) }
acscd \fp_eval:n { acscd( ⟨fp expr⟩ ) }
asecd \fp_eval:n { asecd( ⟨fp expr⟩ ) }
```

Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨*fp expr*⟩ and returns the result in degrees, in the range $[-90, 90]$ for `asind` and `acscd` and $[0, 180]$ for `acosd` and `asecd`. For a result in radians, use `asin`, etc. If the argument of `asind` or `acosd` lies outside the range $[-1, 1]$, or the argument of `acscd` or `asecd` inside the range $(-1, 1)$, an "invalid operation" exception is raised. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

atan `\fp_eval:n { atan(` ⟨*fp expr*⟩ `) }`
acot `\fp_eval:n { atan(` ⟨*fp expr*$_1$⟩ `,` ⟨*fp expr*$_2$⟩ `) }`
`\fp_eval:n { acot(` ⟨*fp expr*⟩ `) }`
`\fp_eval:n { acot(` ⟨*fp expr*$_1$⟩ `,` ⟨*fp expr*$_2$⟩ `) }`

Those functions yield an angle in radians: `atand` and `acotd` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the ⟨*fp expr*⟩: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates (⟨*fp expr*$_2$⟩, ⟨*fp expr*$_1$⟩): this is the arctangent of ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩, possibly shifted by π depending on the signs of ⟨*fp expr*$_1$⟩ and ⟨*fp expr*$_2$⟩. The two-argument arccotangent computes the angle in polar coordinates of the point (⟨*fp expr*$_1$⟩, ⟨*fp expr*$_2$⟩), equal to the arccotangent of ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩, possibly shifted by π. Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. The "underflow" exception can occur. If any operand is a tuple, "invalid operation" occurs.

atand `\fp_eval:n { atand(` ⟨*fp expr*⟩ `) }`
acotd `\fp_eval:n { atand(` ⟨*fp expr*$_1$⟩ `,` ⟨*fp expr*$_2$⟩ `) }`
`\fp_eval:n { acotd(` ⟨*fp expr*⟩ `) }`
`\fp_eval:n { acotd(` ⟨*fp expr*$_1$⟩ `,` ⟨*fp expr*$_2$⟩ `) }`

Those functions yield an angle in degrees: `atan` and `acot` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the ⟨*fp expr*⟩: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates (⟨*fp expr*$_2$⟩, ⟨*fp expr*$_1$⟩): this is the arctangent of ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩, possibly shifted by 180 depending on the signs of ⟨*fp expr*$_1$⟩ and ⟨*fp expr*$_2$⟩. The two-argument arccotangent computes the angle in polar coordinates of the point (⟨*fp expr*$_1$⟩, ⟨*fp expr*$_2$⟩), equal to the arccotangent of ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. The "underflow" exception can occur. If any operand is a tuple, "invalid operation" occurs.

sqrt `\fp_eval:n { sqrt(` ⟨*fp expr*⟩ `) }`

Computes the square root of the ⟨*fp expr*⟩. The "invalid operation" is raised when the ⟨*fp expr*⟩ is negative or is a tuple; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\texttt{nan}} = \texttt{nan}$.

rand `\fp_eval:n { rand() }`

Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. This is not available in older versions of XeTeX. The random seed can be queried using `\sys_rand_seed:` and set using `\sys_gset_rand_seed:n`.

TeXhackers note: This is based on pseudo-random numbers provided by the engine's primitive `\pdfuniformdeviate` in pdfTeX, pTeX, upTeX and `\uniformdeviate` in LuaTeX and XeTeX. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of "The Art of Computer Programming, Volume 2".

While we are more careful than `\uniformdeviate` to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

randint `\fp_eval:n { randint( ⟨fp expr⟩ ) }`
`\fp_eval:n { randint( ⟨fp expr₁⟩ , ⟨fp expr₂⟩ ) }`

Produces a pseudo-random integer between 1 and ⟨*fp expr*⟩ or between ⟨*fp expr*$_1$⟩ and ⟨*fp expr*$_2$⟩ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See `rand` for important comments on how these pseudo-random numbers are generated.

inf
nan
The special values $+\infty$, $-\infty$, and `nan` are represented as `inf`, `-inf` and `nan` (see `\c_inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi The value of π (see `\c_pi_fp`).

deg The value of 1° in radians (see `\c_one_degree_fp`).

`em` `ex` `in` `pt` `pc` `cm` `mm` `dd` `cc` `nd` `nc` `bp` `sp`

Those units of measurement are equal to their values in `pt`, namely

$$
\begin{aligned}
1\,\mathtt{in} &= 72.27\,\mathtt{pt}\\
1\,\mathtt{pt} &= 1\,\mathtt{pt}\\
1\,\mathtt{pc} &= 12\,\mathtt{pt}\\
1\,\mathtt{cm} &= \frac{1}{2.54}\,\mathtt{in} = 28.45275590551181\,\mathtt{pt}\\
1\,\mathtt{mm} &= \frac{1}{25.4}\,\mathtt{in} = 2.845275590551181\,\mathtt{pt}\\
1\,\mathtt{dd} &= 0.376065\,\mathtt{mm} = 1.07000856496063\,\mathtt{pt}\\
1\,\mathtt{cc} &= 12\,\mathtt{dd} = 12.84010277952756\,\mathtt{pt}\\
1\,\mathtt{nd} &= 0.375\,\mathtt{mm} = 1.066978346456693\,\mathtt{pt}\\
1\,\mathtt{nc} &= 12\,\mathtt{nd} = 12.80374015748031\,\mathtt{pt}\\
1\,\mathtt{bp} &= \frac{1}{72}\,\mathtt{in} = 1.00375\,\mathtt{pt}\\
1\,\mathtt{sp} &= 2^{-16}\,\mathtt{pt} = 1.52587890625\times 10^{-5}\,\mathtt{pt}.
\end{aligned}
$$

The values of the (font-dependent) units `em` and `ex` are gathered from TeX when the surrounding floating point expression is evaluated.

`true` `false`

Other names for 1 and +0.

`\fp_abs:n` ⋆

`\fp_abs:n {⟨fp expr⟩}`

Evaluates the ⟨*fp expr*⟩ as described for `\fp_eval:n` and leaves the absolute value of the result in the input stream. If the argument is $\pm\infty$, `nan` or a tuple, "invalid operation" occurs. Within floating point expressions, `abs()` can be used; it accepts $\pm\infty$ and `nan` as arguments.

`\fp_max:nn` ⋆
`\fp_min:nn` ⋆

`\fp_max:nn {⟨fp expr₁⟩} {⟨fp expr₂⟩}`

Evaluates the ⟨*fp exprs*⟩ as described for `\fp_eval:n` and leaves the resulting larger (`max`) or smaller (`min`) value in the input stream. If the argument is a tuple, "invalid operation" occurs, but no other case raises exceptions. Within floating point expressions, `max()` and `min()` can be used.

30.14 Disclaimer and roadmap

This module may break if the escape character is among `0123456789_+`, or if it receives a TeX primitive conditional affected by `\exp_not:N`.

The following need to be done. I'll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.

- Support signaling `nan`.
- Modulo and remainder, and rounding function `quantize` (and its friends analogous to `trunc`, `ceil`, `floor`).
- `\fp_format:nn {`⟨*fp expr*⟩`} {`⟨*format*⟩`}`, but what should ⟨`format`⟩ be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add $\log(x, b)$ for logarithm of x in base b.
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Factorial (not with `!`), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide an `isnan` function analogue of `\fp_if_nan:nTF`?
- Support keyword arguments?

Pgfmath also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n {0pt}` should return -0, not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a TEX "number too large" error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that l3trial/l3fp-types introduces tools for adding new types.
- In subsection 30.13.1, write a grammar.

- It would be nice if the parse auxiliaries for each operation were set up in the corresponding module, rather than centeralizing in l3fp-parse.
- Some functions should get an _o ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the ternary set of functions is ugly.
- There are many ~ missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t. However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (l3fp-basics).
- Understand and document __fp_basics_pack_weird_low:NNNNw and __fp_basics_pack_weird_high:NNNNNNNNw better. Move the other basics_pack auxiliaries to l3fp-aux under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan's articles, some previous TeX fp packages, the international standards,...
- Also take into account the "inexact" exception?
- Support multi-character prefix operators (*e.g.*, @/ or whatever)?

Chapter 31

The l3fparray module Fast global floating point arrays

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). The interface is very close to that of l3intarray. The size of the array is fixed and must be given at point of initialization

31.1 Creating and initializing floating point array variables

`\fparray_new:Nn`
`\fparray_new:cn`

`\fparray_new:Nn` ⟨*fparray var*⟩ {⟨*size*⟩}

Evaluates the integer expression ⟨*size*⟩ and allocates an ⟨*fparray var*⟩ with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

`\fparray_gzero:N`
`\fparray_gzero:c`

`\fparray_gzero:N` ⟨*fparray var*⟩

Sets all entries of the ⟨*fparray var*⟩ to +0. Assignments are always global.

31.2 Adding data to floating point arrays

`\fparray_gset:Nnn`
`\fparray_gset:cnn`

`\fparray_gset:Nnn` ⟨*fparray var*⟩ {⟨*position*⟩} {⟨*value*⟩}

Stores the result of evaluating the floating point expression ⟨*value*⟩ into the ⟨*fparray var*⟩ at the (integer expression) ⟨*position*⟩. If the ⟨*position*⟩ is not between 1 and the `\fparray_count:N`, an error occurs. Assignments are always global.

31.3 Counting entries in floating point arrays

\fparray_count:N ⋆
\fparray_count:c ⋆

\fparray_count:N ⟨*fparray var*⟩

Expands to the number of entries in the ⟨*fparray var*⟩. This is performed in constant time.

31.4 Using a single entry

\fparray_item:Nn ⋆
\fparray_item:cn ⋆
\fparray_item_to_tl:Nn ⋆
\fparray_item_to_tl:cn ⋆

\fparray_item:Nn ⟨*fparray var*⟩ {⟨*position*⟩}

Applies \fp_use:N or \fp_to_tl:N (respectively) to the floating point entry stored at the (integer expression) ⟨*position*⟩ in the ⟨*fparray var*⟩. If the ⟨*position*⟩ is not between 1 and the \fparray_count:N ⟨*fparray var*⟩, an error occurs.

31.5 Floating point array conditional

\fparray_if_exist_p:N ⋆
\fparray_if_exist_p:c ⋆
\fparray_if_exist:N*TF* ⋆
\fparray_if_exist:c*TF* ⋆

New: 2024-03-31

\fparray_if_exist_p:N ⟨*fparray var*⟩
\fparray_if_exist:NTF ⟨*fparray var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*fparray var*⟩ is currently defined. This does not check that the ⟨*fparray var*⟩ really is a floating point array variable.

Chapter 32

The l3bitset module
Bitsets

This module defines and implements the data type `bitset`, a vector of bits. The size of the vector may grow dynamically. Individual bits can be set and unset by names pointing to an index position. The names `1`, `2`, `3`, ... are predeclared and point to the index positions 1, 2, 3,.... More names can be added and existing names can be changed. The index is like all other indices in expl3 modules *1-based*. A `bitset` can be output as binary number or—as needed e.g. in a PDF dictionary—as decimal (arabic) number. Currently only a small subset of the functions provided by the bitset package are implemented here, mainly the functions needed to use bitsets in PDF dictionaries.

The bitset is stored as a string (but one shouldn't rely on the internal representation) and so the vector size is theoretically unlimited, only restricted by TeX-memory. But the functions to set and clear bits use integer functions for the index so bitsets can't be longer than $2^{31} - 1$. The export function `\bitset_to_arabic:N` can use functions from the `int` module only if the largest index used for this bitset is smaller than 32, for longer bitsets `fp` is used and this is slower.

32.1 Creating bitsets

\bitset_new:N
\bitset_new:c
\bitset_new:Nn
\bitset_new:cn

New: 2023-11-15

```
\bitset_new:N  ⟨bitset var⟩
\bitset_new:Nn ⟨bitset var⟩
  {
    ⟨name1⟩ = ⟨index1⟩ ,
    ⟨name2⟩ = ⟨index2⟩ , ...
  }
```

Creates a new ⟨`bitset var`⟩ or raises an error if the name is already taken. The declaration is global. The ⟨`bitset var`⟩ is initially 0.

Bitsets are implemented as string variables consisting of `1`'s and `0`'s. The rightmost number is the index position 1, so the string variable can be viewed directly as the binary number. But one shouldn't rely on the internal representation, but use the dedicated `\bitset_to_bin:N` instead to get the binary number.

The name–index pairs given in the second argument of `\bitset_new:Nn` declares names for some indices, which can be used to set and unset bits. The names `1`, `2`, `3`, ... are predeclared and point to the index positions 1, 2, 3, ...

⟨`index...`⟩ should be a positive number or an ⟨`integer expression`⟩ which evaluates to a positive number. The expression is evaluated when the index is used, not at declaration time. The names ⟨`name...`⟩ should be unique. Using a number as name, e.g. `10=1`, is allowed, it then overwrites the predeclared name `10`, but the index position 10 can then only be reached if some other name for it exists, e.g. `ten=10`. It is not necessary to give every index a name, and an index can have more than one name. The named index can be extended or changed with the next function.

\bitset_addto_named_index:Nn

New: 2023-11-15

```
\bitset_addto_named_index:Nn ⟨bitset var⟩
  {
    ⟨name1⟩ = ⟨index1⟩ ,
    ⟨name2⟩ = ⟨index2⟩ , ...
  }
```

This extends or changes the name–index pairs for ⟨`bitset var`⟩ globally as described for `\bitset_new:Nn`.

For example after these settings

```
\bitset_new:Nn \l_pdfannot_F_bitset
  {
    Invisible      = 1,
    Hidden         = 2,
    Print          = 3,
    NoZoom         = 4,
    NoRotate       = 5,
    NoView         = 6,
    ReadOnly       = 7,
    Locked         = 8,
    ToggleNoView   = 9,
    LockedContents = 10
  }
\bitset_addto_named_index:Nn \l_pdfannot_F_bitset
  {
```

```
    print = 3
  }
```

it is possible to set bit 3 by using any of these alternatives:

```
\bitset_set_true:Nn \l_pdfannot_F_bitset {Print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {3}
```

\bitset_if_exist_p:N ⋆
\bitset_if_exist_p:c ⋆
\bitset_if_exist:NTF ⋆
\bitset_if_exist:cTF ⋆

New: 2023-11-15

\bitset_if_exist_p:N ⟨*bitset var*⟩
\bitset_if_exist:NTF ⟨*bitset var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨bitset var⟩ exist.

32.2 Setting and unsetting bits

\bitset_set_true:Nn
\bitset_set_true:cn
\bitset_gset_true:Nn
\bitset_gset_true:cn

New: 2023-11-15

\bitset_set_true:Nn ⟨*bitset var*⟩ {⟨*name*⟩}

This sets the bit of the index position represented by {⟨*name*⟩} to 1. ⟨name⟩ should be either one of the predeclared names 1, 2, 3, ..., or one of the names added manually. Index position are 1-based. If needed the length of the bit vector is enlarged.

\bitset_set_false:Nn
\bitset_set_false:cn
\bitset_gset_false:Nn
\bitset_gset_false:cn

New: 2023-11-15

\bitset_set_false:Nn ⟨*bitset var*⟩ {⟨*name*⟩}

This unsets the bit of the index position represented by {⟨*name*⟩} (sets it to 0). ⟨name⟩ should be either one of the predeclared names 1, 2, 3, ..., or one of the names added manually. The index is 1-based. If the index position is larger than the current length of the bit vector nothing happens. If the leading (left most) bit is unset, zeros are not trimmed but stay in the bit vector and are still shown by \bitset_show:N.

\bitset_clear:N
\bitset_clear:c
\bitset_gclear:N
\bitset_gclear:c

New: 2023-11-15

\bitset_clear:N ⟨*bitset var*⟩

This resets the bitset to the initial state. The declared names are not changed.

32.3 Using bitsets

\bitset_item:Nn ⋆
\bitset_item:cn ⋆

New: 2023-11-15

\bitset_item:Nn ⟨*bitset var*⟩ {⟨*name*⟩}

\bitset_item:Nn outputs 1 if the bit with the index number represented by ⟨name⟩ is set and 0 otherwise. ⟨name⟩ is either one of the predeclared names 1, 2, 3, ..., or one of the names added manually.

\bitset_to_bin:N ⋆
\bitset_to_bin:c ⋆

New: 2023-11-15

\bitset_to_bin:N ⟨*bitset var*⟩

This leaves the current value of the bitset expressed as a binary (string) number in the input stream. If no bit has been set yet, the output is zero.

\bitset_to_arabic:N ⋆
\bitset_to_arabic:c ⋆

New: 2023-11-15

\bitset_to_arabic:N ⟨*bitset var*⟩

This leaves the current value of the bitset expressed as a decimal number in the input stream. If no bit has been set yet, the output is zero. The function uses **\int_from_-bin:n** if the largest index that have been set or unset is smaller than 32, and a slower implementation based on **\fp_eval:n** otherwise.

\bitset_use:N ⋆
\bitset_use:c ⋆

New: 2024-11-12

\bitset_use:N ⟨*bitset var*⟩

This leaves the current value of the bitset expressed as a binary (string) number in the input stream. If no bit has been set yet, the output is zero. This is functionally equivalent to **\bitset_to_bin:N**.

\bitset_show:N
\bitset_show:c

New: 2023-11-15

\bitset_show:N ⟨*bitset var*⟩

Displays the binary and decimal values of the ⟨*bitset var*⟩ on the terminal.

\bitset_log:N
\bitset_log:c

New: 2023-11-15

\bitset_log:N ⟨*bitset var*⟩

Writes the binary and decimal values of the ⟨*bitset var*⟩ in the log file.

\bitset_show_named_index:N
\bitset_show_named_index:c

New: 2023-11-15

\bitset_show_named_index:N ⟨*bitset var*⟩

Displays declared name–index pairs of the ⟨*bitset var*⟩ on the terminal.

\bitset_log_named_index:N
\bitset_log_named_index:c

New: 2023-12-11

\bitset_log_named_index:N ⟨*bitset var*⟩

Writes declared name–index pairs of the ⟨*bitset var*⟩ in the log file.

Chapter 33

The l3cctab module
Category code tables

A category code table enables rapid switching of all category codes in one operation. For LuaTeX, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables. The implementation of category code tables in expl3 also saves and restores the TeX `\endlinechar` primitive value, meaning they could be used for example to implement `\ExplSyntaxOn`.

33.1 Creating and initializing category code tables

`\cctab_new:N`
`\cctab_new:c`

Updated: 2020-07-02

`\cctab_new:N` ⟨*category code table*⟩

Creates a new ⟨*category code table*⟩ variable or raises an error if the name is already taken. The declaration is global. The ⟨*category code table*⟩ is initialized with the codes as used by iniTeX.

`\cctab_const:Nn`
`\cctab_const:cn`

Updated: 2020-07-07

`\cctab_const:Nn` ⟨*category code table*⟩ {⟨*category code set up*⟩}

Creates a new ⟨*category code table*⟩, applies (in a group) the ⟨*category code set up*⟩ on top of iniTeX settings, then saves them globally as a constant table. The ⟨*category code set up*⟩ can include a call to `\cctab_select:N`.

`\cctab_gset:Nn`
`\cctab_gset:cn`

Updated: 2020-07-07

`\cctab_gset:Nn` ⟨*category code table*⟩ {⟨*category code set up*⟩}

Starting from the iniTeX category codes, applies (in a group) the ⟨*category code set up*⟩, then saves them globally in the ⟨*category code table*⟩. The ⟨*category code set up*⟩ can include a call to `\cctab_select:N`.

`\cctab_gsave_current:N`
`\cctab_gsave_current:c`

New: 2023-05-26

`\cctab_gsave_current:N` ⟨*category code table*⟩

Saves the current prevailing category codes in the ⟨*category code table*⟩.

33.2 Using category code tables

\cctab_begin:N
\cctab_begin:c

Updated: 2020-07-02

\cctab_begin:N ⟨*category code table*⟩

Switches locally the category codes in force to those stored in the ⟨*category code table*⟩. The prevailing codes before the function is called are added to a stack, for use with \cctab_end:. This function does not start a TeX group.

\cctab_end:

Updated: 2020-07-02

\cctab_end:

Ends the scope of a ⟨*category code table*⟩ started using \cctab_begin:N, returning the codes to those in force before the matching \cctab_begin:N was used. This must be used within the same TeX group and at the same TeX group level as the matching \cctab_begin:N.

\cctab_select:N
\cctab_select:c

New: 2020-05-19
Updated: 2020-07-02

\cctab_select:N ⟨*category code table*⟩

Selects the ⟨*category code table*⟩ for the scope of the current group. This is in particular useful in the ⟨*setup*⟩ arguments of \tl_set_rescan:Nnn, \tl_rescan:nn, \cctab_const:Nn, and \cctab_gset:Nn.

\cctab_item:Nn ⋆
\cctab_item:cn ⋆

New: 2021-05-10

\cctab_item:Nn ⟨*category code table*⟩ {⟨*int expr*⟩}

Determines the ⟨*character*⟩ with character code given by the ⟨*int expr*⟩ and expands to its category code specified by the ⟨*category code table*⟩.

33.3 Category code table conditionals

\cctab_if_exist_p:N ⋆
\cctab_if_exist_p:c ⋆
\cctab_if_exist:N*TF* ⋆
\cctab_if_exist:c*TF* ⋆

\cctab_if_exist_p:N ⟨*category code table*⟩
\cctab_if_exist:NTF ⟨*category code table*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*category code table*⟩ is currently defined. This does not check that the ⟨*category code table*⟩ really is a category code table.

33.4 Constant and scratch category code tables

\c_code_cctab

Updated: 2020-07-10

Category code table for the expl3 code environment; this does *not* include @, which is retained as an "other" character. Sets the \endlinechar value to 32 (a space).

\c_document_cctab

Updated: 2020-07-08

Category code table for a standard LaTeX document, as set by the LaTeX kernel. In particular, the upper-half of the 8-bit range will be set to "active" with pdfTeX *only*. No babel shorthands will be activated. Sets the \endlinechar value to 13 (normal line ending).

`\c_initex_cctab` Category code table as set up by iniTeX.

Updated: 2020-07-02

`\c_other_cctab` Category code table where all characters have category code 12 (other). Sets the `\endlinechar` value to −1.

Updated: 2020-07-02

`\c_str_cctab` Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space). Sets the `\endlinechar` value to −1.

Updated: 2020-07-02

`\g_tmpa_cctab`
`\g_tmpb_cctab` Scratch category code tables.

New: 2023-05-26

Part V

Text manipulation

Chapter 34

The **l3unicode** module Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. Most of the code here is internal, but there are a small set of public functions. These work with Unicode ⟨*codepoints*⟩ and are designed to give usable results with both Unicode-aware and 8-bit engines.

\codepoint_generate:nn ⋆ `\codepoint_generate:nn {⟨codepoint⟩} {⟨catcode⟩}`

New: 2022-10-09
Updated: 2022-11-09

Generates one or more character tokens representing the ⟨*codepoint*⟩. With Unicode engines, exactly one character token will be generated, and this will have the ⟨*catcode*⟩ specified as the second argument:

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the ⟨*codepoint*⟩. For all codepoints outside of the classical ASCII range, the generated character tokens will be active (category code 13); for codepoints in the ASCII range, the given ⟨*catcode*⟩ will be used. To allow the result of this function to be used inside an expansion context, the result is protected by `\exp_not:n`.

TEXhackers note: Users of (u)pTEX note that these engines are treated as 8-bit in this context. In particular, for upTEX, irrespective of the `\kcatcode` of the ⟨*codepoint*⟩, any value outside the ASCII range will result in a series of active bytes being generated.

\codepoint_str_generate:n ⋆ `\codepoint_str_generate:n {⟨codepoint⟩}`

New: 2022-10-09

Generates one or more character tokens representing the ⟨*codepoint*⟩. With Unicode engines, exactly one character token will be generated. For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the ⟨*codepoint*⟩. All of the generated character tokens will be of category code 12, except any spaces (codepoint 32), which will be category code 10.

`\codepoint_to_category:n` ⋆ `\codepoint_to_category:n {⟨codepoint⟩}`

New: 2023-06-19

Expands to the Unicode general category identifier of the ⟨*codepoint*⟩. The general category identifier is a string made up of two letter characters, the first uppercase and the second lowercase. The uppercase letters divide codepoints into broader groups, which are then refined by the lowercase letter. For example, codepoints representing letters all have identifiers starting `L`, for example `Lu` (uppercase letter), `Lt` (titlecase letter), etc. Full details are available in the documentation provided by the Unicode Consortium: see https://www.unicode.org/reports/tr44/#General_Category_Values

`\codepoint_to_nfd:n` ⋆ `\codepoint_to_nfd:n {⟨codepoint⟩}`

New: 2022-10-09

Converts the ⟨*codepoint*⟩ to the Unicode Normalization Form Canonical Decomposition. The generated character(s) will have the current category code as they would if typed in directly for Unicode engines; for 8-bit engines, active characters are used for all codepoints outside of the ASCII range.

Chapter 35

The l3text module
Text processing

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the ⟨*text*⟩ are normalized and become `{` and `}`, respectively.

35.1 Expanding text

`\text_expand:n` ⋆

Updated: 2023-06-09

`\text_expand:n {⟨text⟩}`

Takes user input ⟨*text*⟩ and expands the content. Protected commands (typically formatting) are left in place, and no processing of math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) takes place. Commands which are neither engine- nor LaTeX-protected are expanded exhaustively. Any commands listed in `\l_text_expand_exclude_tl` are excluded from expansion, as are those in `\l_text_case_exclude_arg_tl` and `\l_text_math_arg_tl`.

`\text_declare_expand_equivalent:Nn`
`\text_declare_expand_equivalent:cn`

`\text_declare_expand_equivalent:Nn ⟨cmd⟩ {⟨replacement⟩}`

Declares that the ⟨*replacement*⟩ tokens should be used whenever the ⟨*cmd*⟩ (a single token) is encountered. The ⟨*replacement*⟩ tokens should be expandable. A token can be "replaced" by itself if the defined replacement wraps it in `\exp_not:n`, for example

```
\text_declare_expand_equivalent:Nn \' { \exp_not:n { \' } }
```

35.2 Case changing

\text_lowercase:n ⋆
\text_uppercase:n ⋆
\text_titlecase_all:n ⋆
\text_titlecase_first:n ⋆
\text_lowercase:nn ⋆
\text_uppercase:nn ⋆
\text_titlecase_all:nn ⋆
\text_titlecase_first:nn ⋆

Updated: 2023-07-08

`\text_uppercase:n {⟨tokens⟩}`
`\text_uppercase:nn {⟨BCP-47⟩} {⟨tokens⟩}`

Takes user input ⟨*text*⟩ first applies `\text_expand:n`, then transforms the case of character tokens as specified by the function name. The category code of letters are not changed by this process when Unicode engines are used; in 8-bit engines, case changed charters in the ASCII range will have the current prevailing category code, while those outside of it will be represented by active characters.

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first *non-space* character of the ⟨*tokens*⟩ to uppercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example `ij` in Dutch which becomes `IJ`. There are two functions available for titlecasing: one which applies the change to each "word" and a second which only applies at the start of the input. (Here, "word" boundaries are spaces: at present, full Unicode word breaking is not attempted.)

Importantly, notice that these functions are intended for working with user *text for typesetting*. For case changing programmatic data see the l3str module and discussion there of `\str_lowercase:n`, `\str_uppercase:n` and `\str_casefold:n`.

Case changing does not take place within math mode material so for example

```
\text_uppercase:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

The first mandatory argument of commands listed in `\l_text_case_exclude_arg_tl` is excluded from case changing; the latter are entirely non-textual content (such as labels).

The standard mappings here follow those defined by the Unicode Consortium in `UnicodeData.txt` and `SpecialCasing.txt`. For pTEX, only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Locale-sensitive conversions are enabled using the ⟨*BCP-47*⟩ argument, and follow Unicode Consortium guidelines. Currently, the locale strings recognized for special handling are as follows.

- Armenian (`hy` and `hy-x-yiwn`) The setting `hy` maps the codepoint U+0587, the ligature of letters ech and yiwn, to the codepoints for capital ech and vew when uppercasing: this follows the spelling reform which is used in Armenia. The alternative `hy-x-yiwn` maps U+0587 to capital ech and yiwn on uppercasing (also the output if Armenian is not selected at all).
- Azeri and Turkish (`az` and `tr`). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lowercasing I-dot and introduced when upper casing i-dotless.
- German (`de-x-eszett`). An alternative mapping for German in which the lowercase *Eszett* maps to a *großes Eszett*.

- Greek (el). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. A variant el-x-iota is available which converts the *ypogegrammeni* (subscript muted iota) to capital iota when uppercasing: the standard version retains the subscript versions.
- Lithuanian (lt). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that *only* the accents used in Lithuanian are covered: the behavior of other accents are not modified.
- Medieval Latin (la-x-medieval). The characters u and V are interchanged on case changing.
- Dutch (nl). Capitalization of ij at the beginning of titlecased input produces IJ rather than Ij.

Determining whether non-letter characters at the start of text should count as the uppercase element is controllable. When \l_text_titlecase_check_letter_bool is true, codepoints which are not letters (Unicode general category L) are not changed, and only the first *letter* is uppercased. When \l_text_titlecase_check_letter_bool is false, the first codepoint is uppercased, irrespective of the general code of the character.

\text_declare_case_equivalent:Nn

New: 2022-07-04

\text_declare_case_equivalent:Nn ⟨*cmd*⟩ {⟨*replacement*⟩}

Declares that the ⟨*replacement*⟩ tokens should be used whenever the ⟨*cmd*⟩ (a single token) is encountered during case changing.

\text_declare_lowercase_mapping:nn
\text_declare_lowercase_mapping:nnn
\text_declare_titlecase_mapping:nn
\text_declare_titlecase_mapping:nnn
\text_declare_uppercase_mapping:nn
\text_declare_uppercase_mapping:nnn

New: 2023-04-11
Updated: 2023-04-20

\text_declare_lowercase_mapping:nn {⟨*codepoint*⟩} {⟨*replacement*⟩}
\text_declare_lowercase_mapping:nnn {⟨*BCP-47*⟩} {⟨*codepoint*⟩} {⟨*replacement*⟩}

Declares that the ⟨*replacement*⟩ tokens should be used when case mapping the ⟨*codepoint*⟩, rather than the standard mapping given in the Unicode data files. The nnn version takes a BCP-47 tag, which can be used to specify that the customization only applies to that locale.

\text_declare_lowercase_exclusion:n
\text_declare_titlecase_exclusion:n
\text_declare_uppercase_exclusion:n

New: 2025-06-24

\text_declare_lowercase_exclusion:n {⟨*word*⟩}

Declares that the ⟨*word*⟩ is excluded from case changing for the appropriate operation.

`\text_case_switch:nnnn` ⋆ `\text_case_switch:nnnn {⟨normal⟩} {⟨upper⟩} {⟨lower⟩} {⟨title⟩}`

New: 2022-07-04

Context-sensitive function which will expand to one of the ⟨*normal*⟩, ⟨*upper*⟩, ⟨*lower*⟩ or ⟨*title*⟩ tokens depending on the current case changing operation. Outside of case changing, the ⟨*normal*⟩ tokens are produced. Within case changing, the appropriate mapping tokens are inserted.

35.3 Removing formatting from text

`\text_purify:n` ⋆ `\text_purify:n {⟨text⟩}`

New: 2020-03-05
Updated: 2020-05-14

Takes user input ⟨*text*⟩ and expands as described for `\text_expand:n`, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) is left contained in a pair of `$` delimiters. Non-expandable functions present in the ⟨*text*⟩ must either have a defined equivalent (see `\text_declare_purify_equivalent:Nn`) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

`\text_declare_purify_equivalent:Nn` `\text_declare_purify_equivalent:Nn ⟨cmd⟩ {⟨replacement⟩}`
`\text_declare_purify_equivalent:Ne`

New: 2020-03-05

Declares that the ⟨*replacement*⟩ tokens should be used whenever the ⟨*cmd*⟩ (a single token) is encountered. The ⟨*replacement*⟩ tokens should be expandable.

35.4 Control variables

`\l_text_math_arg_tl`

Lists commands present in the ⟨*text*⟩ where the argument of the command should be treated as math mode material. The treatment here is similar to `\l_text_math_delims_tl` but for a command rather than paired delimiters.

`\l_text_math_delims_tl`

Lists pairs of tokens which delimit (in-line) math mode content; such content *may* be excluded from processing.

`\l_text_case_exclude_arg_tl`

Lists commands where the first mandatory argument is excluded from case changing.

`\l_text_expand_exclude_tl`

Lists commands which are excluded from expansion. This protection includes everything up to and including their first braced argument.

`\l_text_titlecase_check_letter_bool`

Controls how the start of titlecasing is handled: when `true`, the first *letter* in text is considered. The standard setting is `true`.

35.5 Mapping to text

Grapheme splitting is implemented using the algorithm described in Unicode Standard Annex #29. This includes support for extended grapheme clusters. Leading line feeds or carriage returns will be dropped due to standard TeX processing. At present extended pictograms are not supported: these may be added in a future release. Some aspects of Indic grapheme breaking, introduced in Unicode 15, are also currently absent. In these functions, math mode is treated as a single indivisible unit, i.e. one "word" or grapheme.

`\text_map_function:nN` ☆

New: 2022-08-04
Updated: 2025-07-11

`\text_map_function:nN {⟨text⟩} ⟨function⟩`

This takes the user input in ⟨text⟩ and expands it as with `\text_expand:n`; it then maps over the *graphemes* within the result, passing each grapheme to the ⟨function⟩. Broadly a grapheme is a "user perceived character": the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The ⟨function⟩ should accept one argument as ⟨balanced text⟩: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_inline:nn`.

`\text_map_tokens:nn` ☆

New: 2025-06-22
Updated: 2025-07-11

`\text_map_tokens:nn {⟨text⟩} {⟨code⟩}`

This takes the user input in ⟨text⟩ and expands it as with `\text_expand:n`; it then maps over the *graphemes* within the result, passing each grapheme to the ⟨code⟩ as a trailing brace group. Broadly a grapheme is a "user perceived character": the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The ⟨code⟩ should accept one argument as ⟨balanced text⟩: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_inline:nn`.

`\text_map_inline:nn`

New: 2022-08-04
Updated: 2025-07-11

`\text_map_inline:nn {⟨text⟩} {⟨inline function⟩}`

This takes the user input in ⟨text⟩ and expands it as with `\text_expand:n`; it then maps over the *graphemes* within the result, passing each grapheme to the ⟨inline function⟩. Broadly a grapheme is a "user perceived character": the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The ⟨inline function⟩ should consist of code which receives the grapheme as ⟨balanced text⟩: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_function:nN`.

Word breaking is implemented using the standard algorithm described in Unicode Standard Annex #29. Leading line feeds or carriage returns will be dropped due to standard TeX processing. Spaces are always considered a break even if immediately followed by an extending character. At present extended pictograms are not supported: these may be added in a future release. Language-based tailoring may be added in a future release: at present the algorithm is exactly that described in the annex.

\text_words_map_function:nN ☆ \text_words_map_function:nN {⟨*text*⟩} ⟨*function*⟩

New: 2025-02-12
Updated: 2025-07-11

This takes the user input in ⟨*text*⟩ and expands it as with \text_expand:n; it then maps over the *words* within the result, passing each word to the ⟨*function*⟩. Word boundaries are determined using the standard algorithm described by the Unicode Consortium. The ⟨*function*⟩ should accept one argument as ⟨*balanced text*⟩: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also \text_words_map_inline:nn.

\text_words_map_tokens:nn ☆ \text_words_map_tokens:nn {⟨*text*⟩} {⟨*code*⟩}

New: 2025-06-22
Updated: 2025-07-11

This takes the user input in ⟨*text*⟩ and expands it as with \text_expand:n; it then maps over the *words* within the result, passing each word to the ⟨*code*⟩ as a trailing brace group. Word boundaries are determined using the standard algorithm described by the Unicode Consortium. The ⟨*code*⟩ should accept one argument as ⟨*balanced text*⟩: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also \text_words_map_inline:nn.

\text_words_map_inline:nn ☆ \text_words_map_inline:nn {⟨*text*⟩} {⟨*inline function*⟩}

New: 2025-02-12
Updated: 2025-07-11

This takes the user input in ⟨*text*⟩ and expands it as with \text_expand:n; it then maps over the *words* within the result, passing each word to the ⟨*function*⟩. Word boundaries are determined using the standard algorithm described by the Unicode Consortium. The ⟨*inline function*⟩ should consist of code that will accept one argument as ⟨*balanced text*⟩: this may comprise codepoints or it may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also \text_words_map_function:nN.

\text_map_break: ☆
\text_map_break:n ☆

\text_map_break:
\text_map_break:n {⟨*code*⟩}

New: 2022-08-04

Used to terminate a \text_map_... or \text_words_map_... function before all entries in the ⟨*text*⟩ have been processed. This normally takes place within a conditional statement.

Part VI

Typesetting

Chapter 36

The l3box module
Boxes

Box variables contain typeset material that can be inserted on the page or in other boxes. Their contents cannot be converted back to lists of tokens. There are three kinds of box operations: horizontal mode denoted with prefix \hbox_, vertical mode with prefix \vbox_, and the generic operations working in both modes with prefix \box_. For instance, a new box variable containing the words "Hello, world!" (in a horizontal box) can be obtained by the following code.

```
\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, ~ world! }
```

The argument is typeset inside a TeX group so that any variables assigned during the construction of this box restores its value afterwards.

Box variables from l3box are compatible with those of LaTeX 2ε and plain TeX and can be used interchangeably. The l3box commands to construct boxes, such as \hbox:n or \hbox_set:Nn, are "color-safe", meaning that

```
\hbox:n { \color_select:n { blue } Hello, } ~ world!
```

will result in "Hello," taking the color blue, but "world!" remaining with the prevailing color outside the box.

36.1 Creating and initializing boxes

\box_new:N
\box_new:c

\box_new:N ⟨*box*⟩

Creates a new ⟨*box*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*box*⟩ is initially void.

\box_clear:N
\box_clear:c
\box_gclear:N
\box_gclear:c

\box_clear:N ⟨*box*⟩

Clears the content of the ⟨*box*⟩ by setting the box equal to \c_empty_box.

\box_clear_new:N
\box_clear_new:c
\box_gclear_new:N
\box_gclear_new:c

\box_clear_new:N ⟨*box*⟩

Ensures that the ⟨*box*⟩ exists globally by applying \box_new:N if necessary, then applies \box_(g)clear:N to leave the ⟨*box*⟩ empty.

\box_set_eq:NN
\box_set_eq:(cN|Nc|cc)
\box_gset_eq:NN
\box_gset_eq:(cN|Nc|cc)

\box_set_eq:NN ⟨box_1⟩ ⟨box_2⟩

Sets the content of ⟨box_1⟩ equal to that of ⟨box_2⟩.

\box_if_exist_p:N ⋆
\box_if_exist_p:c ⋆
\box_if_exist:N*TF* ⋆
\box_if_exist:c*TF* ⋆

\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.

36.2 Using boxes

\box_use:N
\box_use:c

\box_use:N ⟨*box*⟩

Inserts the current content of the ⟨*box*⟩ onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

TEXhackers note: This is the TEX primitive \copy.

\box_move_right:nn
\box_move_left:nn

\box_move_right:nn {⟨*dim expr*⟩} {⟨*box function*⟩}

This function operates in vertical mode, and inserts the material specified by the ⟨*box function*⟩ such that its reference point is displaced horizontally by the given ⟨*dim expr*⟩ from the reference point for typesetting, to the right or left as appropriate. The ⟨*box function*⟩ should be a box operation such as \box_use:N \<box> or a “raw” box specification such as \vbox:n { xyz }.

\box_move_up:nn
\box_move_down:nn

\box_move_up:nn {⟨*dim expr*⟩} {⟨*box function*⟩}

This function operates in horizontal mode, and inserts the material specified by the ⟨*box function*⟩ such that its reference point is displaced vertically by the given ⟨*dim expr*⟩ from the reference point for typesetting, up or down as appropriate. The ⟨*box function*⟩ should be a box operation such as \box_use:N \<box> or a “raw” box specification such as \vbox:n { xyz }.

36.3 Measuring and setting box dimensions

\box_dp:N
\box_dp:c

\box_dp:N ⟨*box*⟩

Calculates the depth (below the baseline) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

TEXhackers note: This is the TEX primitive \dp.

\box_ht:N
\box_ht:c

\box_ht:N ⟨*box*⟩

Calculates the height (above the baseline) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

TeXhackers note: This is the TeX primitive \ht.

\box_wd:N
\box_wd:c

\box_wd:N ⟨*box*⟩

Calculates the width of the ⟨*box*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

TeXhackers note: This is the TeX primitive \wd.

\box_ht_plus_dp:N
\box_ht_plus_dp:c

New: 2021-05-05

\box_ht_plus_dp:N ⟨*box*⟩

Calculates the total vertical size (height plus depth) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

\box_set_dp:Nn
\box_set_dp:cn
\box_gset_dp:Nn
\box_gset_dp:cn

\box_set_dp:Nn ⟨*box*⟩ {⟨*dim expr*⟩}

Set the depth (below the baseline) of the ⟨*box*⟩ to the value of the {⟨*dim expr*⟩}.

\box_set_ht:Nn
\box_set_ht:cn
\box_gset_ht:Nn
\box_gset_ht:cn

\box_set_ht:Nn ⟨*box*⟩ {⟨*dim expr*⟩}

Set the height (above the baseline) of the ⟨*box*⟩ to the value of the {⟨*dim expr*⟩}.

\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn

\box_set_wd:Nn ⟨*box*⟩ {⟨*dim expr*⟩}

Set the width of the ⟨*box*⟩ to the value of the {⟨*dim expr*⟩}.

36.4 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.

36.5 The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

\box_set_to_last:N ⟨*box*⟩

Sets the ⟨*box*⟩ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the ⟨*box*⟩ is always void as it is not possible to recover the last added item.

36.6 Constant boxes

\c_empty_box

This is a permanently empty box, which is neither set as horizontal nor vertical.

TeXhackers note: At the TeX level this is a void box.

36.7 Scratch boxes

\l_tmpa_box
\l_tmpb_box

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.

36.8 Viewing box contents

\box_show:N
\box_show:c

\box_show:N ⟨*box*⟩

Shows full details of the content of the ⟨*box*⟩ in the terminal.

\box_show:Nnn
\box_show:cnn

\box_show:Nnn ⟨*box*⟩ {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}

Display the contents of ⟨*box*⟩ in the terminal, showing the first ⟨*int expr*$_1$⟩ items of the box, and descending into ⟨*int expr*$_2$⟩ group levels.

\box_log:N
\box_log:c

\box_log:N ⟨*box*⟩

Writes full details of the content of the ⟨*box*⟩ to the log.

\box_log:Nnn
\box_log:cnn

\box_log:Nnn ⟨*box*⟩ {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}

Writes the contents of ⟨*box*⟩ to the log, showing the first ⟨*int expr*$_1$⟩ items of the box, and descending into ⟨*int expr*$_2$⟩ group levels.

36.9 Boxes and color

All LaTeX3 boxes are "color safe": a color set inside the box stops applying after the end of the box has occurred.

36.10 Horizontal mode boxes

`\hbox:n`

`\hbox:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of natural width and then includes this box in the current list for typesetting.

`\hbox_to_wd:nn`

`\hbox_to_wd:nn {⟨dim expr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of width ⟨*dim expr*⟩ and then includes this box in the current list for typesetting.

`\hbox_to_zero:n`

`\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⟩ {⟨dim expr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ to the width given by the ⟨*dim expr*⟩ and then stores the result inside the ⟨*box*⟩.

`\hbox_overlap_center:n`

New: 2020-08-25

`\hbox_overlap_center:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material protrudes equally to both sides of the insertion point.

`\hbox_overlap_right:n`

`\hbox_overlap_right:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material protrudes to the right of the insertion point.

`\hbox_overlap_left:n`

`\hbox_overlap_left:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material protrudes to the left of the insertion point.

`\hbox_set:Nw`
`\hbox_set:cw`
`\hbox_set_end:`
`\hbox_gset:Nw`
`\hbox_gset:cw`
`\hbox_gset_end:`

`\hbox_set:Nw ⟨box⟩ ⟨contents⟩ \hbox_set_end:`

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩. In contrast to `\hbox_set:Nn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

\hbox_set_to_wd:Nnw
\hbox_set_to_wd:cnw
\hbox_gset_to_wd:Nnw
\hbox_gset_to_wd:cnw

`\hbox_set_to_wd:Nnw ⟨box⟩ {⟨dim expr⟩} ⟨contents⟩ \hbox_set_end:`

Typesets the ⟨*contents*⟩ to the width given by the ⟨*dim expr*⟩ and then stores the result inside the ⟨*box*⟩. In contrast to `\hbox_set_to_wd:Nnn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument

\hbox_unpack:N
\hbox_unpack:c

`\hbox_unpack:N ⟨box⟩`

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set.

TEXhackers note: This is the TEX primitive `\unhcopy`.

36.11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

\vbox:n

`\vbox:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of natural height and includes this box in the current list for typesetting.

\vbox_top:n

`\vbox_top:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the *first* item added to the box.

\vbox_to_ht:nn

`\vbox_to_ht:nn {⟨dim expr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of height ⟨*dim expr*⟩ and then includes this box in the current list for typesetting.

\vbox_to_zero:n

`\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

`\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

`\vbox_set_top:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. The baseline of the box is equal to that of the *first* item added to the box.

\vbox_set_to_ht:Nnn
\vbox_set_to_ht:cnn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cnn

\vbox_set_to_ht:Nnn ⟨*box*⟩ {⟨*dim expr*⟩} {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ to the height given by the ⟨*dim expr*⟩ and then stores the result inside the ⟨*box*⟩.

\vbox_set:Nw
\vbox_set:cw
\vbox_set_end:
\vbox_gset:Nw
\vbox_gset:cw
\vbox_gset_end:

\vbox_set:Nw ⟨*box*⟩ ⟨*contents*⟩ \vbox_set_end:

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. In contrast to \vbox_set:Nn this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

\vbox_set_to_ht:Nnw
\vbox_set_to_ht:cnw
\vbox_gset_to_ht:Nnw
\vbox_gset_to_ht:cnw

\vbox_set_to_ht:Nnw ⟨*box*⟩ {⟨*dim expr*⟩} ⟨*contents*⟩ \vbox_set_end:

Typesets the ⟨*contents*⟩ to the height given by the ⟨*dim expr*⟩ and then stores the result inside the ⟨*box*⟩. In contrast to \vbox_set_to_ht:Nnn this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument

\vbox_set_split_to_ht:NNn
\vbox_set_split_to_ht:(cNn|Ncn|ccn)
\vbox_gset_split_to_ht:NNn
\vbox_gset_split_to_ht:(cNn|Ncn|ccn)

\vbox_set_split_to_ht:NNn ⟨box_1⟩ ⟨box_2⟩ {⟨*dim expr*⟩}

Sets ⟨box_1⟩ to contain material to the height given by the ⟨*dim expr*⟩ by removing content from the top of ⟨box_2⟩ (which must be a vertical box).

\vbox_unpack:N
\vbox_unpack:c

\vbox_unpack:N ⟨*box*⟩

Unpacks the content of the vertical ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set.

TEXhackers note: This is the TEX primitive \unvcopy.

36.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other expl3 variable. However, for efficiency reasons, it is also useful to have functions which *drop* box contents on use. When a box is dropped, the box becomes empty at the group level *where the box was originally set* rather than necessarily *at the current group level*. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
  \hbox_set:Nn \l_tmpa_box { B }
   \group_begin:
   \box_use_drop:N \l_tmpa_box
  \group_end:
  \box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
```

the first use of \box_show:N will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behavior of drop functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

\box_use_drop:N
\box_use_drop:c

\box_use_drop:N ⟨box⟩

Inserts the current content of the ⟨box⟩ onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

TEXhackers note: This is the TEX primitive \box.

\box_set_eq_drop:NN
\box_set_eq_drop:(cN|Nc|cc)

\box_set_eq_drop:NN ⟨box_1⟩ ⟨box_2⟩

Sets the content of ⟨box_1⟩ equal to that of ⟨box_2⟩, then drops ⟨box_2⟩.

\box_gset_eq_drop:NN
\box_gset_eq_drop:(cN|Nc|cc)

\box_gset_eq_drop:NN ⟨box_1⟩ ⟨box_2⟩

Sets the content of ⟨box_1⟩ globally equal to that of ⟨box_2⟩, then drops ⟨box_2⟩.

\hbox_unpack_drop:N
\hbox_unpack_drop:c

\hbox_unpack_drop:N ⟨box⟩

Unpacks the content of the horizontal ⟨box⟩, retaining any stretching or shrinking applied when the ⟨box⟩ was set. The original ⟨box⟩ is then dropped.

TEXhackers note: This is the TEX primitive \unhbox.

\vbox_unpack_drop:N
\vbox_unpack_drop:c

\vbox_unpack_drop:N ⟨box⟩

Unpacks the content of the vertical ⟨box⟩, retaining any stretching or shrinking applied when the ⟨box⟩ was set. The original ⟨box⟩ is then dropped.

TEXhackers note: This is the TEX primitive \unvbox.

36.13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in TEX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

\box_autosize_to_wd_and_ht:Nn
\box_autosize_to_wd_and_ht:cnn
\box_gautosize_to_wd_and_ht:Nnn
\box_gautosize_to_wd_and_ht:cnn

`\box_autosize_to_wd_and_ht:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}`

Resizes the ⟨box⟩ to fit within the given ⟨x-size⟩ (horizontally) and ⟨y-size⟩ (vertically); both of the sizes are dimension expressions. The ⟨y-size⟩ is the height only: it does not include any depth. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. The final size of the ⟨box⟩ is the smaller of {⟨x-size⟩} and {⟨y-size⟩}, i.e., the result fits within the dimensions specified. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_autosize_to_wd_and_ht_plus_dp:Nnn
\box_autosize_to_wd_and_ht_plus_dp:cnn
\box_gautosize_to_wd_and_ht_plus_dp:Nnn
\box_gautosize_to_wd_and_ht_plus_dp:cnn

`\box_autosize_to_wd_and_ht_plus_dp:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}`

Resizes the ⟨box⟩ to fit within the given ⟨x-size⟩ (horizontally) and ⟨y-size⟩ (vertically); both of the sizes are dimension expressions. The ⟨y-size⟩ is the total vertical size (height plus depth). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. The final size of the ⟨box⟩ is the smaller of {⟨x-size⟩} and {⟨y-size⟩}, i.e., the result fits within the dimensions specified. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_ht:Nn
\box_resize_to_ht:cn
\box_gresize_to_ht:Nn
\box_gresize_to_ht:cn

`\box_resize_to_ht:Nn ⟨box⟩ {⟨y-size⟩}`

Resizes the ⟨box⟩ to ⟨y-size⟩ (vertically), scaling the horizontal size by the same amount; ⟨y-size⟩ is a dimension expression. The ⟨y-size⟩ is the height only: it does not include any depth. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative ⟨y-size⟩ causes the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_gresize_to_ht_plus_dp:Nn
\box_gresize_to_ht_plus_dp:cn

`\box_resize_to_ht_plus_dp:Nn ⟨box⟩ {⟨y-size⟩}`

Resizes the ⟨box⟩ to ⟨y-size⟩ (vertically), scaling the horizontal size by the same amount; ⟨y-size⟩ is a dimension expression. The ⟨y-size⟩ is the total vertical size (height plus depth). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative ⟨y-size⟩ causes the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_wd:Nn
\box_resize_to_wd:cn
\box_gresize_to_wd:Nn
\box_gresize_to_wd:cn

\box_resize_to_wd:Nn ⟨box⟩ {⟨x-size⟩}

Resizes the ⟨box⟩ to ⟨x-size⟩ (horizontally), scaling the vertical size by the same amount; ⟨x-size⟩ is a dimension expression. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative ⟨x-size⟩ causes the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨x-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_wd_and_ht:Nnn
\box_resize_to_wd_and_ht:cnn
\box_gresize_to_wd_and_ht:Nnn
\box_gresize_to_wd_and_ht:cnn

\box_resize_to_wd_and_ht:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}

Resizes the ⟨box⟩ to ⟨x-size⟩ (horizontally) and ⟨y-size⟩ (vertically): both of the sizes are dimension expressions. The ⟨y-size⟩ is the height only and does not include any depth. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_wd_and_ht_plus_dp:Nnn
\box_resize_to_wd_and_ht_plus_dp:cnn
\box_gresize_to_wd_and_ht_plus_dp:Nnn
\box_gresize_to_wd_and_ht_plus_dp:cnn

\box_resize_to_wd_and_ht_plus_dp:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}

Resizes the ⟨box⟩ to ⟨x-size⟩ (horizontally) and ⟨y-size⟩ (vertically): both of the sizes are dimension expressions. The ⟨y-size⟩ is the total vertical size (height plus depth). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_rotate:Nn
\box_rotate:cn
\box_grotate:Nn
\box_grotate:cn

\box_rotate:Nn ⟨box⟩ {⟨angle⟩}

Rotates the ⟨box⟩ by ⟨angle⟩ (a ⟨fp expr⟩ in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the rotation is applied.

\box_scale:Nnn
\box_scale:cnn
\box_gscale:Nnn
\box_gscale:cnn

\box_scale:Nnn ⟨box⟩ {⟨x-scale⟩} {⟨y-scale⟩}

Scales the ⟨box⟩ by factors ⟨x-scale⟩ and ⟨y-scale⟩ in the horizontal and vertical directions, respectively (both scales are ⟨fp expr⟩). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the scaling is applied. Negative scalings cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-scale⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

36.14 Viewing part of a box

`\box_set_clipped:N`
`\box_set_clipped:c`
`\box_gset_clipped:N`
`\box_gset_clipped:c`

Updated: 2023-04-14

`\box_set_clipped:N` ⟨*box*⟩

Clips the ⟨*box*⟩ in the output so that only material inside the bounding box is displayed in the output. The updated ⟨*box*⟩ is an hbox, irrespective of the nature of the ⟨*box*⟩ before the clipping is applied. Additional box levels are also generated by this operation.

TEXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

`\box_set_trim:Nnnnn`
`\box_set_trim:cnnnn`
`\box_gset_trim:Nnnnn`
`\box_gset_trim:cnnnn`

`\box_set_trim:Nnnnn` ⟨*box*⟩ {⟨*left*⟩} {⟨*bottom*⟩} {⟨*right*⟩} {⟨*top*⟩}

Adjusts the bounding box of the ⟨*box*⟩: ⟨*left*⟩ is removed from the left-hand edge of the bounding box, ⟨*right*⟩ from the right-hand edge, and so forth. All adjustments are ⟨*dim exprs*⟩. Material outside of the bounding box is still displayed in the output unless `\box_set_clipped:N` is subsequently applied. The updated ⟨*box*⟩ is an hbox, irrespective of the nature of the ⟨*box*⟩ before the trim operation is applied. Additional box levels are also generated by this operation. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

`\box_set_viewport:Nnnnn`
`\box_set_viewport:cnnnn`
`\box_gset_viewport:Nnnnn`
`\box_gset_viewport:cnnnn`

`\box_set_viewport:Nnnnn` ⟨*box*⟩ {⟨*llx*⟩} {⟨*lly*⟩} {⟨*urx*⟩} {⟨*ury*⟩}

Adjusts the bounding box of the ⟨*box*⟩ such that it has lower-left coordinates (⟨*llx*⟩, ⟨*lly*⟩) and upper-right coordinates (⟨*urx*⟩, ⟨*ury*⟩). All four coordinate positions are ⟨*dim exprs*⟩. Material outside of the bounding box is still displayed in the output unless `\box_set_clipped:N` is subsequently applied. The updated ⟨*box*⟩ is an hbox, irrespective of the nature of the ⟨*box*⟩ before the viewport operation is applied. Additional box levels are also generated by this operation.

36.15 Primitive box conditionals

`\if_hbox:N` ⋆

```
\if_hbox:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is a horizontal box.

TEXhackers note: This is the TEX primitive `\ifhbox`.

`\if_vbox:N` ⋆

```
\if_vbox:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is a vertical box.

> **TEXhackers note:** This is the TEX primitive `\ifvbox`.

`\if_box_empty:N` ⋆

```
\if_box_empty:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is an empty (void) box.

> **TEXhackers note:** This is the TEX primitive `\ifvoid`.

Chapter 37

The l3coffins module
Coffin code layer

In expl3 terminology, a "coffin" is a box containing typeset material. Along with the box itself, the coffin structure includes information on the size and shape of the box, which makes it possible to align two or more coffins easily. This is achieved by providing a series of "poles" for each coffin. These are horizontal and vertical lines through the coffin at defined positions, for example the top or horizontal center. The points where these poles intersect are called "handles". Two coffins can then be aligned by describing the relationship between a handle on one coffin with a handle on the second. In words, an example might then read

> Align the top-left handle of coffin A with the bottom-right handle of coffin B.

The locations of coffin handles are much easier to understand visually. Figure 1 shows the standard handle positions for a coffin typeset in horizontal mode (left) and in vertical mode (right). Notice that the later case results in a greater number of handles being available. As illustrated, each handle results from the intersection of two poles. For example, the center of the coffin is marked `(hc,vc)`, i.e., it is the point of intersection of the horizontal center pole with the vertical center pole. New handles are generated automatically when poles are added to a coffin: handles are "dynamic" entities.

37.1 Controlling coffin poles

A number of standard poles are automatically generated when the coffin is set or an alignment takes place. The standard poles for all coffins are:

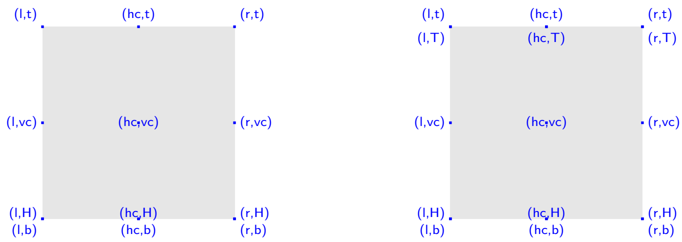

Figure 1: Standard coffin handles: left, horizontal coffin; right, vertical coffin

- l a pole running along the left-hand edge of the bounding box of the coffin;
- hc a pole running vertically through the center of the coffin half-way between the left- and right-hand edges of the bounding box (i.e., the "horizontal center");
- r a pole running along the right-hand edge of the bounding box of the coffin;
- b a pole running along the bottom edge of the bounding box of the coffin;
- vc a pole running horizontally through the center of the coffin half-way between the bottom and top edges of the bounding box (i.e., the "vertical center");
- t a pole running along the top edge of the bounding box of the coffin;
- H a pole running along the baseline of the typeset material contained in the coffin.

In addition, coffins containing vertical-mode material also feature poles which reflect the richer nature of these systems:

- B a pole running along the baseline of the material at the bottom of the coffin.
- T a pole running along the baseline of the material at the top of the coffin.

37.2 Creating and initializing coffins

`\coffin_new:N`
`\coffin_new:c`

`\coffin_new:N ⟨coffin⟩`

Creates a new ⟨*coffin*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*coffin*⟩ is initially empty.

`\coffin_clear:N`
`\coffin_clear:c`
`\coffin_gclear:N`
`\coffin_gclear:c`

`\coffin_clear:N ⟨coffin⟩`

Clears the content of the ⟨*coffin*⟩.

`\coffin_set_eq:NN`
`\coffin_set_eq:(Nc|cN|cc)`
`\coffin_gset_eq:NN`
`\coffin_gset_eq:(Nc|cN|cc)`

`\coffin_set_eq:NN` ⟨*coffin*$_1$⟩ ⟨*coffin*$_2$⟩

Sets both the content and poles of ⟨*coffin*$_1$⟩ equal to those of ⟨*coffin*$_2$⟩.

`\coffin_if_exist_p:N` ⋆
`\coffin_if_exist_p:c` ⋆
`\coffin_if_exist:NTF` ⋆
`\coffin_if_exist:cTF` ⋆

`\coffin_if_exist_p:N ⟨coffin⟩`
`\coffin_if_exist:NTF ⟨coffin⟩ {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*coffin*⟩ is currently defined.

37.3 Setting coffin content and poles

\hcoffin_set:Nn
\hcoffin_set:cn
\hcoffin_gset:Nn
\hcoffin_gset:cn

\hcoffin_set:Nn ⟨*coffin*⟩ {⟨*material*⟩}

Typesets the ⟨*material*⟩ in horizontal mode, storing the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material.

\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:
\hcoffin_gset:Nw
\hcoffin_gset:cw
\hcoffin_gset_end:

\hcoffin_set:Nw ⟨*coffin*⟩ ⟨*material*⟩ \hcoffin_set_end:

Typesets the ⟨*material*⟩ in horizontal mode, storing the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\vcoffin_set:Nnn
\vcoffin_set:cnn
\vcoffin_gset:Nnn
\vcoffin_gset:cnn

Updated: 2023-02-03

\vcoffin_set:Nnn ⟨*coffin*⟩ {⟨*width*⟩} {⟨*material*⟩}

Typesets the ⟨*material*⟩ in vertical mode constrained to the given ⟨*width*⟩ and stores the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material.

\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:
\vcoffin_gset:Nnw
\vcoffin_gset:cnw
\vcoffin_gset_end:

Updated: 2023-02-03

\vcoffin_set:Nnw ⟨*coffin*⟩ {⟨*width*⟩} ⟨*material*⟩ \vcoffin_set_end:

Typesets the ⟨*material*⟩ in vertical mode constrained to the given ⟨*width*⟩ and stores the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn

\coffin_set_horizontal_pole:Nnn ⟨*coffin*⟩
{⟨*pole*⟩} {⟨*offset*⟩}

Sets the ⟨*pole*⟩ to run horizontally through the ⟨*coffin*⟩. The ⟨*pole*⟩ is placed at the ⟨*offset*⟩ from the baseline of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression.

\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnn

\coffin_set_vertical_pole:Nnn ⟨*coffin*⟩ {⟨*pole*⟩} {⟨*offset*⟩}

Sets the ⟨*pole*⟩ to run vertically through the ⟨*coffin*⟩. The ⟨*pole*⟩ is placed at the ⟨*offset*⟩ from the left-hand edge of the bounding box of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression.

\coffin_reset_poles:N
\coffin_greset_poles:N

New: 2023-05-17

\coffin_reset_poles:N ⟨*coffin*⟩

Resets the poles of the ⟨*coffin*⟩ to the standard set, removing any custom or inherited poles. The poles will therefore be equal to those that would be obtained from \hcoffin_set:Nn or similar; the bounding box of the coffin is not reset, so any material outside of the formal bounding box will not influence the poles.

37.4 Coffin affine transformations

\coffin_resize:Nnn
\coffin_resize:cnn
\coffin_gresize:Nnn
\coffin_gresize: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_grotate:Nn
\coffin_grotate:cn

\coffin_rotate:Nn ⟨*coffin*⟩ {⟨*angle*⟩}

Rotates the ⟨*coffin*⟩ by the given ⟨*angle*⟩ (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.

\coffin_scale:Nnn
\coffin_scale:cnn
\coffin_gscale:Nnn
\coffin_gscale:cnn

\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.

37.5 Joining and using coffins

\coffin_attach:NnnNnnnn
\coffin_attach:(cnnNnnnn|Nnncnnnn|cnncnnnn)
\coffin_gattach:NnnNnnnn
\coffin_gattach:(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_gjoin:NnnNnnnn
\coffin_gjoin:(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 covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn

\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.

37.6 Measuring coffins

\coffin_dp:N
\coffin_dp:c

\coffin_dp:N ⟨*coffin*⟩

Calculates the depth (below the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

\coffin_ht:N
\coffin_ht:c

\coffin_ht:N ⟨*coffin*⟩

Calculates the height (above the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

\coffin_ht_plus_dp:N
\coffin_ht_plus_dp:c
New: 2024-10-01

\coffin_ht_plus_dp:N ⟨*coffin*⟩

Calculates the total vertical size (height plus depth) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

\coffin_wd:N
\coffin_wd:c

\coffin_wd:N ⟨*coffin*⟩

Calculates the width of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

37.7 Coffin diagnostics

\coffin_display_handles:Nn
\coffin_display_handles:cn

\coffin_display_handles:Nn ⟨*coffin*⟩ {⟨*color*⟩}

This function first calculates the intersections between all of the ⟨*poles*⟩ of the ⟨*coffin*⟩ to give a set of ⟨*handles*⟩. It then prints the ⟨*coffin*⟩ at the current location in the source, with the position of the ⟨*handles*⟩ marked on the coffin. The ⟨*handles*⟩ are labeled 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

\coffin_mark_handle:Nnnn ⟨*coffin*⟩ {⟨$pole_1$⟩} {⟨$pole_2$⟩} {⟨*color*⟩}

This function first calculates the ⟨*handle*⟩ for the ⟨*coffin*⟩ as defined by the intersection of ⟨$pole_1$⟩ and ⟨$pole_2$⟩. It then marks the position of the ⟨*handle*⟩ on the ⟨*coffin*⟩. The ⟨*handle*⟩ are labeled 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

\coffin_show_structure:N ⟨*coffin*⟩

This function shows the structural information about the ⟨*coffin*⟩ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.

Notice that the poles of a coffin are defined by four values: the x and y coordinates of a point that the pole passes through and the x- and y-components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

\coffin_log_structure:N
\coffin_log_structure:c

\coffin_log_structure:N ⟨*coffin*⟩

This function writes the structural information about the ⟨*coffin*⟩ in the log file. See also \coffin_show_structure:N which displays the result in the terminal.

\coffin_show:N
\coffin_show:c
\coffin_log:N
\coffin_log:c
New: 2021-05-11

\coffin_show:N ⟨*coffin*⟩
\coffin_log:N ⟨*coffin*⟩

Shows full details of poles and contents of the ⟨*coffin*⟩ in the terminal or log file. See \coffin_show_structure:N and \box_show:N to show separately the pole structure and the contents.

\coffin_show:Nnn
\coffin_show:cnn
\coffin_log:Nnn
\coffin_log:cnn
New: 2021-05-11

\coffin_show:Nnn ⟨*coffin*⟩ {⟨*int* $expr_1$⟩} {⟨*int* $expr_2$⟩}
\coffin_log:Nnn ⟨*coffin*⟩ {⟨*int* $expr_1$⟩} {⟨*int* $expr_2$⟩}

Shows poles and contents of the ⟨*coffin*⟩ in the terminal or log file, showing the first ⟨*int* $expr_1$⟩ items in the coffin, and descending into ⟨*int* $expr_2$⟩ group levels. See \coffin_show_structure:N and \box_show:Nnn to show separately the pole structure and the contents.

37.8 Constants and variables

\c_empty_coffin A permanently empty coffin.

`\l_tmpa_coffin`
`\l_tmpb_coffin`

Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_coffin`
`\g_tmpb_coffin`

Scratch coffins for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 38

The l3color module Color support

38.1 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:`

```
\color_group_begin:
...
\color_group_end:
```

Creates a color group: one used to "trap" color settings. This grouping is built in to for example `\hbox_set:Nn`.

`\color_ensure_current:`

```
\color_ensure_current:
```

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin:` ...`\color_group_end:` group.

38.2 Color models

A color *model* is a way to represent sets of colors. Different models are particularly suitable for different output methods, *e.g.* screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of *axes* which define a color space. In contrast, various proprietary models are available which define *spot* colors (more formally separations).

Core models are used to pass color information to output; these are "native" to l3color. Core models use real numbers in the range $[0, 1]$ to represent values. The core models supported here are

- `gray` Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
- `rgb` Red-green-blue color, with three axes, one for each of the components

- `cmyk` Cyan-magenta-yellow-black color, with four axes, one for each of the components

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based: see below for more detail. The supported interface models are

- `Gray` Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)
- `hsb` Hue-saturation-brightness color, with three axes, all real values in the range $[0, 1]$ for hue saturation and brightness
- `Hsb` Hue-saturation-brightness color, with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness
- `HSB` Hue-saturation-brightness color, with three axes, integers in the range $[0, 240]$ for hue, saturation and brightness
- `HTML` HTML format representation of RGB color given as a single six-digit hexadecimal number
- `RGB` Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255
- `oklch` Lightness-chromacity-hue color in the Oklab color space (`https://bottosson.github.io/posts/oklab`), which models human perception, with three axes, real values in the range $[0, 1]$ for lightness, real values in the range $[0, 0.4]$ for chromacity and real values in the range $[0, 360]$ for hue[8]
- `oklab` Oklab color, with three axes, real values in the range $[0, 1]$ for lightness and real values in the range $[-0.4, 0.4]$ for the position on the green/red- and yellow/blue-axes[8]
- `wave` Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as `rgb`.

Finally, there are a small number of models which are parsed to allow data transfer from xcolor but which should not be used by end-users. These are

- `cmy` Cyan-magenta-yellow color with three axes, one for each of the components; converted to `cmyk`
- `tHsb` "Tuned" hue-saturation-brightness color with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness; converted to `rgb` using the standard tuning map defined by xcolor
- `&spot` Spot color tint with one value; treated as a gray tint as spot color data is not available for extraction

[8] Be aware, that for now, this is an input model only. Color blending will not benefit from Oklab color space. It is advised to only input colors inside the sRGB gamut, i.e., mapping to $[0, 1]^3$ in the core `rgb` model. Other colors will be crudely clipped to fit inside. Thus, for most hues and lightnesses, the chroma should actually remain below 0.2.

To allow parsing of data from xcolor, any leading model up the first `:` will be discarded; the approach of selecting an internal form for data is *not* used in l3color.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 38.9 for more detail of color support for additional models.

When color is selected by model, the ⟨*value(s)*⟩ given are specified as a comma-separated list. The length of the list will therefore be determined by the detail of the model involved.

Color models (and interconversion) are complex, and more details are given in the manual to the LaTeX 2ε xcolor package and in the *PostScript Language Reference Manual*, published by Addison–Wesley.

38.3 Color expressions

In addition to allowing specification of color by model and values, l3color also supports color expressions. These are created by combining one or more color names, with the amount of each specified as a value in the range 0–100. The value should be given between `!` symbols in the expression. Thus for example

```
red!50!green
```

is a mixture of 50 % red and 50 % green. A trailing value is interpreted as implicitly followed by `!white`, and so

```
red!25
```

specifies 25 % red mixed with 75 % white.

Where the models for the mixed colors are different, the model of the first color is used. Thus

```
red!50!cyan
```

will result in a color specification using the `rgb` model, made up of 50 % red and 50 % of cyan *expressed in* `rgb`. This may be important as color model interconversion is not exact.

The one exception to the above is where the first model in an expression is `gray`. In this case, the order of mixing is "swapped" internally, so that for example

```
black!50!red
```

has the same result as

```
red!50!black
```

(the predefined colors `black` and `white` use the `gray` model).

Where more than two colors are mixed in an expression, evaluation takes place in a stepwise fashion. Thus in

```
cyan!50!magenta!10!yellow
```

the sub-expression

```
cyan!50!magenta
```

is first evaluated to give an intermediate color specification, before the second step

```
<intermediate>!10!yellow
```

where `<intermediate>` represents this transitory calculated value.

Within a color expression, `.` may be used to represent the color active for typesetting (the current color). This allows for example

```
.!50
```

to mean a mixture of 50 % of current color with white.

(Color expressions supported here are a subset of those provided by the LaTeX 2ε xcolor package. At present, only such features as are clearly useful have been added here.)

38.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta` and `yellow` have special meaning and should not be redefined. Color names should be made up of letters, numbers and spaces only: other characters are reserved for use in color expressions. In particular, `.` represents the current color at the start of a color expression.

`\color_set:nn` `\color_set:nn {⟨name⟩} {⟨color expression⟩}`

Evaluates the ⟨*color expression*⟩ and stores the resulting color specification as the ⟨*name*⟩.

`\color_set:nnn` `\color_set:nnn {⟨name⟩} {⟨model(s)⟩} {⟨value(s)⟩}`

Updated: 2024-12-24

Stores the color specification equivalent to the ⟨*model(s)*⟩ and ⟨*value(s)*⟩ as the ⟨*name*⟩. The ⟨*value(s)*⟩ are expanded before parsing.

`\color_set_eq:nn` `\color_set_eq:nn {⟨name1⟩} {⟨name2⟩}`

Copies the color specification in ⟨*name*$_2$⟩ to ⟨*name*$_1$⟩. The special name `.` may be used to represent the current color, allowing it to be saved to a name.

`\color_if_exist_p:n` ⋆
`\color_if_exist:nTF` ⋆

`\color_if_exist_p:n {⟨name⟩}`
`\color_if_exist:nTF {⟨name⟩} {⟨true code⟩} {⟨false code⟩}`

New: 2022-08-12

Tests whether ⟨*name*⟩ is currently defined to provide a color specification.

`\color_show:n`
`\color_log:n`

`\color_show:n {⟨name⟩}`
`\color_log:n {⟨name⟩}`

New: 2021-05-11

Displays the color specification stored in the ⟨*name*⟩ on the terminal or log file.

38.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color (`.`): other more specialized functions such as fill and stroke selectors do *not* adjust this value.

`\color_select:n`
`\color_select:V`
Updated: 2024-12-24

`\color_select:n {⟨color expression⟩}`

Parses the ⟨*color expression*⟩ and then activates the resulting color specification for typeset material.

`\color_select:nn`
`\color_select:(nV|Vn|VV)`

`\color_select:nn {⟨model(s)⟩} {⟨value(s)⟩}`

Activates the color specification equivalent to the ⟨*model(s)*⟩ and ⟨*value(s)*⟩ for typeset material. The ⟨*value(s)*⟩ are fully expanded before parsing.

`\l_color_fixed_model_tl`

When this is set to a non-empty value, colors will be converted to the specified model when they are selected. Note that included images and similar are not influenced by this setting.

38.6 Colors for fills and strokes

Colors for drawing operations and so forth are split into strokes and fills (the latter may also be referred to as non-stroke color). The fill color is used for text under normal circumstances. Depending on the backend, stroke color may use a *stack*, in which case it exhibits the same page breaking behavior as general color. However, `dvips`/`dvisvgm` do not support this, and so color will need to be contained within a scope, such as `\draw_begin:`/`\draw_end:`.

`\color_fill:n`
`\color_stroke:n`

`\color_fill:n {⟨color expression⟩}`

Parses the ⟨*color expression*⟩ and then activates the resulting color specification for filling or stroking.

`\color_fill:nn`
`\color_stroke:nn`
Updated: 2024-12-24

`\color_fill:nn {⟨model(s)⟩} {⟨value(s)⟩}`

Activates the color specification equivalent to the ⟨*model(s)*⟩ and ⟨*value(s)*⟩ for filling or stroking. The ⟨*value(s)*⟩ are fully expanded before parsing.

`color.sc`

When using `dvips`, this PostScript variable holds the stroke color.

38.6.1 Coloring math mode material

Coloring math mode material using `\color_select:nn(n)` has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating `\mathord` atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

\color_math:nn \color_math:nn {⟨*color expression*⟩} {⟨*content*⟩}
\color_math:nnn \color_math:nnn {⟨*model(s)*⟩} {⟨*value(s)*⟩} {⟨*content*⟩}

New: 2022-01-26
Updated: 2024-12-24

Works as for \color_select:n(n) but applies color only to the math mode ⟨*content*⟩. The ⟨*value(s)*⟩ are fully expanded before parsing. The function does not generate a group and the ⟨*content*⟩ therefore retains its math atom states. Sub/superscripts are also properly handled.

\l_color_math_active_tl

New: 2022-01-26

This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.

38.7 Multiple color models

When selecting or setting a color with an explicit model, it is possible to give values for more than one model at one time. This is particularly useful where automated conversion between models does not give the desired outcome. To do this, the list of models and list of values are both subdivided using / characters (as for the similar function in xcolor). For example, to save a color with explicit cmyk and rgb values, one could use

```
\color_set:nnn { foo } { cmyk / rgb }
  { 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

The manually-specified conversion will be used in preference to automated calculation whenever the model(s) listed are used: both in expressions and when a fixed model is active.

Similarly, the same syntax can be applied to directly selecting a color.

```
\color_select:nn { cmyk / rgb }
  { 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

Again, this list is used when a fixed model is active: the first entry is used unless there is a fixed model matching one of the other entries.

38.8 Exporting color specifications

The major use of color expressions is in setting typesetting output, but there are other places in which some form of color information is required. These may need data in a different format or using a different model to the internal representation. Thus a set of functions are available to export colors in different formats.

Valid export targets are

- backend Two brace groups: the first containing the model, the second containing space-separated values appropriate for the model; this is the format required by backend functions of expl3
- comma-sep-cmyk Comma-separated cyan-magenta-yellow-black values
- comma-sep-rgb Comma-separated red-green-blue values suitable for use as a PDF annotation color

- `HTML` Uppercase two-digit hexadecimal values, expressing a red-green-blue color; the digits are *not* separated
- `space-sep-cmyk` Space-separated cyan-magenta-yellow-black values
- `space-sep-rgb` Space-separated red-green-blue values suitable for use as a PDF annotation color

`\color_export:nnN`

`\color_export:nnN {⟨color expression⟩} {⟨format⟩} ⟨tl var⟩`

Parses the ⟨*color expression*⟩ as described earlier, then converts to the ⟨*format*⟩ specified and assigns the data to the ⟨*tl var*⟩.

`\color_export:nnnN`

Updated: 2024-12-24

`\color_export:nnnN {⟨model⟩} {⟨value(s)⟩} {⟨format⟩} ⟨tl var⟩`

Expresses the combination of ⟨*model*⟩ and ⟨*value(s)*⟩ in an internal representation, then converts to the ⟨*format*⟩ specified and assigns the data to the ⟨*tl var*⟩. The ⟨*value(s)*⟩ are fully expanded before parsing.

38.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see https://helpx.adobe.com/indesign/using/spot-process-colors.html for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (`DeviceCMYK`, `DeviceRGB`, `DeviceGray`), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that l3color uses the shorter names `cmyk`, etc.)

`\color_model_new:nnn`

`\color_model_new:nnn {⟨model⟩} {⟨family⟩} {⟨params⟩}`

Creates a new ⟨*model*⟩ which is derived from the color model ⟨*family*⟩. The latter should be one of

- `DeviceN`
- `ICCBased`
- `Separation`

(The ⟨*family*⟩ may be given in mixed case as-in the PDF reference: internally, case of these strings is folded.) Depending on the ⟨*family*⟩, one or more ⟨*params*⟩ are mandatory or optional.

For a `Separation` space, there are three *compulsory* keys.

- `name` The name of the Separation, for example the formal name of a spot color ink. Such a ⟨*name*⟩ may contain spaces, etc., which are not permitted in the ⟨*model*⟩.
- `alternative-model` An alternative device colorspace, one of `cmyk`, `rgb`, `gray` or `CIELAB`. The three parameter-based models work as described above; see below for details of CIELAB colors.
- `alternative-values` A comma-separated list of values appropriate to the `alternative-model`. This information is used by the PDF application if the `Separation` is not available.

CIELAB color separations are created using the `alternative-model = CIELAB` setting. These colors must also have an `illuminant` key, one of `a`, `c`, `e`, `d50`, `d55`, `d65` or `d75`. The `alternative-values` in this case are the three parameters $L*$, $a*$ and $b*$ of the CIELAB model. Full details of this device-independent color approach are given in the documentation to the colorspace package.

CIELAB colors *cannot* be converted into other device-dependent color spaces, and as such, mixing can only occur if colors set up using the CIELAB model are also given with an alternative parameter-based model. If that is not the case, l3color will fallback to using black as the colorant in any mixing.

For a `DeviceN` space, there is one *compulsory* key.

- `names` The names of the components of the `DeviceN` space. Each should be either the ⟨*name*⟩ of a `Separation` model, a process color name (`cyan`, etc.) or the special name `none`.

For a `ICCBased` space, there is one *compulsory* key.

- `file` The name of the file containing the profile.

38.9.1 Color profiles

Color profiles are used to ensure color accuracy by linking to collaboration. Applying a profile can be used to standardize color which is otherwise device-dependent.

`\color_profile_apply:nn`

New: 2021-02-23

`\color_profile_apply:nn {⟨profile⟩} {⟨model⟩}`

This function applies a ⟨*profile*⟩ to one of the device ⟨*models*⟩. The profile will then apply to all color of the selected ⟨*model*⟩. The ⟨*profile*⟩ should specify an ICC profile file. The ⟨*model*⟩ has to be one the standard device models: `cmyk`, `gray` or `rgb`.

Chapter 39

The l3graphics module Graphics inclusion support

39.1 Graphics keys

Inclusion of graphic files requires a range of low-level data be passed to the backend. This is set up using a small number of key–value settings, which are stored in the `graphics` tree.

`decodearray` Array to decode color in bitmap graphic: when non-empty, this should be in the form of one, two or three pairs of real numbers in the range $[0, 1]$, separated by spaces.

`draft` Switch to enable draft mode: graphics are read but not included when this is true.

`interpolate` Switch which indicates whether interpolation should be applied to bitmap graphic files.

`page` The page to extract from a multi-page graphic file: used for `.pdf` files which may contain multiple pages.

`pdf-attr` Additional PDF-focussed attributes: available to allow control of extended `.pdf` structures beyond those needed for graphic inclusion. Due to backend restrictions, this key is only functional with direct PDF mode (pdfTEX and LuaTEX).

`pagebox` The nature of the page box setting used to determine the bounding box of material: used for `.pdf` files which feature multiple page box specifications. A choice from `art`, `bleed`, `crop`, `media`, `trim`. The standard setting is `crop`.

`type` The type of graphic file being included: if this key is not set, the *type* is determined from the file extension.

39.2 Including graphics

\graphics_include:nn
\graphics_include:nV

New: 2025-03-14

\graphics_include:nn {⟨*keys*⟩} {⟨*file*⟩}

Horizontal-mode command which includes the ⟨*file*⟩ as a graphic at the current location. The file ⟨*type*⟩ may be given as one of the ⟨*keys*⟩, or will otherwise be determined from file extension. The ⟨*keys*⟩ is used to pass settings as detailed above.

\l_graphics_ext_type_prop

New: 2025-03-14

Defines mapping between file extensions and file types; where there is no entry for an extension, the type is assumed to be the extension with the leading . removed. Entries should be made in lower case, and the key should be an extension including the leading ., for example

```
\prop_put:Nnn \l_graphics_ext_type_prop { .ps } { eps }
```

\l_graphics_search_ext_seq

New: 2025-03-14

Extensions to use for graphic searching when the given ⟨*file*⟩ name is not found by \graphics_get_full_name:nN.

\l_graphics_search_path_seq

New: 2025-03-14

Each entry is the path to a directory which should be searched when seeking a graphic file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.

39.3 Utility functions

\graphics_get_full_name:nN
\graphics_get_full_name:nN*TF*

New: 2025-03-14

\graphics_get_full_name:nN {⟨*file*⟩} ⟨*tl var*⟩
\graphics_get_full_name:nNTF {⟨*file*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Searches for ⟨*file*⟩ first as given and then using the extensions listed in \l_graphics_search_ext_seq. The search path used will be the entries of \l_graphics_search_path_seq. If found, the full file name including any path and extension will be returned in the ⟨*tl var*⟩. In the non-branching version, the ⟨*tl var*⟩ will be set to \q_no_value in the case that the graphics is not found.

\graphics_get_pagecount:nN

New: 2025-03-14

\graphics_get_pagecount:nn {⟨*file*⟩} ⟨*tl var*⟩

Reads the graphics ⟨*file*⟩ and extracts the number of pages, which are stored in the ⟨*tl var*⟩.

39.4 Showing and logging included graphics

\graphics_show_list:
\graphics_log_list:

New: 2025-03-14

\graphics_show_list:
\graphics_log_list:

These functions list all graphic files loaded in a similar manner to \file_show_list: and \file_log_list:. While \graphics_show_list: displays the list in the terminal, \graphics_log_list: outputs it to the log file only. In both cases, only graphics loaded by l3graphics are listed.

Chapter 40

The l3opacity module Opacity (transparency) support

40.1 Selecting opacity

Opacity (transparency) shares many characteristics with color. However, limitations in terms of backends mean that it is not always possible to use a dedicated stack for tracking opacity. The best results when breaking pages are therefore likely to result using direct PDF output (pdfTEX, LuaTEX).

For users of PostScript-based routes, note that there are security restrictions which can prevent opacity being available in output. In particular, using Adobe Distiller, you will need to enable transparency in the (text-based) configuration: this is not selectable from the GUI.

For users of PDF-based routes, note that opacity only takes effect if a `\DocumentMetadata{}` is added *before* `\documentclass`, which loads and activates the PDF management. See `pdfmanagement-testphase.pdf` for more info.

`\opacity_select:n`
New: 2025-03-27

`\opacity_select:n {⟨expression⟩}`

Evaluates the ⟨*expression*⟩, which should yield a value in the range $[0, 1]$. This is then activated as an opacity for both filling and stroking.

`\opacity_fill:n`
`\opacity_stroke:n`
New: 2025-03-27

`\opacity_fill:n {⟨expression⟩}`

Evaluates the ⟨*expression*⟩, which should yield a value in the range $[0, 1]$. This is then activated as an opacity for filling or stroking, respectively.

Chapter 41

The l3pdf module
Core PDF support

41.1 Objects

41.1.1 Named objects

An ⟨*object*⟩ name should fully expand to tokens suitable for use in a label-like context.

\pdf_object_new:n
New: 2022-08-23

\pdf_object_new:n {⟨*object*⟩}

Declares ⟨*object*⟩ as a PDF object. The object may be referenced from this point on, and written later using \pdf_object_write:nnn.

\pdf_object_write:nnn
\pdf_object_write:nne
New: 2022-08-23

\pdf_object_write:nnn {⟨*object*⟩} {⟨*type*⟩} {⟨*content*⟩}

Writes the ⟨*content*⟩ as content of the ⟨*object*⟩. Depending on the ⟨*type*⟩ declared for the object, the format required for ⟨*content*⟩ will vary:

- array A space-separated list of values
- dict Key–value pairs in the form /⟨*key*⟩ ⟨*value*⟩
- fstream Two brace groups: ⟨*file name*⟩ and ⟨*file content*⟩
- stream Two brace groups: ⟨*attributes (dictionary)*⟩ and ⟨*stream contents*⟩

\pdf_object_ref:n ⋆
New: 2021-02-10

\pdf_object_ref:n {⟨*object*⟩}

Inserts the appropriate information to reference the ⟨*object*⟩ in for example page resource allocation. If the ⟨*object*⟩ does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

\pdf_object_if_exist_p:n ⋆
\pdf_object_if_exist:nTF ⋆
New: 2020-05-15

\pdf_object_if_exist_p:n {⟨*object*⟩}
\pdf_object_if_exist:nTF {⟨*object*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether an object with name {⟨*object*⟩} has been defined.

41.1.2 Indexed objects

Objects can also be created using a pair of ⟨*class*⟩ and *index*; the ⟨*class*⟩ argument should expand to character tokens, whilst the ⟨*index*⟩ is an ⟨*int expr*⟩ and starts at 1. For large families of objects, this approach is more efficient than using individual names.

`\pdf_object_new_indexed:nn`

New: 2024-04-01

`\pdf_object_new_indexed:nn {⟨class⟩} {⟨index⟩}`

Declares a PDF object of ⟨*class*⟩ and ⟨*index*⟩. The object may be referenced from this point on, and written later using `\pdf_object_write_indexed:nnnn`.

`\pdf_object_write_indexed:nnnn`
`\pdf_object_write_indexed:nnne`

New: 2024-04-01

`\pdf_object_write_indexed:nnnn {⟨class⟩} {⟨index⟩} {⟨type⟩} {⟨content⟩}`

Writes the ⟨*content*⟩ as content of the object of ⟨*class*⟩ and ⟨*index*⟩. Depending on the ⟨*type*⟩ declared for the object, the format required for the ⟨*content*⟩ will vary

`array` A space-separated list of values

`dict` Key–value pairs in the form `/`⟨*key*⟩ ⟨*value*⟩

`fstream` Two brace groups: ⟨*file name*⟩ and ⟨*file content*⟩

`stream` Two brace groups: ⟨*attributes (dictionary)*⟩ and ⟨*stream contents*⟩

`\pdf_object_ref_indexed:nn` ⋆

New: 2024-04-01

`\pdf_object_ref_indexed:nn {⟨class⟩} {⟨index⟩}`

Inserts the appropriate information to reference the object of ⟨*class*⟩ and ⟨*index*⟩ in for example page resource allocation. If the ⟨*class*⟩/⟨*index*⟩ combination does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

41.1.3 General functions

`\pdf_object_unnamed_write:nn`
`\pdf_object_unnamed_write:ne`

New: 2021-02-10

`\pdf_object_unnamed_write:nn {⟨type⟩} {⟨content⟩}`

Writes the ⟨*content*⟩ as content of an anonymous object. Depending on the ⟨*type*⟩, the format required for ⟨*content*⟩ will vary:

`array` A space-separated list of values

`dict` Key–value pairs in the form `/`⟨*key*⟩ ⟨*value*⟩

`fstream` Two brace groups: ⟨*attributes (dictionary)*⟩ and ⟨*file name*⟩

`stream` Two brace groups: ⟨*attributes (dictionary)*⟩ and ⟨*stream contents*⟩

\pdf_object_ref_last: ⋆

New: 2021-02-10

\pdf_object_ref_last:

Inserts the appropriate information to reference the last ⟨*object*⟩ created. This is particularly useful for anonymous objects.

\pdf_pageobject_ref:n ⋆

New: 2021-02-10
Updated: 2024-04-22

\pdf_pageobject_ref:n {⟨*abspage*⟩}

Inserts the appropriate information to reference the ⟨*abspage*⟩; the latter is expanded fully before further processing.

41.2 Version

\pdf_version_compare_p:Nn ⋆
\pdf_version_compare:Nn*TF* ⋆

New: 2021-02-10

\pdf_version_compare_p:Nn ⟨*relation*⟩ {⟨*version*⟩}
\pdf_version_compare:NnTF ⟨*relation*⟩ {⟨*version*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the version of the PDF being created with the ⟨*version*⟩ string specified, using the ⟨*relation*⟩. Either the ⟨*true code*⟩ or ⟨*false code*⟩ will be left in the output stream.

\pdf_version_gset:n
\pdf_version_min_gset:n

New: 2021-02-10

\pdf_version_gset:n {⟨*version*⟩}

Sets the ⟨*version*⟩ of the PDF being created. The `min` version will not alter the output version unless it is currently lower than the ⟨*version*⟩ requested.

This function may only be used up to the point where the PDF file is initialized. With `dvips` it sets `\pdf_version_major:` and `\pdf_version_minor:` and allows to compare the values with `\pdf_version_compare:Nn`, but the PDF version itself still has to be set with the command line option `-dCompatibilityLevel` of `ps2pdf`.

\pdf_version: ⋆
\pdf_version_major: ⋆
\pdf_version_minor: ⋆

New: 2021-02-10

\pdf_version:

Expands to the currently-active PDF version.

With `dvips`, the PDF version is initialized to `-1.-1`. With `dvipdfmx`, it is initialized to `1.7` in releases since 2025 June, following the default TeX Live 2025 setting; and `1.5` in previous releases.

41.3 Page (media) size

\pdf_pagesize_gset:nn

New: 2023-01-14

\pdf_pagesize_gset:nn {⟨*width*⟩} {⟨*height*⟩}

Sets the page size (`mediabox`) of the PDF being created to the ⟨*width*⟩ and ⟨*height*⟩, both of which are ⟨*dimexpr*⟩. The page size can only be set at the start of the output with `dvips`; with other backends, this can be adjusted on a per-page basis.

41.4 Compression

\pdf_uncompress:

New: 2021-02-10

\pdf_uncompress:

Disables any compression of the PDF, where possible.

This function may only be used up to the point where the PDF file is initialized.

41.5 Destinations

Destinations are the places a link jumped to. Unlike the name may suggest, they don't describe an exact location in the PDF. Instead, a destination contains a reference to a page along with an instruction how to display this page. The normally used "XYZ *top left zoom*" for example instructs the viewer to show the page with the given *zoom* and the top left corner at the *top left* coordinates—which then gives the impression that there is an anchor at this position.

If an instruction takes a coordinate, it is calculated by the following commands relative to the location the command is issued. So to get a specific coordinate one has to move the command to the right place.

`\pdf_destination:nn`
New: 2021-01-03

`\pdf_destination:nn {⟨name⟩} {⟨type or integer⟩}`

This creates a destination. `{⟨type or integer⟩}` can be one of `fit`, `fith`, `fitv`, `fitb`, `fitbh`, `fitbv`, `fitr`, `xyz` or an integer representing a scale factor in percent. `fitr` here gives only a lightweight version of `/FitR`: The backend code defines `fitr` so that it will with pdfLaTeX and LuaLaTeX use the coordinates of the surrounding box, with `dvips` and `dvipdfmx` it falls back to `fit`. For full control use `\pdf_destination:nnnn`.

The keywords match to the PDF names as described in the following tabular.

Keyword	PDF	Remarks
`fit`	`/Fit`	Fits the page to the window
`fith`	`/FitH` *top*	Fits the width of the page to the window
`fitv`	`/FitV` *left*	Fits the height of the page to the window
`fitb`	`/FitB`	Fits the page bounding box to the window
`fitbh`	`/FitBH` *top*	Fits the width of the page bounding box to the window.
`fitbv`	`/FitBV` *left*	Fits the height of the page bounding box to the window.
`fitr`	`/FitR` *left bottom right top*	Fits the rectangle specified by the four coordinates to the window (see above for the restrictions)
`xyz`	`/XYZ` *left top* null	Sets a coordinate but doesn't change the zoom.
`{⟨integer⟩}`	`/XYZ` *left top zoom*	Sets a coordinate and a zoom meaning `{⟨integer⟩}`%.

`\pdf_destination:nnnn`
New: 2021-01-17

`\pdf_destination:nnnn {⟨name⟩} {⟨width⟩} {⟨height⟩} {⟨depth⟩}`

This creates a destination with `/FitR` type with the given dimensions relative to the current location. The destination is in a box of size zero, but it doesn't switch to horizontal mode.

Part VII
Utilities

Chapter 42

The l3benchmark module Benchmarking

42.1 Benchmark

\g_benchmark_duration_target_fp

New: 2025-03-17

This variable (default value: 1) controls roughly for how long \benchmark:n will repeat code to more accurately benchmark it. The actual duration of one call to \benchmark:n typically lasts between half and twice \g_benchmark_duration_target_fp seconds, unless of course running the code only once already lasts longer than this.

\g_benchmark_time_fp
\g_benchmark_ops_fp

New: 2025-03-17

These variables store the results of the most recently run benchmark. \g_benchmark_time_fp stores the time TEX took in seconds, and \g_benchmark_ops_fp stores the estimated number of elementary operations. The latter is not set by \benchmark_tic:/\benchmark_toc:.

\benchmark_once:n
\benchmark_once_silent:n

New: 2025-03-17

\benchmark_once_silent:n {⟨*code*⟩}
\benchmark_once:n {⟨*code*⟩}

Determines the time \g_benchmark_time_fp (in seconds) taken by TEX to run the ⟨*code*⟩, and an estimated number \g_benchmark_ops_fp of elementary operations. In addition, \benchmark_once:n prints these values to the terminal. The ⟨*code*⟩ is run only once so the time may be quite inaccurate for fast code.

\benchmark:n
\benchmark_silent:n

New: 2025-03-17

\benchmark:n {⟨*code*⟩}

Determines the time \g_benchmark_time_fp (in seconds) taken by TEX to run the ⟨*code*⟩, and an estimated number \g_benchmark_ops_fp of elementary operations. In addition, \benchmark:n prints these values to the terminal. The ⟨*code*⟩ may be run many times and not within a group, thus code with side-effects may cause problems.

\benchmark_tic:
\benchmark_toc:

New: 2025-03-17

`\benchmark_tic: ⟨slow code⟩ \benchmark_toc:`

When it is not possible to run `\benchmark:n` (e.g., the code is part of the execution of a package which cannot be looped) the tic/toc commands can be used instead to time between two points in the code. When executed, `\benchmark_tic:` will print a line to the terminal, and `\benchmark_toc:` will print a matching line with a time to indicate the duration between them in seconds. These commands can be nested.

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.

F

G

H

I

N

O

P

Q

R