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 \eTeX{} 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 \LaTeX{}3 programming language.

The expl3 modules are designed to be loaded on top of \LaTeX{}2e. With an up-to-date \LaTeX{}2e 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.
Contents

I  Introduction  
1  Introduction to expl3 and this document  
   1.1  Naming functions and variables  
1.1.1  Scratch variables  
1.1.2  Terminological inexactitude  
1.2  Documentation conventions  
1.3  Formal language conventions which apply generally  
1.4  \TeX concepts not supported by L\TeX3  

II  Bootstrapping  
2  The l3bootstrap module: Bootstrap code  
   2.1  Using the \LaTeX3 modules  

III  Programming Flow  
4  The l3basics module: Basic definitions  
   4.1  No operation functions  
   4.2  Grouping material  
   4.3  Control sequences and functions  
   4.3.1  Defining functions  
   4.3.2  Defining new functions using parameter text  
   4.3.3  Defining new functions using the signature  
   4.3.4  Copying control sequences  
   4.3.5  Deleting control sequences  
   4.3.6  Showing control sequences  
   4.3.7  Converting to and from control sequences  
   4.4  Analysing control sequences  
   4.5  Using or removing tokens and arguments  
   4.5.1  Selecting tokens from delimited arguments  
   4.6  Predicates and conditionals  
   4.6.1  Tests on control sequences  
   4.6.2  Primitive conditionals  
   4.7  Starting a paragraph  
   4.8  Debugging support
5 The \texttt{l3exp} module: Argument expansion 31
5.1 Defining new variants 31
5.2 Methods for defining variants 32
5.3 Introducing the variants 34
5.4 Manipulating the first argument 35
5.5 Manipulating two arguments 37
5.6 Manipulating three arguments 37
5.7 Unbraced expansion 38
5.8 Preventing expansion 39
5.9 Controlled expansion 41
5.10 Internal functions 43

6 The \texttt{l3s}ort module: Sorting functions 44
6.1 Controlling sorting 44

7 The \texttt{l3tl}-analysis module: Analysing token lists 46

8 The \texttt{l3reg}ex module: Regular expressions in \TeX{} 47
8.1 Syntax of regular expressions 48
8.1.1 Regular expression examples 48
8.1.2 Characters in regular expressions 49
8.1.3 Characters classes 49
8.1.4 Structure: alternatives, groups, repetitions 50
8.1.5 Matching exact tokens 51
8.1.6 Miscellaneous 53
8.2 Syntax of the replacement text 53
8.3 Pre-compiling regular expressions 55
8.4 Matching 56
8.5 Submatch extraction 57
8.6 Replacement 58
8.7 Scratch regular expressions 60
8.8 Bugs, misfeatures, future work, and other possibilities 60

9 The \texttt{l3pr}g module: Control structures 64
9.1 Defining a set of conditional functions 64
9.2 The boolean data type 66
9.2.1 Constant and scratch booleans 68
9.3 Boolean expressions 69
9.4 Logical loops 71
9.5 Producing multiple copies 72
9.6 Detecting \TeX{}'s mode 72
9.7 Primitive conditionals 73
9.8 Nestable recursions and mappings 73
9.8.1 Simple mappings 74
9.9 Internal programming functions 74
10 The \texttt{l3sys} module: System/runtime functions 75
  10.1 The name of the job ........................................... 75
  10.2 Date and time .................................................. 75
  10.3 Engine ................................................................ 76
  10.4 Output format ...................................................... 77
  10.5 Platform ................................................................ 77
  10.6 Random numbers .................................................. 77
  10.7 Access to the shell ............................................... 78
  10.8 Loading configuration data ..................................... 79
    10.8.1 Final settings ................................................. 80

11 The \texttt{l3msg} module: Messages 81
  11.1 Creating new messages ......................................... 81
  11.2 Customizable information for message modules .......... 82
  11.3 Contextual information for messages ....................... 83
  11.4 Issuing messages .................................................. 84
    11.4.1 Messages for showing material ......................... 88
    11.4.2 Expandable error messages ............................... 88
  11.5 Redirecting messages .......................................... 89

12 The \texttt{l3file} module: File and I/O operations 91
  12.1 Input–output stream management ............................ 91
    12.1.1 Reading from files ....................................... 93
    12.1.2 Reading from the terminal ............................... 97
    12.1.3 Writing to files .......................................... 97
    12.1.4 Wrapping lines in output ............................... 99
    12.1.5 Constant input–output streams, and variables ...... 100
    12.1.6 Primitive conditionals ................................... 100
  12.2 File operations .................................................... 100
    12.2.1 Basic file operations ................................. 100
    12.2.2 Information about files and file contents ........ 101
    12.2.3 Accessing file contents ................................ 104

13 The \texttt{l3luatex} module: \LaTeX-specific functions 106
  13.1 Breaking out to Lua ............................................ 106
  13.2 Lua interfaces .................................................. 107

14 The \texttt{l3legacy} module: Interfaces to legacy concepts 109

IV Data types 110
33 The \texttt{unicode} module: Unicode support functions 291
34 The \texttt{text} module: Text processing 294
  34.1 Expanding text ................................................................. 294
  34.2 Case changing ................................................................. 295
  34.3 Removing formatting from text ............................................. 297
  34.4 Control variables ............................................................. 297
  34.5 Mapping to graphemes ....................................................... 298
VI Typesetting 299
35 The \texttt{box} module: Boxes 300
  35.1 Creating and initialising boxes ........................................... 300
  35.2 Using boxes ................................................................. 301
  35.3 Measuring and setting box dimensions ................................... 302
  35.4 Box conditionals ............................................................ 303
  35.5 The last box inserted ....................................................... 303
  35.6 Constant boxes .............................................................. 303
  35.7 Scratch boxes ............................................................... 303
  35.8 Viewing box contents ...................................................... 304
  35.9 Boxes and color ............................................................. 304
  35.10 Horizontal mode boxes .................................................... 304
  35.11 Vertical mode boxes ...................................................... 305
  35.12 Using boxes efficiently ................................................... 307
  35.13 Affine transformations .................................................... 308
  35.14 Viewing part of a box ..................................................... 311
  35.15 Primitive box conditionals ............................................... 312
36 The \texttt{coffins} module: Coffin code layer 313
  36.1 Creating and initialising coffins ....................................... 313
  36.2 Setting coffin content and poles ....................................... 314
  36.3 Coffin affine transformations ........................................... 315
  36.4 Joining and using coffins ................................................ 316
  36.5 Measuring coffins .......................................................... 316
  36.6 Coffin diagnostics ........................................................ 317
  36.7 Constants and variables .................................................. 318
37 The \texttt{color} module: Color support 319
  37.1 Color in boxes ............................................................. 319
  37.2 Color models ............................................................... 319
  37.3 Color expressions .......................................................... 321
  37.4 Named colors ............................................................... 322
  37.5 Selecting colors ............................................................ 322
  37.6 Colors for fills and strokes .............................................. 323
  37.7.1 Coloring math mode material ........................................ 323
  37.7 Multiple color models ..................................................... 323
  37.8 Exporting color specifications ......................................... 324
  37.9 Creating new color models ............................................... 325
  37.9.1 Color profiles .......................................................... 326
38 The l3pdf module: Core PDF support 327
  38.1 Objects ................................................................. 327
  38.2 Version ................................................................. 328
  38.3 Page (media) size ..................................................... 329
  38.4 Compression ........................................................... 329
  38.5 Destinations ........................................................... 329

VII Additions and removals 331

39 The l3candidates module: Experimental additions to l3kernel 332
  39.1 Important notice ..................................................... 332
  39.2 Additions to l3seq ................................................... 333

Index 334
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 \TeXX{} structure containing the data. A V argument will be a single token (similar to N), for example
\texttt{\textbackslash foo:\texttt{V \textbackslash MyVariable}}: on the other hand, using \texttt{v} a csname is constructed first, and then the value is recovered, for example \texttt{\textbackslash foo:\texttt{v \{MyVariable\}}.

\begin{itemize}
  \item This means \textit{expansion once}. In general, the \texttt{V} and \texttt{v} specifiers are favoured over \texttt{o} for recovering stored information. However, \texttt{o} is useful for correctly processing information with delimited arguments.
  \item The \texttt{x} specifier stands for \textit{exhaustive expansion}: every token in the argument is fully expanded until only unexpandable ones remain. The \TeX \texttt{\textbackslash def} primitive carries out this type of expansion. Functions which feature an \texttt{x}-type argument are \textit{not} expandable.
  \item The \texttt{e} specifier is in many respects identical to \texttt{x}, but uses \texttt{\expanded} primitive. Parameter character (usually \texttt{#}) in the argument need not be doubled. Functions which feature an \texttt{e}-type argument may be expandable.
  \item The \texttt{f} specifier stands for \textit{full expansion}, and in contrast to \texttt{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 \textit{(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

\begin{verbatim}
\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 }
\end{verbatim}

will leave \texttt{\l_mya_tl} with the content \texttt{A\l_myb_tl}, as \texttt{A} cannot be expanded and so terminates expansion before \texttt{\l_myb_tl} is considered.
  \item \texttt{T} and \texttt{F} For logic tests, there are the branch specifiers \texttt{T} (\textit{true}) and \texttt{F} (\textit{false}). Both specifiers treat the input in the same way as \texttt{n} (no change), but make the logic much easier to see.
  \item The letter \texttt{p} indicates \TeX \textit{parameters}. Normally this will be used for delimited functions as \texttt{expl3} provides better methods for creating simple sequential arguments.
  \item Finally, there is the \texttt{w} specifier for \textit{weird} arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).
\end{itemize}

\texttt{D} The \texttt{D} stands for \texttt{Do not use}. All of the \TeX primitives are initially \texttt{\let} to a \texttt{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 \textit{strongly discouraged} in package code: programmers should instead use the interfaces documented in \texttt{interface3.pdf}.

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

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

\begin{itemize}
  \item \texttt{c} Constant: global parameters whose value should not be changed.
\end{itemize}
Parameters whose value should only be set globally.

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 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.
- **seq** “Sequence”: a data-type used to implement lists (with access at both ends) and stacks.
- **skip** “Rubber” lengths.
- **str** String variables: contain character data.
- **tl** Token list variables: placeholder for a token list.

Applying \textit{V-} or \textit{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 \textit{box} alignment operations.
- **flag** 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.

\footnote{The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the \texttt{int} module contains some scratch variables called \texttt{\_tmpa.int}, \texttt{\_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 \texttt{\_\_int.tma\_int} would be very unreadable.}
1.1.1 Scratch variables

Modules focussed on variable usage typically provide four scratch variables, two local and two global, with names of the form \((\text{scope})\_\text{tmpa}_(\text{type})/\text{scope}\_\text{tmpb}_(\text{type})\). 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.

1.1.2 Terminological inexactitude

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

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

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

1.2 Documentation conventions

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

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

\begin{verbatim}
\ExplSyntaxOn
\ExplSyntaxOff
\end{verbatim}

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:

\footnote{\TeX\nically, functions with no arguments are \texttt{\textbackslash long} while token list variables are not.}

\[^2\]
When a number of variants are described, the arguments are usually illustrated only for the base function. Here, \texttt{\texttt{sequence}} indicates that \texttt{\texttt{seq_new:N}} expects the name of a sequence. From the argument specifier, \texttt{\texttt{seq_new:c}} also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

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

\begin{verbatim}
\cs_to_str:N ⋆ \cs_to_str:N ⟨cs⟩
\end{verbatim}

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 \texttt{(cs)}, shorthand for a \texttt{(control sequence)}.

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

\begin{verbatim}
\seq_map_function:NN ⋆ \seq_map_function:NN ⟨seq⟩ ⟨function⟩
\end{verbatim}

**Conditional functions** Conditional (if) functions are normally defined in three variants, with \texttt{T}, \texttt{F} and \texttt{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:

\begin{verbatim}
\sys_if_engine_xetex:TF ⋆ \sys_if_engine_xetex:TF ⟨true code⟩ ⟨false code⟩
\end{verbatim}

The underlining and italic of \texttt{TF} indicates that three functions are available:

- \texttt{\sys_if_engine_xetex:T}
- \texttt{\sys_if_engine_xetex:F}
- \texttt{\sys_if_engine_xetex:TF}

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

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

\begin{verbatim}
\l_tmpa_tl
\end{verbatim}

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{} or plain \TeX{}. In these cases, the text will include an extra “\TeX{}hackers note” section:
\token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N \token_to_str:N

The normal description text.

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

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

### 1.3 Formal language conventions which apply generally

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

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

### 1.4 \TeX concepts not supported by \LaTeX3

The \TeX concept of an “\outer” macro is not supported at all by \LaTeX3. As such, the functions provided here may break when used on top of \LaTeX\ 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 \LaTeX2ε and are already pre-loaded since \LaTeX2ε 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 \LaTeX2ε it provides a few functions for setting it up.

\ExplSyntaxOn \ExplSyntaxOff

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

\ProvidesExplPackage \ProvidesExplClass \ProvidesExplFile

These functions act broadly in the same way as the corresponding \LaTeX2ε 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 \LaTeX2ε provides in turning on \makeatletter within package and class code.) The \langle date\rangle should be given in the format ⟨year⟩/⟨month⟩/⟨day⟩ or in the ISO date format ⟨year⟩-⟨month⟩-⟨day⟩. If the \langle version\rangle is given then a leading v is optional: if given as a “pure” version string, a v will be prepended.

\Updated: 2011-08-13
\Updated: 2023-08-03
\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{} category codes and the \LaTeX{} 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 \texttt{l3names} module
Namespace for primitives

3.1 Setting up the \LaTeX{} programming language

This module is at the core of the \LaTeX{} programming language. It performs the following tasks:

- defines new names for all \TeX{} primitives;
- emulate required primitives not provided by default in Lua\TeX{};
- 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 \LaTeX{} code (outside of “kernel-level” code). As such, the primitives are not documented here: \textit{The \TeXbook{}, \TeX{} by Topic} and the manuals for pdf\TeX{}, Xe\TeX{}, Lua\TeX{}, \pdf\TeX{} and up\TeX{} should be consulted for details of the primitives. These are named `$\backslash$tex$\langle$\textit{name}$\rangle$\texttt{\texttt{D}}$, typically based on the primitive’s \langle\textit{name}\rangle in pdf\TeX{} and omitting a leading pdf when the primitive is not related to pdf output.
Part III
Programming Flow
Chapter 4

The \texttt{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

\begin{verbatim}
\prg_do_nothing: *
\end{verbatim}

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

\begin{verbatim}
\scan_stop:
\end{verbatim}

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

4.2 Grouping material

\begin{verbatim}
\group_begin: \group_end: \group_begin:
\end{verbatim}

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

Adds \langle token⟩ to the list of \langle tokens⟩ to be inserted when the current group level ends.
The list of \langle 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 \langle token⟩ at a time. The current group level may be closed by a \group_end: function or by a token with category code 2 (close-group), namely a \} if standard category codes apply.

\TeXhackers note: This is the \TeX primitive \aftergroup.

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

New: 2021-05-11

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

\TeXhackers note: This is a wrapper around the \e-T\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, \langle 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 expansions. 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:Npn \cs_new_nopar:Npn \cs_new_protected:Npn
\cs_new:cpn \cs_new:cpn \cs_new_protected:cpn
\cs_new:Npe \cs_new:Npe \cs_new_protected:Npe
\cs_new:cpe \cs_new:cpe \cs_new_protected:cpe
\cs_new:Npx \cs_new:Npx \cs_new_protected:Npx
\cs_new:cpx \cs_new:cpx \cs_new_protected:cpx

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

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

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

Create (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.
```

```
Create (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.
```

```
Create (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 `x`-type argument. The definition is
global and an error results if the (function) is already defined.
```

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

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

Sets \texttt{function} to expand to \texttt{code} as replacement text. Within the \texttt{code}, the \texttt{parameters} (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \texttt{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

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

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

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

4.3.3 Defining new functions using the signature

\texttt{\cs_new:Nn \cs_new:cn{Ne} \cs_new:cn{Ne}|ce}}

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

\texttt{\cs_new_nopar:Nn \cs_new_nopar:cn{Ne} \cs_new_nopar:cn{Ne}|ce}}

Creates \texttt{(function)} to expand to \texttt{(code)} as replacement text. Within the \texttt{(code)}, the number of \texttt{(parameters)} is detected automatically from the function signature. These \texttt{(parameters)} \texttt{(#1, #2, etc.)} will be replaced by those absorbed by the function. When the \texttt{(function)} is used the \texttt{(parameters)} absorbed cannot contain \texttt{\par} tokens. The definition is global and an error results if the \texttt{(function)} is already defined.
\cs_set_protected:Nn
\cs_set_protected:(cn|Ne|ce)

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

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

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

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

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

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

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

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

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

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

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

18
\texttt{\textbackslash cs\_gset:Nn} \texttt{\textbackslash cs\_gset:(cn|Ne|ce)} \texttt{\textbackslash cs\_gset:Nn \{function\} \{\{code\}\}}

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

\texttt{\textbackslash cs\_gset\_nopar:Nn} \texttt{\textbackslash cs\_gset\_nopar:(cn|Ne|ce)} \texttt{\textbackslash cs\_gset\_nopar:Nn \{function\} \{\{code\}\}}

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

\texttt{\textbackslash cs\_gset\_protected:Nn} \texttt{\textbackslash cs\_gset\_protected:(cn|Ne|ce)} \texttt{\textbackslash cs\_gset\_protected:Nn \{function\} \{\{code\}\}}

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

\texttt{\textbackslash cs\_gset\_protected\_nopar:Nn} \texttt{\textbackslash cs\_gset\_protected\_nopar:(cn|Ne|ce)} \texttt{\textbackslash cs\_gset\_protected\_nopar:Nn \{function\} \{\{code\}\}}

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

\texttt{\textbackslash cs\_generate\_from\_arg\_count:NNnn} \texttt{\textbackslash cs\_generate\_from\_arg\_count:NNnn \{function\} \{creator\}} \texttt{\{number\}} \texttt{\{\{code\}\}}

Uses the \texttt{\{creator\}} function (which should have signature \texttt{Npn}, for example \texttt{\textbackslash cs\_new:Npn}) to define a \texttt{\{function\}} which takes \texttt{\{number\}} arguments and has \texttt{\{code\}} as replacement text. The \texttt{\{number\}} of arguments is an integer expression, evaluated as detailed for \texttt{\textbackslash 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 \textit{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”.

Updated: 2012-01-14
\cs_new_eq:NN ⟨cs⟩ ⟨cs⟩
\cs_new_eq:NN ⟨cs⟩ ⟨token⟩

Globally creates ⟨control sequence1⟩ and sets it to have the same meaning as ⟨control sequence2⟩ or ⟨token⟩. The second control sequence may subsequently be altered without affecting the copy.

\cs_set_eq:NN ⟨cs⟩ ⟨cs⟩
\cs_set_eq:NN ⟨cs⟩ ⟨token⟩

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

\cs_gset_eq:NN ⟨cs⟩ ⟨cs⟩
\cs_gset_eq:NN ⟨cs⟩ ⟨token⟩

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

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 ⟨control sequence⟩
\cs_undefine:c

Sets ⟨control sequence⟩ to be globally undefined.

4.3.6 Showing control sequences

\cs_meaning:N * \cs_meaning:c *

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

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.
\texttt{\cs_log:N} \texttt{⟨control sequence⟩}

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

\texttt{\cs_log:N} \texttt{\cs_log:c}

\texttt{\cs_if_exist_use:N} \texttt{⟨control sequence⟩}

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

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

\texttt{\cs:w ⟨control sequence name⟩} \texttt{\cs_end:}

Converts the given \texttt{⟨control sequence name⟩} into a single control sequence token. This process requires one expansion. The content for \texttt{⟨control sequence name⟩} may be literal material or from other expandable functions. The \texttt{⟨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.

**\LaTeX**hackers note: These are the \texttt{\LaTeX} primitives \texttt{\csname} and \texttt{\endcsname}.

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

\texttt{\cs:w a b c \cs_end:}

and

4.3.7 Converting to and from control sequences

\texttt{\use:c {⟨control sequence name⟩}}

Expands the \texttt{⟨control sequence name⟩} until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other \texttt{c-type} arguments the \texttt{⟨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 \texttt{\use:c} function, both

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

and

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

would be equivalent to

\texttt{\abc}

after two expansions of \texttt{\use:c}.

\texttt{\cs:w} \texttt{\cs_end:}

\texttt{\cs:w ⟨control sequence name⟩} \texttt{\cs_end:}

Converts the given \texttt{⟨control sequence name⟩} into a single control sequence token. This process requires one expansion. The content for \texttt{⟨control sequence name⟩} may be literal material or from other expandable functions. The \texttt{⟨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.

**\LaTeX**hackers note: These are the \texttt{\LaTeX} primitives \texttt{\csname} and \texttt{\endcsname}.

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

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

4.4 Analysing control sequences

\cs_split_function:N \cs_split_function:N (function)

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

The next three functions decompose \texttt{TeX} macros into their constituent parts: if the \emph{(token)} passed is not a macro then no decomposition can occur. In the latter case, all three functions leave \texttt{scan_stop:} in the input stream.

\cs_prefix_spec:N \cs_prefix_spec:N (token)

If the \emph{(token)} is a macro, this function leaves the applicable \texttt{TeX} prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

\begin{verbatim}
\cs_set:Npn \next:nn #1#2 { x #1-y #2 }
\cs_prefix_spec:N \next:nn
\end{verbatim}

leaves \texttt{\long} in the input stream. If the \emph{(token)} is not a macro then \texttt{scan_stop:} is left in the input stream.

\textbf{\texttt{TeX} hackers note}: The prefix can be empty, \texttt{\long}, \texttt{protected} or \texttt{protected\long} with backslash replaced by the current escape character.
\texttt{\textbackslash cs\_parameter\_spec:N} \langle\text{token}\rangle

If the \langle\text{token}\rangle 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

\begin{verbatim}
\texttt{\textbackslash cs\_set:Npn \textbackslash next:nn \#1\#2 \{ x \#1 y \#2 \}}
\texttt{\textbackslash cs\_parameter\_spec:N \textbackslash next:nn}
\end{verbatim}

leaves \#1\#2 in the input stream. If the \langle\text{token}\rangle is not a macro then \texttt{\textbackslash scan\_stop:} is left in the input stream.

\textbf{\TeXhackers note:} If the parameter specification contains the string \texttt{->}, then the function produces incorrect results.

\texttt{\textbackslash cs\_replacement\_spec:N} \langle\text{token}\rangle

If the \langle\text{token}\rangle 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

\begin{verbatim}
\texttt{\textbackslash cs\_set:Npn \textbackslash next:nn \#1\#2 \{ x \#1-y \#2 \}}
\texttt{\textbackslash cs\_replacement\_spec:N \textbackslash next:nn}
\end{verbatim}

leaves \texttt{x\#1\#2} in the input stream. If the \langle\text{token}\rangle is not a macro then \texttt{\textbackslash scan\_stop:} is left in the input stream.

\textbf{\TeXhackers note:} If the parameter specification contains the string \texttt{->}, 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).
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.

\TeX{} hackers note: The \use:n function is equivalent to \LaTeX{} 2e’s \@firstofone.
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, \texttt{\textbackslash 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.
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_i: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.

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.

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

\use:e \{expandable tokens\}

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

Absorb the (balanced text) from the input stream delimited by the marker given in
the function name, leaving nothing in the input stream.
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

\LaTeX{} 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

\begin{verbatim}
\cs_if_free:cTF {abc} {⟨true code⟩} {⟨false code⟩}
\end{verbatim}

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 \texttt{p} in the description part. For example,

\begin{verbatim}
\cs_if_free_p:N
\end{verbatim}

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

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

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 \TeX\,\textit{2e}. Their use is discouraged in \texttt{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

\texttt{\cs_if_eq_p:NN} \langle cs_1 \rangle \langle cs_2 \rangle
\texttt{\cs_if_eq:NNTF} \langle cs_1 \rangle \langle cs_2 \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}

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

\texttt{\cs_if_exist_p:N} \langle control sequence \rangle
\texttt{\cs_if_exist:NTF} \langle control sequence \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}

Tests whether the \textit{control sequence} is currently defined (whether as a function or another control sequence type). Any definition of \textit{control sequence} other than \texttt{relax} evaluates as \texttt{true}.

\texttt{\cs_if_free_p:N} \langle control sequence \rangle
\texttt{\cs_if_free:NTF} \langle control sequence \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}

Tests whether the \textit{control sequence} is currently free to be defined. This test is \texttt{false} if the \textit{control sequence} currently exists (as defined by \texttt{cs_if_exist:NTF}).

4.6.2 Primitive conditionals

The \textit{e-\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 \texttt{:w} part but higher level functions are often available. See for instance \texttt{\int_compare_p:nNn} which is a wrapper for \texttt{\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 \texttt{\if_}, except for \texttt{\if:w}.

\texttt{\if_true:} \langle true code \rangle \texttt{\else:} \langle false code \rangle \texttt{\fi:}
\texttt{\if_false:} \langle true code \rangle \texttt{\else:} \langle false code \rangle \texttt{\fi:}
\texttt{\reverse_if:N} \langle primitive conditional \rangle

\texttt{\if_true:} always executes \langle true code \rangle, while \texttt{\if_false:} always executes \langle false code \rangle. \texttt{\reverse_if:N} reverses any two-way primitive conditional. \texttt{\else:} and \texttt{\fi:} delimit the branches of the conditional. The function \texttt{\or:} is documented in \texttt{\texttt{\textbackslash int}} and used in case switches.

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

28
\if_meaning:w \else: \fi:
\if_meaning:w \langle arg_1 \rangle \langle arg_2 \rangle \langle true code \rangle \else: \langle false code \rangle \fi:
\if_meaning:w \langle true code \rangle \if_meaning:w \langle arg_1 \rangle \langle arg_2 \rangle \else: \langle false code \rangle \fi:
\if_meaning:w
\if:w \langle token_1 \rangle \langle token_2 \rangle \langle true code \rangle \else: \langle false code \rangle \fi:
\if:w \langle token_1 \rangle \langle token_2 \rangle \langle true code \rangle \else: \langle false code \rangle \fi:
\if:w \langle false code \rangle \fi:
\if:w \langle token_1 \rangle \langle token_2 \rangle \langle true code \rangle \else: \langle false code \rangle \fi:
\if:w \langle false code \rangle \fi:
\else:
\fi:
\else:
\fi:
\else:
\else:
\else:
\else:
\else:
\else:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
\fi:
4.8 Debugging support

\debug_on:n \debug_on:n { ⟨comma-separated list⟩ }
\debug_off:n \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 soon-to-be-deprecated commands produce errors;
- **log-functions** that logs function definitions;
- **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:

Suppress (locally) errors and logging from debug commands, except for the deprecation errors or warnings. 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 l3coffins.
Chapter 5

The \texttt{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 \texttt{exp}.

Not all possible variations are implemented for every base function. Instead only those that are used within the \LaTeX{} 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 \texttt{\exp_\ldots}. They all look alike, an example would be \texttt{\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 \texttt{\exp_args:NNo} expands the content of third argument once before any expansion of the first and second arguments. If \texttt{\seq_gpush:No} was not defined it could be coded in the following way:

\begin{verbatim}
\exp_args:NNo \seq_gpush:Nn \g_file_name_stack { \l_tmpa_tl }
\end{verbatim}

In other words, the first argument to \texttt{\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 \texttt{\exp_} function followed by the desired base function, \textit{e.g.}.

\begin{verbatim}
\cs_generate_variant:Nn \seq_gpush:Nn { No }
\end{verbatim}

results in the definition of \texttt{\seq_gpush:No}
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.

- \textit{N} is used for single-token arguments while \textit{c} constructs a control sequence from its name and passes it to a parent function as an \textit{N}-type argument.

- Many argument types extract or expand some tokens and provide it as an \textit{n}-type argument, namely a braced multiple-token argument: \textit{V} extracts the value of a variable, \textit{v} extracts the value from the name of a variable, \textit{n} uses the argument as it is, \textit{o} expands once, \textit{f} expands fully the front of the token list, \textit{e} and \textit{x} expand fully all tokens (differences are explained later).

- A few odd argument types remain: \textit{T} and \textit{F} for conditional processing, otherwise identical to \textit{n}-type arguments, \textit{p} for the parameter text in definitions, \textit{w} for arguments with a specific syntax, and \textit{D} to denote primitives that should not be used directly.
This function is used to define argument-specifier variants of the \textit{parent control sequence} for \LaTeX{} code-level macros. The \textit{parent control sequence} is first separated into the \textit{base name} and \textit{original argument specifier}. The comma-separated list of \textit{variant argument specifiers} is then used to define variants of the \textit{original argument specifier} if these are not already defined; entries which correspond to existing functions are silently ignored. For each \textit{variant} given, a function is created that expands its arguments as detailed and passes them to the \textit{parent control sequence}. So for example

\begin{verbatim}
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
\end{verbatim}

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

\begin{verbatim}
\cs_generate_variant:Nn \foo:Nn \foo:Nn { NV , cV }
\end{verbatim}

generates the functions \texttt{\foo:NV} and \texttt{\foo:cV} in the same way. The \texttt{\cs_generate_variant:Nn} function should only be applied if the \textit{parent control sequence} is already defined. (This is only enforced if debugging support \texttt{check-declarations} is enabled.) If the \textit{parent control sequence} is protected or if the \textit{variant} involves any \texttt{x} argument, then the \textit{variant control sequence} is also protected. The \textit{variant} is created globally, as is any \texttt{\exp_args:N\texttt{variant}} function needed to carry out the expansion. There is no need to re-apply \texttt{\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 \texttt{\cs_generate_variant:Nn} will only create new definitions if there is not already one available.

Only \texttt{n} and \texttt{N} arguments can be changed to other types. The only allowed changes are

\begin{itemize}
  \item \texttt{c} variant of an \texttt{N} parent;
  \item \texttt{o}, \texttt{V}, \texttt{v}, \texttt{f}, \texttt{e}, or \texttt{x} variant of an \texttt{n} parent;
  \item \texttt{N}, \texttt{n}, \texttt{T}, \texttt{F}, or \texttt{p} argument unchanged.
\end{itemize}

This means the \textit{parent} of a \textit{variant} form is always unambiguous, even in cases where both an \texttt{n}-type parent and an \texttt{N}-type parent exist, such as for \texttt{\tl_count:n} and \texttt{\tl_count:N}.

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

For backward compatibility it is currently possible to make \texttt{n}, \texttt{o}, \texttt{V}, \texttt{v}, \texttt{f}, \texttt{e}, or \texttt{x}-type variants of an \texttt{N}-type argument or \texttt{N} or \texttt{c}-type variants of an \texttt{n}-type argument. Both are deprecated. The first because passing more than one token to an \texttt{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 \texttt{V}-type or \texttt{v}-type variant instead of \texttt{c}-type. In those cases, using the lower-level \texttt{\exp_args:No} or \texttt{\exp_args:Nc} functions explicitly is preferred to defining confusing variants.

Updated: 2017-11-28
5.3 Introducing the variants

The \texttt{V} type returns the value of a register, which can be one of \texttt{tl}, \texttt{clist}, \texttt{int}, \texttt{skip}, \texttt{dim}, \texttt{muskip}, or built-in \TeX{} registers. The \texttt{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 \texttt{V} specifier should be used. For those referred to by (cs)name, the \texttt{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 \texttt{o} specifiers be employed.

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

The \texttt{x} type expands all tokens fully, starting from the first. In contrast to \texttt{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 \texttt{x} in their signature do not themselves expand when appearing inside \texttt{e} or \texttt{x} expansion.

The \texttt{f} type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then \texttt{x}-expansion cannot be used, and \texttt{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 \texttt{\example:n}. For this, one should define a variant using \texttt{\cs_generate_variant:Nn \example:n { f }}, then do

\begin{verbatim}
\example:f { \int_eval:n { 3 + 4 } }
\end{verbatim}

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

\begin{verbatim}
\example:f { \int_eval:n { 1 + 2 }, \int_eval:n { 3 + 4 } }
\end{verbatim}

results in the call

\begin{verbatim}
\example:n { 3, \int_eval:n { 3 + 4 } }
\end{verbatim}

while using \texttt{\example:x} or \texttt{\example:e} instead results in

\begin{verbatim}
\example:n { 3, 7 }
\end{verbatim}
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: \l\_tmpa\_tl 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.
This function absorbs two arguments (the \textit{function} name and the \textit{tokens}). The \textit{tokens} are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream \textit{after} reinsertion of the \textit{function}. Thus the \textit{function} may take more than one argument: all others are left unchanged.

The \texttt{:cc} variant constructs the \textit{function} name in the same manner as described for the \textit{tokens}.

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

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

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

\[\exp_args:NNc (token_1) (token_2) \{\text{tokens}\}\]

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.

\[\exp_args:Nco \exp_args:Nno \exp_args:NnV \exp_args:Nnv \exp_args:Nne \exp_args:Nnf \exp_args:Noc \exp_args:Noo \exp_args:Nof \exp_args:NVo \exp_args:Nfo \exp_args:Nff \exp_args:Nee\]

Updated: 2018-05-15

\[\exp_args:Ncx \exp_args:Nnx \exp_args:Nox \exp_args:Nxo \exp_args:Nxx\]

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 \textit{x}-type argument.

5.6 Manipulating three arguments

\[\exp_args:NNNo (token_1) (token_2) (token_3) \{\text{tokens}\}\]

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, \textit{etc.}

\[\exp_args:NNNo \exp_args:NNV \exp_args:NNV \exp_args:NNV \exp_args:Ncc \exp_args:NNe \exp_args:NNe \exp_args:NNe \exp_args:NNe \exp_args:NNe \exp_args:NNe \exp_args:NNe \exp_args:NNe \exp_args:NNe\]

Updated: 2018-05-15
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, \emph{etc}.

\begin{Verbatim}
\exp_args:NNcf \exp_args:NNno \exp_args:NNnV \exp_args:NNoo \exp_args:NNVV \exp_args:Ncno \exp_args:NcnV \exp_args:Ncoo \exp_args:NcVV \exp_args:Nnnc \exp_args:Nnno \exp_args:Nnff \exp_args:Nooo \exp_args:Noof \exp_args:Nffo \exp_args:Neee
\end{Verbatim}

\begin{Verbatim}
\exp_args:NNNx \exp_args:NNnx \exp_args:NNox \exp_args:Nccx \exp_args:Ncnx \exp_args:Nnnx \exp_args:Nnox \exp_args:Nnox
\end{Verbatim}

\textbf{5.7 Unbraced expansion}

\begin{Verbatim}
\exp_last_unbraced:No \exp_last_unbraced:NV \exp_last_unbraced:Nv \exp_last_unbraced:Ne \exp_last_unbraced:Nf \exp_last_unbraced:Noo \exp_last_unbraced:NNV \exp_last_unbraced:NNf \exp_last_unbraced:NcV \exp_last_unbraced:Nnc \exp_last_unbraced:NnV \exp_last_unbraced:NNV \exp_last_unbraced:NNnV \exp_last_unbraced:Nnfo \exp_last_unbraced:NNno \exp_last_unbraced:NNVV \exp_last_unbraced:Nnno \exp_last_unbraced:NNVV \exp_last_unbraced:NNnV \exp_last_unbraced:NnVf \exp_last_unbraced:Nnff \exp_last_unbraced:NnNo \exp_last_unbraced:NNNNo \exp_last_unbraced:NNNf
\end{Verbatim}

\textbf{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, \texttt{\exp_last_unbraced:Nf \foo_bar:w } \texttt{\q_stop} leads to an infinite loop, as the quark is \texttt{f}-expanded.

\textbf{Updated: 2018-05-15}
This function fully expands the \langle tokens \rangle and leaves the result in the input stream after reinsertion of the \langle function \rangle. This function is not expandable.

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.

This function carries out a single expansion of \langle token2 \rangle (which may consume arguments) prior to the expansion of \langle token1 \rangle. If \langle token2 \rangle has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that \langle token1 \rangle 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_after:wN \langle variant \rangle function.

\TeXhackers note: This is the \TeX primitive \expandafter.

\subsection{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.

Prevents expansion of the \langle token \rangle 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 \langle token \rangle is removed even if it appears as \exp_not:N \c_space_token. In an e-expanding definition \langle cs_new:Npe \rangle, a macro parameter introduces an argument even if it appears as \exp_not:N # 1. This differs from \exp_not:n.

Expands the \langle tokens \rangle 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 \{\langle tokens\rangle\}

Prevents expansion of the \langle tokens\rangle in an e-type or x-type argument. In all other cases the \langle tokens\rangle 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.

\textbf{Tex hackers note}: This is the \texttt{\epsilon-\TeX} primitive \unexpanded. In an e-expanding definition (\cs_new:Npe), \exp_not:n \{\#1\} is equivalent to \texttt{##1} rather than to \texttt{#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 \{\langle tokens\rangle\}

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

\exp_not:V = \exp_not:V \langle variable\rangle

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

\exp_not:v = \exp_not:v \{\langle tokens\rangle\}

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

\exp_not:e = \exp_not:e \{\langle tokens\rangle\}

Expands \langle tokens\rangle 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 \{\langle tokens\rangle\}

Expands \langle tokens\rangle 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 \{ \langle tokens\rangle \exp_stop_f: \langle more tokens\rangle \}

This function terminates an f-type expansion. Thus if a function \foo_bar:f starts an f-type expansion and all of \langle tokens\rangle are expandable \exp_stop_f: terminates the expansion of tokens even if \langle more tokens\rangle 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 (\texttt{\textbackslash \␣}).
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 \langle expandable-tokens \rangle as that will break badly if unexpandable tokens are encountered in that place!

\begin{verbatim}
\exp:w \langle expandable tokens \rangle \exp_end:  
\exp_end:  
\end{verbatim}

Expands \langle expandable-tokens \rangle until reaching \exp_end: at which point expansion stops. The full expansion of \langle expandable tokens \rangle has to be empty. If any token in \langle expandable tokens \rangle 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.\footnote{Due to the implementation you might get the character in position 0 in the current font (typically ‘”) in the output without any error message!}

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

\begin{verbatim}
\exp:w \@@_case:NnTF #1 {#2} { } { }
\end{verbatim}

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

\TeXXhackers note: The current implementation uses \texttt{romannumeral} hence ignores space tokens and explicit signs + and - in the expansion of the \langle expandable tokens \rangle, but this should not be relied upon.

\texttt{3}
\exp:w * \exp:w \langle \text{expandable-tokens} \rangle \exp_end_continue_f:w (\text{further-tokens})

Expands \langle \text{expandable-tokens} \rangle\text{ until reaching }\exp_end_continue_f:w\text{ at which point expansion continues as an }f\text{-type expansion expanding }\langle \text{further-tokens} \rangle\text{ until an un-
expandable token is encountered (or the }f\text{-type expansion is explicitly terminated by }
\exp_stop_f:\text{). As with all }f\text{-type expansions a space ending the expansion gets re-
moved.}

The full expansion of \langle \text{expandable-tokens} \rangle has to be empty. If any token in
\langle \text{expandable-tokens} \rangle 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 \langle \text{expandable-tokens} \rangle 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\text{-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\text{-expansion).

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

\exp:w \langle \text{expandable-tokens} \rangle \exp_end:

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

\exp:w \exp_end_continue_f:w \langle \text{expandable-tokens} \rangle \exp_stop_f:

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

\exp:w * \exp:w \langle \text{expandable-tokens} \rangle \exp_end_continue_f:nw (\text{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 \langle \text{further-tokens} \rangle 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\text{-type expansion.

⁴In this particular case you may get a character into the output as well as an error message.
5.10 Internal functions

\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.

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

\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.

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

\begin{verbatim}
\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: } }
\end{verbatim}

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

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

\textbf{\textit{\LaTeX}hackers note}: The current implementation is limited to sorting approximately 20000 items (40000 in \LaTeX), depending on what other packages are loaded.

Internally, the code from \texttt{l3sort} stores items in \texttt{\toks} registers allocated locally. Thus, the \langle comparison code \rangle should not call \texttt{\newtoks} or other commands that allocate new \texttt{\toks} registers. On the other hand, altering the value of a previously allocated \texttt{\toks} register is not a problem.
\sort_return_same: \seq_sort:Nn \seq var
\sort_return_swapped: \lbrace \ldots \ \sort_return_same: \text{ or } \sort_return_swapped: \ldots \rbrace

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the \sort_return_... functions should be used by the code, according to the results of some tests on the items #1 and #2 to be compared.
Chapter 7

The \l3tl-analysis module

Analysing token lists

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

\begin{itemize}
\item \texttt{(tokens)}, which both \texttt{o-expand} and \texttt{e/x-expand} to the \texttt{token}. The detailed form of \texttt{(tokens)} may change in later releases.
\item \texttt{(char code)}, a decimal representation of the character code of the \texttt{token}, \texttt{-1} if it is a control sequence.
\item \texttt{(catcode)}, a capital hexadecimal digit which denotes the category code of the \texttt{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 \texttt{"\(\langle\text{catcode}\rangle\)}.
\end{itemize}

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

\begin{verbatim}
\tl_analysis_show:N \tl_analysis_show:n \{(token list)\}
\tl_analysis_log:n \{(token list)\}
\tl_analysis_log:N
\end{verbatim}

Displays to the terminal (or log) the detailed decomposition of the \texttt{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.

\begin{verbatim}
\tl_analysis_map_inline:nn \tl_analysis_map_inline:nn \{(token list)\} \{(inline function)\}
\tl_analysis_map_inline:Nn
\end{verbatim}

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

46
Chapter 8

The \texttt{l3regex} module

Regular expressions in \TeX

The \texttt{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 \texttt{pcre} syntax (and very close to \texttt{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

\begin{verbatim}
\tl_set:Nn \l_my_tl { That\text{-}cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
\end{verbatim}

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

\begin{verbatim}
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
\end{verbatim}

The \texttt{\w} sequence represents any “word” character, and + indicates that the \texttt{\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, \texttt{\0} denotes the full match (here, a word). The command \texttt{\emph} is inserted using \texttt{\c{emph}}, and its argument \texttt{\0} is put between braces \texttt{\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 \texttt{\regex_set:Nn}. For example,

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

stores in \texttt{\l_foo_regex} a regular expression which matches the starting marker for an environment: \texttt{\begin}, followed by a begin-group token (\texttt{\cB.}), then any number of tokens which are neither begin-group nor end-group character tokens (\texttt{\c[^BE].*}), ending with an end-group token (\texttt{\cE.}). As explained in the next section, the parentheses “capture” the result of \texttt{\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 \texttt{\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 \texttt{\bCat\b} to match a complete word only.

- **[abc]** matches one letter among “a”, “b”, “c”; the pattern \texttt{(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 any number of characters other than underscore, and another underscore; it is equivalent to \texttt{\_.?\_.} 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+\_\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 \texttt{[.,]} instead of \_ would allow the comma as a decimal marker.

- **[(\+-)\d+\d+\d+]** matches an explicit dimension with any unit that \TeX{} knows, where \texttt{(?!pt|in|[cem]m|ex|[bs]p|[dn]d|[pc]c)} matches an explicit dimension with any unit that \TeX{} knows, where \texttt{(?i)} means to treat lowercase and uppercase letters identically.

- **[(\+-)\d+\d+\d+]** matches an explicit floating point number or the special values \texttt{nan} and \texttt{inf} (with signs and spaces allowed).

- **[(\+-)\d+\cC.]** matches an explicit integer or control sequence (without checking whether it is an integer variable).

- **\G.** at the beginning of a regular expression matches and discards (due to \texttt{\G}) everything between the end of the previous match (\texttt{\G}) and what is matched by the rest of the regular expression; this is useful in \texttt{\regex_replace_all:nnN} when the goal is to extract matches or submatches in a finer way than with \texttt{\regex_extract_all:nnN}.

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

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

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

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

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

Characters.

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

\xhh Character with hex code hh.

\a Alarm (hex 07).
\e Escape (hex 1B).
\f Form-feed (hex 0C).
\n New line (hex 0A).
\r Carriage return (hex 0D).
\t Horizontal tab (hex 09).

8.1.3 Characters classes

Character properties.

. A single period matches any token.
\d Any decimal digit.
\h Any horizontal space character, equivalent to [\ \^I]: space and tab.
\s Any space character, equivalent to [\ \^I\^J\^L\^M].
\v Any vertical space character, equivalent to [\^-J\^-K\^-L\^-M]. Note that \^-K is a vertical space, but not a space, for compatibility with Perl.

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

\D Any token not matched by \d.

\H Any token not matched by \h.

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

\S Any token not matched by \s.

\V Any token not matched by \v.

\W Any token not matched by \w.

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

Character classes match exactly one token in the subject.

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

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

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

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

[:^\(name)\:] Negative POSIX character class.

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

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

8.1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

? 0 or 1, greedy.

?? 0 or 1, lazy.

* 0 or more, greedy.

*? 0 or more, lazy.

+ 1 or more, greedy.
+? 1 or more, lazy.
{n} Exactly \( n \).
{n,} \( n \) or more, greedy.
{n,}? \( n \) or more, lazy.
{n, m} At least \( n \), no more than \( m \), greedy.
{n, m}? At least \( n \), no more than \( m \), lazy.

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

Alternation and capturing groups.

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

(... ) Capturing group.

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

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

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

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

\[
\text{\regex_extract_all:nnN \{ a \K . \} \{ a123aaxyz \} \l_foo_seq}
\]

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,

\[
\text{\regex_extract_once:nnN \{ (. \K c)+ \d \} \{ acbc3 \} \l_foo_seq}
\]

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;
• 0 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 \texttt{CBEMTPUDSLOA}). For instance, \cL[A-Z\d] matches uppercase letters and digits of category code letter, \cC. matches any control sequence, and \c0(abc) matches where each character has category other.\footnote{This last example also captures “abc” as a regex group; to avoid this use a non-capturing group \c0(?:abc).}

\c[XYZ] Applies to the next object, and forces it to only match tokens with category X, Y, or Z (each being any of \texttt{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 \texttt{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, [\c0\d \c[LO][A-F]] matches what \texttt{TpX} 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\c0\*cd) matches \texttt{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 \texttt{regex} variable into a larger regular expression. For instance, A\ur{1_{tmpa_regex}}D matches the tokens A and
\section*{8.1.6 Miscellaneous}

Anchors and simple assertions.

\begin{itemize}
  \item \texttt{\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.
  \item \texttt{\B} Not a word boundary: between two \w tokens or two \W tokens (including the boundary).
  \item ^ or \texttt{\A} Start of the subject token list.
  \item \$ or \texttt{\Z} or \texttt{\z} End of the subject token list.
  \item \texttt{\G} Start of the current match. This is only different from \^ in the case of multiple matches: for instance \texttt{\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.
\end{itemize}

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 \[{?-b}\] (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\[[:lower:]\].

\section*{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:

\begin{itemize}
  \item \texttt{\0} is the whole match;
  \item \texttt{\1} is the submatch that was matched by the first (capturing) group (…); similarly for \texttt{\2}, …, \texttt{\9} and \texttt{\g{⟨number⟩}};
  \item \texttt{\␣} inserts a space (spaces are ignored when not escaped);
\end{itemize}
• \a, \e, \f, \n, \r, \t, \xzh, \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)(1d--l)!

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

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

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

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

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

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

The escape sequence \u{⟨var name⟩} allows to insert the contents of the variable with name ⟨var name⟩ directly into the replacement, giving an easier control of category codes. When nested in \c{⟨…⟩} and \u{⟨…⟩} constructions, the \u and \c escape sequences perform \tl_to_str:v, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of \c and \u. For instance,
\let\set:Nn \l_my_one_tl { first }
\let\set:Nn \l_my_two_tl { \textit{second} }
\let\set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \l_my_\0_tl } \l_my_tl

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

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

\let\clear:N \l_tmpa_tl
\regex_replace_all:nnN { } { \textup{U}\% \textup{A}\~ } \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 \texttt{3regex} module’s functions can be given their regular
expression argument either as an explicit string or as a compiled regular expression.

\begin{verbatim}
\regex_new:N \regex_new:N \langle regex var \rangle
\regex_set:Nn \regex_set:Nn \langle regex var \rangle { \langle regex \rangle }
\regex_const:Nn \regex_const:Nn \langle regex var \rangle { \langle regex \rangle }
\end{verbatim}

- \regex_new:N \regex_new:N \langle regex var \rangle
  Creates a new \langle regex var \rangle or raises an error if the name is already taken. The declaration
  is global. The \langle regex var \rangle is initially such that it never matches.

- \regex_set:Nn \regex_set:Nn \langle regex var \rangle { \langle regex \rangle }
  Stores a compiled version of the \langle regular expression \rangle in the \langle regex var \rangle. The as-
  signment is local for \regex_set:Nn and global for \regex_gset:Nn. For instance, this
  function can be used as

  \begin{verbatim}
  \regex_new:N \l_my_regex
  \regex_set:Nn \l_my_regex { my\ (simple\ )? \texttt{reg(\texttt{e}x|\texttt{u}l\texttt{a}r\ \texttt{e}xpression)} }
  \end{verbatim}

- \regex_const:Nn \regex_const:Nn \langle regex var \rangle { \langle regex \rangle }
  Creates a new constant \langle regex var \rangle or raises an error if the name is already taken.
  The value of the \langle regex var \rangle is set globally to the compiled version of the \langle regular
  expression \rangle.
Displays in the terminal or writes in the log file (respectively) how l3regex interprets the \langle regex\rangle. For instance, \regex_show:n \{\A X|Y\} shows

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

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

### 8.4 Matching

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

\regex_match:nnTF {\langle regex\rangle} {\langle token list\rangle} {\langle true code\rangle} {\langle false code\rangle}

Tests whether the \langle regular expression\rangle matches any part of the \langle token list\rangle. For instance,

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

leaves TRUE then FALSE in the input stream.

\regex_count:nnN {\langle regex\rangle} {\langle token list\rangle} {\langle int var\rangle}

Sets \langle int var\rangle within the current \TeX group level equal to the number of times \langle regular expression\rangle appears in \langle token list\rangle. 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) } { abababcb } \l_foo_int

results in \l_foo_int taking the value 5.
\regex_match_case:nnTF
\regex_match_case:nnTF
\regex_match_case:nnTF
\regex_match_case:nnTF
\regex_match_case:nnTF
\regex_match_case:nnTF
Determines which of the \textit{regular expressions} matches at the earliest point in the \textit{token list}, and leaves the corresponding \textit{code} followed by the \textit{true code} in the input stream. If several \textit{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 \textit{regex} match, the \textit{false code} is left in the input stream. Each \textit{regex} can either be given as a regex variable or as an explicit regular expression.

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

### 8.5 Submatch extraction

\regex_extract_once:nnN \{ \regex \} \{ \textit{token list} \} \{ \textit{seq var} \}
\regex_extract_once:nVN \{ \regex \} \{ \textit{token list} \} \{ \textit{seq var} \} \{ \textit{true code} \} \{ \textit{false code} \}

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

For instance, assume that you type

\begin{verbatim}
\regex_extract_once:nVN \{ A(La)?TeX(!*) \} \{ LaTeX!!! \} \l_foo_seq \{ true \} \{ false \}
\end{verbatim}

Then the regular expression (anchored at the start with \verb|A| and at the end with \verb|Z|) must match the whole token list. The first capturing group, (La)?, matches La, and the second capturing group, (!*), matches !!. Thus, \verb|\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 \verb|\l_foo_seq|, as obtained using \texttt{\seq_item:Nn}, correspond to the submatch numbered \((n - 1)\) in functions such as \texttt{\regex_replace_once:nnN}.
\regex_extract_all:nnN \regex_extract_all:nVN \regex_extract_all:nnNTF \regex_extract_all:NNN \regex_extract_all:NNNTF
\regex_split:nnN \regex_split:nVN \regex_split:nnNTF \regex_split:NNN \regex_split:NNNTF
\regex_replace_once:nnN \regex_replace_once:nVN \regex_replace_once:nnNTF \regex_replace_once:NNN \regex_replace_once:NNNTF

Finds all matches of the \textit{regular expression} in the \textit{token list}, and stores all
the submatch information in a single sequence (concatenating the results of multiple
\regex_extract_once:nnN calls). The \textit{seq var} is assigned locally. If there is no
match, the \textit{seq var} is cleared. The testing versions insert the \textit{(true code)} into the
input stream if a match was found, and the \textit{(false code)} otherwise. For instance,
assume that you type
\begin{verbatim}
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
\end{verbatim}
Then the regular expression matches twice, the resulting sequence contains the two items
\{Hello\} and \{world\}, and the \textit{true} branch is left in the input stream.

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

\begin{verbatim}
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
\end{verbatim}
the sequence \\textbackslash l\_path\_seq contains the items \{the\}, \{path\}, \{for\}, \{this\}, and
\{file.tex\}, and the \textit{true} branch is left in the input stream.

8.6 Replacement

\begin{verbatim}
\regex_replace_once:nnN \regex_replace_once:nVN \regex_replace_once:nnNTF \regex_replace_once:NNN \regex_replace_once:NNNTF
\end{verbatim}
Searches for the \textit{regular expression} in the contents of the \textit{tl var} and replaces
the first match with the \textit{replacement}. In the \textit{replacement}, \texttt{\textbackslash 0}
represents the full
match, \texttt{\textbackslash 1} represent the contents of the first capturing group, \texttt{\textbackslash 2} of the second, etc. The
result is assigned locally to \textit{tl var}.
\regex_replace_all:nnN \{⟨regular expression⟩\} \{⟨replacement⟩\} \{tl var\}
\regex_replace_all:nVN \{⟨regular expression⟩\} \{⟨replacement⟩\} \{tl var\} \{true code\} \{false code\}

Replaces all occurrences of the \{⟨regular expression⟩\} in the contents of the \{tl var\} by the \{⟨replacement⟩\}, where \0 represents the full match, \1 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⟩\} \{⟨replacement⟩\}
\regex_replace_case_once:nNTF \{⟨regex⟩\} \{⟨replacement⟩\} \{true code\} \{false code\}

Replaces the earliest match of the regular expression \(?|\{⟨regex⟩\}|...|\{⟨regex⟩\}\) in the \{token list variable\} by the \{replacement\} corresponding to which \{regex\} 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.
Replaces all occurrences of all \texttt{regex} in the \texttt{token list} by the corresponding \texttt{replacement}. Every match is treated independently, and matches cannot overlap. The result is assigned locally to \texttt{tl var}, and the \texttt{true code} or \texttt{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 \texttt{token list}, each of the \texttt{regex} is searched in turn. If one of them matches then it is replaced by the corresponding \texttt{replacement}, and the search resumes at the position that follows this match (and replacement). For instance

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

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

8.7 Scratch regular expressions

\begin{verbatim}
\l_tmpa_regex \l_tmpb_regex
\end{verbatim}

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.

\begin{verbatim}
\g_tmpa_regex \g_tmpb_regex
\end{verbatim}

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 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.

• Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with `\cs_if_exist` tests.

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.
\texttt{\textbackslash \texttt{ddd}}, matching the character with octal code \texttt{ddd}: we already have \texttt{\textbackslash \{...\}} and the syntax is confusingly close to what we could have used for backreferences (\texttt{\textbackslash 1}, \texttt{\textbackslash 2}, \ldots), making it harder to produce useful error message.

\texttt{\textbackslash \texttt{cx}}, similar to \LaTeX’s own \texttt{\textbackslash ^\texttt{x}}.

Comments: \LaTeX already has its own system for comments.

\texttt{\textbackslash \texttt{Q...}\texttt{E}} escaping: this would require to read the argument verbatim, which is not in the scope of this module.

\texttt{\textbackslash \texttt{C}} single byte in UTF-8 mode: \TeX and \LaTeX 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 \LaTeX{} 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 \langle true \rangle and \langle false \rangle.

\LaTeX{} 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 \langle true \rangle or \langle false \rangle. For example, the function \texttt{\cs_if_free_p:N} checks whether the control sequence given as its argument is free and then returns the boolean \langle true \rangle or \langle false \rangle values to be used in testing with \texttt{\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 \texttt{\cs_if_free:NTF} which also takes one argument (the N) and then executes either \texttt{true} or \texttt{false} depending on the result.

\texttt{\LaTeX} hackers note: The arguments are executed after exiting the underlying \texttt{\if...\fi} structure.

9.1 Defining a set of conditional functions

\begin{verbatim}
\prg_new_conditional:Npnn ⟨name⟩:(⟨arg spec⟩) ⟨parameters⟩ {{⟨conditions⟩}} {{⟨code⟩}}
\prg_new_conditional:Nnn ⟨name⟩:⟨arg spec⟩ {{⟨conditions⟩}} {{⟨code⟩}}
\prg_gset_conditional:Npnn ⟨name⟩:(⟨arg spec⟩) ⟨parameters⟩ {{⟨conditions⟩}} {{⟨code⟩}}
\prg_gset_conditional:Nnn ⟨name⟩:⟨arg spec⟩ {{⟨conditions⟩}} {{⟨code⟩}}
\end{verbatim}

These functions create a family of conditionals using the same \langle ⟨code⟩ \rangle to perform the test created. Those conditionals are expandable if \langle ⟨code⟩ \rangle is. The new versions check for existing definitions and perform assignments globally (cf. \texttt{\cs_new:Npn}) whereas the set versions do no check and perform assignments locally (cf. \texttt{\cs_set:Npn}). The conditionals created are dependent on the comma-separated list of \langle ⟨conditions⟩ \rangle, which should be one or more of p, T, F and TF.
These functions create a family of protected conditionals using the same \{\code\} to perform the test created. The \code{} does not need to be expandable. The new version check for existing definitions and perform assignments globally (cf. \cs_new:Npn) whereas the set version do not (cf. \cs_set:Npn). The conditionals created are depended on the comma-separated list of \conditions{}, which should be one or more of T, F and TF (not p).

The conditionals are defined by \prg_new_protected_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{}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{}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{}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:

\begin{verbatim}
\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:
\end{verbatim}

Updated: 2012-02-06
This defines the function \(\texttt{foo_if_bar_p:NN}\), \(\texttt{foo_if_bar:NNTF}\) and \(\texttt{foo_if_bar:NNT}\) but not \(\texttt{foo_if_bar:NNF}\) (because \(F\) is missing from the \texttt{⟨conditions⟩} list). The return statements take care of resolving the remaining \texttt{\else:} and \texttt{\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.

\begin{verbatim}
\prg_new_eq_conditional:NNn \langle name1⟩:⟨arg spec1⟩ \langle name2⟩:⟨arg spec2⟩
\prg_set_eq_conditional:NNn \{⟨conditions⟩\}
\prg_gset_eq_conditional:NNn
\end{verbatim}

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

\begin{verbatim}
\prg_return_true: * \prg_return_true:
\prg_return_false: * \prg_return_false:
\end{verbatim}

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by \texttt{\prg_set_conditional:Npnn}, \texttt{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 \texttt{\prg_return_true:} or \texttt{\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.

\begin{verbatim}
\prg_generate_conditional_variant:Nnn \langle name⟩:⟨arg spec⟩ \{⟨variant argument specifiers⟩\} \{⟨condition specifiers⟩\}
\end{verbatim}

Defines argument-specifier variants of conditionals. This is equivalent to running \texttt{\cs_generate_variant:Nn} \langle conditional⟩ \{⟨variant argument specifiers⟩\} on each \texttt{⟨conditional⟩} described by the \texttt{⟨condition specifiers⟩}. These base-form \texttt{⟨conditionals⟩} are obtained from the \texttt{⟨name⟩} and \texttt{⟨arg spec⟩} as described for \texttt{\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 (\textit{e.g.}, draft/final) and other times you perhaps want to use it as a predicate function in an \texttt{\if_predicate:w} test. The problem of the primitive \texttt{\if_false:} and \texttt{\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).

**\TeX hackers note:** The bool data type is not implemented using the \iffalse/\iftrue primitives, in contrast to \newif, etc., in plain \TeX, \LaTeX, \TeX2ε and so on. Programmers should not base use of bool switches on any particular expectation of the implementation.

```latex
\def\bool_new:N #1 {\cs_new:Nn {\bool_use:N #1} {false}}
\def\bool_new:c #1 {\cs_new:cN {\bool_use:cN #1} {false}}
\def\bool_set_false:N #1 {\bool_set:Nn {#1} {false}}
\def\bool_set_false:c #1 {\bool_set:cn {#1} {false}}
\def\bool_set_true:N #1 {\bool_set:Nn {#1} {true}}
\def\bool_set_true:c #1 {\bool_set:cn {#1} {true}}
\def\bool_set_eq:NN #1 #2 {\bool_set_eq:NNn {#1} {#2}}
\def\bool_set_eq:nnn #1 #2 #3 {\bool_set_eq:nnnn {#1} {#2} {#3}}
\def\bool_set_inverse:N #1 {\bool_set_inverse:Nn {#1} {false}}
\def\bool_set_inverse:c #1 {\bool_set_inverse:cn {#1} {false}}
```

New: 2018-05-10

67
\bool_if_p:N \bool_if_p:c \bool_if:N \bool_if:c \true \false
\bool_if:NTF \bool_if:NTF \false \true
Tests the current truth of \texttt{boolean}, and continues expansion based on this result.

\bool_to_str:N \bool_to_str:c \bool_to_str:n \true \false
\bool_to_str:N \bool_to_str:c \bool_to_str:n
Expands to the string \texttt{true} or \texttt{false} depending on the logical truth of the \texttt{boolean} or \texttt{boolean expression}.

\bool_show:N \bool_show:c \bool_show:n \true \false
\bool_show:N \bool_show:c \bool_show:n
Displays the logical truth of the \texttt{boolean} on the terminal.

\bool_log:N \bool_log:c \bool_log:n \true \false
\bool_log:N \bool_log:c \bool_log:n
Writes the logical truth of the \texttt{boolean} in the log file.

\bool_if_exist_p:N \bool_if_exist_p:c \bool_if_exist:N \bool_if_exist:c \true \false
\bool_if_exist:NTF \bool_if_exist:NTF \false \true
Tests whether the \texttt{boolean} is currently defined. This does not check that the \texttt{boolean} really is a boolean variable.

9.2.1 Constant and scratch booleans

\c_true_bool \c_false_bool
Constants that represent \texttt{true} and \texttt{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 \LaTeX\-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.
A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any \TeX3-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 \langle true \rangle or \langle false \rangle values, it seems only fitting that we also provide a parser for \langle boolean expressions \rangle.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean \langle true \rangle or \langle false \rangle. 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,

\begin{verbatim}
\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 }
\end{verbatim}

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.

\begin{verbatim}
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
\end{verbatim}

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

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

\begin{verbatim}
\bool_if_p:n *
\bool_if_p:n \{\langle boolean expression\rangle\} \{\langle true code\rangle\} \{\langle false code\rangle\}
\end{verbatim}

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

\begin{verbatim}
\bool_lazy_all_p:n *
\bool_lazy_all_p:n \{\langle boolean expr1\rangle \langle boolean expr2\rangle \ldots \langle boolean exprn\rangle\} \{\langle true code\rangle\}
\end{verbatim}

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

\begin{verbatim}
\bool_lazy_any_p:n *
\bool_lazy_any_p:n \{\langle boolean expr1\rangle \langle boolean expr2\rangle \ldots \langle boolean exprn\rangle\} \{\langle true code\rangle\} \{\langle false code\rangle\}
\end{verbatim}

Implements the "Or" operation on the \langle boolean expressions\rangle, hence is \texttt{true} if any of them is \texttt{true} and \texttt{false} if all of them are \texttt{false}. Contrarily to the infix operator \texttt{||}, only the \langle boolean expressions\rangle which are needed to determine the result of \texttt{\bool_lazy_any:nTF} are evaluated. See also \texttt{\bool_lazy_all:nTF} when there are more than two \langle boolean expressions\rangle.

\begin{verbatim}
\bool_not_p:n *
\bool_not_p:n \{\langle boolean expression\rangle\}
\end{verbatim}

Function version of !\langle boolean expression\rangle within a boolean expression.
\bool_xor_p:nn \bool_xor_p:nn \{boolexpr_1\} \{boolexpr_2\}
\bool_xor:nnTF \bool_xor:nnTF \{boolexpr_1\} \{boolexpr_2\} \{true\} \{false\}

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: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 \texttt{false} then the \{code\} is inserted into the input stream
again and the process loops until the \{boolean\} is \texttt{true}.

\bool_do_until:cn \bool_do_until:cn

\bool_do_while:Nn \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 \texttt{true} then the \{code\} is inserted into the input stream
again and the process loops until the \{boolean\} is \texttt{false}.

\bool_until_do:Nn \bool_until_do:Nn \{boolean\} \{code\}
This function first checks the logical value of the \{boolean\}. If it is \texttt{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 \texttt{true}.

\bool_until_do:cn \bool_until_do:cn

\bool_while_do:Nn \bool_while_do:Nn \{boolean\} \{code\}
This function first checks the logical value of the \{boolean\}. If it is \texttt{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 \texttt{false}.

\bool_until_do:nn \bool_until_do: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 \texttt{false} then the
\{code\} is inserted into the input stream again and the process loops until the \{boolean expression\}
evaluates to \texttt{true}.

\bool_do_until:nn \bool_do_until:nn \{boolean expression\} \{code\}
Updated: 2017-07-15

\bool_do_while:nn \bool_do_while:nn \{boolean expression\} \{code\}
Updated: 2017-07-15

\bool_until_do:nn \bool_until_do:nn \{boolean expression\} \{code\}
Updated: 2017-07-15

This function first checks the logical value of the \{boolean expression\} (as described
for \bool_if:nTF). If it is \texttt{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 \texttt{true}.
\bool_while_do:nn \bool_while_do:nn \bool_if:nTF \bool_case:nTF \bool_case:nF

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

\bool_case:n \bool_case:nTF \bool_case:nF

\texttt{\{\textit{expr case 1}\} \{\textit{code case 1}\}}
\texttt{\{\textit{expr case 2}\} \{\textit{code case 2}\}}
... \texttt{\{\textit{expr case n}\} \{\textit{code case n}\}}
\texttt{\{\texttt{true code}\}}
\texttt{\{\texttt{false code}\}}

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

\begin{verbatim}
\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! }
\end{verbatim}

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

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

9.6 Detecting \LaTeX{}’s mode

\texttt{\mode_if_horizontal_p: \mode_if_horizontal_p: \mode_if_horizontal:TF \mode_if_horizontal:TF}

Detects if \LaTeX{} is currently in horizontal mode.
Detects if \TeX{} is currently in inner mode.

\texttt{\mode_if_inner_p: \* \mode_if_inner:TF \{true code\} \{false code\}}

Detects if \TeX{} is currently in maths mode.

\texttt{\mode_if_math_p: \* \mode_if_math:TF \{true code\} \{false code\}}

Detects if \TeX{} is currently in vertical mode.

\texttt{\mode_if_vertical_p: \* \mode_if_vertical:TF \{true code\} \{false code\}}

\section{Primitive conditionals}

\texttt{\if_predicate:w \langle predicate \rangle \{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 \texttt{\langle predicate \rangle} but to make the coding clearer this should be done through \texttt{\if_bool:N}.)

\texttt{\if_bool:N \langle boolean \rangle \{true code\} \else: \{false code\} \fi:}

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

\section{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.

\texttt{\prg_break_point:Nn \langle type \rangle \_map_break: \{\langle code \rangle\}}

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

\prg_break_point:Nn \( \langle \text{type} \rangle \) _map_break: {\langle \text{ending code} \rangle}

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

For types with mappings defined in the kernel, \langle type \rangle _map_break: and \langle type \rangle _map_break:n are defined as \prg_map_break:Nn \langle type \rangle _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 \langle \langle \text{code} \rangle \rangle ...
\prg_break_point:
\prg_break:n *

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

9.9 Internal programming functions

\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.

\TeX{}hackers 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.

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

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.

\TeX{}hackers 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.

Nov: 2015-09-22

\c_sys_timestamp_str

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

Nov: 2023-08-27
10.3 Engine

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

\begin{verbatim}
\sys_if_engine_luatex_p: ⋆\sys_if_engine_luatex: ⋆\sys_if_engine_pdfTeX_p: ⋆\sys_if_engine_pdfTeX:TF \{\texttt{(true code)}\} \{\texttt{(false code)}\}
\end{verbatim}

\texttt{\c_sys_engine_str}

The current engine given as a lower case string: one of \texttt{luatex, pdfTeX, pTeX, upTeX} or \texttt{xetex}.

\texttt{\c_sys_engine_exec_str}

The name of the standard executable for the current \TeX\ engine given as a lower case string: one of \texttt{luatex, luahbtex, pdfTeX, eptex, euptex} or \texttt{xetex}.

\texttt{\c_sys_engine_format_str}

The name of the preloaded format for the current \TeX\ run given as a lower case string: one of \texttt{lualatex} (or \texttt{dvilualatex}), \texttt{pdflatex} (or \texttt{latex}), \texttt{platex}, \texttt{uplatex} or \texttt{xelatex} for \TeX, similar names for plain \TeX\ (except \texttt{pdflatex} in DVI mode yields \texttt{etex}), and \texttt{con-t-en} for \ConTeXt\ (i.e. the \texttt{\fmtname}).

\texttt{\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 \pdflatex\ and \LuaTeX\ this is of the form

\texttt{⟨major⟩.⟨minor⟩.⟨revision⟩}

For \XeTeX, the form is

\texttt{⟨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-p\TeX\ version.

\texttt{p⟨major⟩.⟨minor⟩.⟨revision⟩-u⟨major⟩.⟨minor⟩-⟨epTeX⟩}

where the \texttt{u} part is only present for \upTeX.

\texttt{\sys_timer: ⋆\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.

76
Tests whether current engine has timer support.

### 10.4 Output format

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

The current output mode given as a lower case string: one of \texttt{dvi} or \texttt{pdf}.

### 10.5 Platform

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

The current platform given as a lower case string: one of \texttt{unix}, \texttt{windows} or unknown.

### 10.6 Random numbers

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.
\textbf{\sysgsetrandseed:n} \{(int expr)\}

Globally sets the seed for the engine's pseudo-random number generator to the \textit{integer expression}. This random seed affects all \texttt{\textbackslash\ldots\_rand} functions (such as \texttt{\int_rand:nn} or \texttt{\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.

\textbf{\texttt{\TeX}hackers 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

\textbf{\sysgetshell:nnN \sysgetshell:nnNTF}

\texttt{\sysgetshell:nnN \{(shell command) \{\texttt{\textbackslash setup}\} \{tl var\}}

\texttt{\sysgetshell:nnNTF \{(shell command) \{\texttt{\textbackslash setup}\} \{tl var\} \{\{true code\} \{false code\}\}}

Defines \texttt{(tl var)} to the text returned by the \texttt{(shell command)}. The \texttt{(shell command)} is converted to a string using \texttt{\tl_to_str:n}. Category codes may need to be set appropriately via the \texttt{(setup)} argument, which is run just before running the \texttt{(shell command)} (in a group). If shell escape is disabled, the \texttt{(tl var)} will be set to \texttt{\q_no_value} in the non-branching version. Note that quote characters ("" ) cannot be used inside the \texttt{(shell command)}. The \texttt{\sysgetshell:nnNTF} conditional inserts the \texttt{(true code)} if the shell is available and no quote is detected, and the \texttt{(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 \texttt{true} branch is taken: if the command is forbidden at this stage, a low-level \TeX{} error will arise.

\textbf{\c_sysshell_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

\textbf{\sysifshell_p: \sysifshell:TF \sysifshell_unrestricted_p: \sysifshell_unrestricted:TF}

\texttt{\sysifshell_p: \star \sysifshell:TF \{(true code) \{(false code)\}}

\texttt{\sysifshell_unrestricted_p: \star \sysifshell_unrestricted: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.

Performs a check for whether \textit{unrestricted} shell escape is enabled.
Perform 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
Execute \token{tokens} through shell escape immediately.

\sys_shell_shipout:n \sys_shell_shipout:e
Execute \token{tokens} through shell escape at shipout.

10.8 Loading configuration data

\sys_load_backend:n \sys_load_backend:n \{\token{backend}\}
Loads the additional configuration file needed for backend support. If the \token{backend} is empty, the standard backend for the engine in use will be loaded. This command may only be used once.

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

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

- \texttt{pdftex}
- \texttt{luatex}
- \texttt{xetex}
- \texttt{dvips}
- \texttt{dvipdfmx}
- \texttt{dvisvgm}

\sys_load_debug:
Load the additional configuration file for debugging support.
10.8.1 Final settings

\sys_finalise: \sys_finalise:

Rev: 2019-10-06 Finalises all system-dependent functionality: required before loading a backend.
Chapter 11

The \texttt{l3msg} module

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

The system used by \texttt{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 \texttt{error}, \texttt{warning} or \texttt{info}.

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

11.1 Creating new messages

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

Messages may be subdivided \textit{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

\begin{verbatim}
\msg_new:nnnn { mymodule } { submodule / message } ...
\end{verbatim}

will allow to filter out specifically messages from the \texttt{submodule}.

Some authors may find the need to include spaces as - characters tedious. This can be avoided by locally reseting the category code of \texttt{\~}.

81
\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 \{⟨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 \{⟨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 \{⟨module⟩\} \{⟨message⟩\} \{⟨true code⟩\} \{⟨false code⟩\} Tests whether the ⟨message⟩ for the ⟨module⟩ is currently defined.

\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 \{⟨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 \LaTeX{} 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 \{\langle\text{module}\rangle\} \msg_fatal_text:n
Produces the standard text

\textbf{Fatal Package} \langle\text{module}\rangle Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the \langle\text{module}\rangle to be included.

\msg_critical_text:n \{\langle\text{module}\rangle\} \msg_critical_text:n
Produces the standard text

\textbf{Critical Package} \langle\text{module}\rangle Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the \langle\text{module}\rangle to be included.

\msg_error_text:n \{\langle\text{module}\rangle\} \msg_error_text:n
Produces the standard text

\textbf{Package} \langle\text{module}\rangle Error

This function can be redefined to alter the language in which the message is given, using #1 as the name of the \langle\text{module}\rangle to be included.

\msg_warning_text:n \{\langle\text{module}\rangle\} \msg_warning_text:n
Produces the standard text

\textbf{Package} \langle\text{module}\rangle Warning

This function can be redefined to alter the language in which the message is given, using #1 as the name of the \langle\text{module}\rangle to be included. The \langle\text{type}\rangle of \langle\text{module}\rangle may be adjusted: Package is the standard outcome: see \msg_module_type:n.

\msg_info_text:n \{\langle\text{module}\rangle\} \msg_info_text:n
Produces the standard text:

\textbf{Package} \langle\text{module}\rangle Info

This function can be redefined to alter the language in which the message is given, using #1 as the name of the \langle\text{module}\rangle to be included. The \langle\text{type}\rangle of \langle\text{module}\rangle may be adjusted: Package is the standard outcome: see \msg_module_type:n.
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.
Issues \textit{\texttt{module}} error \textit{\texttt{message}}, passing \textit{\texttt{arg one}} to \textit{\texttt{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).

\TeXhackers note: The \TeX \texttt{\endinput} primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.
Issues \texttt{\langle module\rangle} warning \texttt{\langle message\rangle}, passing \texttt{\langle arg\ one\rangle} to \texttt{\langle arg\ four\rangle} to the text-creating functions. The warning text is added to the log file and the terminal, but the \TeX run is not interrupted.

Issues \texttt{\langle module\rangle} information \texttt{\langle message\rangle}, passing \texttt{\langle arg\ one\rangle} to \texttt{\langle arg\ four\rangle} to the text-creating functions. For the more common \texttt{\langle module\rangle}, the information text is added to the log file only, while \texttt{\langle module\rangle} 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:nnee|nnnee
\msg_term:nnnn
\msg_term:nnV|nnV|nnee|nnnV|nnV
\msg_term:nnV
\msg_term:nn
\msg_log:nnnnnn
\msg_log:nneeee
\msg_log:nnnnn
\msg_log:nneee
\msg_log:nnnn
\msg_log:nnV|nnV|nnV|nnee|nnnee
\msg_log:nnnn
\msg_log:nn
\msg_log:nn
\msg_log:nn

Updated: 2012-08-11

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:nnnee
\msg_none:nnee|nnnee
\msg_none:nnnn
\msg_none:nnV|nnV|nnee|nnnV|nnV
\msg_none:nnV
\msg_none:nn
\msg_none:nn
\msg_none:nn

Updated: 2012-08-11

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 {\langle module \rangle} {\langle message \rangle} {\langle arg one \rangle} {\langle arg two \rangle} {\langle arg three \rangle} {\langle arg four \rangle}

New: 2017-12-04

Issues \langle module \rangle information \langle message \rangle, passing \langle arg one \rangle to \langle arg four \rangle 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 * \seq_map_function:NN \langle seq \rangle \msg_show_item:n
\msg_show_item_unbraced:n * \prop_map_function:NN \langle prop \rangle \msg_show_item:nn
\msg_show_item:nn *
\msg_show_item_unbraced:nn *

New: 2017-12-04

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:

>\langle a \rangle
>\langle \{b\} \rangle
>\langle \c \rangle

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 {module} {message} {arg one} {arg two} {arg three} {arg four}
\msg expandable error:nnffff {module} {message} {arg one} {arg two} {arg three} {arg four}
\msg expandable error:nnnn {module} {message} {arg one} {arg two} {arg three} {arg four}
\msg expandable error:nnff {module} {message} {arg one} {arg two} {arg three}
\msg expandable error:nnn {module} {message} {arg one} {arg two}
\msg expandable error:nnf {module} {message} {arg one}
\msg expandable error:nn {module}

Issues an “Undefined error” message from TeX itself using the undefined control sequence \::error then prints “! ⟨module⟩: "⟨error message⟩", which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

11.5 Redirecting messages

Each message has a “name”, which can be used to alter the behaviour of the message when it is given. Thus we might have

\msg new:nnnn {module} {my-message} {Some-text} {Some-more-text}
to define a message, with

\msg error:nn {module} {my-message}
when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

\msg redirect class:nn {error} {warning}
to turn all errors into warnings, or with

\msg redirect module:nnn {module} {error} {warning}
to alter only messages from that module, or even

\msg redirect name:nnn {module} {my-message} {warning}
to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error.
immediately. Infinite loops are prevented by eliminating the redirection starting from
the target of the redirection that caused the loop to appear. Namely, if redirections are
requested as $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is
cancelled.

\begin{verbatim}
\msg_redirect_class:nn \msg_redirect_class:nn \{class one\} \{class two\}
Updated: 2012-04-27
Changes the behaviour of messages of \{class one\} so that they are processed using the
code for those of \{class two\}. Each \{class\} can be one of fatal, critical, error,
warning, note, info, term, log, none.

\msg_redirect_module:nnn \msg_redirect_module:nnn \{module\} \{class one\} \{class two\}
Updated: 2012-04-27
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\}
Updated: 2012-04-27
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 \}
\end{verbatim}
Chapter 12

The \texttt{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 \texttt{file...}, while others are used to work with files on a line by line basis and have prefix \texttt{ior...} (reading) or \texttt{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 (\texttt{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 \texttt{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.
Globally reserves the name of the \texttt{stream}, either for reading or for writing as appropriate. The \texttt{stream} is not opened until the appropriate \texttt{\textbackslash open:Nn} function is used. Attempting to use a \texttt{stream} which has not been opened is an error, and the \texttt{stream} will behave as the corresponding \texttt{c_term:...}.

\texttt{\ior_open:Nn} \texttt{\ior_open:cn} \texttt{\iow_open:Nn} \texttt{\iow_open:c} \texttt{\ior_shell_open:Nn} \texttt{\iow_shell_open:Nn}
\texttt{\ior_open:NnTF} \texttt{\ior_open:cnTF} \texttt{\iow_open:NnTF} \texttt{\iow_open:cnTF}
\texttt{\ior_open:Nn} \texttt{\ior_open:(NV|cn|cV)}

Opens \texttt{file name} for reading using \texttt{stream} as the control sequence for file access. If the \texttt{stream} was already open it is closed before the new operation begins. The \texttt{stream} is available for access immediately and will remain allocated to \texttt{file name} until a \texttt{\ior_close:N} instruction is given or the \TeX{} run ends. If the file is not found, an error is raised.

Opens \texttt{file name} for writing using \texttt{stream} as the control sequence for file access. If the \texttt{stream} was already open it is closed before the new operation begins. The \texttt{stream} is available for access immediately and will remain allocated to \texttt{file name} until a \texttt{\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).
Closes the \texttt{stream}. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

Display (to the terminal or log file) the file name associated to the (read or write) \texttt{stream}.

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.

\subsection*{12.1.1 Reading from files}

Reading from files and reading from the terminal are separate processes in \texttt{expl3}. The functions \texttt{ior_get:NN} and \texttt{ior_str_get:NN}, and their branching equivalents, are designed to work with \texttt{files}. 
\ior_get:NN \ior_get:NN \langle stream \rangle \langle token list variable \rangle
\ior_get:NNTF \ior_get:NNTF \langle stream \rangle \langle token list variable \rangle \langle true code \rangle \langle false code \rangle

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input \langle stream \rangle and stores the result locally in the \langle token list \rangle variable. The material read from the \langle stream \rangle is tokenized by \TeX according to the category codes and \texttt{\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

\begin{verbatim}
a b c
\end{verbatim}
results in a token list \texttt{a b c}. Any blank line is converted to the token \texttt{\par}. Therefore, blank lines can be skipped by using a test such as

\begin{verbatim}
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl \{ \par \}
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl ...
\end{verbatim}

Also notice that if multiple lines are read to match braces then the resulting token list can contain \texttt{\par} tokens. In the non-branching version, where the \langle stream \rangle is not open the \langle tl var \rangle is set to \texttt{\q_no_value}.

\TeXhackers note: This protected macro is a wrapper around the \TeX primitive \texttt{\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 \texttt{\endlinechar}, omitted if \texttt{\endlinechar} is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

\ior_str_get:NN \ior_str_get:NN \langle stream \rangle \langle token list variable \rangle
\ior_str_get:NNTF \ior_str_get:NNTF \langle stream \rangle \langle token list variable \rangle \langle true code \rangle \langle false code \rangle

Function that reads one line from the file input \langle stream \rangle and stores the result locally in the \langle token list \rangle variable. The material is read from the \langle stream \rangle 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 \langle token list variable \rangle being empty. Unlike \ior_get:NN, line ends do not receive any special treatment. Thus input

\begin{verbatim}
a b c
\end{verbatim}
results in a token list \texttt{a b c} with the letters \texttt{a}, \texttt{b}, and \texttt{c} having category code 12. In the non-branching version, where the \langle stream \rangle is not open the \langle tl var \rangle is set to \texttt{\q_no_value}.

\TeXhackers note: This protected macro is a wrapper around the \texttt{\readline} \TeX primitive \texttt{\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, \textit{i.e.} any local assignments made
by the \textit{function} or \textit{code} discussed below remain in effect after the loop.

\ior_map_inline:Nn  \ior_map_inline:Nn \ior_map_variable:NNn
\texttt{\ior_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn}
\begin{verbatim}
\ior_map_inline:Nn \ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn
\end{verbatim}

\begin{verbatim}
\ior_map_inline:Nn \ior_map_inline:Nn \ior_get:NN \ior_str_map_inline:Nn \ior_map_variable:NNn \ior_str_map_variable:NNn
\end{verbatim}
\ior_map_break:  
\ior_map_break:  
New: 2012-06-29  

Used to terminate a \ior_map... function before all lines from the \langle stream \rangle have been processed. This normally takes place within a conditional statement, for example

\ior_map_inline:Nn \l_my_iormap
{  
\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.

\TeX{}hackers 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  
New: 2012-06-29  

Used to terminate a \ior_map... function before all lines in the (stream) have been processed, inserting the (code) after the mapping has ended. This normally takes place within a conditional statement, for example

\ior_map_inline:Nn \l_my_iormap
{  
\str_if_eq:nnTF { #1 } { bingo }  
{ \ior_map_break:n { \langle code \rangle } }  
{  
% Do something useful  
}  
}

Use outside of a \ior_map... scenario leads to low level \TeX{} errors.

\TeX{}hackers 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:NTF  
\ior_if_eof_p:N  
\ior_if_eof:NTF  

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.

96
12.1.2 Reading from the terminal

\lor_get_term:nN ⟨prompt⟩ ⟨token list variable⟩

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 \lor_get:NN or \lor_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. \lor_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

\lor_now:Nn ⟨stream⟩ {⟨tokens⟩}

This function writes ⟨tokens⟩ to the specified ⟨stream⟩ immediately (i.e. the write operation is called on expansion of \lor_now:Nn).

\lor_log:n {⟨tokens⟩}

This function writes the given ⟨tokens⟩ to the log (transcript) file immediately: it is a dedicated version of \lor_now:Nn.

\lor_term:n {⟨tokens⟩}

This function writes the given ⟨tokens⟩ to the terminal file immediately: it is a dedicated version of \lor_now:Nn.

\lor_shipout:Nn ⟨stream⟩ {⟨tokens⟩}

This function writes ⟨tokens⟩ to the specified ⟨stream⟩ when the current page is finalised (i.e. at shipout). The e-type variants expand the ⟨tokens⟩ at the point where the function is used but not when the resulting tokens are written to the ⟨stream⟩ (cf. \lor_shipout_e:Nn).

\TeX hackers note: When using expl3 with a format other than \LaTeX, new line characters inserted using \lor_newline: or using the line-wrapping code \lor_wrap:nnnN are not recognized in the argument of \lor_shipout:Nn. This may lead to the insertion of additional unwanted line-breaks.
This function writes \langle \text{tokens} \rangle to the specified \langle \text{stream} \rangle when the current page is finalised (\emph{i.e.} at shipout). The \langle \text{tokens} \rangle are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

\textbf{Text hackers note:} This is a wrapper around the \TeX{} primitive \texttt{\write}. When using expl3 with a format other than \LaTeX{}, new line characters inserted using \texttt{\iow_newline}: or using the line-wrapping code \texttt{\iow_wrap:nnnN} are not recognized in the argument of \texttt{\iow_shipout:Nn}. This may lead to the insertion of additional unwanted line-breaks.

\texttt{\iow_char:N} \langle \text{char} \rangle

Inserts \langle \text{char} \rangle into the output stream. Useful when trying to write difficult characters such as \%, \{, \}, \emph{etc.} in messages, for example:

\begin{verbatim}
\iow_now:Ne \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
\end{verbatim}

The function has no effect if writing is taking place without expansion (\emph{e.g.} in the second argument of \texttt{\iow_now:Nn}).

\texttt{\iow_newline:}

Function to add a new line within the \langle \text{tokens} \rangle written to a file. The function has no effect if writing is taking place without expansion (\emph{e.g.} in the second argument of \texttt{\iow_now:Nn}).

\textbf{Text hackers note:} When using expl3 with a format other than \LaTeX{}, the character inserted by \texttt{\iow_newline:} is not recognized by \TeX{}, which may lead to the insertion of additional unwanted line-breaks. This issue only affects \texttt{\iow_shipout:Nn}, \texttt{\iow_shipout_e:Nn} and direct uses of primitive operations.
12.1.4 Wrapping lines in output

This function wraps the \langle text\rangle to a fixed number of characters per line. At the start of each line which is wrapped, the \langle run-on text\rangle is inserted. The line character count targeted is the value of \l_iow_line_count_int minus the number of characters in the \langle run-on text\rangle 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 \langle text\rangle and \langle run-on text\rangle are exhaustively expanded by the function, with the following substitutions:

- \\\ or \texttt{\iow_newline}: may be used to force a new line,
- \\texttt{\textbackslash h}, \\texttt{\textbackslash h}, \\texttt{\textbackslash h}, \texttt{\textbackslash h} 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 \langle text\rangle (not the \langle run-on text\rangle).

Additional functions may be added to the wrapping by using the \langle set up\rangle, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the \langle text\rangle 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 \langle function\rangle, which is typically a wrapper around a write operation. The output of \iow_wrap:nnnN (i.e. the argument passed to the \langle function\rangle) 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 \langle text\rangle 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 \langle text\rangle.

\iow_wrap_allow_break: \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 \langle text\rangle by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the \langle text\rangle. In case the indented \langle text\rangle should appear on separate lines from the surrounding text, use \\\ to force line breaks.
12.1.5 Constant input–output streams, and variables

\texttt{\l_iow\_line\_count\_int}

The maximum number of characters in a line to be written by the \texttt{\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 MiK\TeX{} systems.

12.2 File operations

12.2.1 Basic file operations

\texttt{\g\_file\_curr\_dir\_str}
\texttt{\g\_file\_curr\_name\_str}
\texttt{\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 (\textit{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 \texttt{\langle name\rangle} and \texttt{\langle ext\rangle} parts together make up the file name, thus the \texttt{\langle name\rangle} part may be thought of as the “job name” for the current file.

Note that \TeX{} does not provide information on the \texttt{\langle dir\rangle} and \texttt{\langle ext\rangle} part for the main (top level) file and that this file always has empty \texttt{\langle dir\rangle} and \texttt{\langle ext\rangle} components. Also, the \texttt{\langle name\rangle} here will be equal to \texttt{\c\_sys\_jobname\_str}, which may be different from the real file name (if set using \texttt{--jobname}, for example).
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, expl3 will automatically append the current \inputpath to the set of values from \file_search_path_seq.

```
\file_if_exist_p:n {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}
\file_if_exist_p:V
\file_if_exist:nTF
\file_if_exist:VnTF
```

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 \file_search_path_seq.

```
\file_hex_dump:n {⟨file name⟩}
\file_hex_dump:V
\file_hex_dump:nnn
\file_hex_dump:Vnnn
```

Searches for \file name using the current \TeX\ search path and the additional paths controlled by \file_search_path_seq. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most \TeX\ behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The \file_hex_dump can be used with \str_range to get a specific part of the file.

```
\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 to the \file. If the file is not found, the \tl var will be set to \q_no_value.
\file_mdfive_hash:n \file_mdfive_hash:V

New: 2019-09-03

Searches for ⟨file name⟩ using the current \TeX\ search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most \TeX\ behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.

\file_get_mdfive_hash:n \file_get_mdfive_hash:VN \file_get_mdfive_hash:nN

\file_get_mdfive_hash:nN \langle file name⟩ \langle tl var⟩

Sets the ⟨tl var⟩ to the result of applying \file_mdfive_hash:n to the ⟨file⟩. If the file is not found, the ⟨tl var⟩ will be set to \q_no_value.

\file_size:n \file_size:V

New: 2019-09-03

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:n \file_get_size:VN \file_get_size:nN

\file_get_size:nN \langle file name⟩ \langle 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. This is not available in older versions of \Xe\TeX.

\file_timestamp:n \file_timestamp:V

New: 2019-09-03

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. This is not available in older versions of \Xe\TeX.

\file_get_timestamp:n \file_get_timestamp:VN \file_get_timestamp:nN

\file_get_timestamp:nN \langle file name⟩ \langle 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. This is not available in older versions of \Xe\TeX.
\file_compare_timestamp_p:nNn × \file_compare_timestamp_p:nNn \{\text{file-1}\} \text{(comparator)}
\file_compare_timestamp_p:nNn \{\text{file-2}\}
\file_compare_timestamp:nNnTF × \file_compare_timestamp:nNnTF \{\text{file-1}\} \text{(comparator)}
\file_compare_timestamp:nNnTF \{\text{file-2}\} \{\text{true code}\} \{\text{false code}\}

Updated: 2019-05-13
New: 2019-09-20

Compares the file stamps on the two \textit{files} as indicated by the \textit{comparator}, and inserts either the \textit{true code} or \textit{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

\begin{verbatim}
\file_compare_timestamp:nNnTF \{ source-file \} > \{ derived-file \}
{ % Code to regenerate derived file }
\end{verbatim}

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different. This is not available in older versions of X\TeX.  

\file_get_full_name:nN \{\text{file name}\} \{\text{tl}\}
\file_get_full_name:nNTF \{\text{file name}\} \{\text{tl}\} \{\text{true code}\} \{\text{false code}\}

Updated: 2019-02-16
New: 2019-09-03

Searches for \textit{file name} in the path as detailed for \texttt{\file_if_exist:nTF}, and if found sets the \texttt{\tl\ var} the fully-qualified name of the file, \textit{i.e.} the path and file name. This includes an extension \texttt{.tex} when the given \textit{file name} has no extension but the file found has that extension. In the non-branching version, the \texttt{\tl\ var} will be set to \texttt{\q_no_value} in the case that the file does not exist.

\file_full_name:n \{\text{file name}\}

Updated: 2019-02-16

Searches for \textit{file name} in the path as detailed for \texttt{\file_if_exist:nTF}, and if found leaves the fully-qualified name of the file, \textit{i.e.} the path and file name, in the input stream. This includes an extension \texttt{.tex} when the given \textit{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 \{\text{full name}\} \{\text{dir}\} \{\text{name}\} \{\text{ext}\}
\file_parse_full_name:VNNN

New: 2017-06-23
Updated: 2020-06-24

Parses the \textit{full name} and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The \texttt{\dir}: everything up to the last / (path separator) in the \textit{file path}. As with system \texttt{PATH} variables and related functions, the \texttt{\dir} does not include the trailing / unless it points to the root directory. If there is no path (only a file name), \texttt{\dir} is empty.
- The \texttt{\name}: everything after the last / up to the last ., where both of those characters are optional. The \texttt{\name} may contain multiple . characters. It is empty if \texttt{\full name} consists only of a directory name.
- The \texttt{\ext}: everything after the last . (including the dot). The \texttt{\ext} is empty if there is no . after the last /.

Before parsing, the \textit{full name} is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (\texttt{"}) are invalid in file names and are discarded from the input.
\file_parse_full_name:n * \file_parse_full_name:V * \file_parse_full_name:apply:nN * \file_parse_full_name:apply:WN *  
\file_parse_full_name:apply:nN \{\text{full name}\} \{\text{function}\}  

Parses the \textit{full name} as described for \file_parse_full_name:nNNN, and leaves \textit{dir}, \textit{name}, and \textit{ext} in the input stream, each inside a pair of braces.

12.2.3 Accessing file contents

\file_get:nnN \{\textit{file name}\} \{\textit{setup}\} \{\textit{tl}\}  
\file_get:VnN \{\textit{file name}\} \{\textit{setup}\} \{\textit{true code}\} \{\textit{false code}\}  
\file_get:nNTF \{\textit{file name}\} \{\textit{setup}\} \{\textit{true code}\} \{\textit{false code}\}  
\file_get:VnNTF \{\textit{file name}\} \{\textit{setup}\} \{\textit{true code}\} \{\textit{false code}\}  

Defines \textit{tl} to the contents of \textit{file name}. Category codes may need to be set appropriately via the \textit{setup} argument. The non-branching version sets the \textit{tl} to \texttt{no_value} if the file is not found. The branching version runs the \textit{true code} after the assignment to \textit{tl} if the file is found, and \textit{false code} otherwise. The file content will be tokenized using the current category code régime.

\file_input:n \{\textit{file name}\}  
\file_input:V  
\file_input_raw:n \{\textit{file name}\}  
\file_input_raw:V  

Searches for \textit{file name} in the path as detailed for \file_if_exist:nTF, and if found reads in the file as additional \LaTeX{} source. No data concerning the file is tracked. If the file is not found, no action is taken.

\TeX{}hackers note: This function is intended only for contexts where files must be read purely by expansion, for example at the start of a table cell in an \verb|\halign|.

\file_if_exist_input:n \{\textit{file name}\}  
\file_if_exist_input:V \{\textit{false code}\}  
\file_if_exist_input:nF \{\textit{false code}\}  
\file_if_exist_input:VF \{\textit{false code}\}  

Searches for \textit{file name} using the current \TeX{} search path and the additional paths included in \texttt{\_file_search_path_seq}. If found then reads in the file as additional \TeX{} source as described for \file_input:n, otherwise inserts the \textit{false code}. Note that these functions do not raise an error if the file is not found, in contrast to \file_input:n.
\texttt{\textbackslash file_input_stop:}\ New: 2017-07-07

Ends the reading of a file started by \texttt{\textbackslash file_input:n} or similar before the end of the file is reached. Where the file reading is being terminated due to an error, \texttt{\textbackslash 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!

\texttt{\textbackslash file_show_list:}\ \texttt{\textbackslash file_log_list:}

These functions list all files loaded by \LTX commands that populate \texttt{@filelist} or by \texttt{\textbackslash file_input:n}. While \texttt{\textbackslash file_show_list:} displays the list in the terminal, \texttt{\textbackslash file\_log\_list:} outputs it to the log file only.
Chapter 13

The l3luatex module
Lua\TeX-\emph{specific} functions

The Lua\TeX\ 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 pdf\TeX, d\TeX, up\TeX\ or X\TeX\ these raise an error: use \texttt{\sys_if_engine_luatex:T} to avoid this. Details on using Lua with the Lua\TeX\ engine are given in the Lua\TeX\ manual.

13.1 Breaking out to Lua

\begin{center}
\begin{tabular}{ll}
\texttt{lua_now:n} & \texttt{lua_now:n \{token list\}}
\end{tabular}
\end{center}

The \texttt{\{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 \texttt{\{Lua input\}} is passed to the Lua interpreter for processing. Each \texttt{\lua_now:n} block is treated by Lua as a separate chunk. The Lua interpreter will execute the \texttt{\{Lua input\}} immediately, and in an expandable manner.

\textbf{\TeXhackers note:} \texttt{\lua_now:e} is a macro wrapper around \texttt{\directlua}: when Lua\TeX\ is in use two expansions are required to yield the result of the Lua code.

\begin{center}
\begin{tabular}{ll}
\texttt{lua_shipout:n} & \texttt{lua_shipout:n \{token list\}}
\end{tabular}
\end{center}

The \texttt{\{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 \texttt{\{Lua input\}} is passed to the Lua interpreter when the current page is finalised (\textit{i.e.} at shipout). Each \texttt{\lua_shipout:n} block is treated by Lua as a separate chunk. The Lua interpreter will execute the \texttt{\{Lua input\}} during the page-building routine: no \TeX\ expansion of the \texttt{\{Lua input\}} will occur at this stage.

In the case of the \texttt{\lua_shipout_e:n} version the input is fully expanded by \TeX\ in an \texttt{e}-type manner during the shipout operation.

\textbf{\TeXhackers note:} At a \TeX\ level, the \texttt{\{Lua input\}} is stored as a “whatsit”.

106
\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 \LaTeX{} is in use two expansions are required to yield the result of the Lua code.

\lua_load_module:n \{Lua module name\}

Loads a Lua module into the Lua interpreter. \lua_now:n passes its \{token list\} argument to the Lua interpreter as a single line, with characters interpreted under the current catcode regime. These two facts mean that \lua_now:n rarely behaves as expected for larger pieces of code. Therefore, package authors should not write significant amounts of Lua code in the arguments to \lua_now:n. Instead, it is strongly recommended that they write the majority 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 \LaTeX{}, there are a small number of Lua functions provided here.

\texttt{ltx.utils}

Most public interfaces provided by the module are stored within the \texttt{ltx.utils} table.

\texttt{ltx.utils.filedump}

\begin{quote}
\texttt{(dump) = ltx.utils.filedump\{(file),(offset),(length)\}}
\end{quote}

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\}.

\texttt{ltx.utils.filemd5sum}

\begin{quote}
\texttt{(hash) = ltx.utils.filemd5sum\{(file)\}}
\end{quote}

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 \LaTeX{} behaviour. If the \{file\} is not found, nothing is returned with no error raised.

\texttt{ltx.utils.filemoddate}

\begin{quote}
\texttt{(date) = ltx.utils.filemoddate\{(file)\}}
\end{quote}

Returns the date/time of last modification of the \{file\} in the format

\begin{center}
D:\{year\}\{month\}\{day\}\{hour\}\{minute\}\{second\}\{offset\}
\end{center}

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.
\texttt{ltx_utils.filesize} \texttt{size} = \texttt{ltx_utils.filesize}((\texttt{file}))

Returns the size of the \texttt{(file)} in bytes. If the \texttt{(file)} is not found, nothing is returned with \textit{no error raised}. 
Chapter 14

The \texttt{l3}legacy module

Interfaces to legacy concepts

There are a small number of \TeX{} or \LaTeX{} concepts which are not used in \texttt{expl3} code but which need to be manipulated when working as a \LaTeX{} \texttt{2e} package. To allow these to be integrated cleanly into \texttt{expl3} code, a set of legacy interfaces are provided here.

\begin{verbatim}
\legacy_if_p:n    \legacy_if_p:n \langle name \rangle
\legacy_if:nTF   \legacy_if:nTF \langle name \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}
\end{verbatim}

Tests if the \LaTeX\texttt{}/plain \TeX{} conditional (generated by \texttt{\newif}) is \texttt{true} or \texttt{false} and branches accordingly. The \langle name \rangle of the conditional should \emph{omit} the leading \texttt{if}.

\begin{verbatim}
\legacy_if_set_true:n \legacy_if_set_true:n \langle name \rangle
\legacy_if_set_false:n \legacy_if_set_false:n \langle name \rangle
\legacy_if_gset_true:n \legacy_if_gset_true:n \langle name \rangle
\legacy_if_gset_false:n \legacy_if_gset_false:n \langle name \rangle
\end{verbatim}

Sets the \LaTeX\texttt{}/plain \TeX{} conditional \texttt{\if\langle name \rangle} (generated by \texttt{\newif}) to be \texttt{true} or \texttt{false}.

\begin{verbatim}
\legacy_if_set:nn \legacy_if_set:nn \langle name \rangle \{\langle boolean expression \rangle\}
\legacy_if_gset:nn \legacy_if_gset:nn \langle name \rangle \{\langle boolean expression \rangle\}
\end{verbatim}

Sets the \LaTeX\texttt{}/plain \TeX{} conditional \texttt{\if\langle name \rangle} (generated by \texttt{\newif}) to the result of evaluating the \langle boolean expression \rangle.
Part IV

Data types
Chapter 15

The \texttt{l3tl} module

Token lists

\LaTeX{} works with tokens, and \LaTeX{}3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

\begin{verbatim}
\foo:n { a collection of \texttt{tokens} }
\end{verbatim}

or may be stored in a so-called “token list variable”, which have the suffix \texttt{tl}: a token list variable can also be used as the argument to a function, for example

\begin{verbatim}
\foo:N \l_some_tl
\end{verbatim}

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix \texttt{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 \texttt{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 \texttt{N} argument, or \texttt{␣}, \texttt{\{} or \texttt{\}} (assuming normal \TeX{} category codes). Thus for example

\begin{verbatim}
{ Hello } \texttt{–} world
\end{verbatim}

contains six items (Hello, w, o, r, l and d), but thirteen tokens (\texttt{\{} H, e, l, o, \texttt{\}}, \texttt{\␣}, w, o, r, l and d). Functions which act on items are often faster than their analogue acting directly on tokens.

15.1 Creating and initialising token list variables

\begin{verbatim}
\tl_new:N \tl_new:N \tl_new:C
\end{verbatim}

Creates a new \texttt{(tl \texttt{var})} or raises an error if the name is already taken. The declaration is global. The \texttt{(tl \texttt{var})} is initially empty.
\tl_const:Nn \tl_const:(Ne|cn|ce)
\tl_const:Nn \tl var \{\token list\}

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

\tl_clear:N \tl_clear:c \tl_gclear:N \tl_gclear:c
\tl_clear:N \tl_clear:c
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_clear_new:c
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}
Sets the content of \tl var equal to that of \tl var.

\tl_concat:NNN \tl_concat:ccc \tl_gconcat:NNN \tl_gconcat:ccc
\tl_concat:NNN \tl_concat:ccc
Concatenates the content of \tl var and \tl var together and saves the result in \tl var. The \tl var is placed at the left side of the new token list.

\tl_if_exist_p:N \tl_if_exist_p:c \tl_if_exist:NTF \tl_if_exist:c TF
\tl_if_exist_p:N \tl_if_exist_p:c \tl_if_exist:NTF \tl_if_exist:c TF
Tests whether the \tl var is currently defined. This does not check that the \tl var really is a token list variable.

\tl_if_exist_p:⋆ \tl_if_exist_p:c ⋆ \tl_if_exist:⋆ \tl_if_exist:c ⋆
\tl_if_exist_p:⋆ \tl_if_exist_p:c ⋆ \tl_if_exist:⋆ \tl_if_exist:c ⋆

15.2 Adding data to token list variables

\tl_set:Nn \tl_set:Nn \tl_set:(NV|Nv|Ne|Nf|cn|cV|cv|co|ce|cf)
\tl_gset:Nn \tl_gset:Nn \tl_gset:(NV|Nv|Ne|Nf|cn|cV|cv|co|ce|cf)

Sets \tl var to contain \token, removing any previous content from the variable.

\tl_put_left:Nn \tl_put_left:Nn \tl_put_left:(NV|Nv|Ne|Nf|cn|cV|cv|co|ce)
\tl_gput_left:Nn \tl_gput_left:Nn \tl_gput_left:(NV|Nv|Ne|Nf|cn|cV|cv|co|ce)

Appends \token to the left side of the current content of \tl var.
\texttt{\tl_put_right:Nn} \langle \textit{tokens} \rangle \texttt{\tl_put_right:(NV|Nv|No|cn|cV|cV|ce|co)} \texttt{\tl_put_right:Nn} \langle \textit{tl var} \rangle \texttt{\tl_put_right:(NV|Nv|No|cn|cV|cV|ce|co)}

Appends \textit{tokens} to the right side of the current content of \textit{tl var}.

### 15.3 Token list conditionals

\texttt{\tl_if_blank_p:n} {\langle \textit{token list} \rangle} \texttt{\tl_if_blank:nTF} {\langle \textit{token list} \rangle} \{\langle \textit{true code} \rangle\} \{\langle \textit{false code} \rangle\}

Tests if the \textit{token list} consists only of blank spaces \textit{(i.e. contains no item)}. The test is true if \langle \textit{token list} \rangle is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is false otherwise.

\texttt{\tl_if_empty_p:N} \langle \textit{tl var} \rangle \texttt{\tl_if_empty:NTF} \langle \textit{tl var} \rangle \{\langle \textit{true code} \rangle\} \{\langle \textit{false code} \rangle\}

Tests if the \langle \textit{token list variable} \rangle is entirely empty \textit{(i.e. contains no tokens at all)}.

\texttt{\tl_if_eq_p:NN} \langle \textit{tl var}_1 \rangle \langle \textit{tl var}_2 \rangle \texttt{\tl_if_eq:NNTF} \langle \textit{tl var}_1 \rangle \langle \textit{tl var}_2 \rangle \{\langle \textit{true code} \rangle\} \{\langle \textit{false code} \rangle\}

Compares the content of two \langle \textit{token list variables} \rangle and is logically true if the two contain the same list of tokens \textit{(i.e. identical in both the list of characters they contain and the category codes of those characters)}. Thus for example

\begin{verbatim}
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Ne \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NN \l_tmpa_tl \l_tmpb_tl \{ true \} \{ false \}
\end{verbatim}

yields false. See also \texttt{\str_if_eq:nnTF} for a comparison that ignores category codes.

\texttt{\tl_if_eq:NN} \langle \textit{tl var}_1 \rangle \langle \textit{tl var}_2 \rangle \{\langle \textit{token list}_2 \rangle\} \{\langle \textit{false code} \rangle\}

Tests if the \langle \textit{token list variable}_1 \rangle and the \langle \textit{token list}_2 \rangle contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see \texttt{\tl_if_eq:NNTF} for an expandable version when both token lists are stored in variables, or \texttt{\str_if_eq:nnTF} if category codes are not important.
\tl_if_in: \{token list\} \{token list\} \{true code\} \{false code\}
Tests if \{token list\} and \{token list\} contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see \tl_if_eq:NTF for an expandable version when token lists are stored in variables, or \str_if_eq:ntTF if category codes are not important.

\tl_if_in: \{begin-group\} \{end-group\}, and tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\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 var\}
\tl_if_single_p:c \{tl var\} \{true code\} \{false code\}
Tests if the content of the \{tl var\} contains a single \{item\}, i.e. 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 \{token list\}
\tl_if_single_p:ntF \{token list\} \{true code\} \{false code\}
Tests if the \{token list\} contains 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 \{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 \{\ldots\} are not single tokens.
15.3.1 Testing the first token

\texttt{tl_if_head_eq_catcode_p:nN} \star \texttt{tl_if_head_eq_catcode_p:nN} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_eq_catcode_p:(VN|eN|oN):nN} \star \texttt{tl_if_head_eq_catcode:nNTF} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_eq_catcode:(VN|eN|oN):nN} \star \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the first \texttt{token} in the \texttt{token list} has the same category code as the \texttt{test token}. In the case where the \texttt{token list} is empty, the test is always \texttt{false}.

\texttt{tl_if_head_eq_charcode_p:nN} \star \texttt{tl_if_head_eq_charcode_p:nN} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_eq_charcode:(VN|eN|fN):nN} \star \texttt{tl_if_head_eq_charcode:nNTF} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_eq_charcode:(VN|eN|fN):nN} \star \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the first \texttt{token} in the \texttt{token list} has the same character code as the \texttt{test token}. In the case where the \texttt{token list} is empty, the test is always \texttt{false}.

\texttt{tl_if_head_eq_meaning_p:nN} \star \texttt{tl_if_head_eq_meaning_p:nN} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_eq_meaning:(VN|eN):nN} \star \texttt{tl_if_head_eq_meaning:nNTF} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_eq_meaning:(VN|eN):nN} \star \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the first \texttt{token} in the \texttt{token list} has the same meaning as the \texttt{test token}. In the case where a \texttt{token list} is empty, the test is always \texttt{false}.

\texttt{tl_if_head_is_group_p:nN} \star \texttt{tl_if_head_is_group_p:nN} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_is_group:(VN|eN):nN} \star \texttt{tl_if_head_is_group:nNTF} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_is_group:(VN|eN):nN} \star \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the first \texttt{token} in the \texttt{token list} is an explicit begin-group character (with code 1 and any character code), in other words, if the \texttt{token list} starts with a brace group. In particular, the test is \texttt{false} if the \texttt{token list} starts with an implicit token such as \texttt{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.

\texttt{tl_if_head_is_N_type_p:nN} \star \texttt{tl_if_head_is_N_type_p:nN} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_is_N_type:(VN|eN):nN} \star \texttt{tl_if_head_is_N_type:nNTF} \{\texttt{token list}\} \{\texttt{test token}\}
\texttt{tl_if_head_is_N_type:(VN|eN):nN} \star \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the first \texttt{token} in the \texttt{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 code 1 and any character code). An empty argument yields \texttt{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.

Updated: 2012-07-08

Updated: 2012-07-09
Tests if the first \textit{token} in the \textit{token list} is an explicit space character (explicit token with character code 32 and category code 10). In particular, the test is \textit{false} if the \textit{token list} starts with an implicit token such as \texttt{\cs{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

\texttt{\tl{to_str:n}}\*\texttt{\tl{to_str:n}}\{\langle token list\rangle\}

Converts the \textit{token list} to a \textit{string}, leaving the resulting character tokens in the input stream. A \textit{string} is a series of tokens with category code 12 \textit{(other)} with the exception of spaces, which retain category code 10 \textit{(space)}. The base function requires only a single expansion. Its argument \textit{must} be braced.

\textbf{TeXhackers note:} This is the \eTeX{} primitive \texttt{\detokenize}. Converting a \textit{token list} to a \textit{string} yields a concatenation of the string representations of every token in the \textit{token list}. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter \texttt{\escapechar}, absent if the \texttt{\escapechar} is negative or greater than the largest character code;
- the control sequence name, as defined by \texttt{\cs{to_str:N}};
- a space, unless the control sequence name is a single character whose category at the time of expansion of \texttt{\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 \texttt{\escapechar}: for instance \texttt{\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 \texttt{a} is currently not a letter.

\texttt{\tl{use:N}}\*\texttt{\tl{use:N}}\{\langle tl var\rangle\}

Recovers the content of a \textit{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 \textit{tl var} directly without an accessor function.
15.4.2 Counting and reversing token lists

\tl_count:n \{tokens\}
\tl_count:(V|eo) *
\tl_count_tokens:n \{tokens\}

Counts the number of \{items\} in \{tokens\} and leaves this information in the input stream. Unbraced tokens count as one element as do each token group (\{\ldots\}). This process ignores any unprotected spaces within \{tokens\}. See also \tl_count:N. This function requires three expansions, giving an \langle integer\ denotation\rangle.

\tl_count:N = \tl_count:N \{tl\ var\}
\tl_count:c *

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 (\{\ldots\}). This process ignores any unprotected spaces within the \{tl\ var\}. See also \tl_count:n. This function requires three expansions, giving an \langle integer\ denotation\rangle.

\tl_reverse:n \{token list\}
\tl_reverse:(V|fe) *

Reverses the order of the \{items\} in the \{token list\}, so that \{item\}{item}\{item\} \ldots \{item\}{item}\{item\}{item}\{item\} becomes \{item\}{item}\{item\}{item}\{item\}{item}\{item\}. 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.

\textbf{\LaTeX{}hackers note}: The result is returned within \langle unexpanded\rangle, 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\ var\}
\tl_reverse:c
\tl_reverse:V
\tl_reverse:c

Sets the \{tl\ var\} to contain the result of reversing the order of its \{items\}, so that \{item\}{item}\{item\}{item}\{item\} \ldots \{item\}{item}\{item\}{item}\{item\}{item}\{item\}{item}\{item\} becomes \{item\}{item}\{item\}{item}\{item\}{item}\{item\}{item}\{item\}{item}. This process preserves unprotected spaces within the \{token list variable\}. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. 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 \{token list\}
\tl_reverse_items:n \{token list\}

Reverses the order of the \{items\} in the \{token list\}, so that \{item\}{item}\{item\}{item}\{item\} \ldots \{item\}{item}\{item\}{item}\{item\}{item}\{item\}{item}\{item\}{item} becomes \{item\}{item}\{item\}{item}\{item\}{item}\{item\}{item}\{item\}{item}. 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.

\textbf{\LaTeX{}hackers note}: The result is returned within \langle unexpanded\rangle, 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|x|o) \*  
Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the \texttt{\{token list\}} and leaves the result in the input stream.  

\textbf{\TeXhackers note:} The result is returned within \texttt{\textbackslash 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:nN \{\texttt{\{token list\}}\} \{\texttt{\{function\}}\}  
Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the \texttt{\{token list\}} and passes the result to the \texttt{\{function\}} as an $n$-type argument.

\tl_trim_spaces:N \tl_trim_spaces:c \tl_gtrim_spaces:N \tl_gtrim_spaces:c  
Sets the \texttt{\{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.

\subsection{15.4.3 Viewing token lists}

\tl_show:N \tl_show:c  
Displays the content of the \texttt{\{tl var\}} on the terminal.  

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

\tl_show:n \tl_show:e  
Displays the \texttt{\{token list\}} on the terminal.  

\textbf{\TeXhackers note:} This is similar to the $\varepsilon\text{-}\TeX$ primitive \texttt{\showtokens}, wrapped to a fixed number of characters per line.

\tl_log:N \tl_log:c  
Writes the content of the \texttt{\{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)  
Writes the \texttt{\{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

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:cN

Updated: 2012-06-29

\tl_map_function:nN \tl_map_function:cn

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_function:nN \tl_map_function:cn

Updated: 2012-06-29

\tl_map_tokens:Nn \tl_map_tokens:cn \tl_map_tokens:nn

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 ⟨tokens⟩ as a trailing brace group. For instance,

\tl_map_tokens:Nn \l_my_tl \prg_replicate:nn \{ \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:NN \tl_map_variable:cn

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 \lbrace \langle \text{token list} \rangle \rbrace \langle \text{variable} \rangle \lbrace \langle \text{code} \rangle \rbrace

Stores each \langle \text{item} \rangle of the \langle \text{token list} \rangle in turn in the \langle \text{token list} \rangle \langle \text{variable} \rangle and applies the \langle \text{code} \rangle. The \langle \text{code} \rangle will usually make use of the \langle \text{variable} \rangle, but this is not enforced. The assignments to the \langle \text{variable} \rangle are local. Its value after the loop is the last \langle \text{item} \rangle in the \langle \text{tl var} \rangle, or its original value if the \langle \text{tl var} \rangle 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 \langle \text{token list variable} \rangle have been processed. This normally takes place within a conditional statement, for example

\tl_map_inline:Nn \l_my_tl
\lbrace
  \str_if_eq:nnT \lbrace \text{#1} \rbrace \lbrace \text{bingo} \rbrace \lbrace \tl_map_break: \rbrace
  \lbrace \text{Do something useful} \rbrace
\rbrace

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 the \langle \text{tokens} \rangle are inserted into the input stream. This depends on the design of the mapping function.

\tl_map_break:n \tl_map_break:n \lbrace \langle \text{code} \rangle \rbrace

Used to terminate a \tl_map_... function before all entries in the \langle \text{token list variable} \rangle have been processed, inserting the \langle \text{code} \rangle after the mapping has ended. This normally takes place within a conditional statement, for example

\tl_map_inline:Nn \l_my_tl
\lbrace
  \str_if_eq:nnT \lbrace \text{#1} \rbrace \lbrace \text{bingo} \rbrace
  \lbrace \tl_map_break:n \lbrace \text{<code>} \rbrace \rbrace
  \lbrace \text{Do something useful} \rbrace
\rbrace

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 \langle \text{code} \rangle 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.
Leaves in the input stream the first \langle item \rangle in the \langle token list \rangle, discarding the rest of the \langle token list \rangle. 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 \texttt{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 \texttt{\_ab}. A blank \langle token list \rangle (see \tl_if_blank:nTF) results in \tl_head:n leaving nothing in the input stream.

\textbf{TEXhackers note}: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion.

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first \langle item \rangle in the \langle token list \rangle, 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 \texttt{\_bc}d in the input stream. A blank \langle token list \rangle (see \tl_if_blank:nTF) results in \tl_tail:n leaving nothing in the input stream.

\textbf{TEXhackers note}: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an \texttt{e}-type or \texttt{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 {⟨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.

\textbf{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 \langle tl var \rangle
\tl_rand_item:c \langle token list \rangle
\tl_rand_item:n \langle token list \rangle

New: 2016-12-06

Selects a pseudo-random item of the ⟨token list⟩. If the ⟨token list⟩ is blank, the result is empty. This is not available in older versions of Xe\TeX.

\textbf{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.
Leaves in the input stream the items from the \( \text{(start index)} \) to the \( \text{(end index)} \) inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here \( \text{(start index)} \) and \( \text{(end index)} \) should be \( \text{(integer expressions)} \). 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 \texttt{\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

\[
\begin{align*}
\texttt{\tl_range:nnn \{ abcd-{e{}}fg \} \{ 1 \} \{ 7 \}} \\
\texttt{\tl_range:nnn \{ abcd-{e{}}fg \} \{ 1 \} \{ 12 \}} \\
\texttt{\tl_range:nnn \{ abcd-{e{}}fg \} \{ -7 \} \{ 7 \}} \\
\texttt{\tl_range:nnn \{ abcd-{e{}}fg \} \{ -12 \} \{ 7 \}}
\end{align*}
\]

Here are some more interesting examples. The calls

\[
\begin{align*}
\texttt{\iow_term:e \{ \tl_range:nnn \{ abcd{e{}}fg \} \{ 2 \} \{ 5 \} \}} \\
\texttt{\iow_term:e \{ \tl_range:nnn \{ abcd{e{}}fg \} \{ 2 \} \{ -3 \} \}} \\
\texttt{\iow_term:e \{ \tl_range:nnn \{ abcd{e{}}fg \} \{ -6 \} \{ 5 \} \}} \\
\texttt{\iow_term:e \{ \tl_range:nnn \{ abcd{e{}}fg \} \{ -6 \} \{ -3 \} \}}
\end{align*}
\]

are all equivalent and will print \( bcd{e{}} \) on the terminal; similarly

\[
\begin{align*}
\texttt{\iow_term:e \{ \tl_range:nnn \{ abcd-{e{}}fg \} \{ 2 \} \{ 5 \} \}} \\
\texttt{\iow_term:e \{ \tl_range:nnn \{ abcd-{e{}}fg \} \{ 2 \} \{ -3 \} \}} \\
\texttt{\iow_term:e \{ \tl_range:nnn \{ abcd-{e{}}fg \} \{ -6 \} \{ 5 \} \}} \\
\texttt{\iow_term:e \{ \tl_range:nnn \{ abcd-{e{}}fg \} \{ -6 \} \{ -3 \} \}}
\end{align*}
\]

are all equivalent and will print \( bcd \ {e{}} \) on the terminal (note the space in the middle).

To the contrary,

\[
\begin{align*}
\texttt{\tl_range:nnn \{ abcd-{e{}}f \} \{ 2 \} \{ 4 \}}
\end{align*}
\]

will discard the space after ‘d’.

If we want to get the items from, say, the third to the last in a token list \(<\texttt{tl}>\), the call is \texttt{\tl_range:nnn \{ <\texttt{tl}> \} \{ 3 \} \{ -1 \}}. Similarly, for discarding the last item, we can do \texttt{\tl_range:nnn \{ <\texttt{tl}> \} \{ 1 \} \{ -2 \}}.

\textbf{\TeXhackers note:} The result is returned within the \texttt{\unexpanded} primitive \texttt{\exp_not:n}, which means that the \texttt{(item)} does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion.
15.5.4 Sorting token lists

\tl_sort:Nn \tl_sort:cn
\tl_gsort:Nn \tl_gsort:cn

Sorts the items in the \langle tl var \rangle according to the \langle comparison code \rangle, and assigns the result to \langle tl var \rangle. The details of sorting comparison are described in Section 6.1.

\tl_sort:nN *
\tl_sort:nN \langle\text{token list}\rangle \langle\text{conditional}\rangle

Sorts the items in the \langle token list \rangle, using the \langle conditional \rangle to compare items, and leaves the result in the input stream. The \langle conditional \rangle 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.

\textit{\TeX hackers note}: The result is returned within \texttt{\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|cnn|cVn|cnV|cen|cne|cee)
\tl_greplace_once:Nnn \tl_greplace_once:(NVn|NnV|Nen|Nne|cnn|cVn|cnV|cen|cne|cee)

Replaces the first (leftmost) occurrence of \langle old tokens \rangle in the \langle tl var \rangle with \langle new tokens \rangle. \langle Old tokens \rangle cannot contain \{, \} or \# (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).
Replaces all occurrences of \(\langle\text{old tokens}\rangle\) in the \(\langle\text{tl var}\rangle\) with \(\langle\text{new tokens}\rangle\). (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 \(\texttt{\tl_remove_all:Nn}\) for an example).

Removes the first (leftmost) occurrence of \(\langle\text{tokens}\rangle\) from the \(\langle\text{tl var}\rangle\). (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).

Removes all occurrences of \(\langle\text{tokens}\rangle\) from the \(\langle\text{tl var}\rangle\). (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,

\[
\texttt{\tl_set:Nn } \l_{\text{tmpa_tl}} \{a b c d c d} \texttt{ \tl_remove_all:Nn } \l_{\text{tmpa_tl}} \{b c}\]
results in \(\l_{\text{tmpa_tl}}\) 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 \(\texttt{\char_generate:nn}\)).
\tl_set_rescan:Nnn \tl_set_rescan:Nnn \tl_set_rescan:Nnn \tl_set_rescan:Nnn \tl_set_rescan:Nnn \tl_set_rescan:Nnn
\tl_gset_rescan:Nnn \tl_gset_rescan:Nnn \tl_gset_rescan:Nnn \tl_gset_rescan:Nnn
Updated: 2015-08-11

Sets \texttt{tl \ var} to contain \texttt{tokens}, applying the category code régime specified in the \texttt{setup} before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the \texttt{setup} are those in force at the point of use of \texttt{\tl_set_rescan:Nnn}.) This allows the \texttt{tl \ var} to contain material with category codes other than those that apply when \texttt{tokens} are absorbed. The \texttt{setup} is run within a group and may contain any valid input, although only changes in category codes, such as uses of \texttt{\cctab_select:N}, are relevant. See also \texttt{\tl_rescan:nn}.

\textbf{\TeXhackers note:} The \texttt{tokens} are first turned into a string (using \texttt{\tl_to_str:n}). If the string contains one or more characters with character code \texttt{\newlinechar} (set equal to \texttt{\endlinechar} unless that is equal to 32, before the user \texttt{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:nn \tl_rescan:nn \tl_rescan:nn
\tl_rescan:nn \tl_rescan:nn
Updated: 2015-08-11

Rescans \texttt{tokens} applying the category code régime specified in the \texttt{setup}, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the \texttt{setup} are those in force at the point of use of \texttt{\tl_rescan:nn}.) The \texttt{setup} is run within a group and may contain any valid input, although only changes in category codes, such as uses of \texttt{\cctab_select:N}, are relevant. See also \texttt{\tl_set_rescan:Nnn}, which is more robust than using \texttt{\tl_set:Nn} in the \texttt{tokens} argument of \texttt{\tl_rescan:nn}.

\textbf{\TeXhackers note:} The \texttt{tokens} are first turned into a string (using \texttt{\tl_to_str:n}). If the string contains one or more characters with character code \texttt{\newlinechar} (set equal to \texttt{\endlinechar} unless that is equal to 32, before the user \texttt{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 \texttt{\scantokens} \TeX primitive, \texttt{\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 \texttt{\verb} that rely on changing category codes will not function properly.

### 15.7 Constant token lists

\texttt{\c_empty_tl} Constant that is always empty.
\c_novaluetl A marker for the absence of an argument. This constant \tl can safely be typeset (cf. \q_{-nil}), with the result being \texttt{-NoValue-}. It is important to note that \c_novaluetl is constructed such that it will not match the simple text input \texttt{-NoValue-}, i.e. that

\tl_if_eq:NnTF \c_novaluetl { -NoValue- }

is logically false. The \c_novaluetl 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_spacetl 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 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.

\l_tmpb_tl

\g_tmpa_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.

\g_tmpb_tl
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.

\tl_build_begin:N  \tl_build_gbegin:N
Clears the ⟨tl var⟩ and sets it up to support other \tl_build_... functions. Until \tl_build_end: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_gput_right:Nn
\tl_build_gput_right:Nn\tl_build_end:N  \tl_build_gend:N
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
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.

128
\tl_build_get_intermediate:NN \tl_build_get_intermediate:NN \tl_build_get_intermediate:NN \tl_build_get_intermediate:NN \tl_build_get_intermediate:NN \tl_build_get_intermediate:NN

Stores the contents of the \tl_var{} in the \tl_var{}. The \tl_var{} must have been set up with \tl_build_begin:N or \tl_build_gbegin:N. The \tl_var{} is a “normal” token list variable, assigned locally using \tl_set:Nn.
Chapter 17

The \texttt{l3str} module

Strings

\LaTeX{} 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 \LaTeX{} sense.

A \LaTeX{} string (and thus an \texttt{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 \LaTeX{} string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix \texttt{...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 \texttt{\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 \texttt{\cs_to_str:N}, \texttt{\tl_to_str:n}, \texttt{\tl_to_str:N} and \texttt{\token_to_str:N} (and variants) generate strings from the appropriate input: these are documented in \texttt{l3basics}, \texttt{l3tl} and \texttt{l3token}, respectively.

Most expandable functions in this module come in three flavours:

- \texttt{\str_{...:N}}, which expect a token list or string variable as their argument;
- \texttt{\str_{...:n}}, taking any token list (or string) as an argument;
- \texttt{\str_{...-ignore_spaces:n}}, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.
17.1 Creating and initialising string variables

\str_new:N \str_new:c
\New: 2015-09-18

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:
\New: 2015-09-18
\Updated: 2018-07-28

Creates a new constant \str var or raises an error if the name is already taken. The value of the \str var is set globally to the \token list, converted to a string.

\str_clear:N \str_clear:c \str_gclear:N \str_gclear:c
\New: 2015-09-18

Clears the content of the \str var.

\str_clear_new:N \str_clear_new:c \str_gclear_new:N \str_gclear_new:c
\New: 2015-09-18

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: \str_gset_eq:NN \str_gset_eq:
\New: 2015-09-18

Sets the content of \str var1 equal to that of \str var2.

\str_concat:NNN \str_concat:ccc \str_gconcat:NNN \str_gconcat:ccc
\New: 2015-09-18
\New: 2017-10-08

Concatenates the content of \str var2 and \str var3 together and saves the result in \str var1. The \str var2 is placed at the left side of the new string variable. The \str var2 and \str var3 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:NTF \str_if_exist:cFF
\New: 2015-09-18

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

\texttt{\textbackslash{}str\_set:Nn} \langle str var \rangle \{\langle token list \rangle\}

Converts the \langle token list \rangle to a \langle string \rangle, and stores the result in \langle str var \rangle.

\texttt{\textbackslash{}str\_set:Nn} \str_set: (NV|Ne|cn|cV|ce)
\texttt{\textbackslash{}str\_gset:Nn} \str_gset: (NV|Ne|cn|cV|ce)

New: 2015-09-18
Updated: 2018-07-28

\texttt{\textbackslash{}str\_put\_left:Nn} \langle str var \rangle \{\langle token list \rangle\}
\texttt{\textbackslash{}str\_put\_left:Nn} \str_put\_left: (NV|Ne|cn|cV|ce)
\texttt{\textbackslash{}str\_gput\_left:Nn} \str_gput\_left: (NV|Ne|cn|cV|ce)

New: 2015-09-18
Updated: 2018-07-28

Converts the \langle token list \rangle to a \langle string \rangle, and prepends the result to \langle str var \rangle. The current contents of the \langle str var \rangle are not automatically converted to a string.

\texttt{\textbackslash{}str\_put\_right:Nn} \langle str var \rangle \{\langle token list \rangle\}
\texttt{\textbackslash{}str\_put\_right:Nn} \str_put\_right: (NV|Ne|cn|cV|ce)
\texttt{\textbackslash{}str\_gput\_right:Nn} \str_gput\_right: (NV|Ne|cn|cV|ce)

New: 2015-09-18
Updated: 2018-07-28

Converts the \langle token list \rangle to a \langle string \rangle, and appends the result to \langle str var \rangle. The current contents of the \langle str var \rangle are not automatically converted to a string.

17.3 String conditionals

\texttt{\textbackslash{}str\_if\_empty\_p:N} \* \str_if_empty:p:N \langle str var \rangle
\texttt{\textbackslash{}str\_if\_empty\_p:c} \* \str_if_empty:p:c \langle str var \rangle \{true code\} \{false code\}
\texttt{\textbackslash{}str\_if\_empty:NTF} \* \str_if_empty:NTF \langle str var \rangle \{true code\} \{false code\}
\texttt{\textbackslash{}str\_if\_empty:n} \* \str_if_empty:n \langle str var \rangle \{true code\} \{false code\}

Tests if the \langle string variable \rangle is entirely empty \textit{i.e.} contains no characters at all.

\texttt{\textbackslash{}str\_if\_eq:p:NN} \* \str_if_eq:p:NN \langle str var \rangle \langle str var \rangle
\texttt{\textbackslash{}str\_if\_eq:p:(Nc|cN|cc)} \* \str_if_eq:p:(Nc|cN|cc) \langle str var \rangle \langle str var \rangle \{true code\} \{false code\}
\texttt{\textbackslash{}str\_if\_eq:NN} \* \str_if_eq:NN \langle str var \rangle \langle str var \rangle \{true code\} \{false code\}
\texttt{\textbackslash{}str\_if\_eq:(Nc|cN|cc)} \* \str_if_eq:(Nc|cN|cc) \langle str var \rangle \langle str var \rangle \{true code\} \{false code\}

Compares the content of two \langle str variables \rangle and is logically true if the two contain the same characters in the same order. See \texttt{\textbackslash{}tl\_if\_eq:NNTF} to compare tokens (including their category codes) rather than characters.
Compares the two \langle token lists \rangle on a character by character basis (namely after converting them to strings), and is \textit{true} if the two \langle strings \rangle contain the same characters in the same order. Thus for example

\[
\text{\str_if_eq_p:nn \{ \{tl_1\} \} \{ \{tl_2\} \}}
\]

is logically \textit{true}. See \texttt{\tl_if_eq:nnTF} to compare tokens (including their category codes) rather than characters.

\[
\text{\str_if_eq_p:nn (Vn|on|no|VV|vn|ee) \ast \str_if_eq_p:nn \{ \{tl_1\} \} \{ \{tl_2\} \}}
\]

Converts the \langle token lists \rangle to \langle strings \rangle and tests whether \langle string \rangle 2 is found inside \langle string \rangle 1.

\[
\text{\str_if_equal:nnTF (Vn|on|en|nV|nv) \ast \str_if_equal:nnTF \{ \{tl_1\} \} \{ \{tl_2\} \}}
\]

Converts the \langle token list \rangle to a \langle string \rangle and tests if that \langle string \rangle is found in the content of the \langle str var \rangle.

\[
\text{\str_case:nnTF \{ \{test string\} \} \{ \{true code\} \} \{ \{false code\} \}}
\]

Compares each \langle test string \rangle in turn with each of the \langle string case \rangle\rangles (all token lists are converted to strings). If the two are equal (as described for \texttt{\str_if_eq:nnTF}) then the associated \langle code \rangle is left in the input stream and other cases are discarded. If any of the cases are matched, the \langle true code \rangle is also inserted into the input stream (after the code for the appropriate case), while if none match then the \langle false code \rangle is inserted.

\texttt{\str_case:nn} is also available. This set of functions performs no expansion on each \langle string case \rangle argument, so any variable in there will be compared as a string. If expansion is needed in the \langle string case \rangle\rangles, then \texttt{\str_case_e:nnTF} should be used instead.
\str_case_e:nn \ { \langle \text{test string} \rangle } \ * \\
\str_case_e:en \ * \\
\str_case_e:nnTF \ * \\
\str_case_e:enTF \ * \\
\ Rev: 2018-06-19

Compares the full expansion of the \langle \text{test string} \rangle in turn with the full expansion of the \langle \text{string case} \rangle 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 \langle \text{code} \rangle is left in the input stream and other cases are discarded. If any of the cases are matched, the \langle \text{true code} \rangle is also inserted into the input stream (after the code for the appropriate case), while if none match then the \langle \text{false code} \rangle is inserted. In \str_case_e:nn(TF), the \langle \text{test string} \rangle 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 \ { \langle \text{tl} 1 \rangle } \ { \langle \text{relation} \rangle } \ { \langle \text{tl} 2 \rangle } \ * \\
\str_compare_p:eNe \ * \\
\str_compare:nNnTF \ { \langle \text{tl} 1 \rangle } \ { \langle \text{relation} \rangle } \ { \langle \text{tl} 2 \rangle } \ { \langle \text{true code} \rangle } \ { \langle \text{false code} \rangle } \\
\ Rev: 2021-05-17

Compares the two \langle \text{token lists} \rangle 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 \langle \text{relation} \rangle can be \langle < \rangle, \langle = \rangle, or \langle > \rangle and the test is \text{true} under the following conditions:

- for \langle < \rangle, if the first string is earlier than the second in lexicographic order;
- for \langle = \rangle, if the two strings have exactly the same characters;
- for \langle > \rangle, if the first string is later than the second in lexicographic order.

Thus for example the following is logically \text{true}:

\str_compare_p:nNn \ { \langle ab \rangle } \ < \ { \langle abc \rangle } 

\TeXhackers note: This is a wrapper around the \TeX{} primitive \langle (pdf)strcmp \rangle. 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.

\subsection{17.4 Mapping over strings}

All mappings are done at the current group level, \textit{i.e.} any local assignments made by the \langle \text{function} \rangle or \langle \text{code} \rangle discussed below remain in effect after the loop.

\str_map_function:nN \ * \\
\str_map_function:NN \ * \\
\str_map_function:cN \ * \\
\ Rev: 2017-11-14

Converts the \langle \text{token list} \rangle to a \langle \text{string} \rangle then applies (\text{function}) to every \langle \text{character} \rangle in the \langle \text{string} \rangle including spaces.
\texttt{\textbackslash str\_map\_inline:n} \hspace{.5em} \texttt{\textbackslash str\_map\_inline:Nn} \hspace{.5em} \texttt{\textbackslash str\_map\_inline:cn} \hspace{.5em}

Converts the \texttt{\langle token list\rangle} to a \texttt{\langle string\rangle} then applies the \texttt{\langle inline function\rangle} to every \texttt{\langle character\rangle} in the \texttt{\langle str var\rangle} including spaces. The \texttt{\langle inline function\rangle} should consist of code which receives the \texttt{\langle character\rangle} as \#1.

\texttt{\textbackslash str\_map\_tokens:nn} \hspace{.5em} \texttt{\textbackslash str\_map\_tokens:Nn} \hspace{.5em} \texttt{\textbackslash str\_map\_tokens:cn} \hspace{.5em} \hspace{.5em} \hspace{.5em}

Converts the \texttt{\langle token list\rangle} to a \texttt{\langle string\rangle} then applies \texttt{\langle code\rangle} to every \texttt{\langle character\rangle} in the \texttt{\langle string\rangle} including spaces. The \texttt{\langle code\rangle} receives each character as a trailing brace group. This is equivalent to \texttt{\textbackslash str\_map\_function:n} if the \texttt{\langle code\rangle} consists of a single function.

\texttt{\textbackslash str\_map\_variable:nN} \hspace{.5em} \texttt{\textbackslash str\_map\_variable:NNn} \hspace{.5em} \texttt{\textbackslash str\_map\_variable:cn} \hspace{.5em} \hspace{.5em} \hspace{.5em}

Converts the \texttt{\langle token list\rangle} to a \texttt{\langle string\rangle} then stores each \texttt{\langle character\rangle} in the \texttt{\langle string\rangle} (including spaces) in turn in the \texttt{\langle string\rangle or token list\rangle} \texttt{\langle variable\rangle} and applies the \texttt{\langle code\rangle}. The \texttt{\langle code\rangle} will usually make use of the \texttt{\langle variable\rangle}, but this is not enforced. The assignments to the \texttt{\langle variable\rangle} are local. Its value after the loop is the last \texttt{\langle character\rangle} in the \texttt{\langle string\rangle}, or its original value if the \texttt{\langle string\rangle} is empty. See also \texttt{\textbackslash str\_map\_inline:n}.

\texttt{\textbackslash str\_map\_break:} \hspace{.5em} \hspace{.5em} \hspace{.5em} \hspace{.5em} \hspace{.5em}

Used to terminate a \texttt{\textbackslash str\_map\_\ldots} function before all characters in the \texttt{\langle string\rangle} have been processed. This normally takes place within a conditional statement, for example

\begin{verbatim}
\texttt{\textbackslash str\_map\_inline:Nn \l_my_str}
{\textbackslash str\_if\_eq:nnT \{ #1 \} \{ bingo \} \{ \textbackslash str\_map\_break: \} % Do something useful}
\end{verbatim}

See also \texttt{\textbackslash str\_map\_break:n}. Use outside of a \texttt{\textbackslash str\_map\_\ldots} scenario leads to low level \LaTeX{} errors.

\textbf{\LaTeX{}hackers 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 \langle \texttt{code} \rangle

Used to terminate a \str_map\ldots function before all characters in the \langle string \rangle have been processed, inserting the \langle code \rangle 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 \{ <\texttt{code}> \} \}
% Do something useful}

Use outside of a \str_map\ldots scenario leads to low level \TeX{} errors.

\textbf{\TeX{}hackers note:} When the mapping is broken, additional tokens may be inserted before the \langle code \rangle 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:N \langle \texttt{str var} \rangle

Recovers the content of a \langle \texttt{str var} \rangle 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 \langle \texttt{str} \rangle directly without an accessor function.

\str_count:n \str_count:n \langle \texttt{token list} \rangle
\str_count:N \str_count:n \str_count_ignore_spaces:n

Leaves in the input stream the number of characters in the string representation of \langle token list \rangle, 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:n \langle \texttt{token list} \rangle

Leaves in the input stream the number of space characters in the string representation of \langle token list \rangle, as an integer denotation. Of course, this function has no \_ignore_spaces variant.
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.

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.

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, \texttt{−1} is the right-most character, \textit{etc.}. 

\str_head:N /* \str_head:n {⟨token list⟩} 
\str_head:c *
\str_head:n *
\str_head_ignore_spaces:n *

Rev: 2015-09-18

\str_tail:N /* \str_tail:n {⟨token list⟩} 
\str_tail:c *
\str_tail:n *
\str_tail_ignore_spaces:n *

Rev: 2015-09-18

\str_item:Nn /* \str_item:nn {⟨token list⟩} {⟨integer expression⟩} 
\str_item:nn *
\str_item_ignore_spaces:nn *

Rev: 2015-09-18
\str_range:nnn \* \str_range:nnn \{\token list\} \{\start index\} \{\end index\}
\str_range:nnn \*
\str_range:nnn \*
\str_range_ignore_spaces:nnn \*

\begin{verbatim}
New: 2015-09-18

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,

\begin{verbatim}
\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 \} \}
\end{verbatim}

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

\begin{verbatim}
\iow_term:e \{ \str_range:nnn { abcdef } \{ 5 \} \{ 2 \} \}
\iow_term:e \{ \str_range:nnn { abcdef } \{ -1 \} \{ -4 \} \}
\end{verbatim}

both yield empty strings.

The behavior of \str_range_ignore_spaces:nnn is similar, but spaces are removed before starting the job. The input

\begin{verbatim}
\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 \} \}
\end{verbatim}

\end{verbatim}

138
will print four instances of bcde, four instances of bc e and eight instances of bcde.

17.6 Modifying string variables

\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 } }

\str_replace_once:Nnn \str_replace_once:cn \str_greplace_once:Nnn \str_greplace_once:cn

\str_replace_all:Nnn \str_replace_all:cn \str_greplace_all:Nnn \str_greplace_all:cn

\str_remove_once:Nn \str_remove_once:cn \str_gremove_once:Nn \str_gremove_once:cn

\str_remove_all:Nn \str_remove_all:cn \str_gremove_all:Nn \str_gremove_all:cn

\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

Converts the input \langle tokens \rangle 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
\begin{verbatim}
\cs_set_protected:cpn
\langle user \rangle
\str_uppercase:f { \tl_head:n {#1} }
\str_lowercase:f { \tl_tail:n {#1} }
\end{verbatim}

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|once):n(n) functions which correctly deal with context-dependence and other factors appropriate to text case changing.
Converts the input \langle tokens \rangle to their string representation, as described for \texttt{tl_to_str:n}, and then folds the case of the resulting \langle string \rangle 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 \texttt{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 \texttt{str_casefold:n} follows the “full” scheme defined by the Unicode Consortium (\textit{e.g.} SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (\textit{i.e.} I always folds to i and not to ı).

\texttt{str_mdfive_hash:n} \xrightarrow{\langle tl \rangle} \texttt{str_mdfive_hash:n} \langle tl \rangle

Expands to the MD5 sum generated from the \langle tl \rangle, which is converted to a \langle string \rangle as described for \texttt{tl_to_str:n}.

### 17.8 Viewing strings

\texttt{str_show:N} \xrightarrow{\langle str \ var \rangle} \texttt{str_show:N} \langle str \ var \rangle

Displays the content of the \langle str \ var \rangle on the terminal.

\texttt{str_log:N} \xrightarrow{\langle str \ var \rangle} \texttt{str_log:N} \langle str \ var \rangle

Writes the content of the \langle str \ var \rangle in the log file.
17.9 Constant strings

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

\c_ampersand_str
\c_atsign_str
\c_backslash_str
\c_left_brace_str
\c_right_brace_str
\c_circumflex_str
\c_colon_str
\c_dollar_str
\c_hash_str
\c_percent_str
\c_tilde_str
\c_underscore_str
\c_zero_str

New: 2015-09-19
Updated: 2020-12-22

\c_empty_str
Constant that is always empty.

New: 2023-12-07

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 \texttt{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 (\textit{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 \texttt{utf-16}, \texttt{iso 8859-1}, \textit{etc}. See Table 1 for a list of supported encodings.\footnote{Encodings and escapings will be added as they are requested.}

- Bytes are translated to \LaTeX{} tokens through a given “escaping”. Those are defined for the most part by the \texttt{pdf} file format. See Table 2 for a list of escaping methods supported.\footnote{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.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>utf8</td>
<td>UTF-8</td>
</tr>
<tr>
<td>utf16</td>
<td>UTF-16, with byte-order mark</td>
</tr>
<tr>
<td>utf16be</td>
<td>UTF-16, big-endian</td>
</tr>
<tr>
<td>utf16le</td>
<td>UTF-16, little-endian</td>
</tr>
<tr>
<td>utf32</td>
<td>UTF-32, with byte-order mark</td>
</tr>
<tr>
<td>utf32be</td>
<td>UTF-32, big-endian</td>
</tr>
<tr>
<td>utf32le</td>
<td>UTF-32, little-endian</td>
</tr>
<tr>
<td>iso88591, latin1</td>
<td>ISO 8859-1</td>
</tr>
<tr>
<td>iso88592, latin2</td>
<td>ISO 8859-2</td>
</tr>
<tr>
<td>iso88593, latin3</td>
<td>ISO 8859-3</td>
</tr>
<tr>
<td>iso88594, latin4</td>
<td>ISO 8859-4</td>
</tr>
<tr>
<td>iso88595</td>
<td>ISO 8859-5</td>
</tr>
<tr>
<td>iso88596</td>
<td>ISO 8859-6</td>
</tr>
<tr>
<td>iso88597</td>
<td>ISO 8859-7</td>
</tr>
<tr>
<td>iso88598</td>
<td>ISO 8859-8</td>
</tr>
<tr>
<td>iso88599, latin5</td>
<td>ISO 8859-9</td>
</tr>
<tr>
<td>iso885910, latin6</td>
<td>ISO 8859-10</td>
</tr>
<tr>
<td>iso885911</td>
<td>ISO 8859-11</td>
</tr>
<tr>
<td>iso885913, latin7</td>
<td>ISO 8859-13</td>
</tr>
<tr>
<td>iso885914, latin8</td>
<td>ISO 8859-14</td>
</tr>
<tr>
<td>iso885915, latin9</td>
<td>ISO 8859-15</td>
</tr>
<tr>
<td>iso885916, latin10</td>
<td>ISO 8859-16</td>
</tr>
<tr>
<td>clist</td>
<td>comma-list of integers</td>
</tr>
<tr>
<td>(empty)</td>
<td>native (Unicode) string</td>
</tr>
<tr>
<td>default</td>
<td>like utf8 with 8-bit engines, and like native with unicode-engines</td>
</tr>
</tbody>
</table>

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

<table>
<thead>
<tr>
<th>Escaping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes, or empty</td>
<td>arbitrary bytes</td>
</tr>
<tr>
<td>hex, hexadecimal name</td>
<td>byte = two hexadecimal digits</td>
</tr>
<tr>
<td></td>
<td>see \pdfescauname</td>
</tr>
<tr>
<td>string</td>
<td>see \pdfescapestring</td>
</tr>
<tr>
<td>url</td>
<td>encoding used in URLs</td>
</tr>
</tbody>
</table>
18.2 Conversion functions

\texttt{\textbackslash str\_set\_convert:Nnnn} \texttt{\textbackslash str\_gset\_convert:Nnnn}

This function converts the \langle string \rangle from the encoding given by \langle name 1 \rangle to the encoding given by \langle name 2 \rangle, and stores the result in the \langle str var \rangle. Each \langle name \rangle can have the form \langle encoding \rangle or \langle encoding \rangle/\langle escaping \rangle, where the possible values of \langle encoding \rangle and \langle escaping \rangle are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty \langle name \rangle indicates the use of “native” strings, 8-bit for pdfTEX, and Unicode strings for the other two engines.

For example,

\texttt{\textbackslash str\_set\_convert:Nnnn \l\_foo\_str \{ Hello! \} \{ \} \{ utf16\textbackslash 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 \langle string \rangle is not valid according to the \langle escaping 1 \rangle and \langle encoding 1 \rangle, or if it cannot be reencoded in the \langle encoding 2 \rangle and \langle escaping 2 \rangle (for instance, if a character does not exist in the \langle encoding 2 \rangle). Erroneous input is replaced by the Unicode replacement character “\textbackslash FFFD, and characters which cannot be reencoded are replaced by either the replacement character "\textbackslash FFFD if it exists in the \langle encoding 2 \rangle, or an encoding-specific replacement character, or the question mark character.

\texttt{\textbackslash str\_set\_convert:Nnnn\_TF} \texttt{\textbackslash str\_gset\_convert:Nnnn\_TF}

\texttt{\textbackslash str\_set\_convert:Nnnn\_TF \{ str var \} \{ \langle string \rangle \} \{ \langle name 1 \rangle \} \{ \langle name 2 \rangle \} \{ \langle true code \rangle \} \{ \langle false code \rangle \}}

As \texttt{\str_set_convert:Nnnn}, converts the \langle string \rangle from the encoding given by \langle name 1 \rangle to the encoding given by \langle name 2 \rangle, and assigns the result to \langle str var \rangle. Contrarily to \texttt{\str_set_convert:Nnnn}, the conditional variant does not raise errors in case the \langle string \rangle is not valid according to the \langle name 1 \rangle encoding, or cannot be expressed in the \langle name 2 \rangle encoding. Instead, the \langle false code \rangle 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.

\texttt{\str\_convert\_pdfname:n}

\texttt{\textbackslash str\_convert\_pdfname:n \{ \langle string \rangle \}}

As \texttt{\str_set_convert:Nnnn}, converts the \langle string \rangle 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 \ldots 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 \texttt{\textbackslash scantokens}.

• Change \texttt{\textbackslash str\_set\_convert:Nnnn} to expand its last two arguments.

• Describe the internal format in the code comments. Refuse code points in \texttt{[\textbackslash D800,\textbackslash DFFF]} in the internal representation?

• Add documentation about each encoding and escaping method, and add examples.

• The \texttt{hex} unescaping should raise an error for odd-token count strings.

• Decide what bytes should be escaped in the \texttt{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 \texttt{stringenc}). CESU?

• More escapings: \texttt{asch85}, shell escapes, lua escapes, \textit{etc}.

Chapter 19

The \texttt{\LaTeX}3 \texttt{quark} 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 \texttt{\q{...}}, and scan marks start with \texttt{\s{...}}.

19.1 Quarks

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore \textit{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. \texttt{\q{stop}}). For example, when writing a macro to parse a user-defined date

\begin{verbatim}
\date_parse:n {19/June/1981}
\end{verbatim}

one might write a command such as

\begin{verbatim}
\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> }
\end{verbatim}

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function \texttt{\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 \texttt{\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 \texttt{\tl_if_eq:NNTF}. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster.
19.2 Defining quarks

\quark_new:N  ⟨quark⟩

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

\q_stop

Used as a marker for delimited arguments, such as

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

\q_mark

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

\q_nil

Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to \q_stop, which is only ever used as a delimiter).

\q_no_value

A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as \prop_get:NnN if there is no data to return.

19.3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

\quark_if_nil_p:N  * \quark_if_nil_p:n {token}
\quark_if_nil:NTF  * \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:n {token list}
\quark_if_nil:NTF  * \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:n {token}
\quark_if_no_value:NTF  * \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_p:n {token list}
\quark_if_no_value:NTF  * \quark_if_no_value:nTF {token list} {(true code)} {(false code)}

Tests if the ⟨token list⟩ contains only \q_no_value (distinct from ⟨token list⟩ being empty or containing \q_no_value plus one or more other tokens).
19.4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 19.4.1.

\q_recursion_tail
This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

\q_recursion_stop
This quark is added after the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

\quark_if_recursion_tail_stop:N \token
Tests if the \{token\} contains only the marker \q_recursion_tail, and if so uses \use_\texttt{-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 \{token list\} \quark_if_recursion_tail_stop:o
Tests if the \{token list\} contains only \q_recursion_tail, and if so uses \use_\texttt{-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 \token \{insertion\}
Tests if the \{token\} contains only \q_recursion_tail, and if so uses \use_\texttt{-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.
Tests if \textit{token list} contains only \texttt{\textbackslash q\_recursion\_tail}, and if so terminates the recursion using \texttt{\textbackslash (type)\_map\_break}. The recursion end should be marked by \texttt{\textbackslash prg\_break\_point:Nn \textbackslash (type)\_map\_break}.

19.4.1 An example of recursion with quarks

Quarks are mainly used internally in the expl3 code to define recursion functions such as \texttt{\textbackslash 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 \texttt{\textbackslash my\_map\_dbl:nn} which takes a token list and applies an operation to every pair of tokens. For example, \texttt{\textbackslash my\_map\_dbl:nn \{abcd\} \{-[#1--#2--]\-\}} would produce \texttt{\{-a-b-\} \{-c-d-\}}. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here's the definition of \texttt{\textbackslash my\_map\_dbl:nn}. First of all, define the function that does the processing based on the inline function argument \texttt{#2}. Then initiate the recursion using an internal function. The token list \texttt{#1} is terminated using \texttt{\textbackslash q\_recursion\_tail}, with delimiters according to the type of recursion (here a pair of \texttt{\textbackslash q\_recursion\_tail}), concluding with \texttt{\textbackslash q\_recursion\_stop}. These quarks are used to mark the end of the token list being operated upon.

\begin{verbatim}
\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}
\end{verbatim}

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.

\begin{verbatim}
\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\}}
\end{verbatim}

Finally, recurse:

\begin{verbatim}
\__my_map_dbl:nn
\end{verbatim}

Note that contrarily to \LaTeX{}3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of \texttt{\__my\_map\_dbl\_fn:nn}.
19.5 Scan marks

Scan marks are control sequences set equal to \scan_stop:, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by \TeX\ in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see \l3regex).

\scan_new:N \scan_stop \langle scan mark \rangle

Creates a new \langle scan mark \rangle which is set equal to \scan_stop:. The \langle scan mark \rangle 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 \langle tokens \rangle \s_stop

Removes the \langle tokens \rangle and \s_stop from the input stream. This leads to a low-level \TeX\ error if \s_stop is absent.
Chapter 20

The \texttt{l3seq} module

Sequences and stacks

\LaTeX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any balanced text. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in \LaTeX3. This is achieved using a number of dedicated stack functions.

20.1 Creating and initialising sequences

\begin{verbatim}
\seq_new:N \seq_new:N \seq_new:c
\seq_new:N \seq_new:c
\seq_clear:N \seq_clear:N \seq_clear:N \seq_clear:N \seq_clear:N \seq_clear:N
\seq_gclear:N \seq_gclear:N \seq_gclear:N \seq_gclear:N \seq_gclear:N \seq_gclear:N
\seq_set_eq:NN \seq_set_eq:NN \seq_gset_eq:NN \seq_gset_eq:NN \seq_gset_eq:NN \seq_gset_eq:NN
\end{verbatim}

\seq_new:N \seq_new:N \seq_new:c \seq_new:N \seq_new:c \seq_new:N \seq_new:c

Creates a new \langle seq var \rangle or raises an error if the name is already taken. The declaration is global. The \langle seq var \rangle initially contains no items.

\seq_clear:N \seq_clear:N \seq_clear:N \seq_clear:N \seq_clear:N \seq_clear:N \seq_clear:N
Clears all items from the \langle seq var \rangle.

\seq_gclear:N \seq_gclear:N \seq_gclear:N \seq_gclear:N \seq_gclear:N \seq_gclear:N
Ensures that the \langle seq var \rangle exists globally by applying \seq_new:N if necessary, then applies \seq_{(g)clear}:N to leave the \langle seq var \rangle empty.

\seq_set_eq:NN \seq_set_eq:NN \seq_set_eq:NN \seq_set_eq:NN \seq_gset_eq:NN \seq_gset_eq:NN
Sets the content of \langle seq var_1 \rangle equal to that of \langle seq var_2 \rangle.
Converts the data in the \langle comma list \rangle into a \langle seq var \rangle: the original \langle comma list \rangle is unchanged.

Creates a new constant \langle seq var \rangle or raises an error if the name is already taken. The \langle seq var \rangle is set globally to contain the items in the \langle comma list \rangle.

Splits the \langle token list \rangle into \langle items \rangle separated by \langle delimiter \rangle, and assigns the result to the \langle seq var \rangle. Spaces on both sides of each \langle item \rangle are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of l3clist functions. Empty \langle items \rangle are preserved by \seq_set_split:Nnn, and can be removed afterwards using \seq_remove_all:Nn \langle seq var \rangle \{}. The \langle delimiter \rangle may not contain {, } or # (assuming \TeX’s normal category code régime). If the \langle delimiter \rangle is empty, the \langle token list \rangle is split into \langle items \rangle as a \langle token list \rangle. See also \seq_set_split_-keep_spaces:Nnn, which omits space stripping.

Splits the \langle token list \rangle into \langle items \rangle separated by \langle delimiter \rangle, and assigns the result to the \langle seq var \rangle. 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 \langle items \rangle are preserved by \seq_set_split_keep_spaces:Nnn, and can be removed afterwards using \seq_remove_all:Nn \langle seq var \rangle \{}. The \langle delimiter \rangle may not contain {, } or # (assuming \TeX’s normal category code régime). If the \langle delimiter \rangle is empty, the \langle token list \rangle is split into \langle items \rangle as a \langle token list \rangle. See also \seq_set_split:Nnn, which removes spaces around the delimiters.
\seq_concat:NN\n\seq_concat:ccc
\seq_gconcat:NN\n\seq_gconcat:ccc
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_concat:NN\n\seq_concat:ccc
\seq_gconcat:NN\n\seq_gconcat:ccc
Tests whether the \seq var is currently defined. This does not check that the \seq var really is a sequence variable.

20.2 Appending data to sequences

\seq_put_left:NN\n\seq_put_left:NV|Nv|Ne|No|cn|cV|cv|ce|co\n\seq_gput_left:NN\n\seq_gput_left:NV|Nv|Ne|No|cn|cV|cv|ce|co
Appends the \item to the left of the \seq var.

\seq_put_right:NN\n\seq_put_right:NV|Nv|Ne|No|cn|cV|cv|ce|co\n\seq_gput_right:NN\n\seq_gput_right:NV|Nv|Ne|No|cn|cV|cv|ce|co
Appends the \item to the right of the \seq var.

20.3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, i.e., setting the \token list variable used with \tl_set:NN and never \tl_gset:NN.

\seq_get_left:NN\n\seq_get_left:cn\nUpdated: 2012-05-14
Stores the left-most item from a \seq var in the \token list variable without removing it from the \seq var. The \token list variable is assigned locally. If \seq var is empty the \token list variable is set to the special marker \q_no_value.

\seq_get_right:NN\n\seq_get_right:cn\nUpdated: 2012-05-19
Stores the right-most item from a \seq var in the \token list variable without removing it from the \seq var. The \token list variable is assigned locally. If \seq var is empty the \token list variable is set to the special marker \q_no_value.
\seq_pop_left:N\seq_pop_left:c

Pops the left-most item from a \seq var into the \token list variable, i.e. removes the item from the sequence and stores it in the \token list variable. Both of the variables are assigned locally. If \seq var is empty the \token list variable is set to the special marker \q_no_value.

\seq_gpop_left:N\seq_gpop_left:c

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

\seq_pop_right:N\seq_pop_right:c

Pops the right-most item from a \seq var into the \token list variable, i.e. removes the item from the sequence and stores it in the \token list variable. Both of the variables are assigned locally. If \seq var is empty the \token list variable is set to the special marker \q_no_value.

\seq_gpop_right:N\seq_gpop_right:c

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

\seq_item:Nn
\seq_item:N\seq_item:NV\seq_item:Ne\seq_item:cn\seq_item:cV\seq_item:ce

* \seq_item:Nn \seq var \langle integer expression \rangle

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.

\TeXhacksnote: 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. This is not available in older versions of \TeX.

\TeXhacksnote: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the \item does not expand further when appearing in an e-type or x-type argument expansion.
20.4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

\seq_get_left:NNTF
\seq_get_left:NNTF \seq var \{ \text{token list variable} \} \{ \text{true code} \} \{ \text{false code} \}

If the \seq var is empty, leaves the \text{false code} in the input stream. The value of the \text{token list variable} 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 \text{token list variable} without removing it from the \seq var, then leaves the \text{true code} in the input stream. The \text{token list variable} is assigned locally.

\seq_get_right:NNTF
\seq_get_right:NNTF \seq var \{ \text{token list variable} \} \{ \text{true code} \} \{ \text{false code} \}

If the \seq var is empty, leaves the \text{false code} in the input stream. The value of the \text{token list variable} 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 \text{token list variable} without removing it from the \seq var, then leaves the \text{true code} in the input stream. The \text{token list variable} is assigned locally.

\seq_pop_left:NNTF
\seq_pop_left:NNTF \seq var \{ \text{token list variable} \} \{ \text{true code} \} \{ \text{false code} \}

If the \seq var is empty, leaves the \text{false code} in the input stream. The value of the \text{token list variable} 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 \text{token list variable}, i.e. removes the item from the \seq var, then leaves the \text{true code} in the input stream. Both the \seq var and the \text{token list variable} are assigned locally.

\seq_gpop_left:NNTF
\seq_gpop_left:NNTF \seq var \{ \text{token list variable} \} \{ \text{true code} \} \{ \text{false code} \}

If the \seq var is empty, leaves the \text{false code} in the input stream. The value of the \text{token list variable} 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 \text{token list variable}, i.e. removes the item from the \seq var, then leaves the \text{true code} in the input stream. The \seq var is modified globally, while the \text{token list variable} is assigned locally.

\seq_pop_right:NNTF
\seq_pop_right:NNTF \seq var \{ \text{token list variable} \} \{ \text{true code} \} \{ \text{false code} \}

If the \seq var is empty, leaves the \text{false code} in the input stream. The value of the \text{token list variable} 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 \text{token list variable}, i.e. removes the item from the \seq var, then leaves the \text{true code} in the input stream. Both the \seq var and the \text{token list variable} are assigned locally.
If the \seq var is empty, leaves the \false code in the input stream. The value of the \token list variable is not defined in this case and should not be relied upon.

If the \seq var is non-empty, pops the right-most item from the \seq var in the \token list variable, 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 \token list variable is assigned locally.

20.5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

\seq_remove_duplicates:N \seq_remove_duplicates:c \seq_gremove_duplicates:N \seq_gremove_duplicates:c

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.

\seq_remove_all:Nn \seq_remove_all:NV \seq_remove_all:Ne \seq_remove_all:cn \seq_remove_all:cV \seq_remove_all:ce

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_gset_item:Nnn \seq_gset_item:cnn

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

Reverses the order of the items stored in the \seq var.
\seq_sort:Nn \seq_sort:cn \seq_gsort:Nn \seq_gsort:cn

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

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.

\seq_if_empty_p:N \seq_if_empty:NTF \seq_if_empty:N \seq_if_empty:c

Tests if the \seq var is empty (containing no items).

\seq_if_in:NnTF \seq_if_in:N (%V | %v | %e | %o | %n | %c | %C | %v | %v | %e | %o)

Tests if the \item is present in the \seq var.

\seq_map_function:NN \seq_map_function:cn \seq_map_inline:Nn \seq_map_inline:cn

 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.

\seq_map_inline:Nn \seq_map_inline:cn

 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.

20.6 Sequence conditionals

20.7 Mapping over sequences

All mappings are done at the current group level, i.e. any local assignments made by the \function or \code discussed below remain in effect after the loop.
Analogue of \seq_map_function:NN which maps several tokens instead of a single function. The \textit{code} receives each item in the \textit{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 \textit{seq var}: for each item in \l_my_seq the function \texttt{\prg_replicate:nn} receives 2 and \textit{item} as its two arguments. The function \texttt{\seq_map_inline:Nn} is typically faster but it is not expandable.

\seq_map_variable:NNn \seq_map_variable:NNn \seq_map_variable:\{Ncn\|cNn|ccNn\}

Stores each \textit{item} of the \textit{seq var} in turn in the (token list) \textit{variable} and applies the \textit{code}. The \textit{code} will usually make use of the \textit{variable}, but this is not enforced. The assignments to the \textit{variable} are local. Its value after the loop is the last \textit{item} in the \textit{seq var}, or its original value if the \textit{seq var} is empty. The \textit{items} are returned from left to right.

\seq_map_indexed_function:NN \seq_map_indexed_function:NN \seq_map_indexed_function:\{\{ inline function \}\}

Applies \textit{function} to every entry in the \textit{sequence variable}. The \textit{function} should have signature :nn. It receives two arguments for each iteration: the \textit{index} (namely 1 for the first entry, then 2 and so on) and the \textit{item}.

\seq_map_indexed_inline:Nn \seq_map_indexed_inline:Nn \seq_map_indexed_inline:NNn \{ inline function \}

Applies \textit{inline function} to every entry in the \textit{sequence variable}. The \textit{inline function} should consist of code which receives the \textit{index} (namely 1 for the first entry, then 2 and so on) as \texttt{#1} and the \textit{item} as \texttt{#2}.

\seq_map_pairwise_function:NNN \seq_map_pairwise_function:NNN \seq_map_pairwise_function:\{seq1\} \seq2 \{ function \}

Applies \textit{function} to every pair of items \texttt{seq1-item}–\texttt{seq2-item} from the two sequences, returning items from both sequences from left to right. The \textit{function} receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (\textit{i.e.} whichever sequence has fewer items determines how many iterations occur).
\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.

\TeX{}hackers 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
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.

\TeX{}hackers 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
Applies ⟨inline function⟩ to every ⟨item⟩ stored within the ⟨seq var2⟩. The ⟨inline function⟩ should consist of code which will receive the ⟨item⟩ as #1. The sequence resulting applying ⟨inline function⟩ to each ⟨item⟩ is assigned to ⟨seq var1⟩.

\TeX{}hackers 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

Applies \texttt{inline function} to every \texttt{item} stored within the \texttt{seq var2}. The \texttt{inline function} should consist of code which will receive the \texttt{item} as \texttt{#1}. The sequence resulting from e-expanding \texttt{inline function} applied to each \texttt{item} is assigned to \texttt{seq var1}. As such, the code in \texttt{inline function} should be expandable.

\textbf{\TeXhacksnote}: Contrarily to other mapping functions, \texttt{\seq_map_break:} cannot be used in this function, and would lead to low-level \TeX{} errors.

\seq_count:N
\seq_count:c

Leaves the number of items in the \texttt{seq var} in the input stream as an \texttt{integer denotation}. The total number of items in a \texttt{seq var} includes those which are empty and duplicates, i.e. every item in a \texttt{seq var} is unique.

\section{Using the content of sequences directly}

\seq_use:Nnnn \seq_use:cnnn

Places the contents of the \texttt{seq var} in the input stream, with the appropriate \texttt{separator} between the items. Namely, if the sequence has more than two items, the \texttt{separator between more than two} is placed between each pair of items except the last, for which the \texttt{separator between final two} is used. If the sequence has exactly two items, then they are placed in the input stream separated by the \texttt{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,

\begin{verbatim}
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
\end{verbatim}

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.

\textbf{\TeXhacksnote}: The result is returned within the \texttt{\unexpanded} primitive \texttt{\exp_not:n}, which means that the \texttt{items} do not expand further when appearing in an e-type or x-type argument expansion.
\seq_use:Nn \seq_use:cn

Places the contents of the \seq var in the input stream, with the \{separator\} between the items. If the sequence has a single item, it is placed in the input stream with no \{separator\}, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | \{de\} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
inserts “a and b and c and de and f” in the input stream.

**TeXhackers note:** The result is returned within the \unexpanded primitive (\exp_not:n), which means that the \{items\} do not expand further when appearing in an e-type or x-type argument expansion.

## 20.9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

\seq_get:NN \seq_get:cn

Reads the top item from a \seq var into the \{token list variable\} without removing it from the \seq var. The \{token list variable\} is assigned locally. If \seq var is empty the \{token list variable\} is set to the special marker \q_no_value.

\seq_pop:NN \seq_pop:cn

Pops the top item from a \seq var into the \{token list variable\}. Both of the variables are assigned locally. If \seq var is empty the \{token list variable\} is set to the special marker \q_no_value.

\seq_gpop:NN \seq_gpop:cn

Pops the top item from a \seq var into the \{token list variable\}. The \seq var is modified globally, while the \{token list variable\} is assigned locally. If \seq var is empty the \{token list variable\} is set to the special marker \q_no_value.

\seq_get:NNTf \seq_get:cn

If the \seq var is empty, leaves the \{false code\} in the input stream. The value of the \{token list variable\} is not defined in this case and should not be relied upon. If the \seq var is non-empty, stores the top item from a \seq var in the \{token list variable\} without removing it from the \seq var. The \{token list variable\} is assigned locally.
If the \texttt{seq var} is empty, leaves the \texttt{(false code)} in the input stream. The value of the \texttt{token list variable} is not defined in this case and should not be relied upon.

If the \texttt{seq var} is non-empty, pops the top item from the \texttt{seq var} in the \texttt{token list variable}, i.e. removes the item from the \texttt{seq var}. Both the \texttt{seq var} and the \texttt{token list variable} are assigned locally.

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

\texttt{seq push:Nn (seq var) \{item\}}
\texttt{seq push:(NV|Nv|Ne|No|cn|cV|cv|ce|co)}
\texttt{seq gpush:Nn}
\texttt{seq gpush:(NV|Nv|Ne|No|cn|cV|cv|ce|co)}

Adds the \{item\} to the top of the \texttt{seq var}.

### 20.10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a \texttt{sequence variable} only has distinct items, use \texttt{seq remove duplicates:N} \texttt{sequence variable}. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set \texttt{seq var} are straightforward. For instance, \texttt{seq count:N} \texttt{seq var} expands to the number of items, while \texttt{seq if in:NnTF} \texttt{seq var} \{\texttt{item}\} tests if the \texttt{item} is in the set.

Adding an \texttt{item} to a set \texttt{seq var} can be done by appending it to the \texttt{seq var} if it is not already in the \texttt{seq var}:

\texttt{seq if in:NnF seq var \{item\}}
\{ \texttt{seq put right:Nn seq var \{item\}} \}

Removing an \texttt{item} from a set \texttt{seq var} can be done using \texttt{seq remove all:Nn},

\texttt{seq remove all:Nn seq var \{item\}}

The intersection of two sets \texttt{seq var}_1 and \texttt{seq var}_2 can be stored into \texttt{seq var}_3 by collecting items of \texttt{seq var}_1 which are in \texttt{seq var}_2.
\seq_clear:N \seq_var_3
\seq_map_inline:Nn \seq_var_1
{  \seq_if_in:NnT \seq_var_2 \{#1\}
   { \seq_put_right:Nn \seq_var_3 \{#1\} }
}

The code as written here only works if \seq_var_3 is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence \l__⟨pkg⟩_internal_seq, then \seq_var_3 should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets \seq_var_1 and \seq_var_2 can be stored into \seq_var_3 through
\seq_concat:NNN \seq_var_3 \seq_var_1 \seq_var_2
\seq_remove_duplicates:N \seq_var_3
or by adding items to (a copy of) \seq_var_3 one by one
\seq_set_eq:NN \seq_var_3 \seq_var_1
\seq_map_inline:Nn \seq_var_2
{  \seq_if_in:NnF \seq_var_3 \{#1\}
   { \seq_put_right:Nn \seq_var_3 \{#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_var_3 \seq_var_1
\seq_map_inline:Nn \seq_var_2
{  \seq_remove_all:Nn \seq_var_3 \{#1\} }

The symmetric difference of two sets \seq_var_1 and \seq_var_2 can be stored into \seq_var_3 by computing the difference between \seq_var_1 and \seq_var_2 and storing the result as \l__⟨pkg⟩_internal_seq, then the difference between \seq_var_2 and \seq_var_1, and finally concatenating the two differences to get the symmetric differences.
\seq_set_eq:NN \l__⟨pkg⟩_internal_seq \seq_var_1
\seq_map_inline:Nn \seq_var_2
{  \seq_remove_all:Nn \l__⟨pkg⟩_internal_seq \{#1\} }
\seq_set_eq:NN \seq_var_3 \seq_var_2
\seq_map_inline:Nn \seq_var_1
{  \seq_remove_all:Nn \seq_var_3 \{#1\} }
\seq_concat:NNN \seq_var_3 \seq_var_3 \l__⟨pkg⟩_internal_seq

\c_empty_seq
Constant that is always empty.

Rev: 2012-07-02

20.11 Constant and scratch sequences
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.

\l_tmpa_seq \l_tmpb_seq

Rev: 2012-04-26

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

\g_tmpa_seq \g_tmpb_seq

Rev: 2012-04-26

20.12 Viewing sequences

\seq_show:N \seq_show:N \seq show:N \seq show:N (seq var)
\seq_show:c

Displays the entries in the \langle seq var \rangle in the terminal.

Updated: 2021-04-29

\seq_log:N \seq_log:N \seq log:N \seq log:N (seq var)
\seq_log:c

Writes the entries in the \langle seq var \rangle in the log file.

Rev: 2014-08-12
Updated: 2021-04-29
Chapter 21

The l3int module

Integers

Calculation and comparison of integer values can be carried out using literal numbers, \texttt{int}
registers, constants and integers stored in token list variables. The standard operators \texttt{+}, \texttt{-}, \texttt{/} and \texttt{*} and parentheses can be used within such expressions to carry arithmetic
operations. This module carries out these functions on integer expressions (\texttt{⟨int expr⟩}).

21.1 Integer expressions

Throughout this module, (almost) all \texttt{n}-type argument allow for an \texttt{⟨int expr⟩} argument
with the following syntax. The \texttt{⟨integer expression⟩} should consist, after expansion,
of \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{(}, \texttt{)} and of course integer operands. The result is calculated by applying
standard mathematical rules with the following peculiarities:

- \texttt{/} 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 oper-
  ations \texttt{a*b/c}, for which \texttt{a*b} may be arbitrarily large (but the operands \texttt{a}, \texttt{b}, \texttt{c}
  are still constrained to an absolute value at most \(2^{31} - 1\));
- parentheses may not appear after unary \texttt{+} or \texttt{-}, namely placing \texttt{+()} or \texttt{-()} at the
  start of an expression or after \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/} or \texttt{()} leads to an error.

Each integer operand can be either an integer variable (with no need for \texttt{\int_use:N}) or
an integer denotation. For example both

\begin{verbatim}
\int_show:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
\end{verbatim}

and

\begin{verbatim}
\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 ) }
\end{verbatim}

166
show the same result $-6$ because $\_\_\_m\_y\_t1$ expands to the integer denotation 5 while the integer variable $\_\_\_m\_y\_i\_n\_t$ 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 $\exp\_n\_o\_t\_n$ and its variants have no effect while $\exp\_n\_o\_t\_N$ may incorrectly interrupt the expression.

\TeXhackers note: Exactly two expansions are needed to evaluate $\int\_e\_v\_a\_l\_n$. The result is not an ⟨internal integer⟩, and therefore should be terminated by a space if used in $\int\_v\_a\_l\_u\_e\_w$ or in a \TeX-style integer assignment.

As all \TeX integers, integer operands can also be: $\value{\langle L\_A\_T\_E\_X2\_c\_o\_n\_t\_r\_u\_e\rangle}$; dimension or skip variables, converted to integers in \sp; the character code of some character given as ‘⟨char⟩’ or ‘\langle char⟩’; octal numbers given as ‘‘ followed by digits from 0 to 7; or hexadecimal numbers given as ‘’ followed by digits and upper case letters from A to F.
\int_eval:n \{ (\text{int expr}) \}

Evaluates the (\text{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. The (\text{int expr}) 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 $a\times b/c$, for which $a\times b$ may be arbitrarily large;
- 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 \texttt{\int_use:N}) or an integer denotation. For example both

\begin{verbatim}
\int_eval:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
\end{verbatim}

and

\begin{verbatim}
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { 5 }
\int_new:N \l_my_int
\int_set:Nn \l_my_int { 4 }
\int_eval:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
\end{verbatim}

evaluate to $-6$ because $\l_my_tl$ expands to the integer denotation 5. As the (\text{int expr}) is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and \texttt{\exp_not:n} and its variants have no effect while \texttt{\exp_not:N} may incorrectly interrupt the expression.

\TeXhackers note: Exactly two expansions are needed to evaluate \texttt{\int_eval:n}. The result is not an (\emph{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 \texttt{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 ’\langle\text{char}\rangle’.

\int_eval:w \{ (\text{int expr}) \}

Nov: 2018-03-30

Evaluates the (\text{int expr}) as described for \texttt{\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 \texttt{\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, \texttt{\int_eval:w 1_9+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 \texttt{\int_eval:n}.
\verb|\int_sign:n| \quad \verb|\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.

\verb|\int_abs:n| \quad \verb|\int_abs:n (\int expr)|

Evaluates the \(\int expr\) as described for \verb|\int_eval:n| and leaves the absolute value of the result in the input stream as an \(\text{integer denotation}\) after two expansions.

\verb|\int_div_round:nn| \quad \verb|\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 \(\text{integer denotation}\) after two expansions.

\verb|\int_div_truncate:nn| \quad \verb|\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 \(\text{integer denotation}\) after two expansions.

\verb|\int_max:nn| \quad \verb|\int_max:nn (\int expr_1) (\int expr_2)|
\verb|\int_min:nn| \quad \verb|\int_min:nn (\int expr_1) (\int expr_2)|

Evaluates the \(\int expr\)s as described for \verb|\int_eval:n| and leaves either the larger or smaller value in the input stream as an \(\text{integer denotation}\) after two expansions.

\verb|\int_mod:nn| \quad \verb|\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 \(\int\_\text{div\_truncate:nn} (\int expr_1) (\int expr_2)\) 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 \(\text{integer denotation}\) after two expansions.

21.2 Creating and initialising integers

\verb|\int_new:N| \quad \verb|\int_new:N (integer)|

\verb|\int_new:c| \quad \verb|\int_new:c (integer)|

\verb|\int_const:Nn| \quad \verb|\int_const:Nn (integer) {\int expr}|\n\verb|\int_const:cn| \quad \verb|\int_const:cn (integer)|

\verb|\int_const:Nn| \quad \verb|\int_const:Nn (integer) {\int expr}|\n\verb|\int_const:cn| \quad \verb|\int_const:cn (integer)|

\verb|\int_const:Nn| \quad \verb|\int_const:Nn (integer) {\int expr}|\n\verb|\int_const:cn| \quad \verb|\int_const:cn (integer)|

Creates a new \(\text{integer}\) or raises an error if the name is already taken. The declaration is global. The \(\text{integer}\) is initially equal to 0.

\verb|\int_new:N| \quad \verb|\int_new:N (integer)|

\verb|\int_new:c| \quad \verb|\int_new:c (integer)|

\verb|\int_const:Nn| \quad \verb|\int_const:Nn (integer) {\int expr}|\n\verb|\int_const:cn| \quad \verb|\int_const:cn (integer)|

\verb|\int_const:Nn| \quad \verb|\int_const:Nn (integer) {\int expr}|\n\verb|\int_const:cn| \quad \verb|\int_const:cn (integer)|

Creates a new constant \(\text{integer}\) or raises an error if the name is already taken. The value of the \(\text{integer}\) is set globally to the \(\int expr\).
\section*{21.3 Setting and incrementing integers}

\begin{itemize}
  \item \texttt{\textbackslash int\_add:Nn} \texttt{\textbackslash int\_add:cn} \texttt{\textbackslash int\_gadd:Nn} \texttt{\textbackslash int\_gadd:cn}
  \begin{itemize}
    \item \texttt{\textbackslash int\_add:Nn} \texttt{\textbackslash int\_add:cn} \texttt{\textbackslash int\_gadd:Nn} \texttt{\textbackslash int\_gadd:cn}
  \end{itemize}
  \begin{itemize}
    \item \texttt{\textbackslash int\_add:Nn} \texttt{\textbackslash int\_add:cn} \texttt{\textbackslash int\_gadd:Nn} \texttt{\textbackslash int\_gadd:cn}
  \end{itemize}
  \begin{itemize}
    \item \texttt{\textbackslash int\_add:Nn} \texttt{\textbackslash int\_add:cn} \texttt{\textbackslash int\_gadd:Nn} \texttt{\textbackslash int\_gadd:cn}
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \texttt{\textbackslash int\_decr:N} \texttt{\textbackslash int\_decr:c} \texttt{\textbackslash int\_gdecr:N} \texttt{\textbackslash int\_gdecr:c}
  \begin{itemize}
    \item \texttt{\textbackslash int\_decr:N} \texttt{\textbackslash int\_decr:c} \texttt{\textbackslash int\_gdecr:N} \texttt{\textbackslash int\_gdecr:c}
  \end{itemize}
  \begin{itemize}
    \item \texttt{\textbackslash int\_decr:N} \texttt{\textbackslash int\_decr:c} \texttt{\textbackslash int\_gdecr:N} \texttt{\textbackslash int\_gdecr:c}
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \texttt{\textbackslash int\_incr:N} \texttt{\textbackslash int\_incr:c} \texttt{\textbackslash int\_gincr:N} \texttt{\textbackslash int\_gincr:c}
  \begin{itemize}
    \item \texttt{\textbackslash int\_incr:N} \texttt{\textbackslash int\_incr:c} \texttt{\textbackslash int\_gincr:N} \texttt{\textbackslash int\_gincr:c}
  \end{itemize}
  \begin{itemize}
    \item \texttt{\textbackslash int\_incr:N} \texttt{\textbackslash int\_incr:c} \texttt{\textbackslash int\_gincr:N} \texttt{\textbackslash int\_gincr:c}
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \texttt{\textbackslash int\_set:Nn} \texttt{\textbackslash int\_set:cn} \texttt{\textbackslash int\_gset:Nn} \texttt{\textbackslash int\_gset:cn}
  \begin{itemize}
    \item \texttt{\textbackslash int\_set:Nn} \texttt{\textbackslash int\_set:cn} \texttt{\textbackslash int\_gset:Nn} \texttt{\textbackslash int\_gset:cn}
  \end{itemize}
  \begin{itemize}
    \item \texttt{\textbackslash int\_set:Nn} \texttt{\textbackslash int\_set:cn} \texttt{\textbackslash int\_gset:Nn} \texttt{\textbackslash int\_gset:cn}
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item \texttt{\textbackslash int\_set:Nn} \texttt{\textbackslash int\_set:cn} \texttt{\textbackslash int\_gset:Nn} \texttt{\textbackslash int\_gset:cn}
  \begin{itemize}
    \item \texttt{\textbackslash int\_set:Nn} \texttt{\textbackslash int\_set:cn} \texttt{\textbackslash int\_gset:Nn} \texttt{\textbackslash int\_gset:cn}
  \end{itemize}
\end{itemize}

Sets \texttt{\textbackslash int\_add:Nn} \texttt{\textbackslash int\_add:cn} \texttt{\textbackslash int\_gadd:Nn} \texttt{\textbackslash int\_gadd:cn} \texttt{\textbackslash int\_decr:N} \texttt{\textbackslash int\_decr:c} \texttt{\textbackslash int\_gdecr:N} \texttt{\textbackslash int\_gdecr:c} \texttt{\textbackslash int\_incr:N} \texttt{\textbackslash int\_incr:c} \texttt{\textbackslash int\_gincr:N} \texttt{\textbackslash int\_gincr:c} \texttt{\textbackslash int\_set:Nn} \texttt{\textbackslash int\_set:cn} \texttt{\textbackslash int\_gset:Nn} \texttt{\textbackslash int\_gset:cn} \texttt{\textbackslash int\_decr:N} \texttt{\textbackslash int\_decr:c} \texttt{\textbackslash int\_gdecr:N} \texttt{\textbackslash int\_gdecr:c} \texttt{\textbackslash int\_incr:N} \texttt{\textbackslash int\_incr:c} \texttt{\textbackslash int\_gincr:N} \texttt{\textbackslash int\_gincr:c} \texttt{\textbackslash int\_set:Nn} \texttt{\textbackslash int\_set:cn} \texttt{\textbackslash int\_gset:Nn} \texttt{\textbackslash int\_gset:cn} \texttt{\textbackslash int\_decr:N} \texttt{\textbackslash int\_decr:c} \texttt{\textbackslash int\_gdecr:N} \texttt{\textbackslash int\_gdecr:c} \texttt{\textbackslash int\_incr:N} \texttt{\textbackslash int\_incr:c} \texttt{\textbackslash int\_gincr:N} \texttt{\textbackslash int\_gincr:c} \texttt{\textbackslash int\_set:Nn} \texttt{\textbackslash int\_set:cn} \texttt{\textbackslash int\_gset:Nn} \texttt{\textbackslash int\_gset:cn}
\int_sub:Nn \langle integer \rangle \{\langle int\ expr\rangle\}
Subtracts the result of the $\langle int\ expr\rangle$ from the current content of the $\langle integer\rangle$.

Updated: 2011-10-22

21.4 Using integers

\int_use:N * \langle integer \rangle
Recover the content of an $\langle integer\rangle$ 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
$\langle integer\rangle$ 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 \TeX{}3
names for this primitive.

21.5 Integer expression conditionals

\int_compare_p:nNn * \langle integer \rangle \{\langle int\ expr\rangle\} \{\langle int\ expr\rangle\}
\int_compare:nNnTF
\{\langle int\ expr\rangle\} \{\langle int\ expr\rangle\}
\{\langle true\ code\rangle\} \{\langle false\ code\rangle\}
This function first evaluates each of the $\langle int\ expr\rangle$s as described for \int_eval:n. The
two results are then compared using the $\langle relation\rangle$:

Equal $=$
Greater than $>$
Less than $<$

This function is less flexible than \int_compare:nTF but around 5 times faster.
\int_compare_p:n \star \\int_compare:nTF \star
\int_compare:nTF
\begin{verbatim}
\int_compare_p:n
\int_compare:nTF
\{
\int{\text{expr}} \, \langle \text{relation}_1 \rangle
\ldots
\int{\text{expr}_{N}} \, \langle \text{relation}_N \rangle
\int{\text{expr}_{N+1}}
\}
\int_compare:nTF
\{
\int{\text{expr}} \, \langle \text{relation}_1 \rangle
\ldots
\int{\text{expr}_{N}} \, \langle \text{relation}_N \rangle
\int{\text{expr}_{N+1}}
\}
\{\langle \text{true code} \rangle\} \{\langle \text{false code} \rangle\}
\end{verbatim}

This function evaluates the ⟨\text{int expr}⟩s as described for \int_eval:n and compares
consecutive result using the corresponding ⟨\text{relation}⟩, namely it compares ⟨\text{int expr}_1⟩
and ⟨\text{int expr}_2⟩ using the ⟨\text{relation}_1⟩, then ⟨\text{int expr}_2⟩ and ⟨\text{int expr}_3⟩ using
the ⟨\text{relation}_2⟩, until finally comparing ⟨\text{int expr}_N⟩ and ⟨\text{int expr}_{N+1}⟩ using the
⟨\text{relation}_N⟩. The test yields \textbf{true} if all comparisons are \textbf{true}. Each ⟨\text{int expr}⟩ is
evaluated only once, and the evaluation is lazy, in the sense that if one comparison is
\textbf{false}, then no other ⟨\text{integer expression}⟩ is evaluated and no other comparison is
performed. The ⟨\text{relations}⟩ can be any of the following:

\begin{center}
\begin{tabular}{ll}
Equal & = or == \\
Greater than or equal to & >= \\
Greater than & > \\
Less than or equal to & <= \\
Less than & < \\
Not equal & != \\
\end{tabular}
\end{center}

This function is more flexible than \int_compare:nNnTF but around 5 times slower.
This function evaluates the \(\text{test int expr}\) and compares this in turn to each of the \(\text{int expr cases}\). If the two are equal then the associated \(\text{code}\) is left in the input stream and other cases are discarded. If any of the cases are matched, the \(\text{true code}\) is also inserted into the input stream (after the code for the appropriate case), while if none match then the \(\text{false code}\) is inserted. The function \(\text{int_case:nn}\), which does nothing if there is no match, is also available. For example

\[
\text{int_case:nnF}
\{ 2 * 5 \}
\{
\{ 5 \} { Small }
\{ 4 + 6 \} { Medium }
\{ -2 * 10 \} { Negative }
\}
\{ No idea! \}
\]

leaves “Medium” in the input stream.

This function first evaluates the \(\text{int expr}\) as described for \(\text{int_eval:n}\). It then evaluates if this is odd or even, as appropriate.

This function first evaluates the \(\text{int expr}\) as described for \(\text{int_eval:n}\). It then evaluates if this is zero or not.

### 21.6 Integer expression loops

Places the \(\text{code}\) in the input stream for \(\text{T}_{\text{e}}\text{X}\) to process, and then evaluates the relationship between the two \(\text{int expr}\)s as described for \(\text{int_compare:nNnTF}\). If the test is \text{false} then the \(\text{code}\) is inserted into the input stream again and a loop occurs until the \(\text{relation}\) is true.
\texttt{\textbackslash int\_do\_while:nNnn} \hspace{1em} \texttt{\textbackslash int\_do\_while:nNnn \{\textit{int expr}\}_1 \{\textit{relation}\} \{\textit{int expr}\}_2 \{\textit{code}\}}

Places the \{\textit{code}\} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \{\textit{int expr}\}s as described for \texttt{\textbackslash int\_compare:nNnTF}. If the test is \texttt{true} then the \{\textit{code}\} is inserted into the input stream again and a loop occurs until the \{\textit{relation}\} is \texttt{false}.

\texttt{\textbackslash int\_until\_do:nNnn} \hspace{1em} \texttt{\textbackslash int\_until\_do:nNnn \{\textit{int expr}\}_1 \{\textit{relation}\} \{\textit{int expr}\}_2 \{\textit{code}\}}

Evaluates the relationship between the two \{\textit{int expr}\}s as described for \texttt{\textbackslash int\_compare:nNnTF}, and then places the \{\textit{code}\} in the input stream if the \{\textit{relation}\} is \texttt{false}. After the \{\textit{code}\} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash int\_while\_do:nNnn} \hspace{1em} \texttt{\textbackslash int\_while\_do:nNnn \{\textit{int expr}\}_1 \{\textit{relation}\} \{\textit{int expr}\}_2 \{\textit{code}\}}

Evaluates the relationship between the two \{\textit{int expr}\}s as described for \texttt{\textbackslash int\_compare:nNnTF}, and then places the \{\textit{code}\} in the input stream if the \{\textit{relation}\} is \texttt{true}. After the \{\textit{code}\} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.

\texttt{\textbackslash int\_do\_until:nNn} \hspace{1em} \texttt{\textbackslash int\_do\_until:nNn \{\textit{integer relation}\} \{\textit{code}\}}

\texttt{Updated: 2013-01-13}

Places the \{\textit{code}\} in the input stream for \TeX{} to process, and then evaluates the \{\textit{integer relation}\} as described for \texttt{\textbackslash int\_compare:nNnTF}. If the test is \texttt{false} then the \{\textit{code}\} is inserted into the input stream again and a loop occurs until the \{\textit{relation}\} is \texttt{true}.

\texttt{\textbackslash int\_while\_do:nNn} \hspace{1em} \texttt{\textbackslash int\_while\_do:nNn \{\textit{integer relation}\} \{\textit{code}\}}

\texttt{Updated: 2013-01-13}

Places the \{\textit{code}\} in the input stream for \TeX{} to process, and then evaluates the \{\textit{integer relation}\} as described for \texttt{\textbackslash int\_compare:nNnTF}. If the test is \texttt{true} then the \{\textit{code}\} is inserted into the input stream again and a loop occurs until the \{\textit{relation}\} is \texttt{false}.

\texttt{\textbackslash int\_until\_do:nNn} \hspace{1em} \texttt{\textbackslash int\_until\_do:nNn \{\textit{integer relation}\} \{\textit{code}\}}

\texttt{Updated: 2013-01-13}

Evaluates the \{\textit{integer relation}\} as described for \texttt{\textbackslash int\_compare:nNnTF}, and then places the \{\textit{code}\} in the input stream if the \{\textit{relation}\} is \texttt{false}. After the \{\textit{code}\} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash int\_while\_do:nNn} \hspace{1em} \texttt{\textbackslash int\_while\_do:nNn \{\textit{integer relation}\} \{\textit{code}\}}

\texttt{Updated: 2013-01-13}

Evaluates the \{\textit{integer relation}\} as described for \texttt{\textbackslash int\_compare:nNnTF}, and then places the \{\textit{code}\} in the input stream if the \{\textit{relation}\} is \texttt{true}. After the \{\textit{code}\} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.
21.7 Integer step functions

\texttt{\textbackslash int\_step\_function:nN } \star \texttt{\textbackslash int\_step\_function:nnN} \star \texttt{\textbackslash int\_step\_function:nnNn} \star

This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, all of which should be integer expressions. The \langle function \rangle is then placed in front of each \langle value \rangle from the \langle initial value \rangle to the \langle final value \rangle in turn (using \langle step \rangle between each \langle value \rangle). The \langle step \rangle must be non-zero. If the \langle step \rangle is positive, the loop stops when the \langle value \rangle becomes larger than the \langle final value \rangle. If the \langle step \rangle is negative, the loop stops when the \langle value \rangle becomes smaller than the \langle final value \rangle. The \langle function \rangle should absorb one numerical argument. For example

\texttt{\textbackslash cs\_set:Npn \textbackslash my\_func:n #1 \{ I-saw-\#1 \} \quad } \texttt{\textbackslash int\_step\_function:nnNn \{ 1 \} \{ 1 \} \{ 5 \} \textbackslash my\_func:n}

would print

[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]

The functions \texttt{\textbackslash int\_step\_function:nN} and \texttt{\textbackslash int\_step\_function:nnN} both use a fixed \langle step \rangle of 1, and in the case of \texttt{\textbackslash int\_step\_function:nN} the \langle initial value \rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\texttt{\textbackslash int\_step\_inline:nn} \texttt{\textbackslash int\_step\_inline:nn} \texttt{\textbackslash int\_step\_inline:nnn}

This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, all of which should be integer expressions. Then for each \langle value \rangle from the \langle initial value \rangle to the \langle final value \rangle in turn (using \langle step \rangle between each \langle value \rangle), the \langle code \rangle is inserted into the input stream with \#1 replaced by the current \langle value \rangle. Thus the \langle code \rangle should define a function of one argument (\#1).

The functions \texttt{\textbackslash int\_step\_inline:nn} and \texttt{\textbackslash int\_step\_inline:nnn} both use a fixed \langle step \rangle of 1, and in the case of \texttt{\textbackslash int\_step\_inline:nnn} the \langle initial value \rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\texttt{\textbackslash int\_step\_variable:nN} \texttt{\textbackslash int\_step\_variable:nNn} \texttt{\textbackslash int\_step\_variable:nNnN}

This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, all of which should be integer expressions. Then for each \langle value \rangle from the \langle initial value \rangle to the \langle final value \rangle in turn (using \langle step \rangle between each \langle value \rangle), the \langle code \rangle is inserted into the input stream, with the \langle tl var \rangle defined as the current \langle value \rangle. Thus the \langle code \rangle should make use of the \langle tl var \rangle.

The functions \texttt{\textbackslash int\_step\_variable:nN} and \texttt{\textbackslash int\_step\_variable:nNnN} both use a fixed \langle step \rangle of 1, and in the case of \texttt{\textbackslash int\_step\_variable:nNn} the \langle initial value \rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.
### 21.8 Formatting integers

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

\begin{verbatim}
\int_to_arabic:n \{\int expr\}
\end{verbatim}

Places the value of the \(\int \text{expr}\) in the input stream as digits, with category code 12 (other).

\begin{verbatim}
\int_to_alph:n \{\int expr\}
\end{verbatim}

Evaluates the \(\int \text{expr}\) 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

\begin{verbatim}
\int_to_alph:n \{ 1 \}
\end{verbatim}

places a in the input stream,

\begin{verbatim}
\int_to_alph:n \{ 26 \}
\end{verbatim}

is represented as z and

\begin{verbatim}
\int_to_alph:n \{ 27 \}
\end{verbatim}

is converted to aa. For conversions using other alphabets, use \texttt{\int_to_symbols:nnn} to define an alphabet-specific function. The basic \texttt{\int_to_alph:n} and \texttt{\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).

\begin{verbatim}
\int_to_symbols:nnn \{(\int expr)\} \{(total symbols)\} \{(value to symbol mapping)\}
\end{verbatim}

This is the low-level function for conversion of an \(\int \text{expr}\) into a symbolic form (often letters). The \(\text{total symbols}\) available should be given as an integer expression. Values are actually converted to symbols according to the \(\text{value to symbol mapping}\). This should be given as \(\text{total symbols}\) pairs of entries, a number and the appropriate symbol. Thus the \texttt{\int_to_alph:n} function is defined as

\begin{verbatim}
\cs_new:Npn \int_to_alph:n #1
{\int_to_symbols:nnn \{#1\} \{26\}
{1} \{a\}
{2} \{b\}
...\n{26} \{z\}}
\end{verbatim}
\int_to_bin:n \{ (int expr) \}

Calculates the value of the \(\langle int \ expr \rangle\) and places the binary representation of the result in the input stream.

\int_to_hex:n \{ (int expr) \}
\int_to_Hex:n \{ (int expr) \}

Calculates the value of the \(\langle int \ expr \rangle\) 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 expr) \}

Calculates the value of the \(\langle int \ expr \rangle\) 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 expr) \} \{ (base) \}
\int_to_Base:nn \{ (int expr) \} \{ (base) \}

Calculates the value of the \(\langle int \ expr \rangle\) and converts it into the appropriate representation in the \(\langle base \rangle\); 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 \(\langle base \rangle\) value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_roman:n \{ (int expr) \}
\int_to_Roman:n \{ (int expr) \}

Places the value of the \(\langle int \ expr \rangle\) 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 \(\bar{v}\) for 5000) is not used. For instance \int_to_roman:n \{ 8249 \} expands to mmmmmmmccxlix.

\int_from_alph:n \{ (letters) \}
\int_from_Alph:n \{ (letters) \}

Converts the \(\langle letters \rangle\) into the integer (base 10) representation and leaves this in the input stream. The \(\langle letters \rangle\) 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 \{ (binary number) \}

Converts the \(\langle binary \ number \rangle\) into the integer (base 10) representation and leaves this in the input stream. The \(\langle binary \ number \rangle\) 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.

21.9 Converting from other formats to integers
\textbf{\int_from_hex:n} \{\langle \text{hexadecimal number} \rangle \}

Converting the \langle \text{hexadecimal number} \rangle into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the \langle \text{hexadecimal number} \rangle by upper or lower case letters. The \langle \text{hexadecimal number} \rangle 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 \texttt{\int_to_hex:n} and \texttt{\int_to_Hex:n}.

\textbf{\int_from_oct:n} \{\langle \text{octal number} \rangle \}

Converting the \langle \text{octal number} \rangle into the integer (base 10) representation and leaves this in the input stream. The \langle \text{octal number} \rangle 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 \texttt{\int_to_oct:n}.

\textbf{\int_from_roman:n} \{\langle \text{roman numeral} \rangle \}

Converting the \langle \text{roman numeral} \rangle into the integer (base 10) representation and leaves this in the input stream. The \langle \text{roman numeral} \rangle is first converted to a string, with no expansion. The \langle \text{roman numeral} \rangle 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 \texttt{\int_to_roman:n} and \texttt{\int_to_Roman:n}.

\textbf{\int_from_base:nn} \{\langle \text{number} \rangle \} \{\langle \text{base} \rangle \}

Converting the \langle \text{number} \rangle expressed in \langle \text{base} \rangle into the appropriate value in base 10. The \langle \text{number} \rangle is first converted to a string, with no expansion. The \langle \text{number} \rangle should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum \langle \text{base} \rangle value is 36. This is the inverse function of \texttt{\int_to_base:nn} and \texttt{\int_to_Base:nn}.

\section*{21.10 Random integers}

\textbf{\int_rand:nn} \{\langle \text{int expr}_1 \rangle \} \{\langle \text{int expr}_2 \rangle \}

Evaluating the two \langle \text{int expr} \rangle s and produces a pseudo-random number between the two (with bounds included). This is not available in older versions of X\LaTeX{}.

\textbf{\int_rand:n} \{\langle \text{int expr} \rangle \}

Evaluating the \langle \text{int expr} \rangle then produces a pseudo-random number between 1 and the \langle \text{int expr} \rangle (included). This is not available in older versions of X\LaTeX{}.

\section*{21.11 Viewing integers}

\textbf{\int_show:N} \{\langle \text{integer} \rangle \}

Displays the value of the \langle \text{integer} \rangle on the terminal.
\int_show:n {\langle int expr\rangle}

Displays the result of evaluating the \langle int expr\rangle on the terminal.

Updated: 2015-08-07

\int_log:N \langle integer\rangle
\int_log:c

Writes the value of the \langle integer\rangle in the log file.

Updated: 2015-08-03

\int_log:n {\langle int expr\rangle}

Writes the result of evaluating the \langle int expr\rangle in the log file.

Updated: 2015-08-07

\section{21.12 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.

New: 2018-05-07

\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.

\section{21.13 Scratch integers}

\l_tmpa_int \l_tmpb_int

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

\g_tmpa_int \g_tmpb_int

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

\int_value:w \ (integer) \\
\int_value:w \ (integer denotation) \ (optional space)

Expands the following tokens until an \textit{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 \textit{integer} can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any \TeX{} register except \texttt{\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 \texttt{\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, \texttt{\int_eval:n} is the preferred approach to generating numbers.

\TeX{}hackers note: This is the \TeX{} primitive \texttt{\number}.

21.15 Primitive conditionals

\if_int_compare:w \ (integer\_1) \ (relation) \ (integer\_2) \\
\{true\ code\} \\
\else:\ \\
\ \{false\ code\} \\
\fi:

Compare two integers using \textit{relation}, which must be one of =, < or > with category code 12. The \texttt{\else:} branch is optional.

\TeX{}hackers note: This is the \TeX{} primitive \texttt{\ifnum}.

\if_case:w \ (integer) \ (case\_0) \\
\or: \ (case\_1) \\
\or: \ ... \\
\else: \ (default) \\
\fi:

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

\TeX{}hackers note: These are the \TeX{} primitives \texttt{\ifcase} and \texttt{\or}. 
\if_int_odd:w \if_int_odd:w \langle tokens \rangle \langle \text{optional space} \rangle
\langle \text{true code} \rangle
\else:
\langle \text{true code} \rangle
\fi:
Expands \langle tokens \rangle until a non-numeric token or a space is found, and tests whether the resulting \langle integer \rangle is odd. If so, \langle \text{true code} \rangle is executed. The \texttt{\textbackslash else:} branch is optional.

\TeXhackers note: This is the \TeX primitive \texttt{\textbackslash ifodd}. 
Chapter 22

The l3flag module
Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any non-negative value, which we call its \( \langle \text{height} \rangle \). In expansion-only contexts, a flag can only be “raised”: this increases the \( \langle \text{height} \rangle \) by 1. The \( \langle \text{height} \rangle \) can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local. They are referenced by a \( \langle \text{flag name} \rangle \) such as \texttt{fp\_overflow}. The \( \langle \text{flag name} \rangle \) is used as part of \use{c} constructions hence is expanded at point of use. It must expand to character tokens only, with no spaces.

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 \texttt{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. Flags should not be used unless unavoidable.

22.1 Setting up flags

\begin{verbatim}
\flag_new:n \flag_new:n {\langle flag name\rangle}
\end{verbatim}

Creates a new flag with a name given by \( \langle \text{flag name} \rangle \), or raises an error if the name is already taken. The \( \langle \text{flag name} \rangle \) may not contain spaces. The declaration is global, but flags are always local variables. The \( \langle \text{flag} \rangle \) initially has zero height.

\begin{verbatim}
\flag_clear:n \flag_clear:n {\langle flag name\rangle}
\end{verbatim}

The \( \langle \text{flag} \rangle \)’s height is set to zero. The assignment is local.
22.2 Expandable flag commands

\flag_clear_new:n \{flag name\}

Ensures that the \texttt{(flag)} exists globally by applying \flag_new:n if necessary, then applies \flag_clear:n, setting the height to zero locally.

\flag_show:n \{flag name\}

Displays the \texttt{(flag)}'s height in the terminal.

\flag_log:n \{flag name\}

Writes the \texttt{(flag)}'s height to the log file.
Chapter 23

The \texttt{l3clist} module

Comma separated lists

Comma lists (in short, \texttt{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 \texttt{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\ 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 \texttt{\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,

\begin{verbatim}
\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\} , }
\end{verbatim}

results in \texttt{\l_my_clist} containing \texttt{a,b,c\d,\{e\},\{f\}} namely the five items \texttt{a}, \texttt{b}, \texttt{c\d}, \texttt{e} and \texttt{f}. Comma lists normally do not contain empty or blank items so the following gives an empty comma list:

\begin{verbatim}
\clist_clear_new:N \l_my_clist
\clist_set:Nn \l_my_clist { , , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
\end{verbatim}

and it leaves \texttt{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 \texttt{n}-type token list is a valid comma list input for \texttt{l3clist} functions, which will split the token list at every comma and process the items as described above. On the other hand, \texttt{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

184
occurred. Because comma list variables are token list variables, expanding them once yields their items separated by commas, and \lst functions such as \tl_show:N can be applied to them. (These functions often have \lclist 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 \lseq) may be advisable if speed is important. The exception is that \clist_if_in:NnTF and \clist_remove_duplicates:N may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains \{, \} or \# (assuming the usual \TeX{} category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

### 23.1 Creating and initialising comma lists

\begin{verbatim}
\clist_new:N \clist_new:Nn \clist_new:c \clist_new:cn \clist_new:cc
\end{verbatim}

Creates a new \clist or raises an error if the name is already taken. The declaration is global. The \clist initially contains no items.

\begin{verbatim}
\clist_new:N \clist_new:Nn \clist_new:c \clist_new:cn \clist_new:cc
\end{verbatim}

Creates a new constant \clist or raises an error if the name is already taken. The value of the \clist is set globally to the \comma list.

\begin{verbatim}
\clist_const:Nn \clist_const:cc \clist_const:cc
\end{verbatim}

New: 2014-07-05

Sets the content of \comma list\_1 equal to that of \comma list\_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.

\begin{verbatim}
\clist_set_eq:NN \clist_set_eq:cc \clist_set_eq:cc
\end{verbatim}

New: 2014-07-17

Converts the data in the \sequence into a \clist; the original \sequence is unchanged. Items which contain either spaces or commas are surrounded by braces.
\clist_concat:NNN \clist_concat:ccc \clist_gconcat:NNN \clist_gconcat:ccc
\clist_concat:NNN (comma list_1) (comma list_2) (comma list_3)
Concatenates the content of (comma list_2) and (comma list_3) together and saves the result in (comma list_1). The items in (comma list_2) are placed at the left side of the new comma list.

\clist_if_exist_p:N \clist_if_exist_p:N \clist_if_exist:NTF \clist_if_exist:N \clist_if_exist:c
\clist_if_exist_p:N \clist_if_exist_p:c \clist_if_exist:N \clist_if_exist:c
Tests whether the (clist var) is currently defined. This does not check that the (clist var) really is a comma list.

Updated: 2011-09-05

23.2 Adding data to comma lists

\clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn
\clist_set:Nn \clist_set:Nn (clist var) {(item_1),...,(item_n)}
\clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn \clist_set:Nn
New: 2011-09-06
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:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn
\clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn
\clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn \clist_put_left:Nn
Updated: 2011-09-05
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:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn
\clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn
\clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn \clist_put_right:Nn
Updated: 2011-09-05
Appends the (items) to the right of the (clist var). Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some (tokens) as a single (item) even if the (tokens) contain commas or spaces, add a set of braces: \clist_put_right:Nn (clist var) { { (tokens) } }.
23.3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

\clist_remove_duplicates:N \clist_remove_duplicates:N \clist_gremove_duplicates:N \clist_gremove_duplicates:c

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:Nn \clist_gremove_all:Nn \clist_gremove_all:c

Updated: 2011-09-06

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

\TeXhackers note: The function may fail if the ⟨item⟩ contains {, }, or # (assuming the usual \TeX category codes apply).

\clist_reverse:N \clist_reverse:c \clist_greverse:N \clist_greverse:c

New: 2014-07-18

Reverses the order of items stored in the ⟨clist var⟩.

\clist_reverse:n \clist_reverse:n

New: 2014-07-18

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.
### 23.4 Comma list conditionals

Tests if the \langle list var\rangle is empty (containing no items).

Tests if the \langle list var\rangle 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.

Tests if the \langle item\rangle is present in the \langle list var\rangle. In the case of an n-type \langle comma list\rangle, the usual rules of space trimming and brace stripping apply. Hence,

\begin{verbatim}
clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
\end{verbatim}
yields true.

\textbf{TeXhackers note:} The function may fail if the \langle item\rangle contains \{,\}, or \# (assuming the usual \TeX category codes apply).

### 23.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, \textit{i.e.} any local assignments made by the \langle function\rangle or \langle code\rangle 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}'.

\begin{verbatim}
clist_if_in:NnTF
clist_if_in:(NV|No|cn|cV|co)TF
clist_if_in:nnTF
\end{verbatim}
When the comma list is given as an \texttt{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 \texttt{n}-type comma lists.

\texttt{clist_map_function:NN} \(\uparrow\) \texttt{clist_map_function:ch} \(\uparrow\) \texttt{clist_map_function:cn} \(\uparrow\) \texttt{clist_map_function:en} \(\uparrow\) \texttt{clist_map_function:nn} \(\uparrow\) \hspace{2cm} updated: 2012-06-29

\texttt{clist_map_function:NN} \texttt{clist var} \texttt{function}

Applies \texttt{function} to every \texttt{item} stored in the \texttt{clist var}. The \texttt{function} receives one argument for each iteration. The \texttt{items} are returned from left to right. The function \texttt{clist_map_inline:Nn} is in general more efficient than \texttt{clist_map_function:NN}.

\texttt{clist_map_function:NN} \hspace{2cm} ⊖ \texttt{clist_map_function:ch} \hspace{2cm} ⊖ \texttt{clist_map_function:cn} \hspace{2cm} ⊖ \texttt{clist_map_function:en} \hspace{2cm} ⊖ \texttt{clist_map_function:nn} \hspace{2cm} ⊖ \texttt{clist_map_function:nn}

\texttt{clist_map_inline:Nn} \texttt{clist var} \{\texttt{inline function}\}

Applies \texttt{inline function} to every \texttt{item} stored within the \texttt{clist var}. The \texttt{inline function} should consist of code which receives the \texttt{item} as \#1. The \texttt{items} are returned from left to right.

\texttt{clist_map_inline:Nn} \hspace{2cm} ⊖ \texttt{clist_map_inline:cn} \hspace{2cm} ⊖ \texttt{clist_map_inline:nn} \hspace{2cm} ⊖ \texttt{clist_map_inline:nn}

\texttt{clist_map_variable:NNn} \texttt{clist var} \texttt{variable} \{\texttt{code}\}

Stores each \texttt{item} of the \texttt{clist var} in turn in the (token list) \texttt{variable} and applies the \texttt{code}. The \texttt{code} will usually make use of the \texttt{variable}, but this is not enforced. The assignments to the \texttt{variable} are local. Its value after the loop is the last \texttt{item} in the \texttt{comma list}, or its original value if there were no \texttt{item}. The \texttt{items} are returned from left to right.

\texttt{clist_map_variable:NNn} \hspace{2cm} ⊖ \texttt{clist_map_variable:cn} \hspace{2cm} ⊖ \texttt{clist_map_variable:nn} \hspace{2cm} ⊖ \texttt{clist_map_variable:nn}

\texttt{clist_map_tokens:Nn} \texttt{clist var} \{\texttt{code}\}

\texttt{clist_map_tokens:nn} \{\texttt{comma list}\} \{\texttt{code}\}

Calls \texttt{code} \{\texttt{item}\} for every \texttt{item} stored in the \texttt{clist var}. The \texttt{code} receives each \texttt{item} as a trailing brace group. If the \texttt{code} consists of a single function this is equivalent to \texttt{clist_map_function:nN}.

\texttt{clist_map_tokens:NNn} \hspace{2cm} ⊖ \texttt{clist_map_tokens:cn} \hspace{2cm} ⊖ \texttt{clist_map_tokens:nn} \hspace{2cm} ⊖ \texttt{clist_map_tokens:nn}

\texttt{clist_map_break:}\hspace{2cm} updated: 2012-06-29

\texttt{clist_map_break:}

Used to terminate a \texttt{clist_map...} function before all entries in the \texttt{comma list} have been processed. This normally takes place within a conditional statement, for example

\texttt{clist_map_inline:Nn} \texttt{l_my_clist} \\
\{ \\
\str_if_eq:nTF \{ #1 \} \{ bingo \} \\
\{ \texttt{clist_map_break:} \} \\
\{ % Do something useful \\
\}

Use outside of a \texttt{clist_map...} scenario leads to low level \TeX{} errors.

\TeX{}hackers 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.

\TeX{}hacker 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:N (clist var)
\clist_count:c \clist_count:c (clist var)
\clist_count:n \clist_count:n (clist var)
\clist_count:e \clist_count:e (clist var)

Leaves the number of items in the ⟨clist var⟩ in the input stream as an ⟨integer denotation⟩. The total number of items in a ⟨clist var⟩ includes those which are duplicates, i.e. every item in a ⟨clist var⟩ is counted.
23.6 Using the content of comma lists directly

\clist_use:Nnnn \clist_use:cn

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

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.
Places the contents of the \texttt{comma list} in the input stream, with the appropriate \texttt{separator} between the items. As for \texttt{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 \texttt{separators} are then inserted in the same way as for \texttt{clist_use:Nnnn} and \texttt{clist_use:Nn}, respectively.

\textbf{\texttt{\LaTeX}hackers note:} The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \texttt{items} do not expand further when appearing in an \texttt{e-type} or \texttt{x-type} argument expansion.

\section{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.

\begin{itemize}
  \item \texttt{clist_get:NN} \texttt{clist_get:cn} \texttt{clist_get:NNTF} \texttt{clist_get:CNTF} \texttt{clist_get:NNNN} \\
  \texttt{\LaTeX}hackers note: The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \texttt{items} do not expand further when appearing in an \texttt{e-type} or \texttt{x-type} argument expansion.

  Stores the left-most item from a \texttt{clist var} in the \texttt{token list variable} without removing it from the \texttt{clist var}. The \texttt{token list variable} is assigned locally. In the non-branching version, if the \texttt{clist var} is empty the \texttt{token list variable} is set to the marker value \texttt{q_no_value}.

  \item \texttt{clist_pop:NN} \texttt{clist_pop:cn} \texttt{clist_pop:NNTF} \texttt{clist_pop:CNTF} \texttt{clist_pop:NNNN} \\
  \texttt{\LaTeX}hackers note: The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \texttt{items} do not expand further when appearing in an \texttt{e-type} or \texttt{x-type} argument expansion.

  Pops the left-most item from a \texttt{clist var} into the \texttt{token list variable}, \texttt{i.e.} removes the item from the comma list and stores it in the \texttt{token list variable}. Both of the variables are assigned locally.

  \item \texttt{clist_gpop:NN} \texttt{clist_gpop:cn} \texttt{clist_gpop:NNTF} \texttt{clist_gpop:CNTF} \texttt{clist_gpop:NNNN} \\
  \texttt{\LaTeX}hackers note: The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \texttt{items} do not expand further when appearing in an \texttt{e-type} or \texttt{x-type} argument expansion.

  Pops the left-most item from a \texttt{clist var} into the \texttt{token list variable}, \texttt{i.e.} removes the item from the comma list and stores it in the \texttt{token list variable}. The \texttt{clist var} is modified globally, while the assignment of the \texttt{token list variable} is local.

  \item \texttt{clist_pop:NNTF} \texttt{clist_pop:CNTF} \texttt{clist_pop:NNNN} \\
  \texttt{\LaTeX}hackers note: The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \texttt{items} do not expand further when appearing in an \texttt{e-type} or \texttt{x-type} argument expansion.

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

\end{itemize}
\clist_gpop:NNTF \clist_gpop:cNTF
\New: 2012-05-14

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

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

Add the \{items\} to the top of the \clist. Spaces are removed from both sides of each item as for any n-type comma list.

23.8 Using a single item

\clist_item:Nn \clist_item:cn \clist_item:nn \clist_item:en
\New: 2014-07-17

Indexing items in the \clist 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 (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:cn \clist_rand_item:cn \clist_rand_item:nn \clist_rand_item:en
\New: 2016-12-06

Selects a pseudo-random item of the \clist/\comma list. If the \comma list has no item, the result is empty.

\TeXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the \item does not expand further when appearing in an e-type or x-type argument expansion.

23.9 Viewing comma lists

\clist_show:N \clist_show:c \clist_show:cn
\Updated: 2021-04-29

Displays the entries in the \clist in the terminal.
\clist_show:n \clist_show:n \{\text{tokens}\}

Displays the entries in the comma list in the terminal.

\clist_log:N \clist_log:N \{\text{clist var}\}
\clist_log:c

Writes the entries in the \text{\{clist var\}} in the log file. See also \clist_show:N which displays the result in the terminal.

\clist_log:n \clist_log:n \{\text{tokens}\}

Writes the entries in the comma list in the log file. See also \clist_show:n which displays the result in the terminal.

### 23.10 Constant and scratch comma lists

\c_empty_clist

Constant that is always empty.

\l_tmpa_clist \l_tmpb_clist

New: 2011-09-06

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

\g_tmpa_clist \g_tmpb_clist

New: 2011-09-06

Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
Chapter 24

The \texttt{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: \texttt{\token{}} for anything that deals with tokens and \texttt{\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, \texttt{\if:w}, \texttt{\ifcharcode:w}, and \texttt{\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 \texttt{\showuntilif:w} defined below takes everything until \texttt{\if:w} as an argument, despite the presence of other copies of \texttt{\if:w} under different names.

\input{\showuntilif.w}

A list of all possible shapes and a list of all possible meanings are given in section 24.7.
24.1 Creating character tokens

\char_set_active_eq:NN \{char\} \{function\}
Sets the behaviour of the \{char\} in situations where it is active (category code 13) to be equivalent to that of the \{function\}. The category code of the \{char\} is unchanged by this process. The \{function\} may itself be an active character.

\char_set_active_eq:nN \{integer expression\} \{function\}
Sets the behaviour of the \{char\} which has character code as given by the \{integer expression\} in situations where it is active (category code 13) to be equivalent to that of the \{function\}. The category code of the \{char\} is unchanged by this process. The \{function\} may itself be an active character.

\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 \TeX{}X. Another way to build token lists with unusual category codes is \regex_replace:nnN \{.*\} \{replacement\} \{tl var\}.

\textbf{\TeX{}hackers 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).
24.2 Manipulating and interrogating character tokens

Sets the category code of the \texttt{character} to that indicated in the function name. Depending on the current category code of the \texttt{token} the escape token may also be needed:

\begin{verbatim}
\char_set_catcode_other:N \%
\end{verbatim}

The assignment is local.
Sets the category code of the \texttt{character} which has character code as given by the \texttt{integer expression}. This version can be used to set up characters which cannot otherwise be given (\textit{cf.} the \texttt{N}-type variants). The assignment is local.

\begin{verbatim}
\char_set_catcode:nn {\text{int expr}} {\text{int expr}}
\end{verbatim}

These functions set the category code of the \texttt{character} which has character code as given by the \texttt{integer expression}. The first \texttt{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 \texttt{\char_set_catcode\_\text{type}} should be preferred, but there are cases where these lower-level functions may be useful.

\begin{verbatim}
\char_set_lccode:nn {\text{int expr}} {\text{int expr}}
\end{verbatim}

Sets up the behaviour of the \texttt{character} when found inside \texttt{text_lowercase:n}, such that \texttt{character}_1 will be converted into \texttt{character}_2. The two \texttt{characters} may be specified using an \texttt{integer expression} for the character code concerned. This may include the TeX \texttt{\char} method for converting a single character into its character code:

\begin{verbatim}
\char_set_lccode:nn { 'A } { 'a } \% Standard behaviour
\char_set_lccode:nn { 'A } { 'A + 32 }
\char_set_lccode:nn { 50 } { 60 }
\end{verbatim}

The setting applies within the current TeX group.
Expands to the current lower case code of the \texttt{character} with character code given by the \texttt{integer expression}.

\char_value_lccode:n \char_value_lccode:n \{(integer expression)\}

Displays the current lower case code of the \texttt{character} with character code given by the \texttt{integer expression} on the terminal.

\char_show_value_lccode:n \char_show_value_lccode:n \{(integer expression)\}

\char_set_uccode:nn \char_set_uccode:nn \{(int expr)\} \{(int expr)\}

Sets up the behaviour of the \texttt{character} when found inside \texttt{text_uppercase:n}, such that \texttt{character1} will be converted into \texttt{character2}. The two \texttt{characters} may be specified using an \texttt{integer expression} for the character code concerned. This may include the \TeX{} \texttt{\char} method for converting a single character into its character code:

- \char_set_uccode:nn \{(\char \a) (\char \A)\} % Standard behaviour
- \char_set_uccode:nn \{(\char \A) (\char \A - 32)\}
- \char_set_uccode:nn \{(60) (50)\}

The setting applies within the current \TeX{} group.

Expands to the current upper case code of the \texttt{character} with character code given by the \texttt{integer expression}.

\char_value_uccode:n \char_value_uccode:n \{(integer expression)\}

Displays the current upper case code of the \texttt{character} with character code given by the \texttt{integer expression} on the terminal.

\char_show_value_uccode:n \char_show_value_uccode:n \{(integer expression)\}

\char_set_mathcode:nn \char_set_mathcode:nn \{(int expr)\} \{(int expr)\}

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

Expands to the current math code of the \texttt{character} with character code given by the \texttt{integer expression}.

\char_value_mathcode:n \char_value_mathcode:n \{(integer expression)\}

Displays the current math code of the \texttt{character} with character code given by the \texttt{integer expression} on the terminal.

\char_show_value_mathcode:n \char_show_value_mathcode:n \{(integer expression)\}

\char_set_sfcode:nn \char_set_sfcode:nn \{(int expr)\} \{(int expr)\}

This function sets up the space factor for the \texttt{character}. The \texttt{character} is specified as an \texttt{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 \expands to the current space factor for the \langle character \rangle with character code given by the \langle integer expression \rangle.

\char_show_value_sfcode:n \display the current space factor for the \langle character \rangle with character code given by the \langle integer expression \rangle on the terminal.

\l_char_active_seq
\new: 2012-01-23
\updated: 2015-11-11

Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category \langle active \rangle (catcode 13). Each entry in the sequence consists of a single escaped token, for example \textbackslash -. Active tokens should be added to the sequence when they are defined for general document use.

\l_char_special_seq
\new: 2012-01-23
\updated: 2015-11-11

Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories \langle letter \rangle (catcode 11) or \langle other \rangle (catcode 12). Each entry in the sequence consists of a single escaped token, for example \textbackslash \ for the backslash or \textbackslash \{ for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

### 24.3 Generic tokens

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

\c_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 and should not be used other than for category code tests.

\c_catcode_letter_token
\c_catcode_other_token

A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.
24.4 Converting tokens

\token_to_meaning:N \token_to_meaning:c

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.

\textbf{\TeX{}hackers 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 \texttt{N}-type arguments.

\token_to_str:N \token_to_str:c

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.

\textbf{\TeX{}hackers 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 \texttt{N}-type arguments.

\token_to_catcode:N

Converts the given \token into a number describing its category code. If \token is a control sequence this expands to 16. This can’t detect the categories 0 (escape character), 5 (end of line), 9 (ignored character), 14 (comment character), or 15 (invalid character). Control sequences or active characters let to a token of one of the detectable category codes will yield that category.

24.5 Token conditionals

\token_if_group_begin_p:N \token_if_group_begin_p:N \token_if_group_begin:NTF \token_if_group_begin:TF \token_if_group_end_p:N \token_if_group_end_p:N \token_if_group_end:NTF \token_if_group_end:TF

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 \texttt{N}-type argument.

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 \texttt{N}-type argument.
Tests if \langle\text{token}\rangle has the category code of a math shift token ($ when normal \TeX\ category codes are in force).

Tests if \langle\text{token}\rangle has the category code of an alignment token (& when normal \TeX\ category codes are in force).

Tests if \langle\text{token}\rangle has the category code of a macro parameter token (# when normal \TeX\ category codes are in force).

Tests if \langle\text{token}\rangle has the category code of a superscript token (^ when normal \TeX\ category codes are in force).

Tests if \langle\text{token}\rangle has the category code of a subscript token (_ when normal \TeX\ category codes are in force).

Tests if \langle\text{token}\rangle 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 \texttt{N}-type argument.

Tests if \langle\text{token}\rangle has the category code of a letter token.

Tests if \langle\text{token}\rangle has the category code of an “other” token.

Tests if \langle\text{token}\rangle has the category code of an active character.

Tests if the two \langle\text{tokens}\rangle have the same category code.
\texttt{\textbackslash token\_if\_eq\_charcode\_p:NN} \texttt{(token\_1) \texttt{(token\_2)}}
\texttt{\textbackslash token\_if\_eq\_charcode:NNTF} \texttt{(token\_1) \texttt{(token\_2)}} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the two \texttt{\{tokens\}} have the same character code.

\texttt{\textbackslash token\_if\_eq\_meaning\_p:NN} \texttt{(token\_1) \texttt{(token\_2)}}
\texttt{\textbackslash token\_if\_eq\_meaning:NNTF} \texttt{(token\_1) \texttt{(token\_2)}} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the two \texttt{\{tokens\}} have the same meaning when expanded.

\texttt{\textbackslash token\_if\_macro\_p:N} \texttt{(token)}
\texttt{\textbackslash token\_if\_macro:NTF} \texttt{(token)} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the \texttt{\{token\}} is a \TeX{} macro.

\texttt{\textbackslash token\_if\_cs\_p:N} \texttt{(token)}
\texttt{\textbackslash token\_if\_cs:NTF} \texttt{(token)} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the \texttt{\{token\}} is a control sequence.

\texttt{\textbackslash token\_if\_expandable\_p:N} \texttt{(token)}
\texttt{\textbackslash token\_if\_expandable:NNTF} \texttt{(token)} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the \texttt{\{token\}} is expandable. This test returns \texttt{false} for an undefined token.

\texttt{\textbackslash token\_if\_long\_macro\_p:N} \texttt{(token)}
\texttt{\textbackslash token\_if\_long\_macro:NTF} \texttt{(token)} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the \texttt{\{token\}} is a long macro.

\texttt{\textbackslash token\_if\_protected\_macro\_p:N} \texttt{(token)}
\texttt{\textbackslash token\_if\_protected\_macro:NTF} \texttt{(token)} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the \texttt{\{token\}} is a protected macro: for a macro which is both protected and long this returns \texttt{false}.

\texttt{\textbackslash token\_if\_protected\_long\_macro\_p:N} \texttt{(token)}
\texttt{\textbackslash token\_if\_protected\_long\_macro:NTF} \texttt{(token)} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the \texttt{\{token\}} is a protected long macro.

\texttt{\textbackslash token\_if\_chardef\_p:N} \texttt{(token)}
\texttt{\textbackslash token\_if\_chardef:NTF} \texttt{(token)} \texttt{{\{true code\}}} \texttt{{\{false code\}}}

Tests if the \texttt{\{token\}} is defined to be a chardef.

\textbf{\TeX{}hackers note}: Booleans, boxes and small integer constants are implemented as chardefs.
\texttt{\textbackslash token\_if\_mathchardef\_p:\texttt{N} \textbackslash token\_if\_mathchardef\texttt{NTF} \texttt{\textbackslash token\_if\_mathchardef\p:\texttt{N} \{\texttt{token}\}} \{\{\texttt{true code}\}\} \{\{\texttt{false code}\}\}}

Updated: 2012-01-20

Tests if the \texttt{\texttt{token}} is defined to be a mathchardef.

\texttt{\texttt{\textbackslash token\_if\_font\_selection\_p:\texttt{N} \star \texttt{\textbackslash token\_if\_font\_selection\_p:\texttt{N} \{\texttt{token}\}}} \{\{\texttt{true code}\}\} \{\{\texttt{false code}\}\}}

New: 2020-10-27

Tests if the \texttt{\texttt{token}} is defined to be a font selection command.

\texttt{\texttt{\textbackslash token\_if\_dim\_register\_p:\texttt{N} \star \texttt{\textbackslash token\_if\_dim\_register\_p:\texttt{N} \{\texttt{token}\}}} \{\{\texttt{true code}\}\} \{\{\texttt{false code}\}\}}

Updated: 2012-01-20

Tests if the \texttt{\texttt{token}} is defined to be a dimension register.

\texttt{\texttt{\textbackslash token\_if\_int\_register\_p:\texttt{N} \star \texttt{\textbackslash token\_if\_int\_register\_p:\texttt{N} \{\texttt{token}\}}} \{\{\texttt{true code}\}\} \{\{\texttt{false code}\}\}}

Updated: 2012-01-20

Tests if the \texttt{\texttt{token}} is defined to be a integer register.

**TexHackers note:** Constant integers may be implemented as integer registers, \texttt{\chardef}s, or \texttt{\mathchardef}s depending on their value.

\texttt{\texttt{\textbackslash token\_if\_muskip\_register\_p:\texttt{N} \star \texttt{\textbackslash token\_if\_muskip\_register\_p:\texttt{N} \{\texttt{token}\}}} \{\{\texttt{true code}\}\} \{\{\texttt{false code}\}\}}

New: 2012-02-15

Tests if the \texttt{\texttt{token}} is defined to be a muskip register.

\texttt{\texttt{\textbackslash token\_if\_skip\_register\_p:\texttt{N} \star \texttt{\textbackslash token\_if\_skip\_register\_p:\texttt{N} \{\texttt{token}\}}} \{\{\texttt{true code}\}\} \{\{\texttt{false code}\}\}}

Updated: 2012-01-20

Tests if the \texttt{\texttt{token}} is defined to be a skip register.

\texttt{\texttt{\textbackslash token\_if\_toks\_register\_p:\texttt{N} \star \texttt{\textbackslash token\_if\_toks\_register\_p:\texttt{N} \{\texttt{token}\}}} \{\{\texttt{true code}\}\} \{\{\texttt{false code}\}\}}

Updated: 2012-01-20

Tests if the \texttt{\texttt{token}} is defined to be a toks register (not used by \LaTeX{}X3).

\texttt{\texttt{\textbackslash token\_if\_primitive\_p:\texttt{N} \star \texttt{\textbackslash token\_if\_primitive\_p:\texttt{N} \{\texttt{token}\}}} \{\{\texttt{true code}\}\} \{\{\texttt{false code}\}\}}

Updated: 2020-09-11

Tests if the \texttt{\texttt{token}} is an engine primitive. In Lua\TeX{} this includes primitive-like commands defined using \texttt{token.set_lua}.
This function compares the \textit{test token} in turn with each of the \textit{token cases}. If the two are equal (as described for \texttt{token_if_eq_catcode:NNTF}, \texttt{token_if_eq_charcode:NNTF} and \texttt{token_if_eq_meaning:NNTF}, respectively) then the associated \textit{code} is left in the input stream and other cases are discarded. If any of the cases are matched, the \texttt{(true code)} is also inserted into the input stream (after the code for the appropriate case), while if none match then the \texttt{(false code)} is inserted. The functions \texttt{token_case_catcode:Nn}, \texttt{token_case_charcode:Nn}, and \texttt{token_case_meaning:Nn}, which do nothing if there is no match, are also available.

\section*{24.6 Peeking ahead at the next token}

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic \texttt{peek_after:Nw} is provided along with a family of predefined tests for common cases. As peeking ahead does \textit{not} skip spaces the predefined tests include both a space-respecting and space-skipping version. In addition, using \texttt{peek_analysis_map_inline:n}, one can map through the following tokens in the input stream and repeatedly perform some tests.

\begin{verbatim}
\peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw \peek_after:Nw
\end{verbatim}

Locally sets the test variable \texttt{l_peek_token} equal to \texttt{token} (as an implicit token, \textit{not} as a token list), and then expands the \texttt{function}. The \texttt{token} remains in the input stream as the next item after the \texttt{function}. The \texttt{token} here may be \texttt{\textasciitilde}, \texttt{\{ or \} (assuming normal \TeX category codes), \textit{i.e.} it is not necessarily the next argument which would be grabbed by a normal function.

Globally sets the test variable \texttt{g_peek_token} equal to \texttt{token} (as an implicit token, \textit{not} as a token list), and then expands the \texttt{function}. The \texttt{token} remains in the input stream as the next item after the \texttt{function}. The \texttt{token} here may be \texttt{\textasciitilde}, \texttt{\{ or \} (assuming normal \TeX category codes), \textit{i.e.} it is not necessarily the next argument which would be grabbed by a normal function.

\texttt{l_peek_token} Token set by \texttt{peek_after:Nw} and available for testing as described above.

\texttt{g_peek_token} Token set by \texttt{peek_gafter:Nw} and available for testing as described above.
\texttt{\textbackslash peek\_catcode:NNTF} {\texttt{\textbackslash peek\_catcode:NTF}} \{\texttt{test token}\} {\{\texttt{true code}\}} {\{\texttt{false code}\}}

Tests if the next \{\texttt{token}\} in the input stream has the same category code as the \{\texttt{test token}\} (as defined by the test \texttt{\token\_if\_eq\_catcode:NNTF}). Spaces are respected by the test and the \{\texttt{token}\} is left in the input stream after the \{\texttt{true code}\} or \{\texttt{false code}\} (as appropriate to the result of the test).

\texttt{\textbackslash peek\_catcode\_remove:NTF} {\texttt{\textbackslash peek\_catcode\_remove:NTF}} \{\texttt{test token}\} {\{\texttt{true code}\}} {\{\texttt{false code}\}}

Tests if the next \{\texttt{token}\} in the input stream has the same category code as the \{\texttt{test token}\} (as defined by the test \texttt{\token\_if\_eq\_catcode:NTF}). Spaces are respected by the test and the \{\texttt{token}\} is removed from the input stream if the test is true. The function then places either the \{\texttt{true code}\} or \{\texttt{false code}\} in the input stream (as appropriate to the result of the test).

\texttt{\textbackslash peek\_charcode:NNTF} {\texttt{\textbackslash peek\_charcode:NTF}} \{\texttt{test token}\} {\{\texttt{true code}\}} {\{\texttt{false code}\}}

Tests if the next \{\texttt{token}\} in the input stream has the same character code as the \{\texttt{test token}\} (as defined by the test \texttt{\token\_if\_eq\_charcode:NNTF}). Spaces are respected by the test and the \{\texttt{token}\} is left in the input stream after the \{\texttt{true code}\} or \{\texttt{false code}\} (as appropriate to the result of the test).

\texttt{\textbackslash peek\_charcode\_remove:NTF} {\texttt{\textbackslash peek\_charcode\_remove:NTF}} \{\texttt{test token}\} {\{\texttt{true code}\}} {\{\texttt{false code}\}}

Tests if the next \{\texttt{token}\} in the input stream has the same character code as the \{\texttt{test token}\} (as defined by the test \texttt{\token\_if\_eq\_charcode:NTF}). Spaces are respected by the test and the \{\texttt{token}\} is removed from the input stream if the test is true. The function then places either the \{\texttt{true code}\} or \{\texttt{false code}\} in the input stream (as appropriate to the result of the test).

\texttt{\textbackslash peek\_meaning:NNTF} {\texttt{\textbackslash peek\_meaning:NTF}} \{\texttt{test token}\} {\{\texttt{true code}\}} {\{\texttt{false code}\}}

Tests if the next \{\texttt{token}\} in the input stream has the same meaning as the \{\texttt{test token}\} (as defined by the test \texttt{\token\_if\_eq\_meaning:NNTF}). Spaces are respected by the test and the \{\texttt{token}\} is left in the input stream after the \{\texttt{true code}\} or \{\texttt{false code}\} (as appropriate to the result of the test).

\texttt{\textbackslash peek\_meaning\_remove:NTF} {\texttt{\textbackslash peek\_meaning\_remove:NTF}} \{\texttt{test token}\} {\{\texttt{true code}\}} {\{\texttt{false code}\}}

Tests if the next \{\texttt{token}\} in the input stream has the same meaning as the \{\texttt{test token}\} (as defined by the test \texttt{\token\_if\_eq\_meaning:NTF}). Spaces are respected by the test and the \{\texttt{token}\} is removed from the input stream if the test is true. The function then places either the \{\texttt{true code}\} or \{\texttt{false code}\} in the input stream (as appropriate to the result of the test).

\texttt{\textbackslash peek\_remove\_spaces:n} \{\texttt{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 \{\texttt{code}\} will be inserted into the input stream. Typically this will contain a \texttt{peek} operation, but this is not required.

Updated: 2012-12-20

Updated: 2011-07-02

Updated: 2018-10-01
\peek_remove_filler:n \peek_remove_filler:n \langle \{\text{code}\} \rangle

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to \texttt{\scan_stop}: . If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using \texttt{I}-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 \texttt{\langle \{\text{code}\} \rangle} will be inserted into the input stream. Typically this will contain a \texttt{\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 \texttt{\everypar}, except that any non-expandable token cleanly ends the \texttt{\langle \text{filler} \rangle} (i.e. it does not lead to a \TeX error).

In contrast to \TeX’s filler removal, a construct \texttt{\exp_not:N \foo} will be treated in the same way as \texttt{\foo}.

\peek_N_type:TF \peek_N_type:TF \{\langle \{\text{true code}\} \rangle\} \{\langle \{\text{false code}\} \rangle\}

Tests if the next \langle \text{token} \rangle in the input stream can be safely grabbed as an \texttt{N}-type argument. The test is \langle \text{false} \rangle if the next \langle \text{token} \rangle 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 L\TeX3) and \langle \text{true} \rangle in all other cases. Note that a \langle \text{true} \rangle result ensures that the next \langle \text{token} \rangle is a valid \texttt{N}-type argument. However, if the next \langle \text{token} \rangle is for instance \texttt{\c_space_token}, the test takes the \langle \text{false} \rangle branch, even though the next \langle \text{token} \rangle is in fact a valid \texttt{N}-type argument. The \langle \text{token} \rangle is left in the input stream after the \langle \text{true code} \rangle or \langle \text{false code} \rangle (as appropriate to the result of the test).
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_analysis_map_break: \peek_analysis_map_inline:n
\peek_analysis_map_break:n { ... \peek_analysis_map_break:n {⟨code⟩} }

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:nTF \peek_regex:NTF \peek_regex:nTF \peek_regex:NTF

New: 2020-12-03

Tests if the \langle tokens \rangle that follow in the input stream match the \langle regular expression \rangle. Any \langle tokens \rangle that have been read are left in the input stream after the \langle true code \rangle or \langle false code \rangle (as appropriate to the result of the test). See \l3regex for documentation of the syntax of regular expressions. The \langle regular expression \rangle is implicitly anchored at the start, so for instance \peek_regex:nTF \{ a \} is essentially equivalent to \peek_charcode:nTF a.

\textbf{\texttt{\TeX}hackers 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:nTF \peek_regex_remove_once:NTF \peek_regex_remove_once:nTF \peek_regex_remove_once:NTF

New: 2020-12-03

Tests if the \langle tokens \rangle that follow in the input stream match the \langle regular expression \rangle. If the test is true, the \langle tokens \rangle are removed from the input stream and the \langle true code \rangle is inserted, while if the test is false, the \langle false code \rangle is inserted followed by the \langle tokens \rangle that were originally in the input stream. See \l3regex for documentation of the syntax of regular expressions. The \langle regular expression \rangle is implicitly anchored at the start, so for instance \peek_regex_remove_once:nTF \{ a \} is essentially equivalent to \peek_charcode_remove:nTF a.

\textbf{\texttt{\TeX}hackers 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.
If the \{tokens\} that follow in the input stream match the \{regex\}, replaces them according to the \{replacement\} as for \regex_replace_once:nn, 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 \texttt{l3regex} for documentation of the syntax of regular expressions and of the \{replacement\}: for instance \texttt{	extbackslash{}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:nn, no error arises if the \{replacement\} leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

\textbf{\textit{TeX}hackers note:} Implicit character tokens are correctly considered by \texttt{\peek_regex_replace_once:nnTF} as control sequences, while functions that inspect individual tokens (for instance \texttt{\peek_charcode:NNTF}) only take into account their meaning.

### 24.7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on \TeX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, \texttt{\use:n} is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for Lua\TeX{} and Xe\TeX{} and less for other engines) and category code \texttt{13}.
- A character token, characterized by its character code and category code (one of \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{10}, \texttt{11} or \texttt{12} whose meaning is described below).

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

- Expanding \texttt{\the\font} results in a token that looks identical to the command that was used to select the current font (such as \texttt{\textsc{tenrm}}) but it differs from it in shape.
- A “frozen” \texttt{\relax}, which differs from the primitive in shape (but has the same meaning), is inserted when the closing \texttt{\fi} of a conditional is encountered before the conditional is evaluated.
- Expanding \texttt{\noexpand \{token\}} (when the \{token\} is expandable) results in an internal token, displayed (temporarily) as \texttt{\notexpanded:\{token\}}, whose shape coincides with the \{token\} and whose meaning differs from \texttt{\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 \textsc{LuaT}eX, there is also the strange case of “bytes” \texttt{\\110xy} where \textit{x, y}
are any two lowercase hexadecimal digits, so that the hexadecimal number ranges
from \texttt{\*110000 = 1114112} to \texttt{\*1100ff = 1114367}. These are used to output individual
bytes to files, rather than \textsc{UTF-8}. For the purposes of token comparisons
they behave like non-expandable primitive control sequences (\textit{not characters}) whose
\meaning is \texttt{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 \texttt{\token_to_str:N} or
\texttt{\token_to_meaning:N} to these tokens is unsafe. Unfortunately, they don’t seem
to be detectable safely by any means except perhaps \textsc{lua} code.

The meaning of a (non-active) character token is fixed by its category code (and
category code) and cannot be changed. We call these tokens \texttt{explicit} character tokens.
Category codes that a character token can have are listed below by giving a sample
output of the \textsc{La}TeX primitive \texttt{\meaning}, together with their \texttt{\LaTeX3} names and most
common example:

1 begin-group character (\texttt{group_begin}, often \{),
2 end-group character (\texttt{group_end}, often \}),
3 math shift character (\texttt{math_toggle}, often $),
4 alignment tab character (\texttt{alignment}, often \&),
6 macro parameter character (\texttt{parameter}, often \#),
7 superscript character (\texttt{math_superscript}, often ^),
8 subscript character (\texttt{math_subscript}, often _),
10 blank space (\texttt{space}, often character code 32),
11 the letter (\texttt{letter}, such as A),
12 the character (\texttt{other}, such as 0).

Category code 13 (active) is discussed below. Input characters can also have sev-
eral other category codes which do not lead to character tokens for later processing:
0 (escape), 5 (\texttt{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 \texttt{implicit}
character tokens. The meaning is otherwise in the following list:

• a macro, used in \texttt{\LaTeX3} for most functions and some variables (\texttt{tl, fp, seq, \ldots}),
• a primitive such as \texttt{\def} or \texttt{\topmark}, used in \texttt{\LaTeX3} for some functions,
• a register such as \texttt{\count123}, used in \texttt{\LaTeX3} for the implementation of some vari-
ables (\texttt{int, dim, \ldots}),

211
• a constant integer such as \texttt{\textbackslash char"56} or \texttt{\textbackslash mathchar"121},
• a font selection command,
• undefined.

Macros can be \texttt{\textbackslash protected} or not, \texttt{\textbackslash long} or not (the opposite of what \LaTeX{}3 calls \texttt{\textbackslash nopar}), and \texttt{\textbackslash outer} or not (unused in \LaTeX{}3). Their \texttt{\textbackslash meaning} takes the form

\begin{verbatim}
  \langle prefix⟩ macro:(argument)->⟨replacement⟩
\end{verbatim}

where \texttt{⟨prefix⟩} is among \texttt{\textbackslash protected\textbackslash long\textbackslash outer}, \texttt{⟨argument⟩} describes parameters that the macro expects, such as \#1\#2\#3, and \texttt{⟨replacement⟩} describes how the parameters are manipulated, such as \texttt{\textbackslash int_eval:n{#2+#1*#3}}.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then \TeX{} scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature :N.

When a macro takes a delimited argument \TeX{} scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are not ignored in this case (and they prevent brace-stripping).
Chapter 25

The l3prop module

Property lists

expl3 implements a \textit{property list} data type, which contain an unordered list of entries each of which consists of a \textit{key} and an associated \textit{value}. The \textit{key} and \textit{value} may both be any \textit{balanced text}, the \textit{key} is processed using \texttt{tl_to_str:n}, meaning that category codes are ignored. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique \textit{key}; if an entry is added to a property list which already contains the \textit{key} then the new entry overwrites the existing one. The \textit{keys} are compared on a string basis, using the same method as \texttt{str_if_-eq:nTF}.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of input parsed by the \l3keys module.

25.1 Creating and initialising property lists

\begin{verbatim}
\prop_new:N \prop_new:c \prop_clear:N \prop_clear:c \prop_gclear:N \prop_gclear:c
\end{verbatim}

\begin{itemize}
  \item \texttt{\prop_new:N \prop_new:c} \textit{property list} \textbf{creates a new \textit{property list} or raises an error if the name is already taken. The declaration is global. The \textit{property list} initially contains no entries.}
  \item \texttt{\prop_clear:N \prop_clear:c \prop_gclear:N \prop_gclear:c} \textit{clears all entries from the \textit{property list}.}
  \item \texttt{\prop_gclear_new:N \prop_gclear_new:c} \textit{ensures that the \textit{property list} exists globally by applying \texttt{\prop_new:N} if necessary, then applies \texttt{\prop_(g)clear:N} to leave the list empty.}
\end{itemize}
\prop_set_eq:NN \prop_set_eq:cn \prop_gset_eq:NN \prop_gset_eq:cn

Sets the content of \emph{property list}_1 to equal to that of \emph{property list}_2.

\prop_set_from_keyval:Nn \prop_set_from_keyval:cn \prop_gset_from_keyval:Nn \prop_gset_from_keyval:cn

\{ 
(\texttt{key1}) = (\texttt{value1}), 
(\texttt{key2}) = (\texttt{value2}), ...
\}

Sets \emph{property list} to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept.

Spaces are trimmed around every \emph{key} and every \emph{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 \emph{key} and the \emph{value} to contain spaces, commas or equal signs. The \emph{key} is then processed by \texttt{tl_to_str:n}. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

Notice that in contrast to most keyval lists (\emph{e.g.} those in \texttt{l3keys}), each key here must be followed with an = sign.

\prop_const_from_keyval:Nn \prop_const_from_keyval:cn

\{ 
(\texttt{key1}) = (\texttt{value1}), 
(\texttt{key2}) = (\texttt{value2}), ...
\}

Creates a new constant \emph{property list} or raises an error if the name is already taken. The \emph{property list} is set globally to contain key–value pairs given in the second argument, processed in the way described for \texttt{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.

Notice that in contrast to most keyval lists (\emph{e.g.} those in \texttt{l3keys}), each key here must be followed with an = sign.
25.2 Adding and updating property list entries

\prop_put:Nnn \prop_put: \langle property list \rangle \{ (key) \} \{ (value) \}
\prop_put_if_new:Nnn \prop_put_if_new: \langle property list \rangle \{ (key) \} \{ (value) \}
\prop_gput_if_new:Nnn \prop_gput_if_new: \langle property list \rangle \{ (key) \} \{ (value) \}

Adds an entry to the \langle property list \rangle which may be accessed using the \langle key \rangle and which has \langle value \rangle. If the \langle key \rangle is already present in the \langle property list \rangle, the existing entry is overwritten by the new \langle value \rangle. Both the \langle key \rangle and \langle value \rangle may contain any \langle balanced text \rangle. The \langle key \rangle is stored after processing with \tl_to_str:n, meaning that category codes are ignored.

\prop_concat:NNN \prop_gconcat:NNN \prop_concat:ccc \prop_gconcat:ccc

If the \langle key \rangle is present in the \langle property list \rangle then no action is taken. Otherwise, a new entry is added as described for \prop_put:Nnn.

\prop_concat:NNN \prop_gconcat:NNN \langle property list_1 \rangle \langle property list_2 \rangle \langle property list_3 \rangle
\prop_concat:ccc \prop_gconcat:ccc

Combines the key–value pairs of \langle property list_2 \rangle and \langle property list_3 \rangle, and saves the result in \langle property list_1 \rangle. If a key appears in both \langle property list_2 \rangle and \langle property list_3 \rangle then the last value, namely the value in \langle property list_3 \rangle is kept.
\prop_put_from_keyval:Nn \prop_put_from_keyval:cn \prop_gput_from_keyval:Nn \prop_gput_from_keyval:cn

Prop: 2021-05-16
Updated: 2021-11-07

\prop_put_from_keyval:Nn \prop_put_from_keyval:cn \prop_gput_from_keyval:Nn \prop_gput_from_keyval:cn

Updates the \textit{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 \textit{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 \textit{property list} with the temporary variable using \prop_concat:NNN. In particular, the \textit{keys} and \textit{values} are space-trimmed and unbraced as described in \prop_set_from_keyval:Nn. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

25.3 Recovering values from property lists

\prop_get:NnN \prop_get:(NVN|NvN|NeN|NoN|cnN|cVN|cvN|ceN|coN|cnc)

Updated: 2011-08-28

Recoveres the \textit{value} stored with \textit{key} from the \textit{property list}, and places this in the \textit{token list variable}. If the \textit{key} is not found in the \textit{property list} then the \textit{token list variable} is set to the special marker \q_no_value. The \textit{token list variable} is set within the current \TeX{} group. See also \prop_get:NnNTF.

\prop_pop:NnN \prop_pop:(NVN|NoN|cnN|cVN|cvN|ceN|coN)

Updated: 2011-08-18

Recoveres the \textit{value} stored with \textit{key} from the \textit{property list}, and places this in the \textit{token list variable}. If the \textit{key} is not found in the \textit{property list} then the \textit{token list variable} is set to the special marker \q_no_value. The \textit{key} and \textit{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|cvN|ceN|coN)

Updated: 2011-08-18

Recoveres the \textit{value} stored with \textit{key} from the \textit{property list}, and places this in the \textit{token list variable}. If the \textit{key} is not found in the \textit{property list} then the \textit{token list variable} is set to the special marker \q_no_value. The \textit{key} and \textit{value} are then deleted from the property list. The \textit{property list} is modified globally, while the assignment of the \textit{token list variable} is local. See also \prop_gpop:NnNTF.
Expands to the \textit{value} corresponding to the \textit{key} in the \textit{property list}. If the \textit{key} is missing, this has an empty expansion.

\textbf{\LaTeX}hacker note: This function is slower than the non-expandable analogue \texttt{\prop_get:Nn}. The result is returned within the \texttt{\unexpanded} primitive \texttt{(\exp_not:n)}, which means that the \textit{value} does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion.

\begin{verbatim}
\prop_count:N \langle property list \rangle
\end{verbatim}
Leaves the number of key–value pairs in the \textit{property list} in the input stream as an \textit{integer denotation}.

\begin{verbatim}
\prop_to_keyval:N \langle property list \rangle
\end{verbatim}
Expands to the \textit{property list} in a key–value notation. Keep in mind that a \textit{property list} is unordered, while key–value interfaces don’t necessarily are, so this can’t be used for arbitrary interfaces.

\textbf{\LaTeX}hacker note: The result is returned within the \texttt{\unexpanded} primitive \texttt{(\exp_not:n)}, which means that the key–value list does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion. It also needs exactly two steps of expansion.

### 25.4 Modifying property lists

\begin{verbatim}
\prop_remove:Nn \langle property list \rangle \{\langle key \rangle\}
\prop_remove:NN \langle property list \rangle \{\langle key \rangle\}
\prop_gremove:Nn \langle property list \rangle \{\langle key \rangle\}
\prop_gremove:NN \langle property list \rangle \{\langle key \rangle\}
\end{verbatim}
Removes the entry listed under \textit{key} from the \textit{property list}. If the \textit{key} is not found in the \textit{property list} no change occurs, \textit{i.e.} there is no need to test for the existence of a key before deleting it.

### 25.5 Property list conditionals

\begin{verbatim}
\prop_if_exist_p:N \langle property list \rangle
\prop_if_exist_p:c \langle property list \rangle
\prop_if_exist:NTF \langle property list \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}
\prop_if_exist:CF \langle property list \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}
\end{verbatim}
Tests whether the \textit{property list} is currently defined. This does not check that the \textit{property list} really is a property list variable.
Tests if the \langle property list \rangle is empty (containing no entries).

\prop_if_empty_p:N * \prop_if_empty_p:NTF \langle property list \rangle \{\{true code\}\} \{\{false code\}\}
\prop_if_empty:p_c * \prop_if_empty:N TF * \prop_if_empty:c TF *

Tests if the \langle key \rangle is present in the \langle property list \rangle, making the comparison using the method described by \str_if_eq:nnTF.

\TeXhacksnote{This function iterates through every key–value pair in the \langle property list \rangle and is therefore slower than using the non-expandable \prop_get:NnNTF.}

25.6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated value. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

\prop_get:NnTF \langle property list \rangle \{\langle key \rangle\} \{\langle true code \rangle\} \{\langle false code \rangle\}
\prop_get:NnN \prop_get: (NVN|NeN|NoN|cnN|cVN|ceN|coN|cnV|cvN|cvO|ceO|coO|cnc)

Updated: 2012-05-19

If the \langle key \rangle is not present in the \langle property list \rangle, leaves the \langle false code \rangle in the input stream. The value of the \langle token list variable \rangle is not defined in this case and should not be relied upon. If the \langle key \rangle is present in the \langle property list \rangle, stores the corresponding \langle value \rangle in the \langle token list variable \rangle without removing it from the \langle property list \rangle, then leaves the \langle true code \rangle in the input stream. The \langle token list variable \rangle is assigned locally.

\prop_pop:NnTF \prop_pop: (NVN|NoN|cnV|cnO|ceO|coO|cnc)

Rev: 2011-09-18
Updated: 2012-05-19

If the \langle key \rangle is not present in the \langle property list \rangle, leaves the \langle false code \rangle in the input stream. The value of the \langle token list variable \rangle is not defined in this case and should not be relied upon. If the \langle key \rangle is present in the \langle property list \rangle, pops the corresponding \langle value \rangle in the \langle token list variable \rangle, i.e., removes the item from the \langle property list \rangle. Both the \langle property list \rangle and the \langle token list variable \rangle are assigned locally.
If the \textit{key} is not present in the \textit{property list}, leaves the \textit{false code} in the input stream. The value of the \textit{token list variable} is not defined in this case and should not be relied upon. If the \textit{key} is present in the \textit{property list}, pops the corresponding \textit{value} in the \textit{token list variable}, i.e. removes the item from the \textit{property list}. The \textit{property list} is modified globally, while the \textit{token list variable} is assigned locally.

25.7 Mapping over property lists

All mappings are done at the current group level, \textit{i.e.} any local assignments made by the \textit{function} or \textit{code} discussed below remain in effect after the loop.

\begin{verbatim}
\prop_map_function:NN \prop_map_function:cN \Updated: 2013-01-08
\end{verbatim}

\begin{verbatim}
\prop_map_inline:NN \prop_map_inline:cn \Updated: 2013-01-08
\end{verbatim}

\begin{verbatim}
\prop_map_tokens:NN \prop_map_tokens:cn \Updated: 2013-01-08
\end{verbatim}

Analogue of \texttt{\prop_map_function:NN} which maps several tokens instead of a single function. The \texttt{\prop_map_tokens:NN} \texttt{\l_my_prop \str_if_eq:nnT \{mykey\}} expands to the value corresponding to \texttt{mykey}: for each pair in \texttt{\l_my_prop} the function \texttt{\str_if_eq:nnT} receives \texttt{mykey}, the \textit{key} and the \textit{value} as its three arguments. For that specific task, \texttt{\prop_item:Nn} is faster.
\prop_map_break: \\
Updated: 2012-06-29

\prop_map_break:

Used to terminate a \prop_map... function before all entries in the ⟨property list⟩ have been processed. This 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
\}
\%
\Do
something
useful
\}
\}

Use outside of a \prop_map... scenario leads to low level \TeX errors.

\TeXhacks{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 \langle\text{code}\rangle

\prop_map_break:n \langle\text{code}\rangle

\prop_map_break:n \langle\text{code}\rangle

\prop_map_break:n \langle\text{code}\rangle

Updated: 2012-06-29

Used to terminate a \prop_map... function before all entries in the ⟨property list⟩ have been processed, inserting the ⟨code⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

\prop_map_inline:Nn \l_my_prop
\{
\str_if_eq:nnTF { #1 } { bingo }
\{
\prop_map_break:n { \langle\text{code}\rangle } 
\%
\Do
something
useful
\}
\%
\Do
something
useful
\}
\}

Use outside of a \prop_map... scenario leads to low level \TeX errors.

\TeXhacks{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.}

\section{25.8 Viewing property lists}

\prop_show:N \langle\text{property list}\rangle

\prop_show:N \langle\text{property list}\rangle

\prop_show:N \langle\text{property list}\rangle

Updated: 2021-04-29

Displays the entries in the ⟨property list⟩ in the terminal.
25.9 Scratch property lists

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

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

25.10 Constants

\c_empty_prop A permanently-empty property list used for internal comparisons.
Chapter 26

The \texttt{l3skip} module

Dimensions and skips

\LaTeX{}3 provides two general length variables: \texttt{dim} and \texttt{skip}. Lengths stored as \texttt{dim} variables have a fixed length, whereas \texttt{skip} lengths have a rubber (stretch/shrink) component. In addition, the \texttt{muskip} type is available for use in math mode: this is a special form of \texttt{skip} where the lengths involved are determined by the current math font (in \texttt{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 \textit{dimension expressions} (\texttt{"\{dim expr\}}) or \textit{skip expressions} (\texttt{"\{skip expr\}}) as arguments.

### 26.1 Creating and initialising \texttt{dim} variables

\begin{verbatim}
\dim_new:N \dim_new:c \dim_const:Nn \dim_const:cn \dim_zero:N \dim_zero:c \dim_gzero:N \dim_gzero:c
\end{verbatim}

- \texttt{\dim_new:N \{dimension\}} or raises an error if the name is already taken. The declaration is global. The \texttt{\{dimension\}} is initially equal to 0 pt.
- \texttt{\dim_new:c \{dimension\}}
- \texttt{\dim_const:Nn \texttt{\{dimension\}} \{\texttt{\{dim expr\}\}}}
- \texttt{\dim_const:cn \texttt{\{dimension\}} \{\texttt{\{dim expr\}\}}}
- \texttt{\dim_zero:N \texttt{\{dimension\}}}
- \texttt{\dim_zero:c \texttt{\{dimension\}}}
- \texttt{\dim_gzero:N \texttt{\{dimension\}}}
- \texttt{\dim_gzero:c \texttt{\{dimension\}}}

\texttt{\dim_zero_new:N \texttt{\{dimension\}}}

Ensures that the \texttt{\{dimension\}} exists globally by applying \texttt{\dim_new:N} if necessary, then applies \texttt{\dim_\{g\}zero:N} to leave the \texttt{\{dimension\}} set to zero.
26.2 Setting dim variables

\dim_add:Nn \dim_add:N \{\text{dim expr}\}
\dim_add:cn
\dim_gadd:Nn \dim_gadd:cn

Adds the result of the \textit{dim expr} to the current content of the \textit{dim}.

Updated: 2011-10-22

\dim_set:Nn \dim_set:N \{\text{dim expr}\}
\dim_set:cn
\dim_gset:Nn \dim_gset:cn

Sets \textit{dim} to the value of \textit{dim expr}, which must evaluate to a length with units.

Updated: 2011-10-22

\dim_set_eq:NN \dim_set_eq: \{\text{dim1}\} \{\text{dim2}\}
\dim_set_eq:(cN|Nc|cc)
\dim_gset_eq:NN \dim_gset_eq: \{\text{dim1}\} \{\text{dim2}\}
\dim_gset_eq:(cN|Nc|cc)

Sets the content of \textit{dim1} equal to that of \textit{dim2}.

\dim_sub:Nn \dim_sub:N \{\text{dim expr}\}
\dim_sub:cn
\dim_gsub:Nn \dim_gsub:cn

Subtracts the result of the \textit{dim expr} from the current content of the \textit{dim}.

Updated: 2011-10-22

26.3 Utilities for dimension calculations

\dim_abs:n \dim_abs:n \{\text{dim expr}\}

Converts the \textit{dim expr} to its absolute value, leaving the result in the input stream as a \textit{dimension denotation}.

Updated: 2012-09-26

\dim_max:nn \dim_max:nn \{\text{dim1}\} \{\text{dim2}\}
\dim_min:nn \dim_min:nn \{\text{dim1}\} \{\text{dim2}\}

Evaluates the two \textit{dim exprs} and leaves either the maximum or minimum value in the input stream as appropriate, as a \textit{dimension denotation}.

Updated: 2012-09-26

Rev: 2012-03-03
\texttt{\textbackslash dim\_ratio:nn} \textasciitilde \texttt{\textbackslash dim\_ratio:nn \{(dim expr_1\}\) \{(dim expr_2\}\)}

\texttt{\textbackslash dim\_ratio:nn} parses the two (\textit{dim exps}) and converts the ratio of the two to a form suitable for use inside a (\textit{dim expr}). This ratio is then left in the input stream, allowing syntax such as

\begin{verbatim}
\def\l_my_dim{10 pt * \dim\_ratio:nn {5 pt} {10 pt}}
\end{verbatim}

The output of \texttt{\textbackslash dim\_ratio:nn} on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

\begin{verbatim}
\tl_set:N \l_my_tl { \dim\_ratio:nn {5 pt} {10 pt}}
\tl_show:N \l_my_tl
\end{verbatim}

displays \texttt{327680/655360} on the terminal.

### 26.4 Dimension expression conditionals

\texttt{\textbackslash dim\_compare:p:nNn} \textasciitilde \texttt{\textbackslash dim\_compare:p:nNn \{(dim expr_1\}\) \{\textit{relation}\) \{(dim expr_2\}\)}}

\texttt{\textbackslash dim\_compare:p:nNnTF} \texttt{\{(dim expr_1\}\) \{\textit{relation}\) \{(dim expr_2\}\)}}

\texttt{\{(true code\}\) \{(false code\\)}}

This function first evaluates each of the (\textit{dim exps}) as described for \texttt{\textbackslash dim\_eval:n}. The two results are then compared using the (\textit{relation}):

\begin{verbatim}
Equal =
Greater than >
Less than <
\end{verbatim}

This function is less flexible than \texttt{\textbackslash dim\_compare:nTF} but around 5 times faster.
\dim_compare_p:n \dim_compare:nTF

\dim_compare_p:n
{ (dim expr_1) (relation_1)
  ...
  (dim expr_N) (relation_N)
  (dim expr_{N+1})
}
\dim_compare:nTF
{ (dim expr_1) (relation_1)
  ...
  (dim expr_N) (relation_N)
  (dim expr_{N+1})
}
{(true code)} {(false code)}

This function evaluates the \textit{dim exprs} as described for \texttt{\dim_eval:n} and compares consecutive result using the corresponding \textit{relation}, namely it compares \texttt{(dim expr_1)} and \texttt{(dim expr_2)} using the \texttt{(relation_1)}, then \texttt{(dim expr_2)} and \texttt{(dim expr_3)} using the \texttt{(relation_2)}, until finally comparing \texttt{(dim expr_N)} and \texttt{(dim expr_{N+1})} using the \texttt{(relation_N)}. The test yields true if all comparisons are true. Each \texttt{(dim expr)} is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is false, then no other \texttt{(dim expr)} is evaluated and no other comparison is performed. The \texttt{(relations)} can be any of the following:

\begin{verbatim}
  Equal        = or ==
  Greater than or equal to  >=
  Greater than         >
  Less than or equal to  <=
  Less than            <
  Not equal          !=
\end{verbatim}

This function is more flexible than \texttt{\dim_compare:nNnTF} but around 5 times slower.
This function evaluates the \textit{test dim expr} and compares this in turn to each of the \textit{dim expr cases}. If the two are equal then the associated \textit{code} is left in the input stream and other cases are discarded. If any of the cases are matched, the \textit{true code} is also inserted into the input stream (after the code for the appropriate case), while if none match then the \textit{false code} is inserted. The function \texttt{\texttt{dim_case:nn}}, which does nothing if there is no match, is also available. For example

\begin{verbatim}
\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! }
\end{verbatim}

leaves “Medium” in the input stream.

\newpage

26.5 Dimension expression loops

\begin{verbatim}
\dim_do_until:nNnn { \textit{dim expr}} \textit{relation} { \textit{dim expr2}} \textit{code}
\end{verbatim}

Places the \textit{code} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{dim exprs} as described for \texttt{\texttt{dim_compare:nNnTF}}. If the test is \textit{false} then the \textit{code} is inserted into the input stream again and a loop occurs until the \textit{relation} is \textit{true}.

\begin{verbatim}
\dim_do_while:nNnn { \textit{dim expr}} \textit{relation} { \textit{dim expr2}} \textit{code}
\end{verbatim}

Places the \textit{code} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{dim exprs} as described for \texttt{\texttt{dim_compare:nNnTF}}. If the test is \textit{true} then the \textit{code} is inserted into the input stream again and a loop occurs until the \textit{relation} is \textit{false}.

\begin{verbatim}
\dim_until_do:nNnn { \textit{dim expr}} \textit{relation} { \textit{dim expr2}} \textit{code}
\end{verbatim}

Evaluates the relationship between the two \textit{dim exprs} as described for \texttt{\texttt{dim_compare:nNnTF}}, and then places the \textit{code} in the input stream if the \textit{relation} is \textit{false}. After the \textit{code} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \textit{true}.
\texttt{\dimwhile_do:nNnn} \hspace{1em} \texttt{\dimwhile_do:nNnn \{\dimexpr\} \{\text{relation}\} \{\dimexpr2\} \{\text{code}\}}

Evaluates the relationship between the two \texttt{\dimexpr}s as described for \texttt{\dim_compare:nTF}, and then places the \texttt{\dimexpr} in the input stream if the \texttt{\text{relation}} is \texttt{true}. After the \texttt{\text{code}} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.

\texttt{\dimdo_until:nn} \hspace{1em} \texttt{\dimdo_until:nn \{\text{dimension relation}\} \{\text{code}\}}

Placed the \texttt{\text{code}} in the input stream for \TeX{} to process, and then evaluates the \texttt{\text{dimension relation}} as described for \texttt{\dim_compare:nTF}. If the test is \texttt{false} then the \texttt{\text{code}} is inserted into the input stream again and a loop occurs until the \texttt{\text{relation}} is \texttt{true}.

\texttt{\dimwhile_do:nn} \hspace{1em} \texttt{\dimwhile_do:nn \{\text{dimension relation}\} \{\text{code}\}}

Placed the \texttt{\text{code}} in the input stream for \TeX{} to process, and then evaluates the \texttt{\text{dimension relation}} as described for \texttt{\dim_compare:nTF}. If the test is \texttt{true} then the \texttt{\text{code}} is inserted into the input stream again and a loop occurs until the \texttt{\text{relation}} is \texttt{false}.

\texttt{\dimuntil_do:nn} \hspace{1em} \texttt{\dimuntil_do:nn \{\text{dimension relation}\} \{\text{code}\}}

Evaluates the \texttt{\text{dimension relation}} as described for \texttt{\dim_compare:nTF}, and then places the \texttt{\text{code}} in the input stream if the \texttt{\text{relation}} is \texttt{false}. After the \texttt{\text{code}} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\dimwhile_do:nn} \hspace{1em} \texttt{\dimwhile_do:nn \{\text{dimension relation}\} \{\text{code}\}}

Evaluates the \texttt{\text{dimension relation}} as described for \texttt{\dim_compare:nTF}, and then places the \texttt{\text{code}} in the input stream if the \texttt{\text{relation}} is \texttt{true}. After the \texttt{\text{code}} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.

\section{26.6 Dimension step functions}

\texttt{\dim_step_function:nnnN} \hspace{1em} \texttt{\dim_step_function:nnnN \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{\text{function}\}}

This function first evaluates the \texttt{\text{initial value}}, \texttt{\text{step}} and \texttt{\text{final value}}, all of which should be dimension expressions. The \texttt{\text{function}} is then placed in front of each \texttt{\text{value}} from the \texttt{\text{initial value}} to the \texttt{\text{final value}} in turn (using \texttt{\text{step}} between each \texttt{\text{value}}). The \texttt{\text{step}} must be non-zero. If the \texttt{\text{step}} is positive, the loop stops when the \texttt{\text{value}} becomes larger than the \texttt{\text{final value}}. If the \texttt{\text{step}} is negative, the loop stops when the \texttt{\text{value}} becomes smaller than the \texttt{\text{final value}}. The \texttt{\text{function}} should absorb one argument.

\texttt{\dim_step_inline:nnn} \hspace{1em} \texttt{\dim_step_inline:nnn \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{\text{code}\}}

This function first evaluates the \texttt{\text{initial value}}, \texttt{\text{step}} and \texttt{\text{final value}}, all of which should be dimension expressions. Then for each \texttt{\text{value}} from the \texttt{\text{initial value}} to the \texttt{\text{final value}} in turn (using \texttt{\text{step}} between each \texttt{\text{value}}), the \texttt{\text{code}} is inserted into the input stream with \#1 replaced by the current \texttt{\text{value}}. Thus the \texttt{\text{code}} should define a function of one argument (\#1).
\texttt{\dim_step_variable:nnnNn}

This function first evaluates the \texttt{(initial value)}, \texttt{(step)} and \texttt{(final value)}, all of which should be dimension expressions. Then for each \texttt{(value)} from the \texttt{(initial value)} to the \texttt{(final value)} in turn (using \texttt{(step)} between each \texttt{(value)}), the \texttt{(code)} is inserted into the input stream, with the \texttt{(tl var)} defined as the current \texttt{(value)}. Thus the \texttt{(code)} should make use of the \texttt{(tl var)}.

### 26.7 Using \texttt{dim} expressions and variables

\texttt{\dim_eval:n}

Evaluates the \texttt{(dim expr)}, expanding any dimensions and token list variables within the \texttt{(expression)} to their content (without requiring \texttt{\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 \texttt{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 \textit{not} an \texttt{internal dimension}.

\texttt{\dim_sign:n}

Evaluates the \texttt{(dim expr)} then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

\texttt{\dim_use:N}

Recovers the content of a \texttt{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 \texttt{dimension} is required (such as in the argument of \texttt{\dim_eval:n}).

\texttt{\dim_use:c}

\texttt{\dim_to_decimal:n}

Evaluates the \texttt{(dim expr)}, and leaves the result, expressed in points (pt) in the input stream, with \textit{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

\texttt{\dim_to_decimal:n \{ 1bp \}}

leaves 1.00374 in the input stream, \textit{i.e.} the magnitude of one “big point” when converted to (\TeX{}) points.
\dim_to_decimal_in_bp:n \{\langle dim expr\rangle\}

Evaluates the \langle dim expr\rangle, 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.

\TeX{}hackers note: The implementation of this function is re-entrant: the result of

\dim_compare:nNnTF
{ xbp } =
{ \dim_to_decimal_in_bp:n \{ xbp \} bp }

will be logically true. The decimal representations may differ provided they produce the same \TeX{} dimension.

\dim_to_decimal_in_cm:n \{\langle dim expr\rangle\}

Evaluates the \langle dim expr\rangle, 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{itemize}
  \item 1276.00215 cc
  \item 575.83174 cm
  \item 15312.02584 dd
  \item 226.70540 in
  \item 5758.31742 mm
  \item 1365.33333 pc
\end{itemize}

(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.

\TeX{}hackers 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 \definemacro* \dim_to_decimal_in_sp:n {\langle dim expr\rangle}

Evaluates the \langle dim expr\rangle, 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 \definemacro* \dim_to_decimal_in_unit:nn {\langle dim expr_1\rangle} {\langle dim expr_2\rangle}

Evaluates the \langle dim exprs\rangle, and leaves the value of \langle dim expr_1\rangle, expressed in a unit given by \langle dim expr_2\rangle, 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 \definemacro* \dim_to_fp:n {\langle dim expr\rangle}

Expands to an internal floating point number equal to the value of the \langle dim expr\rangle in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, \dim_to_fp:n can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

### 26.8 Viewing dim variables

\dim_show:N \definemacro* \dim_show:N \langle dimension\rangle
\dim_show:C
Displays the value of the \langle dimension\rangle on the terminal.

\dim_show:n \definemacro* \dim_show:n \langle dim expr\rangle

Displays the result of evaluating the \langle dim expr\rangle on the terminal.

\dim_log:N \definemacro* \dim_log:N \langle dimension\rangle
\dim_log:C

Writes the value of the \langle dimension\rangle in the log file.
26.9 Constant dimensions

\c_max_dim The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

\c_zero_dim A zero length as a dimension. This can also be used as a component of a skip.

26.10 Scratch dimensions

\l_tmpa_dim \l_tmpb_dim Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any \TeX3-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 \TeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

26.11 Creating and initialising skip variables

\skip_new:N \skip_new:c 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 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 Sets \skip to 0 pt.
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.

Tests whether the (skip) is currently defined. This does not check that the (skip) really is a skip variable.

26.12 Setting skip variables

Adds the result of the (skip expr) to the current content of the (skip).

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).

Sets the content of (skip_1) equal to that of (skip_2).

Subtracts the result of the (skip expr) from the current content of the (skip).
26.13 Skip expression conditionals

\skip_if_eq_p:nn \skip_if_eq:nnTF
\skip_if_finite_p:n \skip_if_finite:nTF

This function first evaluates each of the \langle \text{skip exprs} \rangle as described for \texttt{\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.

26.14 Using \texttt{skip} expressions and variables

\skip_eval:n
\skip_use:N
\skip_use:c

\texttt{\skip_show:N} \langle \text{skip expr} \rangle

\texttt{\skip_show:n} \langle \text{skip expr} \rangle

**Notes:** \texttt{\skip_use:N} is the \texttt{\the} | \texttt{\the} primitive | \texttt{\the}: this is one of several \texttt{\the} names for this primitive.

26.15 Viewing \texttt{skip} variables

\texttt{\skip_show:N} \langle \text{skip} \rangle

\texttt{\skip_show:n} \langle \text{skip expr} \rangle

Displays the value of the \langle \text{skip} \rangle on the terminal.

Displays the result of evaluating the \langle \text{skip expr} \rangle on the terminal.
\skip_log:N \skip_log:C

Wrote the value of the \langle \text{skip} \rangle in the log file.

\skip_log:n
\skip_log:c

New: 2014-08-22
Updated: 2015-08-03

\skip_log:n \{\langle \text{skip expr} \rangle\}

Wrote the result of evaluating the \langle \text{skip expr} \rangle in the log file.

\skip_log:n
\skip_log:c

New: 2014-08-22
Updated: 2015-08-07

\skip_horizontal:N \langle \text{skip} \rangle
\skip_horizontal:c
\skip_horizontal:n \{\langle \text{skip expr} \rangle\}

Inserts a horizontal \langle \text{skip} \rangle into the current list. The argument can also be a \langle \text{dim} \rangle.

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

26.16 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.

26.17 Scratch skips

\l_tmpa_skip \l_tmpb_skip

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

\g_tmpa_skip \g_tmpb_skip

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

26.18 Inserting skips into the output
\skip_vertical:N \skip_vertical:C \skip_vertical:n

Inserts a vertical \skip into the current list. The argument can also be a \dim.

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

26.19 Creating and initialising muskip variables

\muskip_new:N \muskip_new:Nn \muskip_zero:N \muskip_zero:cn

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

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.

\skip_zero:N \skip_zero:cn \muskip_gzero:N \muskip_gzero:cn

Sets \muskip to 0\mu.

\muskip_zero_new:N \muskip_zero_new:cn \muskip_gzero_new:N \muskip_gzero_new:cn

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:NTF \muskip_if_exist_p:cn \muskip_if_exist:CTF

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

26.20 Setting muskip variables

\muskip_add:Nn \muskip_add:cn \muskip_gadd:Nn \muskip_gadd:cn

Adds the result of the \muskip expr to the current content of the \muskip.
\muskip_set:Nn \muskip_set:cn \muskip_gset:Nn \muskip_gset:cn

Updated: 2011-10-22

\muskip_set_eq:NN \muskip eq eq \muskip \muskip
Sets the content of \muskip to that of \muskip.

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

Updated: 2011-10-22

26.21 Using \muskip expressions and variables

\muskip eval:n \muskip eval:n \muskip eval:n

Updated: 2011-10-22

\muskip use:N \muskip use:c

\muskip show:N \muskip show:c

Updated: 2015-08-03

236
Displays the result of evaluating the \( \text{<muskip expr>} \) on the terminal.

\( \text{New: 2011-11-22} \)
\( \text{Updated: 2015-08-07} \)

\( \text{\texttt{muskip_log:N}} \)
\( \text{\texttt{muskip_log:c}} \)

Writes the value of the \( \text{<muskip>} \) in the log file.

\( \text{New: 2014-08-22} \)
\( \text{Updated: 2015-08-03} \)

\( \text{\texttt{muskip_log:n}} \)
\( \text{\texttt{muskip_log:n}} \)

Writes the result of evaluating the \( \text{<muskip expr>} \) in the log file.

\( \text{New: 2014-08-22} \)
\( \text{Updated: 2015-08-07} \)

\section*{26.23 Constant muskips}

\( \text{\texttt{c_max_muskip}} \)

The maximum value that can be stored as a muskip, with no stretch nor shrink component.

\( \text{\texttt{c_zero_muskip}} \)

A zero length as a muskip, with no stretch nor shrink component.

\section*{26.24 Scratch muskips}

\( \text{\texttt{l_tmpa_muskip}} \)
\( \text{\texttt{l_tmpb_muskip}} \)

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.

\( \text{\texttt{g_tmpa_muskip}} \)
\( \text{\texttt{g_tmpb_muskip}} \)

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.

\section*{26.25 Primitive conditional}

\( \text{\texttt{if_dim:w}} \)
\( \langle \text{dimen}_1 \rangle \langle \text{relation} \rangle \langle \text{dimen}_2 \rangle \langle \text{true code} \rangle \langle \text{false} \rangle \langle \text{fi} \rangle \)

Compare two dimensions. The \( \langle \text{relation} \rangle \) is one of \( <, = \) or \( > \) with category code 12.

\textbf{\textit{\LaTeX}xhackers note:} This is the \LaTeX primitive \texttt{ifdim}.
Chapter 27

The l3keys module
Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

\MyModuleSetup{  
  key-one = value one,  
  key-two = value two  
}

or

\MyModuleMacro[  
  key-one = value one,  
  key-two = value two  
]{argument}

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more properties of the key:

\keys_define:nn { mymodule }  
{  
  key-one .code:n = code including parameter #1,  
  key-two .tl_set:N = \l_mymodule_store_tl  
}

These values can then be set as with other key–value approaches:

\keys_set:nn { mymodule }  
{  
  key-one = value one,  
  key-two = value two  
}
As illustrated, keys are created inside a \texttt{module}: a set of related keys, typically those for a single module/\LaTeX{} package. See Section for suggestions on how to divide large numbers of keys for a single module.

At a document level, \texttt{\keys_set:nn} is used within a document function, for example

\begin{verbatim}
\ 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:
}
\end{verbatim}

Key names may contain any tokens, as they are handled internally using \texttt{\tl_to_str:n}. As discussed in section 27.2, it is suggested that the character / is reserved for sub-division of keys into logical groups. Functions and variables are not expanded when creating key names, and so

\begin{verbatim}
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
  { \l_mymodule_tmp_tl .code:n = code }
\end{verbatim}

creates a key called \texttt{\l_mymodule_tmp_tl}, and not one called \texttt{key}.

\section{Creating keys}

\begin{verbatim}
\keys_define:nn \keys_define:ne
\end{verbatim}

parses the \texttt{(keyval list)} and defines the keys listed there for \texttt{(module)}. The \texttt{(module)} name is treated as a string. In practice the \texttt{(module)} should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The \texttt{(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 \texttt{\keys_define:nn} might read

\begin{verbatim}
\keys_define:nn { mymodule }
  { 
    keyname .code:n = Some-code-using-#1, 
    keyname .value_required:n = true 
  }
\end{verbatim}

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 \texttt{(key)}, which when used may be supplied with a \texttt{(value)}. All key \texttt{definitions} are local.
Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as \texttt{.code:n}, \texttt{.tl_set:N}, etc., override one another. Some other properties are mutually exclusive, notably \texttt{.value_required:n} and \texttt{.value_forbidden:n}, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

\begin{verbatim}
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some-code-using-#1,
    keyname .value_required:n = true
  }
\end{verbatim}

\begin{verbatim}
\keys_define:nn { mymodule }
  {
    keyname .value_required:n = true,
    keyname .code:n = Some-code-using-#1
  }
\end{verbatim}

Note that all key properties define the key within the current \TeX{} group, with an exception that the special \texttt{.undefine:} property \textit{undefines} the key within the current \TeX{} group.

\begin{verbatim}
\bool_set:N \bool_set:c \bool_gset:N \bool_gset:c
\end{verbatim}

\begin{verbatim}
\bool_set_inverse:N \bool_set_inverse:c \bool_gset_inverse:N \bool_gset_inverse:c
\end{verbatim}

\begin{verbatim}
\clist_set:N \clist_set:c \clist_gset:N \clist_gset:c
\end{verbatim}

\begin{verbatim}
\choice: \choices:nn \choices:(Vn|en|on)
\end{verbatim}

\begin{verbatim}
\clist_set:N \clist_set:c \clist_gset:N \clist_gset:c
\end{verbatim}

\begin{verbatim}
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some-code-using-#1,
    keyname .value_required:n = true
  }
\end{verbatim}

\begin{verbatim}
\keys_define:nn { mymodule }
  {
    keyname .value_required:n = true,
    keyname .code:n = Some-code-using-#1
  }
\end{verbatim}

Note that all key properties define the key within the current \TeX{} group, with an exception that the special \texttt{.undefine:} property \textit{undefines} the key within the current \TeX{} group.

\begin{verbatim}
\bool_set:N \bool_set:c \bool_gset:N \bool_gset:c
\end{verbatim}

\begin{verbatim}
\bool_set_inverse:N \bool_set_inverse:c \bool_gset_inverse:N \bool_gset_inverse:c
\end{verbatim}

\begin{verbatim}
\clist_set:N \clist_set:c \clist_gset:N \clist_gset:c
\end{verbatim}

\begin{verbatim}
\choice: \choices:nn \choices:(Vn|en|on)
\end{verbatim}

\begin{verbatim}
\clist_set:N \clist_set:c \clist_gset:N \clist_gset:c
\end{verbatim}

\begin{verbatim}
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some-code-using-#1,
    keyname .value_required:n = true
  }
\end{verbatim}

\begin{verbatim}
\keys_define:nn { mymodule }
  {
    keyname .value_required:n = true,
    keyname .code:n = Some-code-using-#1
  }
\end{verbatim}

Note that all key properties define the key within the current \TeX{} group, with an exception that the special \texttt{.undefine:} property \textit{undefines} the key within the current \TeX{} group.

\begin{verbatim}
\bool_set:N \bool_set:c \bool_gset:N \bool_gset:c
\end{verbatim}

\begin{verbatim}
\bool_set_inverse:N \bool_set_inverse:c \bool_gset_inverse:N \bool_gset_inverse:c
\end{verbatim}

\begin{verbatim}
\clist_set:N \clist_set:c \clist_gset:N \clist_gset:c
\end{verbatim}

\begin{verbatim}
\choice: \choices:nn \choices:(Vn|en|on)
\end{verbatim}

\begin{verbatim}
\clist_set:N \clist_set:c \clist_gset:N \clist_gset:c
\end{verbatim}

\begin{verbatim}
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some-code-using-#1,
    keyname .value_required:n = true
  }
\end{verbatim}

\begin{verbatim}
\keys_define:nn { mymodule }
  {
    keyname .value_required:n = true,
    keyname .code:n = Some-code-using-#1
  }
\end{verbatim}

Note that all key properties define the key within the current \TeX{} group, with an exception that the special \texttt{.undefine:} property \textit{undefines} the key within the current \TeX{} group.
Stores the \texttt{code} for execution when \texttt{key} is used. The \texttt{code} can include one parameter (\#1), which will be the \texttt{value} given for the \texttt{key}.

\texttt{cs_set:N\_p} = \langle control sequence \rangle \langle arg. spec. \rangle

Defines \texttt{key} to set \texttt{control sequence} to have \texttt{arg. spec.} and replacement text \texttt{value}.

\texttt{default:n} = \{\langle default \rangle\}

Creates a \texttt{default} value for \texttt{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 \texttt{value} is given:

\begin{verbatim}
\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 '
  }
\end{verbatim}

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 \texttt{\keys_set:nn}, the \texttt{\default:n} value provides the value before key properties are considered. The only exception is when the \texttt{\value_required:n} property is active: a required value cannot be supplied by the default, and must be explicitly given as part of \texttt{\keys_set:nn}.

\texttt{dim_set:N} = \langle dimension \rangle

Defines \texttt{key} to set \texttt{dimension} to \texttt{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.

\texttt{fp_set:N} = \{\langle floating point \rangle\}

Defines \texttt{key} to set \texttt{floating point} to \texttt{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\}

New: 2013-07-14

Defines (key) as belonging to the \{groups\} declared. Groups provide a “secondary axis”
for selectively setting keys, and are described in Section 27.7.

.inherit:n  (key) .\inherit:n = \{parents\}

New: 2016-11-22

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 subgroup. For example, after setting
\begin{verbatim}
\keys_define:nn { foo } { test .\code:n = \tl_show:n \{#1\} }
\keys_define:nn { } { bar .\inherit:n = foo }
\end{verbatim}
setting
\begin{verbatim}
\keys_set:nn { bar } { test = a }
\end{verbatim}

will be equivalent to
\begin{verbatim}
\keys_set:nn { foo } { test = a }
\end{verbatim}

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  (key) .\initial:n = \{value\}

Initialises the \{key\} with the \{value\}, equivalent to
\begin{verbatim}
\keys_set:nn { \{module\} } { \{key\} = \{value\} }
\end{verbatim}

Updated: 2013-07-09

.int_set:N  (key) .\int_set:N = \{integer\}

\int_set:c  (key) .\int_set:c

\int_gset:N  (key) .\int_gset:N

\int_gset:c  (key) .\int_gset:c

Updated: 2020-01-17

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  (key) .\legacy_if_set:n = \{switch\}

.\legacy_if_gset:n  (key) .\legacy_if_gset:n

.\legacy_if_set_inverse:n  (key) .\legacy_if_set_inverse:n

.\legacy_if_gset_inverse:n  (key) .\legacy_if_gset_inverse:n

Updated: 2022-01-15

Defines \{key\} to set legacy \texttt{\if\{switch\}} to \{value\} (which must be either “true” or
“false”). The \{switch\} is the name of the switch without the leading \texttt{if}.
The inverse versions will set the \{switch\} to the logical opposite of the \{value\}.

.meta:n  (key) .\meta:n = \{keyval list\}

Updated: 2013-07-10

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).

242
.meta:nn \{key\}.meta:nn = \{(path)\} \{keyval list\}

Makes \langle key\rangle a meta-key, which will set \langle keyval list\rangle in one go using the \langle path\rangle in place of the current one. The \langle keyval list\rangle can refer as \#1 to the value given at the time the \langle key\rangle is used (or, if no value is given, the \langle key\rangle's default value).

.multichoice: \{key\}.multichoice:

New: 2011-06-21

Sets \langle key\rangle to act as a multiple choice key. Each valid choice for \langle key\rangle must then be created, as discussed in section 27.3.

.multichoices:nn \{key\}.multichoices:nn \{(choices)\} \{(code)\}

Sets \langle key\rangle to act as a multiple choice key, and defines a series \langle choices\rangle which are implemented using the \langle code\rangle. Inside \langle code\rangle, \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 \langle choices\rangle (indexed from 1). Choices are discussed in detail in section 27.3.

.muskip_set:N \{key\}.muskip_set:N = \{muskip\}

Defines \langle key\rangle to set \langle muskip\rangle to \langle value\rangle (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 \{key\}.prop_put:N = \{property list\}

Defines \langle key\rangle to put the \langle value\rangle onto the \langle property list\rangle stored under the \langle key\rangle. If the variable does not exist, it is created globally at the point that the key is set up.

.skip_set:N \{key\}.skip_set:N = \{skip\}

Defines \langle key\rangle to set \langle skip\rangle to \langle value\rangle (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 \{key\}.str_set:N = \{string variable\}

Defines \langle key\rangle to set \langle string variable\rangle to \langle value\rangle. If the variable does not exist, it is created globally at the point that the key is set up.

.str_set_e:N \{key\}.str_set_e:N = \{string variable\}

Defines \langle key\rangle to set \langle string variable\rangle to \langle value\rangle, which will be subjected to an e-type expansion (i.e. using \texttt{\str_set:Ne}). If the variable does not exist, it is created globally at the point that the key is set up.

243
\tl_set:N  \langle \text{key} \rangle .\tl_set:N = \langle \text{token list variable} \rangle 
\tl_set:c
\tl_gset:N
\tl_gset:c

Defines \langle \text{key} \rangle to set \langle \text{token list variable} \rangle to \langle \text{value} \rangle. If the variable does not exist, it is created globally at the point that the key is set up.

\tl_set_e:N  \langle \text{key} \rangle .\tl_set_e:N = \langle \text{token list variable} \rangle 
\tl_set_e:c
\tl_gset_e:N
\tl_gset_e:c

\text{New: 2023-09-18}

Defines \langle \text{key} \rangle to set \langle \text{token list variable} \rangle to \langle \text{value} \rangle, 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.

\text{.undefine: } \langle \text{key} \rangle .\text{.undefine: }
\text{New: 2015-07-14}

Removes the definition of the \langle \text{key} \rangle within the current \TeX{} group.

\text{.value_forbidden:n } \langle \text{key} \rangle .\text{value_forbidden:n} = \text{true|false}
\text{New: 2015-07-14}

Specifies that \langle \text{key} \rangle cannot receive a \langle \text{value} \rangle when used. If a \langle \text{value} \rangle is given then an error will be issued. Setting the property “false” cancels the restriction.

\text{.value_required:n } \langle \text{key} \rangle .\text{value_required:n} = \text{true|false}
\text{New: 2015-07-14}

Specifies that \langle \text{key} \rangle must receive a \langle \text{value} \rangle when used. If a \langle \text{value} \rangle is not given then an error will be issued. Setting the property “false” cancels the restriction.

### 27.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

\keys_define:nn { mymodule / subgroup }
{ key .\text{code:n} = code }

or to the key name:

\keys_define:nn { mymodule }
{ subgroup / key .\text{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/subgroup/key.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.
27.3 Choice and multiple choice keys

The \texttt{\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 \texttt{.choice:} property:

\begin{verbatim}
\keys_define:nn { mymodule }
 { key .choice: }
\end{verbatim}

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the \texttt{.choices:nn} property.

\begin{verbatim}
\keys_define:nn { mymodule }
 { key .choices:nn =
   { choice-a, choice-b, choice-c }
   { You\_gave\_choice\_\texttt{\l\_keys\_choice\_tl},\texttt{\l\_keys\_choice\_int} in-the-list. }
 }
\end{verbatim}

The index \texttt{\l\_keys\_choice\_int} in the list of choices starts at 1.

Inside the code block for a choice generated using \texttt{.choices:nn}, the variables \texttt{\l\_keys\_choice\_tl} and \texttt{\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 \texttt{.code:n}, the value passed to the key (i.e. the choice name) is also available as \texttt{#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 \texttt{.choice:} property of a key, then manually defining sub-keys.

\begin{verbatim}
\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,
 }
\end{verbatim}

It is possible to mix the two methods, but manually-created choices should \textbf{not} use \texttt{\l\_keys\_choice\_tl} or \texttt{\l\_keys\_choice\_int}. These variables do not have defined
behaviour when used outside of code created using \texttt{.choices:nn} \textit{(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 \texttt{unknown} choice. The general behavior of the \texttt{unknown} key is described in Section 27.6. A typical example in the case of a choice would be to issue a custom error message:

\begin{verbatim}
\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  % Valid choices  % Invalid choice given
   \exp_not:n {#1}
}
\end{verbatim}

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties \texttt{.multichoice:} and \texttt{.multichoices:nn}. As with mutually exclusive choices, multiple choices are defined as sub-keys. Thus both

\begin{verbatim}
\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.  }
}
\end{verbatim}

and

\begin{verbatim}
\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,  }
\end{verbatim}

are valid.

When a multiple choice key is set
\keys_set:nn { mymodule }
{
    key = { a , b , c } \% 'key' defined as a multiple choice
}

Each choice is applied in turn, equivalent to a clist mapping or to applying each value individually:

\keys_set:nn { mymodule }
{
    key = a ,
    key = b ,
    key = c ,
}

Thus each separate choice will have passed to it the \l_keys_choice_tl and \l_keys_choice_int in exactly the same way as described for .choices:nn.

### 27.4 Key usage scope

Some keys will be used as settings which have a strictly limited scope of usage. Some will be only available once, others will only be valid until typesetting begins. To allow formats to support this in a structured way, l3keys allows this information to be specified using the .usage:n property.

\begin{verbatim}
\l_keys_usage_load_prop
\l_keys_usage_preamble_prop
\end{verbatim}

\l3keys itself does \textit{not} attempt to redefine keys based on the usage scope. Rather, this information is made available with these two property lists. These hold an entry for each module (prefix); the value of each entry is a comma-separated list of the usage-restricted key(s).

### 27.5 Setting keys

\begin{verbatim}
\keys_set:nn
\keys_set:(nV|nv|ne|no)
\end{verbatim}

\keys_set:nn \{(module) \{(keyval list)\}

 Parses the \{(keyval list)\}, and sets those keys which are defined for \{module\}. The behaviour on finding an unknown key can be set by defining a special unknown key: this is illustrated later.
For each key processed, information of the full path of the key, the name of the key and the value of the key is available within two string and one token list variables. These may be used within the code of the key.

The path of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

\keys_set:nn { mymodule } { key-a = some-value }

has path mymodule/key-a while

\keys_set:nn { mymodule } { subset / key-a = some-value }

has path mymodule/subset/key-a. This information is stored in \l_keys_path_str.

The name of the key is the part of the path after the last /, and thus is not unique. In the preceding examples, both keys have name key-a despite having different paths. This information is stored in \l_keys_key_str.

The value is everything after the =, which may be empty if no value was given. This is stored in \l_keys_value_tl, and is not processed in any way by \keys_set:nn.

### 27.6 Handling of unknown keys

If a key has not previously been defined (is unknown), \keys_set:nn looks for a special unknown key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts. The unknown key also supports the .default:n property.

\keys_define:nn { mymodule }
{
    unknown .code:n =
    You-tried-to-set-key-’\l_keys_key_str’-to-’#1’.
    unknown .default:V = \c_novalue_tl
}

\keys_set_known:nn \keys_set_known:nnN \keys_set_known:nnnN

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 stores the key–value pairs in the ⟨t1⟩ 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.
27.7 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

\begin{verbatim}
\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 ,
}
\end{verbatim}

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

\begin{verbatim}
\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 ,
}
\end{verbatim}

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.

\begin{verbatim}
\keys_set_filter:nnn \keys_set_filter:nnnN \keys_set_filter:nnnnN
\end{verbatim}

Activates key filtering in an “opt-out” sense: keys assigned to any of the \groups specified are ignored. The \groups are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key–value pairs for each key which is filtered out are stored in the \tl in a comma-separated form \textit{i.e.} an edited version of the \keyval list. The \keys_set_filter:nnn version skips this stage.

Use of \keys_set_filter:nnnN can be nested, with the correct residual \keyval list returned at each stage. In the version which takes a \root argument, the key list is returned relative to that point in the key tree. In the cases without a \root argument, only the key names and values are returned.
Activates key filtering 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.

27.8 Digesting keys

Parses the ⟨keyval list⟩ as for \keys_set:nn, placing the resulting code for those which set variables or functions into the ⟨tl⟩. Thus this function “precompiles” the keyval list into a set of results which can be applied rapidly.

27.9 Utility functions for keys

Tests if the ⟨key⟩ exists for ⟨module⟩, i.e. if any code has been defined for ⟨key⟩.

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.

Displays in the terminal the information associated to the ⟨key⟩ for a ⟨module⟩, including the function which is used to actually implement it.

Writes in the log file the information associated to the ⟨key⟩ for a ⟨module⟩. See also \keys_show:nn which displays the result in the terminal.
27.10 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

\[
\text{KeyOne} = \text{ValueOne} , \\
\text{KeyTwo} = \text{ValueTwo} , \\
\text{KeyThree}
\]

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 \((\text{key–value list})\) into \((\text{keys})\) and associated \((\text{values})\). 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

\[
\text{key} = \{\text{value here}\},
\]

and

\[
\text{key} = \text{value here},
\]

are treated identically.
\keyval_parse:nnn \keyval_parse:(nnVnnv)

Parses the \textit{key–value list} into a series of \textit{keys} and associated \textit{values}, or keys alone (if no \textit{value} was given). \textit{code1} receives each \textit{key} (with no \textit{value}) as a trailing brace group, whereas \textit{code2} is appended by two brace groups, the \textit{key} and \textit{value}. The order of the \textit{keys} in the \textit{key–value list} is preserved. Thus

\begin{verbatim}
\keyval_parse:nnn
  { \use_none:nn { code 1 } }
  { \use_none:nn { code 2 } }
  { key1 = value1, key2 = value2, key3 = , key4 }
\end{verbatim}

is converted into an input stream

\begin{verbatim}
\use_none:nn { code 2 } { key1 } { value1 }
\use_none:nn { code 2 } { key2 } { value2 }
\use_none:nn { code 2 } { key3 } { }
\use_none:nn { code 1 } { key4 }
\end{verbatim}

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 \textit{key} and \textit{value}, then one outer set of braces is removed from the \textit{key} and \textit{value} as part of the processing. If you need exactly the output shown above, you’ll need to either \texttt{e}-type or \texttt{x}-type expand the function.

\textbf{\TeXhackers note:} The result of each list element is returned within \texttt{\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:(NN V)Nn

Parses the \texttt{<key–value list>} into a series of \texttt{<keys>} and associated \texttt{<values>}, or keys alone (if no \texttt{<value>} was given). \texttt{<function>} should take one argument, while \texttt{<function2>} should absorb two arguments. After \texttt{\keyval_parse:NNn} has parsed the \texttt{<key–value list>}, \texttt{<function>} is used to process keys given with no value and \texttt{<function2>} is used to process keys given with a value. The order of the \texttt{<keys>} in the \texttt{<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 \texttt{<key>} and \texttt{<value>}, then one outer set of braces is removed from the \texttt{<key>} and \texttt{<value>} as part of the processing.

This shares the implementation of \texttt{\keyval_parse:NNNN}, the difference is only semantically.

\TeXhacksnote{The result is returned within \texttt{\exp_not:n}, which means that the converted input stream does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion.}
Chapter 28

The \texttt{l3intarray} module

Fast global integer arrays

28.1 \texttt{l3intarray} documentation

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast \texttt{l3seq}, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum $2^{30} - 1$ (i.e. one power lower than the usual \texttt{c_max_int} ceiling of $2^{31} - 1$)

The use of \texttt{intarray} data is therefore recommended for cases where the need for fast access is of paramount importance.

\begin{verbatim}
\texttt{\intarray_new:Nn} \langle \texttt{intarray var} \rangle \{ \langle \texttt{size} \rangle \}
\texttt{\intarray_new:cn}
\end{verbatim}

Evaluates the integer expression $\langle \texttt{size} \rangle$ and allocates an \texttt{integer array variable} with that number of (zero) entries. The variable name should start with \texttt{\g_} because assignments are always global.

\begin{verbatim}
\texttt{\intarray_count:N} \langle \texttt{intarray var} \rangle
\texttt{\intarray_count:c}
\end{verbatim}

Expands to the number of entries in the \texttt{integer array variable}. Contrarily to \texttt{\seq_count:N} this is performed in constant time.

\begin{verbatim}
\texttt{\intarray_gset:Nnn} \langle \texttt{intarray var} \rangle \{ \langle \texttt{position} \rangle \} \{ \langle \texttt{value} \rangle \}
\texttt{\intarray_gset:cnn}
\end{verbatim}

Stores the result of evaluating the integer expression $\langle \texttt{value} \rangle$ into the \texttt{integer array variable} at the (integer expression) $\langle \texttt{position} \rangle$. If the $\langle \texttt{position} \rangle$ is not between 1 and the $\texttt{\intarray_count:N}$, or the $\langle \texttt{value} \rangle$'s absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.
\intarray_const_from_clist:Nn \intarray_const_from_clist:Nn \langle \text{intarray var} \rangle \langle \text{int expr clist} \rangle
\intarray_const_from_clist:cn
New: 2018-05-04

Creates a new constant \textit{(integer array variable)} or raises an error if the name is already taken. The \textit{(integer array variable)} is set (globally) to contain as its items the results of evaluating each \textit{(integer expression)} in the \textit{(comma list)}.

\intarray_gzero:N \intarray_gzero:cn
New: 2018-05-04
\intarray_gzero:N \langle \text{intarray var} \rangle
Sets all entries of the \textit{(integer array variable)} to zero. Assignments are always global.

\intarray_item:Nn \intarray_item:cn *
New: 2018-03-29
\intarray_item:Nn \langle \text{intarray var} \rangle \{ \langle \text{position} \rangle \}
Expands to the integer entry stored at the \textit{(integer expression)} \textit{(position)} in the \textit{(integer array variable)}. If the \textit{(position)} is not between 1 and the \textit{\intarray_count:N}, an error occurs.

\intarray_rand_item:N \intarray_rand_item:cn *
New: 2018-05-05
\intarray_rand_item:N \langle \text{intarray var} \rangle
Selects a pseudo-random item of the \textit{(integer array)}. If the \textit{(integer array)} is empty, produce an error.

\intarray_show:N \intarray_show:cn \intarray_log:N \intarray_log:cn
New: 2018-05-04
\intarray_show:N \langle \text{intarray var} \rangle
\intarray_log:N \langle \text{intarray var} \rangle
Displays the items in the \textit{(integer array variable)} in the terminal or writes them in the log file.

### 28.1.1 Implementation notes

It is a wrapper around the \texttt{\fontdimen} primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to \texttt{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 \texttt{\fontdimen} stores dimensions but the \texttt{l3intarray} module transparently converts these from/to integers. Assignments are always global.

While Lua\TeX{}’s memory is extensible, other engines can “only” deal with a bit less than $4 \times 10^9$ entries in all \texttt{\fontdimen} arrays combined (with default \TeX{} Live settings).
Chapter 29

The l3fp module

Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions (“\langle fp expr\rangle”) support the following operations with their usual precedence.

- Basic arithmetic: addition \( x + y \), subtraction \( x - y \), multiplication \( x \times y \), division \( x/y \), square root \( \sqrt{x} \), and parentheses.
- Comparison operators: \( x < y \), \( x \leq y \), \( x > y \), \( x \geq y \) etc.
- Boolean logic: sign \( \text{sign} \ x \), negation \( \neg x \), conjunction \( x \&\& y \), disjunction \( x || y \), ternary operator \( x ? y : z \).
- Exponentials: \( \exp x \), \( \ln x \), \( x^y \), \( \log_b x \).
- Integer factorial: \( \text{fact} x \).
- Trigonometry: \( \sin x \), \( \cos x \), \( \tan x \), \( \cot x \), \( \sec x \), \( \csc x \) expecting their arguments in radians, and \( \text{sind} x \), \( \text{cosd} x \), \( \text{tand} x \), \( \text{cotd} x \), \( \text{secd} x \), \( \text{cscd} x \) expecting their arguments in degrees.
- Inverse trigonometric functions: \( \arcsin x \), \( \arccos x \), \( \arctan x \), \( \arccot x \), \( \text{arcsec} x \), \( \text{arccsc} x \) giving a result in radians, and \( \text{asind} x \), \( \text{acosd} x \), \( \text{atand} x \), \( \text{acotd} x \), \( \text{asecd} x \), \( \text{acscd} x \) giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: \( \sinh x \), \( \cosh x \), \( \tanh x \), \( \coth x \), \( \text{sech} x \), \( \text{csch} x \), and \( \text{asinh} x \), \( \text{acosh} x \), \( \text{atanh} x \), \( \text{acoth} x \), \( \text{asech} x \), \( \text{acsch} x \).
- Extrema: \( \max(x_1, x_2, \ldots) \), \( \min(x_1, x_2, \ldots) \), \( \text{abs}(x) \).
- Rounding functions, controlled by two optional values, \( n \) (number of places, 0 by default) and \( t \) (behavior on a tie, \text{nan} by default):
  - \( \text{trunc}(x, n) \) rounds towards zero,
  - \( \text{floor}(x, n) \) rounds towards \( -\infty \),

256
\[ \textbf{ceil}(x, n) \] rounds towards \(+\infty\),
\[ \textbf{round}(x, n, t) \] 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: \texttt{rand()}, \texttt{randint}(m, n).
- Constants: \texttt{pi}, \texttt{deg} (one degree in radians).
- Dimensions, automatically expressed in points, \textit{e.g.}, \texttt{pc} is 12.
- Automatic conversion (no need for \langle\texttt{type}\_use:N\rangle) 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
\(-0.0001\)), or as a stored floating point variable, which is automatically replaced by its
current value. A “floating point” is a floating point number or a tuple thereof. See section
29.12.1 for a description of what a floating point is, section 29.12.2 for details about
how an expression is parsed, and section 29.12.3 to know what the various operations do.
Some operations may raise exceptions (error messages), described in section 29.10.

An example of use could be the following.

\begin{verbatim}
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2 \cdot 10^{-3}$
= \ExplSyntaxOn \fp_to_decimal:n {sin(3.5)/2 + 2e-3} \.$
\end{verbatim}

The operation \texttt{round} can be used to limit the result’s precision. Adding \(+0\) avoids the
possibly undesirable output \(-0\), replacing it by \(+0\). However, the \texttt{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.

\begin{verbatim}
\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}
\end{verbatim}

See the documentation of \texttt{siunitx} for various options of \texttt{num}.
29.1 Creating and initialising floating point variables

\fp_new:N \langle fp \ var \rangle
\fp_new:c

Creates a new \langle fp \ var \rangle or raises an error if the name is already taken. The declaration is global. The \langle fp \ var \rangle is initially +0.

\fp_const:Nn \langle fp \ var \rangle \{ \langle fp \ expr \rangle \}
\fp_const:cn

Creates a new constant \langle fp \ var \rangle or raises an error if the name is already taken. The \langle fp \ var \rangle is set globally equal to the result of evaluating the \langle fp \ expr \rangle.

\fp_zero:N \langle fp \ var \rangle
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

Sets the \langle fp \ var \rangle to +0.

\fp_zero_new:N \langle fp \ var \rangle
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

Ensures that the \langle fp \ var \rangle exists globally by applying \fp_new:N if necessary, then applies \fp_(g)zero:N to leave the \langle fp \ var \rangle set to +0.

29.2 Setting floating point variables

\fp_set:Nn \langle fp \ var \rangle \{ \langle fp \ expr \rangle \}
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn

Sets \langle fp \ var \rangle equal to the result of computing the \langle fp \ expr \rangle.

\fp_set_eq:NN \langle fp \ var_1 \rangle \langle fp \ var_2 \rangle
\fp_set_eq:cn\langle fp \ var_1 \rangle \langle fp \ var_2 \rangle\langle cn|\Nc|\cc \rangle
\fp_gset_eq:NN
\fp_gset_eq:cn\langle fp \ var_1 \rangle \langle fp \ var_2 \rangle\langle cn|\Nc|\cc \rangle

Sets the floating point variable \langle fp \ var_1 \rangle equal to the current value of \langle fp \ var_2 \rangle.

\fp_add:Nn \langle fp \ var \rangle \{ \langle fp \ expr \rangle \}
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn

Adds the result of computing the \langle fp \ expr \rangle to the \langle fp \ var \rangle. This also applies if \langle fp \ var \rangle and \langle floating point expression \rangle evaluate to tuples of the same size.
29.3 Using floating points

\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.

\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 \(\text{nan}\), then “invalid operation” occurs and the result is 0.

\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 \(±\infty\) and \(\text{nan}\) trigger an “invalid operation” exception. For a tuple, each item is converted using \fp_eval:n and they are combined as \((\langle\fp_1\rangle,\langle\fp_2\rangle,\ldots,\langle\fp_n\rangle)\) if \(n>1\) and \((\langle\fp_1\rangle,)\) or () for fewer items. This function is identical to \fp_to_decimal: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 \(±\infty\) and \(\text{nan}\), trigger an “invalid operation” exception.

\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 \(±\infty\) and \(\text{nan}\), trigger an “invalid operation” exception.
\fp_to_scientific:N * \fp_to_scientific:n \fp_to_scientific:c

Evaluates the \textit{fp expr} and expresses the result in scientific notation:
\[(\text{optional } -)\langle \text{digit} \rangle.\langle 15 \text{ digits} \rangle\text{e}(\text{optional sign})\langle \text{exponent} \rangle\]
The leading \textit{digit} is non-zero except in the case of ±0. The values ±∞ 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 \textit{fp_to_scientific:n} and they are combined as \((\langle \text{fp}_1 \rangle, \langle \text{fp}_2 \rangle, \ldots, \langle \text{fp}_n \rangle)\) if \(n > 1\) and \((\langle \text{fp}_1 \rangle,\) or () for fewer items.

\fp_to_tl:N * \fp_to_tl:n \fp_to_tl:c

Evaluates the \textit{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 \textit{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 \textit{fp_to_decimal:n}). Negative numbers start with -. The special values ±0, ±∞ 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 \textit{fp_to_tl:n} and they are combined as \((\langle \text{fp}_1 \rangle, \langle \text{fp}_2 \rangle, \ldots, \langle \text{fp}_n \rangle)\) if \(n > 1\) and \((\langle \text{fp}_1 \rangle,\) or () for fewer items. This function is identical to \textit{fp_to_decimal:N}.

\fp_use:N * \fp_use:n \fp_use:c

Inserts the value of the \textit{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 ±∞ and nan trigger an “invalid operation” exception. For a tuple, each item is converted using \textit{fp_to_decimal:n} and they are combined as \((\langle \text{fp}_1 \rangle, \langle \text{fp}_2 \rangle, \ldots, \langle \text{fp}_n \rangle)\) if \(n > 1\) and \((\langle \text{fp}_1 \rangle,\) or () for fewer items. This function is identical to \textit{fp_to_decimal:N}.

29.4 Floating point conditionals

\fp_if_exist_p:N * \fp_if_exist_p:n \fp_if_exist_p:c
\fp_if_exist:NNTF * \fp_if_exist:NNTF_c

Tests whether the \textit{fp var} is currently defined. This does not check that the \textit{fp var} really is a floating point variable.
\verb|
\fp_compare_p:nNn| \verb| \fp_compare_p:nNn \{\langle \text{fp expr} \rangle \} \{\langle \text{relation} \rangle \} \{\langle \text{fp expr} \rangle \}
\verb|
\fp_compare:nNnTF| \verb| \fp_compare:nNnTF \{\langle \text{fp expr} \rangle \} \{\langle \text{relation} \rangle \} \{\langle \text{true code} \rangle \} \{\langle \text{false code} \rangle \}

Updated: 2012-05-08

Compares the \langle \text{fp expr} \rangle and the \langle \text{fp expr} \rangle, and returns \text{true} if the \langle \text{relation} \rangle 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 \text{nan} or is a tuple, unless they are equal tuples. Note that a \text{nan} is distinct from any value, even another \text{nan}, hence \( x = x \) is not true for a \text{nan}. To test if a value is \text{nan}, compare it to an arbitrary number with the "not ordered" relation.

\begin{verbatim}
\fp_compare:nNnTF \{<value>\} \{0\}
\{\} \% \text{<value> is nan}
\{\} \% \text{<value> is not nan}
\end{verbatim}

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no \text{nan}). At present any other comparison with tuples yields ? (not ordered). This is experimental.

This function is less flexible than \verb|\fp_compare:nTF| but slightly faster. It is provided for consistency with \verb|\int_compare:nNnTF| and \verb|\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 \texttt{(fp exprs)} as described for \texttt{\fp_eval:n} and compares consecutive result using the corresponding \texttt{(relation)}, namely it compares \texttt{(fp expr_1)} and \texttt{(fp expr_2)} using the \texttt{(relation_1)}, then \texttt{(fp expr_2)} and \texttt{(fp expr_3)} using the \texttt{(relation_2)}, until finally comparing \texttt{(fp expr_N)} and \texttt{(fp expr_{N+1})} using the \texttt{(relation_N)}. The test yields \texttt{true} if all comparisons are \texttt{true}. Each \texttt{(floating point expression)} is evaluated only once. Contrarily to \texttt{\int_compare:nTF}, all \texttt{(fp exprs)} are computed, even if one comparison is \texttt{false}. Two floating points \texttt{x} and \texttt{y} may obey four mutually exclusive relations: \texttt{x < y}, \texttt{x = y}, \texttt{x > y}, or \texttt{x ? y} ("not ordered"). The last case occurs exactly if one or both operands is \texttt{nan} or is a tuple, unless they are equal tuples. Each \texttt{(relation)} can be any (non-empty) combination of \texttt{<, =, >, or ?}, plus an optional leading \texttt{!} (which negates the \texttt{(relation)}), with the restriction that the \texttt{(relation)} may not start with \texttt{?}, as this symbol has a different meaning (in combination with :) within floating point expressions. The comparison \texttt{x (relation) y} is then \texttt{true} if the \texttt{(relation)} does not start with \texttt{!} and the actual relation (\texttt{<, =, >, or ?}) between \texttt{x} and \texttt{y} appears within the \texttt{(relation)}, or on the contrary if the \texttt{(relation)} starts with \texttt{!} and the relation between \texttt{x} and \texttt{y} does not appear within the \texttt{(relation)}. Common choices of \texttt{(relation)} include \texttt{>=} (greater or equal), \texttt{!=} (not equal), \texttt{!?} or \texttt{<=>} (comparable).

\texttt{\fp_if_nan_p:n} \\texttt{\fp_if_nan:nTF} \\texttt{\fp_if_nan_p:n} \\texttt{\fp_if_nan:nTF}

Evaluates the \texttt{(fp expr)} and tests whether the result is exactly \texttt{nan}. The test returns \texttt{false} for any other result, even a tuple containing \texttt{nan}.

\fp_do_until:nNnn \fp_do_until:nNnn
\fp_do_until:nNnn
\fp_do_until:nNnn
\fp_do_until:nNnn

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \texttt{(floating point expressions)} as described for \texttt{\fp_compare:nTF}. If the test is \texttt{false} then the \texttt{(code)} is inserted into the input stream again and a loop occurs until the \texttt{(relation)} is \texttt{true}.

\subsection*{29.5 Floating point expression loops}

\fp_do_until:nNnn \fp_do_until:nNnn
\fp_do_until:nNnn
\fp_do_until:nNnn
\fp_do_until:nNnn


262
\texttt{\textbackslash fp\_do\_while:nNnn} \hspace{1em} \texttt{\textbackslash fp\_do\_while:nNnn \{\textit{fp expr}_1\} \{\textit{relation}\} \{\textit{fp expr}_2\} \{\textit{code}\}}

Places the \texttt{\textit{code}} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \texttt{\textit{floating point expressions}} as described for \texttt{\textbackslash fp\_compare:nNnTF}. If the test is \texttt{true} then the \texttt{\textit{code}} is inserted into the input stream again and a loop occurs until the \texttt{\textit{relation}} is \texttt{false}.

\texttt{\textbackslash fp\_until\_do:nNn} \hspace{1em} \texttt{\textbackslash fp\_until\_do:nNnn \{\textit{fp expr}_1\} \{\textit{relation}\} \{\textit{fp expr}_2\} \{\textit{code}\}}

Evaluates the relationship between the two \texttt{\textit{floating point expressions}} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \texttt{\textit{code}} in the input stream if the \texttt{\textit{relation}} is \texttt{false}. After the \texttt{\textit{code}} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash fp\_do\_while:nNnn} \hspace{1em} \texttt{\textbackslash fp\_do\_while:nNnn \{\textit{fp expr}_1\} \{\textit{relation}\} \{\textit{fp expr}_2\} \{\textit{code}\}}

Evaluates the relationship between the two \texttt{\textit{floating point expressions}} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \texttt{\textit{code}} in the input stream if the \texttt{\textit{relation}} is \texttt{false}. After the \texttt{\textit{code}} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash fp\_do\_until:nn} \hspace{1em} \texttt{\textbackslash fp\_do\_until:nn \{\textit{fp expr}_1\} \{\textit{relation}\} \{\textit{fp expr}_2\} \{\textit{code}\}}

Places the \texttt{\textit{code}} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \texttt{\textit{floating point expressions}} as described for \texttt{\textbackslash fp\_compare:nNnTF}. If the test is \texttt{false} then the \texttt{\textit{code}} is inserted into the input stream again and a loop occurs until the \texttt{\textit{relation}} is \texttt{true}.

\texttt{\textbackslash fp\_while\_do:nn} \hspace{1em} \texttt{\textbackslash fp\_while\_do:nn \{\textit{fp expr}_1\} \{\textit{relation}\} \{\textit{fp expr}_2\} \{\textit{code}\}}

Places the \texttt{\textit{code}} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \texttt{\textit{floating point expressions}} as described for \texttt{\textbackslash fp\_compare:nNnTF}. If the test is \texttt{true} then the \texttt{\textit{code}} is inserted into the input stream again and a loop occurs until the \texttt{\textit{relation}} is \texttt{false}.

\texttt{\textbackslash fp\_until\_do:nn} \hspace{1em} \texttt{\textbackslash fp\_until\_do:nn \{\textit{fp expr}_1\} \{\textit{relation}\} \{\textit{fp expr}_2\} \{\textit{code}\}}

Evaluates the relationship between the two \texttt{\textit{floating point expressions}} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \texttt{\textit{code}} in the input stream if the \texttt{\textit{relation}} is \texttt{false}. After the \texttt{\textit{code}} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash fp\_while\_do:nn} \hspace{1em} \texttt{\textbackslash fp\_while\_do:nn \{\textit{fp expr}_1\} \{\textit{relation}\} \{\textit{fp expr}_2\} \{\textit{code}\}}

Evaluates the relationship between the two \texttt{\textit{floating point expressions}} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \texttt{\textit{code}} in the input stream if the \texttt{\textit{relation}} is \texttt{true}. After the \texttt{\textit{code}} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.
This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The \langle function \rangle is then placed in front of each \langle value \rangle from the \langle initial value \rangle to the \langle final value \rangle in turn (using \langle step \rangle between each \langle value \rangle). The \langle step \rangle must be non-zero. If the \langle step \rangle is positive, the loop stops when the \langle value \rangle becomes larger than the \langle final value \rangle. If the \langle step \rangle is negative, the loop stops when the \langle value \rangle becomes smaller than the \langle final value \rangle. The \langle function \rangle 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]

\TeX\ hackers note: Due to rounding, it may happen that adding the \langle step \rangle to the \langle value \rangle does not change the \langle value \rangle; such cases give an error, as they would otherwise lead to an infinite loop.

This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each \langle value \rangle from the \langle initial value \rangle to the \langle final value \rangle in turn (using \langle step \rangle between each \langle value \rangle), the \langle code \rangle is inserted into the input stream with \#1 replaced by the current \langle value \rangle. Thus the \langle code \rangle should define a function of one argument (#1).

This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each \langle value \rangle from the \langle initial value \rangle to the \langle final value \rangle in turn (using \langle step \rangle between each \langle value \rangle), the \langle code \rangle is inserted into the input stream, with the \langle \tl\ var \rangle defined as the current \langle value \rangle. Thus the \langle code \rangle should make use of the \langle \tl\ var \rangle.

### 29.6 Symbolic expressions

Floating point expressions support variables: these can only be set locally, so act like standard \l... variables.

\begin{verbatim}
\fp_new_variable:n { A }
\fp_set:Nn \l_tmpb_fp { 1 * \sin(A) + 3**2 }
\fp_show:n \l_tmpb_fp
\fp_set_variable:nn { A } { pi/2 }
\end{verbatim}
defines \( A \) to be a variable, then defines \( \l_{tmpb\_fp} \) to stand for \( 1 \cdot \sin(A) + 9 \) (note that \( 3 \cdot 2 \) is evaluated, but the \( 1 \cdot \) product is not simplified away). Until \( \l_{tmpb\_fp} \) is changed, \( \fp:\text{show}\{\l_{tmpb\_fp}\} \) will show \( ((1 \cdot \sin(A)) + 9) \) regardless of the value of \( A \). The next step defines \( A \) to be equal to \( \pi/2 \): then \( \fp:\text{show}\{\l_{tmpb\_fp}\} \) will evaluate \( \l_{tmpb\_fp} \) and show 10. We then redefine \( A \) to be 0: since \( \l_{tmpb\_fp} \) still stands for \( 1 \cdot \sin(A) + 9 \), the value shown is then 9. Variables can be set with \( \fp:\text{set\_variable}\{\langle\text{identifier}\rangle, \langle\text{fp expr}\rangle\} \) to arbitrary floating point expressions including other variables.

\[ \fp:\text{new\_variable}\{\langle\text{identifier}\rangle\} \]

Declares the \( \langle\text{identifier}\rangle \) as a variable, which allows it to be used in floating point expressions. For instance,

\[
\begin{align*}
\fp:\text{new\_variable}\{\ A \} \\
\fp:\text{show}\{\ A^2 - A + 1 \}
\end{align*}
\]

shows \(((A^2) - A) + 1\). If the declaration was missing, the parser would complain about an “Unknown \( \text{fp word 'A'} \)”.

\( \langle\text{identifier}\rangle \) must consist entirely of Latin letters among \([a-zA-Z]\).

\[ \fp:\text{set\_variable}\{\langle\text{identifier}\rangle, \langle\text{fp expr}\rangle\} \]

Defines the \( \langle\text{identifier}\rangle \) to stand in any further expression for the result of evaluating the \( \langle\text{floating point expression}\rangle \) as much as possible. The result may contain other variables, which are then replaced by their values if they have any. For instance,

\[
\begin{align*}
\fp:\text{new\_variable}\{\ A \} \\
\fp:\text{new\_variable}\{\ B \} \\
\fp:\text{new\_variable}\{\ C \} \\
\fp:\text{set\_variable}\{\ A \} \{\ 3 \} \\
\fp:\text{set\_variable}\{\ C \} \{\ A \times 2 + B \times 1 \} \\
\fp:\text{show}\{\ C + 4 \} \\
\fp:\text{set\_variable}\{\ A \} \{\ 4 \} \\
\fp:\text{show}\{\ C + 4 \}
\end{align*}
\]

shows \(((9 + (B \times 1)) + 4)\) twice: changing the value of \( A \) to 4 does not alter \( C \) because \( A \) was replaced by its value 3 when evaluating \( A \times 2 + B \times 1 \).
\texttt{\textbackslash fp\_clear\_variable:n} \{\texttt{identifier}\}\}

Removes any value given by \texttt{\textbackslash fp\_set\_variable:nn} to the variable with this \texttt{identifier}. For instance,

\begin{verbatim}
\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 }
\end{verbatim}

shows 9, then \((A^2)\).

\section*{29.7 User-defined functions}

It is possible to define new user functions which can be used inside the argument to \texttt{\textbackslash fp\_eval:n}, etc. These functions may take one or more named arguments, and should be implemented using expansion methods only.

\texttt{\textbackslash fp\_new\_function:n} \{\texttt{identifier}\}\}

Declares the \texttt{identifier} as a function, which allows it to be used in floating point expressions. For instance,

\begin{verbatim}
\fp_new_function:n { foo } \fp_show:n { foo ( 1 + 2 , foo(3), A ) ** 2 } }
\end{verbatim}

shows \((\text{foo}(3, \text{foo}(3), A))^{2}\). If the declaration was missing, the parser would complain about an “Unknown fp word ‘foo’”. The \texttt{identifier} must consist entirely of Latin letters \([a-zA-Z]\).

\texttt{\textbackslash fp\_set\_function:nnn} \texttt{\{identifier\} \{\texttt{vars}\} \{\texttt{fp expr}\}}

Defines the \texttt{identifier} to stand in any further expression for the result of evaluating the \texttt{floating point expression}, with the \texttt{identifier} accepting the \texttt{vars} (a non-empty comma-separated list). The result may contain other functions, which are then replaced by their results if they have any. For instance,

\begin{verbatim}
\fp_new_function:n { foo } \fp_set_function:nnn { npow } { a,b } { a**b } \fp_show:n { npow(16,0.25) } }
\end{verbatim}

shows 2. The names of the \texttt{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 \texttt{\textbackslash fp\_new\_variable:n}.

\texttt{\textbackslash fp\_clear\_function:n} \{\texttt{identifier}\}\}

Removes any definition given by \texttt{\textbackslash fp\_set\_function:nnn} to the function with this \texttt{identifier}. 266
29.8 Some useful constants, and scratch variables

\c_zero_fp Zero, with either sign.
\c_minus_zero_fp

New: 2012-05-08

\c_one_fp One as an fp: useful for comparisons in some places.

New: 2012-05-08

\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.

New: 2012-05-08

\c_nan_fp Not a number. This can be input directly in a floating point expression as nan.

New: 2012-05-08

\c_e_fp The value of the base of the natural logarithm, e = exp(1).

Updated: 2012-05-08

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

Updated: 2013-11-17

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

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

29.9 Scratch variables

\l_tmpa_fp \l_tmpb_fp Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_fp \g_tmpb_fp Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.


### 29.10 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as $0 / 0$, or $10 ** 1e9999$. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- **Overflow** occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm \infty$.

- **Underflow** occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in $\pm 0$.

- **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 \TeX3.

To each exception we associate a “flag”: \fp_overflow, \fp_underflow, \fp_invalid_operation, and \fp_division_by_zero. The state of these flags can be tested and modified with commands from \l3flag.

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using \fp_trap:nn) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

\[ \text{\texttt{\textbackslash fp\_trap:nn}\{\langle \text{exception}\rangle\}\{\langle \text{trap type}\rangle\} } \]

All occurrences of the \{\langle exception\rangle\} (overflow, underflow, invalid_operation or division_by_zero) within the current group are treated as \{\langle trap type\rangle\}, which can be

- **none**: the \langle exception\rangle will be entirely ignored, and leave no trace;

- **flag**: the \langle exception\rangle will turn the corresponding flag on when it occurs;

- **error**: additionally, the \langle exception\rangle will halt the \TeX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

\[ \text{\texttt{fp\_overflow}} \]
\[ \text{\texttt{fp\_underflow}} \]
\[ \text{\texttt{fp\_invalid\_operation}} \]
\[ \text{\texttt{fp\_division\_by\_zero}} \]

Flags denoting the occurrence of various floating-point exceptions.
29.11 Viewing floating points

\fp_show:N \fp_show:N \langle \text{fp var} \rangle
\fp_show:c \fp_show:n \{\langle \text{fp expr} \rangle\}

Evaluates the \langle \text{fp expr} \rangle and displays the result in the terminal.

\fp_log:N \fp_log:N \langle \text{fp var} \rangle
\fp_log:c \fp_log:n \{\langle \text{fp expr} \rangle\}

Evaluates the \langle \text{fp expr} \rangle and writes the result in the log file.

29.12 Floating point expressions

29.12.1 Input of floating point numbers

We support four types of floating point numbers:

- \( \pm m \cdot 10^n \), a floating point number, with integer \( 1 \leq m \leq 10^{16} \), and \( -10000 \leq n \leq 10000 \);
- \( \pm 0 \), zero, with a given sign;
- \( \pm \infty \), infinity, with a given sign;
- \text{nan}, is “not a number”, and can be either quiet or signalling (\textit{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:

- \langle \text{sign} \rangle: a possibly empty string of + and - characters;
- \langle \text{significand} \rangle: a non-empty string of digits together with zero or one dot;
- \langle \text{exponent} \rangle 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 \langle \text{sign} \rangle contains an even number of -, and otherwise, hence, an empty \langle \text{sign} \rangle denotes a non-negative input. The stored significand is obtained from \langle \text{significand} \rangle by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input \langle \text{significand} \rangle has at most 16 digits. The stored \langle \text{exponent} \rangle is obtained by combining the input \langle \text{exponent} \rangle (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting \langle \text{exponent} \rangle is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by \( \pm \infty \)), or an underflow (resulting in \( \pm 0 \)).
The result is thus ±0 if and only if \( \langle \text{significand} \rangle \) contains no non-zero digit (i.e., consists only in characters 0, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

The \( \langle \text{significand} \rangle \) must be non-empty, so \( e1 \) and \( e-1 \) are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as \( e \) and should be input as \( \exp(1) \) or \( \text{\textbackslash e}_\text{fp} \) (which is faster).

Special numbers are input as follows:

- `inf` represents \(+\infty\), and can be preceded by any \( \langle \text{sign} \rangle \), yielding \( \pm\infty \) as appropriate.
- `nan` represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a `nan`.
- Note that commands such as `\infty`, `\pi`, or `\sin` do not work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

### 29.12.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`\sin`, `\ln`, etc).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (2pi) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance 3(4+5)).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, etc.
- Logical `\text{\textbackslash and}`, denoted by `&amp;`.
- Logical `\text{\textbackslash or}`, denoted by `\text{\textbar bar}`.
- 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

\[
\frac{1}{2\pi} = \frac{1}{2\pi(\pi + \pi)} \approx 1,
\]

\[
\sin(2\pi) = \sin(2\pi)\pi \neq 0.
\]

\[
2^{-2\max(3,5)} = 2^{-2\max(3,5)} = 20.
\]

\[
1\text{in}/1\text{cm} = (1\text{in})/(1\text{cm}) = 2.54.
\]

Functions are called on the value of their argument, contrarily to \TeX{} macros.

### 29.12.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is \texttt{false} if it is \texttt{±0}, and \texttt{true} otherwise, including when it is \texttt{nan} or a tuple such as \texttt{(0,0)}. Tuples are only supported to some extent by operations that work with truth values (\texttt{?,:}, \texttt{||}, \texttt{&&}, \texttt{!}), by comparisons (\texttt{!<=>?}), and by \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a \texttt{nan} result.

\[
\texttt{?:
}\langle \text{operand} 1 \rangle \langle \text{operand} 2 \rangle : \langle \text{operand} 3 \rangle}
\]

The ternary operator \texttt{?::} results in \texttt{\langle operand2 \rangle} if \texttt{\langle operand1 \rangle} is true (not \texttt{±0}), and \texttt{\langle operand3 \rangle} if \texttt{\langle operand1 \rangle} is false (\texttt{±0}). All three \texttt{\langle operands \rangle} are evaluated in all cases; they may be tuples. The operator is right associative, hence

\[
\texttt{\fp_eval:n}{1 + 3 > 4 \ ? 1 : 2 + 4 > 5 \ ? 2 : 3 + 5 > 6 \ ? 3 : 4}
\]

first tests whether \texttt{1 + 3 > 4}; since this isn’t true, the branch following \texttt{:} is taken, and \texttt{2 + 4 > 5} is compared; since this is true, the branch before \texttt{:} 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.

\[
\texttt{|| \fp_eval:n}{\langle operand1 \rangle \ || \langle operand2 \rangle}
\]

If \texttt{\langle operand1 \rangle} is true (not \texttt{±0}), use that value, otherwise the value of \texttt{\langle operand2 \rangle}. Both \texttt{\langle operands \rangle} are evaluated in all cases; they may be tuples. In \texttt{\langle operand1 \rangle \ || \langle operand2 \rangle \ || \ ... \ || \langle operand_n \rangle}, the first true (nonzero) \texttt{\langle operand \rangle} is used and if all are zero the last one (\texttt{±0}) is used.

\[
\texttt{&& \fp_eval:n}{\langle operand1 \rangle \ && \langle operand2 \rangle}
\]

If \texttt{\langle operand1 \rangle} is false (equal to \texttt{±0}), use that value, otherwise the value of \texttt{\langle operand2 \rangle}. Both \texttt{\langle operands \rangle} are evaluated in all cases; they may be tuples. In \texttt{\langle operand1 \rangle \ && \langle operand2 \rangle \ && \ ... \ && \langle operand_n \rangle}, the first false (\texttt{±0}) \texttt{\langle operand \rangle} is used and if none is zero the last one is used.

271
\[
\begin{align*}
&\text{\texttt{abs}} \ \text{\texttt{\{ \ \texttt{fp expr} \ \}} } \\
&\text{\texttt{\{ \ \texttt{fp expr} \ \}} }
\end{align*}
\]
Computes the absolute value of the \texttt{\{ \ \texttt{fp expr} \ \}}. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also \texttt{\{ \ \texttt{fp_abs:n} \ \}.}
\texttt{\textbf{exp} \:\texttt{\var{fp_eval:n \{ exp( \langle \texttt{fp expr} \rangle ) \}}}

Computes the exponential of the \langle \texttt{fp expr} \rangle. \textit{Underflow} and \textit{overflow} occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

\texttt{\textbf{fact} \:\texttt{\var{fp_eval:n \{ fact( \langle \texttt{fp expr} \rangle ) \}}}

Computes the factorial of the \langle \texttt{fp expr} \rangle. If the \langle \texttt{fp expr} \rangle is an integer between \(-0\) and 3248 included, the result is finite and correctly rounded. Larger positive integers give \(+\infty\) with “overflow”, while \texttt{fact}(+\infty) = +\infty and \texttt{fact(nan)} = \texttt{nan} with no exception. All other inputs give \texttt{nan} with the “invalid operation” exception.

\texttt{\textbf{ln} \:\texttt{\var{fp_eval:n \{ ln( \langle \texttt{fp expr} \rangle ) \}}}

Computes the natural logarithm of the \langle \texttt{fp expr} \rangle. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for \texttt{ln}(\texttt{-0}). “Division by zero” occurs when evaluating \texttt{ln}(+\texttt{0}) = -\infty. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

\texttt{\textbf{logb} \:\texttt{\var{fp_eval:n \{ logb( \langle \texttt{fp expr} \rangle ) \}}}

Determines the exponent of the \langle \texttt{fp expr} \rangle, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating \texttt{logb}(\texttt{\pm0}) = -\infty. Other special values are \texttt{logb}(\texttt{\pm\infty}) = +\infty and \texttt{logb(nan)} = \texttt{nan}. If the operand is a tuple or is \texttt{nan}, then “invalid operation” occurs and the result is \texttt{nan}.

\texttt{\textbf{max} \:\texttt{\var{fp_eval:n \{ max( \langle \texttt{fp expr} \rangle 1, \langle \texttt{fp expr} \rangle 2, \ldots \}) \}}

\texttt{\textbf{min} \:\texttt{\var{fp_eval:n \{ min( \langle \texttt{fp expr} \rangle 1, \langle \texttt{fp expr} \rangle 2, \ldots \}) \}}}

Evaluates each \langle \texttt{fp expr} \rangle and computes the largest (smallest) of those. If any of the \langle \texttt{fp expr} \rangle is a \texttt{nan} or tuple, the result is \texttt{nan}. If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.
Only \texttt{round} accepts a third argument. Evaluates $\langle \text{fp expr}_1 \rangle = x$ and $\langle \text{fp expr}_2 \rangle = n$ and $\langle \text{fp expr}_3 \rangle = t$ 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 $\pm 0, \pm \infty$, or \texttt{nan}; if $n = \texttt{nan}$, this yields \texttt{nan}; if $n$ is neither $\pm \infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle \text{fp expr}_2 \rangle$ is omitted, $n = 0$, i.e., $\langle \text{fp expr}_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.

- \texttt{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 \texttt{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 $\infty/\pm \infty$ is chosen (“ties towards positive/negative infinity”).

- \texttt{floor} yields the largest multiple of $10^{-n}$ smaller or equal to $x$ (“round towards negative infinity”);

- \texttt{ceil} yields the smallest multiple of $10^{-n}$ greater or equal to $x$ (“round towards positive infinity”);

- \texttt{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 $\langle \text{fp expr}_2 \rangle < -9984$). If any operand is a tuple, “invalid operation” occurs.

\texttt{sign} \quad \texttt{fp_eval:n \{ sign( \langle \text{fp expr} \rangle ) \}}

Evaluates the $\langle \text{fp expr} \rangle$ and determines its sign: +1 for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, $\pm 0$ for $\pm 0$, and \texttt{nan} for \texttt{nan}. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

\texttt{sin} \quad \texttt{fp_eval:n \{ \sin( \langle \text{fp expr} \rangle ) \}}

\texttt{cos} \quad \texttt{fp_eval:n \{ \cos( \langle \text{fp expr} \rangle ) \}}

\texttt{tan} \quad \texttt{fp_eval:n \{ \tan( \langle \text{fp expr} \rangle ) \}}

\texttt{cot} \quad \texttt{fp_eval:n \{ \cot( \langle \text{fp expr} \rangle ) \}}

\texttt{csc} \quad \texttt{fp_eval:n \{ \csc( \langle \text{fp expr} \rangle ) \}}

\texttt{sec} \quad \texttt{fp_eval:n \{ \sec( \langle \text{fp expr} \rangle ) \}}

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle \text{fp expr} \rangle$ given in radians. For arguments given in degrees, see \texttt{sind}, \texttt{cosd}, etc. Note that since $\pi$ is irrational, $\sin(8\pi)$ is not quite zero, while its analogue $\sin(8 \times 180)$ is exactly zero. The trigonometric functions are undefined for an argument of $\pm \infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
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 \( \pi \) is irrational, \( \sin(8\pi) \) is not quite zero, while its analogue \( \text{sind}(8 \times 180) \) is exactly zero. The trigonometric functions are undefined for an argument of \( \pm \infty \), leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

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.

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.
Those functions yield an angle in radians: `atan` and `acot` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the \( \langle fp \ expr \rangle \): arctangent takes values in the range \([-\pi/2, \pi/2]\), and arccotangent in the range \([0, \pi]\). The two-argument arccotangent computes the angle in polar coordinates of the point with Cartesian coordinates \((\langle fp \ expr_2 \rangle, \langle fp \ expr_1 \rangle)\): this is the arccotangent of \(\langle fp \ expr_1 \rangle/\langle fp \ expr_2 \rangle\), possibly shifted by \(\pi\) depending on the signs of \(\langle fp \ expr_1 \rangle\) and \(\langle fp \ expr_2 \rangle\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\langle fp \ expr_1 \rangle, \langle fp \ expr_2 \rangle)\), equal to the arc tangent of \(\langle fp \ expr_1 \rangle/\langle fp \ expr_2 \rangle\), possibly shifted by \(\pi\). Both two-argument functions take values in the wider range \([-\pi, \pi]\]. The ratio \(\langle fp \ expr_1 \rangle/\langle fp \ expr_2 \rangle\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm 0\), 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.

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 \( \langle fp \ expr \rangle \): arctangent takes values in the range \([-90, 90]\), and arccotangent in the range \([0, 180]\]. The two-argument arccotangent computes the angle in polar coordinates of the point with Cartesian coordinates \((\langle fp \ expr_2 \rangle, \langle fp \ expr_1 \rangle)\): this is the arccotangent of \(\langle fp \ expr_1 \rangle/\langle fp \ expr_2 \rangle\), possibly shifted by 180 depending on the signs of \(\langle fp \ expr_1 \rangle\) and \(\langle fp \ expr_2 \rangle\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\langle fp \ expr_1 \rangle, \langle fp \ expr_2 \rangle)\), equal to the arccotangent of \(\langle fp \ expr_1 \rangle/\langle fp \ expr_2 \rangle\), possibly shifted by 180. Both two-argument functions take values in the wider range \([-180, 180]\]. The ratio \(\langle fp \ expr_1 \rangle/\langle fp \ expr_2 \rangle\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm 0\), 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.

Computes the square root of the \( \langle fp \ expr \rangle \). The “invalid operation” is raised when the \( \langle fp \ expr \rangle \) is negative or is a tuple; no other exception can occur. Special values yield \(\sqrt{-0} = -0\), \(\sqrt{+0} = +0\), \(\sqrt{\pm \infty} = +\infty\) and \(\sqrt{\text{nan}} = \text{nan}\).
\fp_eval:n { rand() }

New: 2016-12-05

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 Xe\TeX. 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 pdf\TeX, \up\TeX and \uniformdeviate in Lua\TeX and Xe\TeX. 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.

\fp_eval:n { randint(⟨fp expr⟩) }
\fp_eval:n { randint(⟨fp expr⟩1, ⟨fp expr⟩2) }

New: 2016-12-05

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  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 $\pi$ (see \c_pi_fp).

deg  The value of $1^\circ$ in radians (see \c_one_degree_fp).
Those units of measurement are equal to their values in pt, namely

\[
\begin{align*}
1 \text{ in} &= 72.27 \text{ pt} \\
1 \text{ pt} &= 1 \text{ pt} \\
1 \text{ pc} &= 12 \text{ pt} \\
1 \text{ cm} &= \frac{1}{2.54} \text{ in} = 28.45275590551181 \text{ pt} \\
1 \text{ mm} &= \frac{1}{25.4} \text{ in} = 2.845275590551181 \text{ pt} \\
1 \text{ dd} &= 0.376065 \text{ mm} = 1.0700856496063 \text{ pt} \\
1 \text{ cc} &= 12 \text{ dd} = 12.84010277952756 \text{ pt} \\
1 \text{ nd} &= 0.375 \text{ mm} = 1.066978346456693 \text{ pt} \\
1 \text{ nc} &= 12 \text{ nd} = 12.80374015748031 \text{ pt} \\
1 \text{ bp} &= \frac{1}{72} \text{ in} = 1.00375 \text{ pt} \\
1 \text{ sp} &= 2^{-16} \text{ pt} = 1.52587890625 \times 10^{-5} \text{ pt}.
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>em</th>
<th>ex</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{em}</td>
<td>\texttt{ex}</td>
</tr>
</tbody>
</table>

\begin{align*}
\text{true} & \quad \text{false}
\end{align*}

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.

\section*{29.13 Disclaimer and roadmap}

This module may break if the escape character is among \texttt{0123456789_+}, or if it receives a \TeX{} primitive conditional affected by \texttt{\exp_not:N}.

The following need to be done. I'll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.
• Support signalling \texttt{nan}.
• Modulo and remainder, and rounding function \texttt{quantize} (and its friends analogous to \texttt{trunc, ceil, floor}).
• \texttt{\textbackslash fp\_format::nn \{\textbackslash fp \ expr\}\ \{(\textbackslash format)\}}, but what should \texttt{(format)} be? More general pretty printing?
• Add \texttt{and, or, xor}? Perhaps under the names \texttt{all, any, xor}?
• Add $\log(x, b)$ for logarithm of $x$ in base $b$.
• \texttt{hypot} (Euclidean length). Cartesian-to-polar transform.
• Hyperbolic functions \texttt{cosh, sinh, tanh}.
• Inverse hyperbolics.
• Base conversion, input such as \texttt{0xAB.CDEF}.
• Factorial (not with \texttt{!}), gamma function.
• Improve coefficients of the \texttt{sin} and \texttt{tan} series.
• Treat upper and lower case letters identically in identifiers, and ignore underscores.
• Add an \texttt{array(1,2,3)} and \texttt{i=complex(0,1)}.
• Provide an experimental \texttt{map} function? Perhaps easier to implement if it is a single character, \texttt{@sin(1,2)}?
• Provide an \texttt{isnan} function analogue of \texttt{\textbackslash fp\_if\_nan::nTF}?
• Support keyword arguments?

\texttt{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 $-\infty$, \texttt{\textbackslash dim\_to\_fp:n \{0pt\}} should return $-0$, not $+0$.
• The result of $(\pm0) + (\pm0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
• \texttt{0e9999999999} gives a \texttt{\LaTeX} “number too large” error.
• Subnormals are not implemented.

Possible optimizations/improvements.

• Document that \texttt{l3trial/l3fp-types} introduces tools for adding new types.
• In subsection 29.12.1, write a grammar.
• It would be nice if the parse auxiliaries for each operation were set up in the corresponding module, rather than centralizing in l3fp-parse.

• Some functions should get an _o ending to indicate that they expand after their result.

• More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.

• The code for the ternary set of functions is ugly.

• There are many - missing in the doc to avoid bad line-breaks.

• The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking \( c = \frac{2000}{(\lfloor 200x \rfloor + 1)} \in [10, 95] \) 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 \texttt{\_\_fp\_basics\_pack\_weird\_low:NNNNw} and \texttt{\_\_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,\ldots

• Also take into account the “inexact” exception?

• Support multi-character prefix operators (\textit{e.g.}, \_\_ or whatever)?
Chapter 30

The \texttt{l3fparray} module
Fast global floating point arrays

30.1 \texttt{l3fparray} documentation

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast \texttt{l3seq}, where access time is linear). The interface is very close to that of \texttt{l3intarray}. The size of the array is fixed and must be given at point of initialisation

\begin{verbatim}
\fparray_new:Nn \fparray_new:Nn \langle fparray var \rangle \{\langle size \rangle\}
\end{verbatim}
Evaluates the integer expression \texttt{\langle size \rangle} and allocates an \texttt{floating point array variable} with that number of (zero) entries. The variable name should start with \texttt{\g_} because assignments are always global.

\begin{verbatim}
\fparray_count:N \fparray_count:N \langle fparray var \rangle
\end{verbatim}
Expands to the number of entries in the \texttt{floating point array variable}. This is performed in constant time.

\begin{verbatim}
\fparray_gset:Nnn \fparray_gset:Nnn \langle fparray var \rangle \{\langle position \rangle\} \{\langle value \rangle\}
\end{verbatim}
Stores the result of evaluating the floating point expression \texttt{\langle value \rangle} into the \texttt{floating point array variable} at the (integer expression) \texttt{\langle position \rangle}. If the \texttt{\langle position \rangle} is not between 1 and the \texttt{\fparray_count:N}, an error occurs. Assignments are always global.

\begin{verbatim}
\fparray_gzero:N \fparray_gzero:N \langle fparray var \rangle
\end{verbatim}
Sets all entries of the \texttt{floating point array variable} to $+0$. Assignments are always global.
Applies \texttt{fp_use:N} or \texttt{fp_to_tl:N} (respectively) to the floating point entry stored at the (integer expression) \texttt{position} in the \texttt{floating point array variable}. If the \texttt{position} is not between 1 and the \texttt{fparray_count:N}, an error occurs.
Chapter 31

The l3bitset module

Bitsets

This module defines and implements the data type `bitset`, a vector of bits. The size of the vector may grow dynamically. Individual bits can be set and unset by names pointing to an index position. The names 1, 2, 3, ... are predeclared and point to the index positions 1, 2, 3,... More names can be added and existing names can be changed. The index is like all other indices in expl3 modules 1-based. A `bitset` can be output as binary number or—as needed e.g. in a PDF dictionary—as decimal (arabic) number. Currently only a small subset of the functions provided by the `bitset` package are implemented here, mainly the functions needed to use bitsets in PDF dictionaries.

The `bitset` is stored as a string (but one shouldn’t rely on the internal representation) and so the vector size is theoretically unlimited, only restricted by \TeX-memory. But the functions to set and clear bits use integer functions for the index so bitsets can’t be longer than $2^{31} - 1$. The export function \texttt{\textbackslash bitset\_to\_arabic:N} can use functions from the \texttt{int} module only if the largest index used for this bitset is smaller than 32, for longer bitsets \texttt{fp} is used and this is slower.
31.1 Creating bitsets

\bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:C \bitset_new:cn
\bitset_new:cn
{ (name_1) = (index_1) ,
  (name_2) = (index_2) , ... }
\bitset_new:cn
\bitset_addto_named_index:N \bitset_addto_named_index:N \bitset_addto_named_index:cn
\bitset_addto_named_index:cn
\bitset_addto_named_index:cn
\bitset_addto_named_index:cn
\bitset_addto_named_index:cn
\bitset_addto_named_index:cn
\bitset_addto_named_index:cn
{ (name_1) = (index_1) ,
  (name_2) = (index_2) , ... }
\bitset_addto_named_index:cn
\bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N
{ Invisible = 1,
  Hidden = 2,
  Print = 3,
  NoZoom = 4,
  NoRotate = 5,
  NoView = 6,
  ReadOnly = 7,
  Locked = 8,
  ToggleNoView = 9,
  LockedContents = 10 }
\bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N \bitset_new:N
{ Invisible = 1,
  Hidden = 2,
  Print = 3,
  NoZoom = 4,
  NoRotate = 5,
  NoView = 6,
  ReadOnly = 7,
  Locked = 8,
  ToggleNoView = 9,
  LockedContents = 10 }

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.
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:N TF ⋆
\bitset_if_exist:c TF ⋆

Tests whether the \texttt{(bitset var)} exist.

31.2 Setting and unsetting bits

\bitset_set_true:Nn \bitset_set_true:cn \bitset_gset_true:Nn \bitset_gset_true:cn
New: 2023-11-15

\bitset_set_false:Nn \bitset_set_false:cn \bitset_gset_false:Nn \bitset_gset_false:cn
New: 2023-11-15

\bitset_clear:N \bitset_clear:c \bitset_gclear:N \bitset_gclear:c
New: 2023-11-15

31.3 Using bitsets

\bitset_item:Nn ⋆ \bitset_item:cn ⋆
New: 2023-11-15

\bitset_item:Nn \bitset_item:cn outputs 1 if the bit with the index number represented by \texttt{(name)} is set and 0 otherwise. \texttt{(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:N \bitset_to_bin:c
\bitset_to_bin:⋆ \bitset_to_bin:c ⋆ New: 2023-11-15

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:N \bitset_to_arabic:c
\bitset_to_arabic:⋆ \bitset_to_arabic:c ⋆ New: 2023-11-15

This leaves the current value of the bitset expressed as a decimal number in the input stream. If no bit has been set yet, the output is zero. The function uses \int_from_-bin:n if the largest index that have been set or unset is smaller than 32, and a slower implementation based on \fp_eval:n otherwise.

\bitset_show:N \bitset_show:N \bitset_show:c
\bitset_show:⋆ \bitset_show:c ⋆ New: 2023-11-15

Displays the binary and decimal values of the \bitset var on the terminal.

\bitset_log:N \bitset_log:N \bitset_log:c
\bitset_log:⋆ \bitset_log:c ⋆ New: 2023-11-15

Writes the binary and decimal values of the \bitset var in the log file.

\bitset_show_named_index:N \bitset_show_named_index:N \bitset_show_named_index:c
\bitset_show_named_index:⋆ \bitset_show_named_index:c ⋆ New: 2023-11-15

Displays declared name–index pairs of the \bitset var on the terminal.

\bitset_log_named_index:N \bitset_log_named_index:N \bitset_log_named_index:c
\bitset_log_named_index:⋆ \bitset_log_named_index:c ⋆ New: 2023-12-11

Writes declared name–index pairs of the \bitset var in the log file.
Chapter 32

The `l3cctab` module

Category code tables

A category code table enables rapid switching of all category codes in one operation. For LuaTeX, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables. The implementation of category code tables in expl3 also saves and restores the \texttt{\endlinechar} primitive value, meaning they could be used for example to implement `\ExplSyntaxOn`.

32.1 Creating and initialising category code tables

\begin{verbatim}
\cctab_new:N \cctab_new:N \texttt{(category code table)}
\cctab_new:N \cctab_new:c
\end{verbatim}

Updates: 2020-07-02

Creating a new `(category code table)` variable or raises an error if the name is already taken. The declaration is global. The `(category code table)` is initialised with the codes as used by iniTeX.

\begin{verbatim}
\cctab_const:Nn \cctab_const:N \texttt{(category code table)} \{\texttt{(category code set up)}\}
\cctab_const:Nn \cctab_const:cn
\end{verbatim}

Updates: 2020-07-07

Creating 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`.

\begin{verbatim}
\cctab_gset:Nn \cctab_gset:N \texttt{(category code table)} \{\texttt{(category code set up)}\}
\cctab_gset:Nn \cctab_gset:cn
\end{verbatim}

Updates: 2020-07-07

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`.

\begin{verbatim}
\cctab_gsave_current:N \cctab_gsave_current:N \texttt{(category code table)}
\cctab_gsave_current:N \cctab_gsave_current:c
\end{verbatim}

New: 2023-05-26

Saves the current prevailing category codes in the `(category code table)`.
32.2 Using category code tables

\cctab_begin:N (category code table)
\cctab_begin:c
Switches locally the category codes in force to those stored in the \textit{category code table}. The prevailing codes before the function is called are added to a stack, for use with \texttt{\cctab_end:}. This function does not start a \TeX group.

\cctab_end:
\cctab_end:
Ends the scope of a \textit{category code table} started using \texttt{\cctab_begin:N}, returning the codes to those in force before the matching \texttt{\cctab_begin:N} was used. This must be used within the same \TeX group (and at the same \TeX group level) as the matching \texttt{\cctab_begin:N}.

\cctab_select:N
\cctab_select:c
Selects the \textit{category code table} for the scope of the current group. This is in particular useful in the \textit{setup} arguments of \texttt{\tl_set_rescan:Nnn}, \texttt{\tl_rescan:nn}, \texttt{\cctab-_const:Nn}, and \texttt{\cctab_gset:Nn}.

\cctab_item:Nn \{\langle int expr\rangle\}
\cctab_item:cn \{\langle int expr\rangle\}
Determines the \textit{character} with character code given by the \textit{int expr} and expands to its category code specified by the \textit{category code table}.

32.3 Category code table conditionals

\cctab_if_exist_p:N \{\langle category code table\rangle\}
\cctab_if_exist_p:c
\cctab_if_exist:NTF \{\langle category code table\rangle\} \{\langle true code\rangle\} \{\langle false code\rangle\}
Tests whether the \textit{category code table} is currently defined. This does not check that the \textit{category code table} really is a category code table.

32.4 Constant and scratch category code tables

\c_code_cctab
Updated: 2020-07-10
Category code table for the expl3 code environment; this does not include \texttt{\&}, which is retained as an “other” character. Sets the \texttt{\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 pdf\LaTeX only. No babel shorthands will be activated. Sets the \texttt{\endlinechar} value to 13 (normal line ending).
\c_inix_cctab

Category code table as set up by iniTeX.

Updated: 2020-07-02

\c_other_cctab

Category code table where all characters have category code 12 (other). Sets the \endlinechar value to $-1$.

Updated: 2020-07-02

\c_str_cctab

Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space). Sets the \endlinechar value to $-1$.

Updated: 2020-07-02

\g_tmpa_cctab
\g_tmpb_cctab

Scratch category code tables.

New: 2023-05-26
Part V

Text manipulation
Chapter 33

The \texttt{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 \texttt{(codepoints)} and are designed to give usable results with both Unicode-aware and 8-bit engines.
Generates one or more character tokens representing the \codepoint{\langle codepoint \rangle}. With Unicode engines, exactly one character token will be generated, and this will have the \catcode{\langle catcode \rangle} 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{\langle codepoint \rangle}. 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{\langle catcode \rangle} will be used. To allow the result of this function to be used inside a expansion context, the result is protected by \exp_not:n.

\TeXhackers note: Users of \upTeX note that these engines are treated as 8-bit in this context. In particular, for \upTeX, irrespective of the \catcode of the \codepoint{\langle codepoint \rangle}, any value outside the ASCII range will result in a series of active bytes being generated.

\codepoint_str_generate:n \codepoint_str_generate:n \langle codepoint \rangle

Generates one or more character tokens representing the \codepoint{\langle codepoint \rangle}. 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{\langle codepoint \rangle}. 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)}

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 \textit{L}, for example \textit{Lu} (uppercase letter), \textit{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

\new: 2023-06-19

\codepoint_to_nfd:n \codepoint_to_nfd:n {(codepoint)}

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.

\new: 2022-10-09
Chapter 34

The l3text module
Text processing

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the \texttt{<text>} are normalized and become \{ and \}, respectively.

34.1 Expanding text

\begin{verbatim}
\texttt{\texttt{text\_expand:n} \{ \texttt{text}\}}
\end{verbatim}

Takes user input \texttt{<text>} and expands the content. Protected commands (typically formatting) are left in place, and no processing takes place of math mode material (as delimited by pairs given in \texttt{\l_text_math_delims_tl} or as the argument to commands listed in \texttt{\l_text_math_arg_tl}). Commands which are neither engine- nor \LaTeX\ protected are expanded exhaustively. Any commands listed in \texttt{\l_text_expand_exclude_tl} are excluded from expansion, as are those in \texttt{\l_text_case_exclude_arg_tl} and \texttt{\l_text_math_arg_tl}.

\begin{verbatim}
\texttt{\texttt{text\_declare\_expand\_equivalent:Nn \texttt{\{ \texttt{cmd} \}} \{ \texttt{replacement}\}}}
\end{verbatim}

\begin{verbatim}
\texttt{\texttt{text\_declare\_expand\_equivalent:cn}}
\end{verbatim}

Declares that the \texttt{<replacement>} tokens should be used whenever the \texttt{<cmd>} (a single token) is encountered. The \texttt{<replacement>} tokens should be expandable. A token can be “replaced” by itself if the defined replacement wraps it in \texttt{\exp_not:n}, for example

\begin{verbatim}
\texttt{\texttt{text\_declare\_expand\_equivalent:Nn \'{ } \{ \texttt{exp\_not:n} \{ \texttt{'} \} \}}
\end{verbatim}
34.2 Case changing

\text_lowercase:n * \text_uppercase:n \{\langle tokens\rangle\}
\text_uppercase:nn * \text_uppercase:nn \{\langle BCP-47\rangle\} \{\langle tokens\rangle\}
\text_titlecase_all:n * \text_uppercase:nn
\text_titlecase_first:n * \text_lowercase:nn
\text_uppercase:nn * \text_titlecase_all:nn
\text_titlecase_first:nn

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 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 \text{ij} in Dutch which becomes \text{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 \text{l3str} module and discussion there of \text{str_lowercase:n}, \text{str_uppercase:n} and \text{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 \text{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 \text{pTHX}, 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 (\text{hy} and \text{hy-x-yiwn}) The setting \text{hy} maps the codepoint U+0587, the ligature of letters ech and yiw, to the codepoints for capital ech and vew when upercasing: this follows the spelling reform which is used in Armenia. The alternative \text{hy-x-yiwn} maps U+0587 to capital ech and yiw on upercasing (also the output if Armenian is not selected at all).

- Azeri and Turkish (\text{az} and \text{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 (\text{de-x-eszett}). An alternative mapping for German in which the lowercase \text{Eszett} maps to a \text{großes Eszett}.
• Greek (el). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. A variant \textit{el-x-iota} is available which converts the \textit{ypogegrammeni} (subscript muted iota) to capital iota when uppercasing: the standard version retains the subscript versions.

• Lithuanian (lt). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that only the accents used in Lithuanian are covered: the behaviour of other accents are not modified.

• Medieval Latin (la-x-medieval). The characters u and V are interchanged on case changing.

• Dutch (nl). Capitalisation of ij at the beginning of titlecased input produces IJ rather than IJ.

Determining whether non-letter characters at the start of text should count as the uppercase element is controllable. When \texttt{\_text_titlecase\_check\_letter\_bool} is \texttt{true}, codepoints which are not letters (Unicode general category \textit{L}) are not changed, and only the first letter is uppercased. When \texttt{\_text_titlecase\_check\_letter\_bool} is \texttt{false}, the first codepoint is uppercased, irrespective of the general code of the character.

\begin{verbatim}
\text_declare_case_equivalent:Nn \text_declare_case_equivalent:cn
\text_declare_lowercase_mapping:nn
\text_declare_lowercase_mapping:nnn
\text_declare_titlecase_mapping:nn
\text_declare_uppercase_mapping:nn
\text_declare_uppercase_mapping:nnn
\end{verbatim}

\texttt{New: 2022-07-04}

Declares that the \texttt{(replacement)} tokens should be used whenever the \texttt{(cmd)} (a single token) is encountered during case changing.

\begin{verbatim}
\text_declare_lower_mapping:nn
\text_declare_lower_mapping:nnn
\text_declare_titlecase_mapping:nnn
\text_declare_uppercase_mapping:nn
\text_declare_uppercase_mapping:nnn
\end{verbatim}

\texttt{Nov: 2023-04-11}

\texttt{Updated: 2023-04-20}

Declares that the \texttt{(replacement)} tokens should be used when case mapping the \texttt{(codepoint)}, rather than the standard mapping given in the Unicode data files. The \texttt{nnn} version takes a BCP-47 tag, which can be used to specify that the customisation only applies to that locale.

\begin{verbatim}
\text_case_switch:nnnn
\end{verbatim}

\texttt{New: 2022-07-04}

Context-sensitive function which will expand to one of the \texttt{(normal)}, \texttt{(upper)}, \texttt{(lower)} or \texttt{(title)} tokens depending on the current case changing operation. Outside of case changing, the \texttt{(normal)} tokens are produced. Within case changing, the appropriate mapping tokens are inserted.
34.3 Removing formatting from text

\text_purify:n * \text_purify:n \{⟨text⟩}\}

New: 2020-03-05
Updated: 2020-05-14

Takes user input \langle 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 \langle 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 \text_declare_purify_equivalent:Nn \langle cmd⟩ \{⟨replacement⟩\}
\text_declare_purify_equivalent:Nn \text_declare_purify_equivalent:Nn

New: 2020-03-05

Declares that the \langle replacement⟩ tokens should be used whenever the \langle cmd⟩ (a single token) is encountered. The \langle replacement⟩ tokens should be expandable.

34.4 Control variables

\l_text_math_arg_tl Lists commands present in the \langle text⟩ where the argument of the command should be treated as math mode material. The treatment here is similar to \l_text_math_delims_tl but for a command rather than paired delimiters.

\l_text_math_delims_tl Lists pairs of tokens which delimit (in-line) math mode content; such content may be excluded from processing.

\l_text_case_exclude_arg_tl Lists commands where the first mandatory argument is excluded from case changing.

\l_text_expand_exclude_tl Lists commands which are excluded from expansion. This protection includes everything up to and including their first braced argument.

\l_text_titlecase_check_letter_bool Controls how the start of titlecasing is handled: when true, the first letter in text is considered. The standard setting is true.
34.5 Mapping to graphemes

Grapheme splitting is implemented using the algorithm described in Unicode Standard Annex #29. This includes support for extended grapheme clusters. Text starting with a line feed or carriage return character will drop this due to standard TeX processing. At present extended pictograms are not supported: these may be added in a future release.

\text_map_function:nN \text_map_function:nN (text) {\langle function\rangle}

Takes user input \langle text\rangle and expands as described for \text_expand:n, then maps over the graphemes within the result, passing each grapheme to the \langle function\rangle. 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 \langle function\rangle should accept one argument as \langle balanced text\rangle: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also \text_map_inline:nn.

\text_map_inline:nn \text_map_inline:nn (text) {\langle inline function\rangle}

Takes user input \langle text\rangle and expands as described for \text_expand:n, then maps over the graphemes within the result, passing each grapheme to the \langle inline function\rangle. 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 \langle inline function\rangle should consist of code which receives the grapheme as \langle balanced text\rangle: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also \text_map_function:nN.

\text_map_break: \text_map_break: \text_map_break:n \text_map_break:n {\langle code\rangle}

Used to terminate a \text_map... function before all entries in the \langle text\rangle have been processed. This normally takes place within a conditional statement.
Part VI
Typesetting
Chapter 35

The l3box module

Boxes

Box variables contain typeset material that can be inserted on the page or in other boxes. Their contents cannot be converted back to lists of tokens. There are three kinds of box operations: horizontal mode denoted with prefix \hbox_, vertical mode with prefix \vbox_, and the generic operations working in both modes with prefix \box_. For instance, a new box variable containing the words “Hello, world!” (in a horizontal box) can be obtained by the following code.

\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, ~ world! }

The argument is typeset inside a \TeX group so that any variables assigned during the construction of this box restores its value afterwards.

Box variables from l3box are compatible with those of \LaTeX{} and plain \TeX{} and can be used interchangeably. The l3box commands to construct boxes, such as \hbox:n or \hbox_set:Nn, are “color-safe”, meaning that

\hbox:n { \color_select:n { blue } Hello, } ~ world!

will result in “Hello,” taking the color blue, but “world!” remaining with the prevailing color outside the box.

35.1 Creating and initialising boxes

\box_new:N \box_new:c

Creates a new \l{box} or raises an error if the name is already taken. The declaration is global. The \l{box} is initially void.

\box_clear:N \box_clear:c \box_gclear:N \box_gclear:c

Clears the content of the \l{box} by setting the box equal to \c_empty_box.
\box_new:N \box_gclear_new:N \box_(g)clear_new:N
Ensures that the \texttt{box} exists globally by applying \texttt{\box_new:N} if necessary, then applies \texttt{\box_(g)clear:N} to leave the \texttt{box} empty.

\box_set_eq:NN \box_set_eq:c \box_gset_eq:NN \box_gset_eq:c
Sets the content of \texttt{box1} equal to that of \texttt{box2}.

\box_if_exist_p:N \box_if_exist:NTF \box_if_exist:c \box_if_exist_p:c
Tests whether the \texttt{box} is currently defined. This does not check that the \texttt{box} really is a box.

New: 2012-03-03

35.2 Using boxes

\box_use:N \box_use:c
Inserts the current content of the \texttt{box} onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

\texttt{\TeX} hackers note: This is the \TeX primitive \texttt{\copy}.

\box_move_right:nn \box_move_left:nn
This function operates in vertical mode, and inserts the material specified by the \texttt{box function} such that its reference point is displaced horizontally by the given \texttt{dim expr} from the reference point for typesetting, to the right or left as appropriate. The \texttt{box function} should be a box operation such as \texttt{\box_use:N \textbackslash box} or a “raw” box specification such as \texttt{\vbox:n \{ \texttt{xyz} \}}.

\box_move_up:nn \box_move_down:nn
This function operates in horizontal mode, and inserts the material specified by the \texttt{box function} such that its reference point is displaced vertically by the given \texttt{dim expr} from the reference point for typesetting, up or down as appropriate. The \texttt{box function} should be a box operation such as \texttt{\box_use:N \textbackslash box} or a “raw” box specification such as \texttt{\vbox:n \{} \texttt{xyz} \}.
35.3 Measuring and setting box dimensions

\box_dp:N\box_dp:c\box_dp:N \langle box \rangle
Calculates the depth (below the baseline) of the \langle box \rangle in a form suitable for use in a \langle dim expr \rangle.

\texttt{\TeX\hackers note:} This is the \TeX\ primitive \texttt{\dp}.

\box_ht:N\box_ht:c\box_ht:N \langle box \rangle
Calculates the height (above the baseline) of the \langle box \rangle in a form suitable for use in a \langle dim expr \rangle.

\texttt{\TeX\hackers note:} This is the \TeX\ primitive \texttt{\ht}.

\box_wd:N\box_wd:c\box_wd:N \langle box \rangle
Calculates the width of the \langle box \rangle in a form suitable for use in a \langle dim expr \rangle.

\texttt{\TeX\hackers note:} This is the \TeX\ primitive \texttt{\wd}.

\box_ht_plus_dp:N\box_ht_plus_dp:c\box_ht_plus_dp:N \langle box \rangle
Calculates the total vertical size (height plus depth) of the \langle box \rangle in a form suitable for use in a \langle dim expr \rangle.

\texttt{Rev: 2021-06-06}
35.4 Box conditionals

\texttt{\box_if_empty_p:N} \star \texttt{\box_if_empty_p:c} \star \texttt{\box_if_empty:N} \star \texttt{\box_if_empty:c} \star

Tests if \(\langle\text{box}\rangle\) is an empty (equal to \texttt{\c_empty_box}).

\texttt{\box_if_horizontal_p:N} \star \texttt{\box_if_horizontal_p:c} \star \texttt{\box_if_horizontal:N} \star \texttt{\box_if_horizontal:c} \star

Tests if \(\langle\text{box}\rangle\) is a horizontal box.

\texttt{\box_if_vertical_p:N} \star \texttt{\box_if_vertical_p:c} \star \texttt{\box_if_vertical:N} \star \texttt{\box_if_vertical:c} \star

Tests if \(\langle\text{box}\rangle\) is a vertical box.

35.5 The last box inserted

\texttt{\box_set_to_last:N} \star \texttt{\box_set_to_last:c} \star \texttt{\box_gset_to_last:N} \star \texttt{\box_gset_to_last:c} \star

Sets the \(\langle\text{box}\rangle\) 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 \(\langle\text{box}\rangle\) is always void as it is not possible to recover the last added item.

35.6 Constant boxes

\texttt{\c_empty_box} \star

This is a permanently empty box, which is neither set as horizontal nor vertical.

\texttt{\TeXhackers note: At the \TeX level this is a void box.}

35.7 Scratch boxes

\texttt{\l_tmpa_box} \star \texttt{\l_tmpb_box} \star

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.

\texttt{\g_tmpa_box} \star \texttt{\g_tmpb_box} \star

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

\box_show:N \box_show:N \box_show:c
Shows full details of the content of the \texttt{box} in the terminal.

\box_show:N \box_show:N \box_show:c
\box_show:N \box_show:N
Updated: 2012-05-11

\box_log:N \box_log:N \box_log:c
Writes full details of the content of the \texttt{box} to the log.

\box_log:N \box_log:N \box_log:c
\box_log:N \box_log:c
Updated: 2012-05-11

35.9 Boxes and color

All \LaTeX{} boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

35.10 Horizontal mode boxes

\hbox:n \hbox:n \hbox:n \hbox:n \hbox:n
Typesets the \texttt{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 \hbox_to_wd:nn \hbox_to_wd:nn \hbox_to_wd:nn
Typesets the \texttt{contents} into a horizontal box of width \texttt{dim expr} and then includes this box in the current list for typesetting.

\hbox_to_zero:n \hbox_to_zero:n \hbox_to_zero:n \hbox_to_zero:n \hbox_to_zero:n
Typesets the \texttt{contents} into a horizontal box of zero width and then includes this box in the current list for typesetting.

\hbox_set:N \hbox_set:N \hbox_set:N \hbox_set:N \hbox_set:N
\hbox_set:C \hbox_set:C \hbox_set:C \hbox_set:C \hbox_set:C
\hbox_gset:N \hbox_gset:N \hbox_gset:N \hbox_gset:N \hbox_gset:N
\hbox_gset:C \hbox_gset:C \hbox_gset:C \hbox_gset:C \hbox_gset:C
Updated: 2017-04-05

Updated: 2017-04-05

Updated: 2017-04-05

Updated: 2017-04-05

Updated: 2017-04-05

304
\hbox_set_to_wd:Nnn \hbox_set_to_wd:cnm \hbox_gset_to_wd:Nnn \hbox_gset_to_wd:cnm

Typesets the ⟨contents⟩ to the width given by the ⟨dim expr⟩ and then stores the result inside the ⟨box⟩.

\hbox_overlap_center:n \hbox_overlap_center:n

New: 2020-08-25

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

Updated: 2017-04-05

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

Updated: 2017-04-05

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_set:Nw \hbox_set:cw \hbox_set:end:
\hbox_gset:Nw \hbox_gset:cw \hbox_gset:end:
\hbox_gset:Nw \hbox_gset:cw \hbox_gset:end:

Updated: 2017-04-05

\hbox_set_to_wd:Nnw \hbox_set_to_wd:cnw \hbox_gset_to_wd:Nnw \hbox_gset_to_wd:cnw

New: 2017-06-08

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

\hbox_unpack:N \hbox_unpack:c

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

\TeXhackers note: This is the \TeX primitive \unhcopy.

35.11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are
_top boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

\vbox:n {\langle contents\rangle}

Typesets the \langle contents\rangle into a vertical box of natural height and includes this box in the current list for typesetting.

\vbox_top:n {\langle contents\rangle}

Typesets the \langle contents\rangle 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_h:nn {\langle dim expr\rangle} {\langle contents\rangle}

Typesets the \langle contents\rangle into a vertical box of height \langle dim expr\rangle and then includes this box in the current list for typesetting.

\vbox_to_zero:n {\langle contents\rangle}

Typesets the \langle contents\rangle into a vertical box of zero height and then includes this box in the current list for typesetting.

\vbox_set:Nn \langle box\rangle \{\langle contents\rangle\}

Typesets the \langle contents\rangle at natural height and then stores the result inside the \langle box\rangle.

\vbox_set_top:Nn \langle box\rangle \{\langle contents\rangle\}

Typesets the \langle contents\rangle at natural height and then stores the result inside the \langle box\rangle. The baseline of the box is equal to that of the first item added to the box.

\vbox_set_to_h:nn \langle box\rangle \{(dim expr)\} \{\langle contents\rangle\}

Typesets the \langle contents\rangle to the height given by the \langle dim expr\rangle and then stores the result inside the \langle box\rangle.

\vbox_set:Nw \langle box\rangle \langle contents\rangle \vbox_set_end:

Typesets the \langle contents\rangle at natural height and then stores the result inside the \langle box\rangle. In contrast to \vbox_set:Nn this function does not absorb the argument when finding the \langle content\rangle, and so can be used in circumstances where the \langle content\rangle may not be a simple argument.
Typesets the \textlangle contents\rangle to the height given by the \textlangle dim expr\rangle and then stores the result inside the \textlangle box\rangle. In contrast to \texttt{vbox\_set\_to\_ht:Nnn} this function does not absorb the argument when finding the \textlangle content\rangle, and so can be used in circumstances where the \textlangle content\rangle may not be a simple argument.

Sets \textlangle box_1\rangle to contain material to the height given by the \textlangle dim expr\rangle by removing content from the top of \textlangle box_2\rangle (which must be a vertical box).

Unpacks the content of the vertical \textlangle box\rangle, retaining any stretching or shrinking applied when the \textlangle box\rangle was set.

\texttt{TEXhacker note:} This is the \TeX{} primitive \texttt{unvcopy}.

### 35.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other \texttt{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 \emph{where the box was originally set} rather than necessarily \emph{at the current group level}. For example, with

\begin{verbatim}
\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
\end{verbatim}

the first use of \texttt{\box\_show:N} will show an entirely cleared (void) box, and the second will show the letter \texttt{A} in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of drop functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.
Inserts the current content of the \texttt{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.

\textbf{\TeX{\kern-.1667em}hackers note: This is the \TeX{} primitive \texttt{\box{}}.}

\texttt{\box\_set\_eq\_drop:NN \langle box1 \rangle \langle box2 \rangle}

Sets the content of \langle box1 \rangle equal to that of \langle box2 \rangle, then drops \langle box2 \rangle.

\texttt{\box\_set\_eq\_drop:(cN|Nc|cc)}

\texttt{New: 2019-01-17}

\texttt{\box\_gset\_eq\_drop:NN \langle box1 \rangle \langle box2 \rangle}

Sets the content of \langle box1 \rangle globally equal to that of \langle box2 \rangle, then drops \langle box2 \rangle.

\texttt{\box\_gset\_eq\_drop:(cN|Nc|cc)}

\texttt{New: 2019-01-17}

\texttt{\hbox\_unpack\_drop:NN \langle box \rangle}

Unpacks the content of the horizontal \langle box \rangle, retaining any stretching or shrinking applied when the \langle box \rangle was set. The original \langle box \rangle is then dropped.

\textbf{\TeX{\kern-.1667em}hackers note: This is the \TeX{} primitive \texttt{\unhbox{}}.}

\texttt{\vbox\_unpack\_drop:NN \langle box \rangle}

Unpacks the content of the vertical \langle box \rangle, retaining any stretching or shrinking applied when the \langle box \rangle was set. The original \langle box \rangle is then dropped.

\textbf{\TeX{\kern-.1667em}hackers note: This is the \TeX{} primitive \texttt{\unvbox{}}.}

\subsection*{35.13 Affine transformations}

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in \TeX{} by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.
Resizes the \langle box\rangle to fit within the given \langle x-size\rangle (horizontally) and \langle y-size\rangle (vertically); both of the sizes are dimension expressions. The \langle y-size\rangle is the height only: it does not include any depth. The updated \langle box\rangle is an hbox, irrespective of the nature of the \langle box\rangle before the resizing is applied. The final size of the \langle box\rangle is the smaller of \{\langle x-size\rangle\} and \{\langle y-size\rangle\}, i.e. the result fits within the dimensions specified. Negative sizes cause the material in the \langle box\rangle to be reversed in direction, but the reference point of the \langle box\rangle is unchanged. Thus a negative \langle y-size\rangle results in the \langle box\rangle having a depth dependent on the height of the original and vice versa.

Resizes the \langle box\rangle to fit within the given \langle x-size\rangle (horizontally) and \langle y-size\rangle (vertically); both of the sizes are dimension expressions. The \langle y-size\rangle is the total vertical size (height plus depth). The updated \langle box\rangle is an hbox, irrespective of the nature of the \langle box\rangle before the resizing is applied. The final size of the \langle box\rangle is the smaller of \{\langle x-size\rangle\} and \{\langle y-size\rangle\}, i.e. the result fits within the dimensions specified. Negative sizes cause the material in the \langle box\rangle to be reversed in direction, but the reference point of the \langle box\rangle is unchanged. Thus a negative \langle y-size\rangle results in the \langle box\rangle having a depth dependent on the height of the original and vice versa.

Resizes the \langle box\rangle to \langle y-size\rangle (vertically), scaling the horizontal size by the same amount; \langle y-size\rangle is a dimension expression. The \langle y-size\rangle is the height only: it does not include any depth. The updated \langle box\rangle is an hbox, irrespective of the nature of the \langle box\rangle before the resizing is applied. A negative \langle y-size\rangle causes the material in the \langle box\rangle to be reversed in direction, but the reference point of the \langle box\rangle is unchanged. Thus a negative \langle y-size\rangle results in the \langle box\rangle having a depth dependent on the height of the original and vice versa.
\texttt{\textbackslash box\_resize\_to\_ht\_plus\_dp:Nn} \texttt{\textbackslash box\_resize\_to\_ht\_plus\_dp:cn} \texttt{\textbackslash box\_gresize\_to\_ht\_plus\_dp:Nn} \texttt{\textbackslash box\_gresize\_to\_ht\_plus\_dp:cn}

Resizes the \texttt{\langle box\rangle} to \texttt{(y-size)} (vertically), scaling the horizontal size by the same amount: \texttt{(y-size)} is a dimension expression. The \texttt{(y-size)} is the total vertical size (height plus depth). The updated \texttt{\langle box\rangle} is an hbox, irrespective of the nature of the \texttt{\langle box\rangle} before the resizing is applied. A negative \texttt{(y-size)} causes the material in the \texttt{\langle box\rangle} to be reversed in direction, but the reference point of the \texttt{\langle box\rangle} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\langle box\rangle} having a depth dependent on the height of the original and \textit{vice versa}.

\texttt{\textbackslash box\_resize\_to\_wd:Nn} \texttt{\textbackslash box\_resize\_to\_wd:cn} \texttt{\textbackslash box\_gresize\_to\_wd:Nn} \texttt{\textbackslash box\_gresize\_to\_wd:cn}

Resizes the \texttt{\langle box\rangle} to \texttt{(x-size)} (horizontally), scaling the vertical size by the same amount: \texttt{(x-size)} is a dimension expression. The updated \texttt{\langle box\rangle} is an hbox, irrespective of the nature of the \texttt{\langle box\rangle} before the resizing is applied. A negative \texttt{(x-size)} causes the material in the \texttt{\langle box\rangle} to be reversed in direction, but the reference point of the \texttt{\langle box\rangle} is unchanged. Thus a negative \texttt{(x-size)} results in the \texttt{\langle box\rangle} having a depth dependent on the height of the original and \textit{vice versa}.

\texttt{\textbackslash box\_resize\_to\_wd\_and\_ht:Nnn} \texttt{\textbackslash box\_resize\_to\_wd\_and\_ht:cnn} \texttt{\textbackslash box\_gresize\_to\_wd\_and\_ht:Nnn} \texttt{\textbackslash box\_gresize\_to\_wd\_and\_ht:cnn}

Resizes the \texttt{\langle box\rangle} to \texttt{(x-size)} (horizontally) and \texttt{(y-size)} (vertically): both of the sizes are dimension expressions. The \texttt{(y-size)} is the height only and does not include any depth. The updated \texttt{\langle box\rangle} is an hbox, irrespective of the nature of the \texttt{\langle box\rangle} before the resizing is applied. Negative sizes cause the material in the \texttt{\langle box\rangle} to be reversed in direction, but the reference point of the \texttt{\langle box\rangle} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\langle box\rangle} having a depth dependent on the height of the original and \textit{vice versa}.

\texttt{\textbackslash box\_resize\_to\_wd\_and\_ht\_plus\_dp:Nnn} \texttt{\textbackslash box\_resize\_to\_wd\_and\_ht\_plus\_dp:cnn} \texttt{\textbackslash box\_gresize\_to\_wd\_and\_ht\_plus\_dp:Nnn} \texttt{\textbackslash box\_gresize\_to\_wd\_and\_ht\_plus\_dp:cnn}

Resizes the \texttt{\langle box\rangle} to \texttt{(x-size)} (horizontally) and \texttt{(y-size)} (vertically): both of the sizes are dimension expressions. The \texttt{(y-size)} is the total vertical size (height plus depth). The updated \texttt{\langle box\rangle} is an hbox, irrespective of the nature of the \texttt{\langle box\rangle} before the resizing is applied. Negative sizes cause the material in the \texttt{\langle box\rangle} to be reversed in direction, but the reference point of the \texttt{\langle box\rangle} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\langle box\rangle} having a depth dependent on the height of the original and \textit{vice versa}.
Rotates the \langle box \rangle by \langle angle \rangle (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 \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the rotation is applied.

Updated: 2019-01-22

Scales the \langle box \rangle by factors \langle x-scale \rangle and \langle y-scale \rangle in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the scaling is applied. Negative scalings cause the material in the \langle box \rangle to be reversed in direction, but the reference point of the \langle box \rangle is unchanged. Thus a negative \langle y-scale \rangle results in the \langle box \rangle having a depth dependent on the height of the original and \textit{vice versa}.

Updated: 2019-01-22

 adjusts the bounding box of the \langle box \rangle \langle left \rangle is removed from the left-hand edge of the bounding box, \langle right \rangle from the right-hand edge and so fourth. All adjustments are \langle dim exprs \rangle. Material outside of the bounding box is still displayed in the output unless \textbf{\textbackslash box_set_clipped:N} is subsequently applied. The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle 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.

Updated: 2019-01-23

35.14 Viewing part of a box

\textbf{\textbackslash box_set_viewport:N}\langle box \rangle\langle llx \rangle\langle lly \rangle\langle urx \rangle\langle ury \rangle

Adjusts the bounding box of the \langle box \rangle such that it has lower-left co-ordinates (\langle llx \rangle, \langle lly \rangle) and upper-right co-ordinates (\langle urx \rangle, \langle ury \rangle). All four co-ordinate positions are \langle dim exprs \rangle. Material outside of the bounding box is still displayed in the output unless \textbf{\textbackslash box_set_clipped:N} is subsequently applied. The updated \langle box \rangle is an hbox, irrespective of the nature of the \langle box \rangle before the viewport operation is applied. Additional box levels are also generated by this operation.

Updated: 2019-01-23

311
35.15 Primitive box conditionals

\if_hbox:N \if_hbox:N \hbox\langle box\rangle
\langle true code\rangle
\else:
\langle false code\rangle
\fi:
Tests is \langle box\rangle is a horizontal box.

\TeXhackers note: This is the \TeX primitive \ifhbox.

\if_vbox:N \if_vbox:N \vbox\langle box\rangle
\langle true code\rangle
\else:
\langle false code\rangle
\fi:
Tests is \langle box\rangle is a vertical box.

\TeXhackers note: This is the \TeX primitive \ifvbox.

\if_box_empty:N \if_box_empty:N \empty\langle box\rangle
\langle true code\rangle
\else:
\langle false code\rangle
\fi:
Tests is \langle box\rangle is an empty (void) box.

\TeXhackers note: This is the \TeX primitive \ifvoid.
Chapter 36

The l3coffins module
Coffin code layer

The material in this module provides the low-level support system for coffins. For details
about the design concept of a coffin, see the \texttt{xcoffins} module (in the l3experimental bundle).

36.1 Creating and initialising coffins

\begin{verbatim}
\coffin_new:N  ⟨coffin⟩
\coffin_new:c  ⟨coffin⟩
\coffin_clear:N  ⟨coffin⟩
\coffin_clear:c  ⟨coffin⟩
\coffin_set_eq:NN  ⟨coffin1⟩ ⟨coffin2⟩
\coffin_gclear:N
clears the content of the \texttt{⟨coffin⟩}.
\coffin_gclear:c
\coffin_set_eq:NN
\coffin_set_eq:(N|c|n|cc)
\coffin_gset_eq:NN
\coffin_gset_eq:(N|c|n|cc)
\coffin_if_exist_p:N  ⟨coffin⟩
\coffin_if_exist:NTF  ⟨coffin⟩ \{⟨true code⟩\} \{⟨false code⟩\}
\coffin_if_exist_p:c
\coffin_if_exist:NTF
\coffin_if_exist:NTF
\coffin_if_exist:cTF
\end{verbatim}

\newpage
36.2 Setting coffin content and poles

\hcoffin_set:Nn \hcoffin_set:cn \hcoffin_gset:Nn \hcoffin_gset:cn

New: 2011-08-17
Updated: 2019-01-21

\hcoffin_set:Nw \hcoffin_set:cw \hcoffin_gset:Nw \hcoffin_gset:cw

New: 2011-09-10
Updated: 2019-01-21

\vcoffin_set:Nnn \vcoffin_set:cnn \vcoffin_gset:Nnn \vcoffin_gset:cnn

New: 2011-08-17
Updated: 2019-01-21

\vcoffin_set:Nnw \vcoffin_set:cnw \vcoffin_gset:Nnw \vcoffin_gset:cnw

New: 2011-09-10
Updated: 2019-01-21

\coffin_set_horizontal_pole:Nnn \coffin_set_horizontal_pole:cnn \coffin_gset_horizontal_pole:Nnn \coffin_gset_horizontal_pole:cnn

New: 2012-07-20
Updated: 2019-01-21

Typesets the \texttt{material} in horizontal mode, storing the result in the \texttt{coffin}. The standard poles for the \texttt{coffin} are then set up based on the size of the typeset material.

Typesets the \texttt{material} in horizontal mode, storing the result in the \texttt{coffin}. The standard poles for the \texttt{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.

Typesets the \texttt{material} in vertical mode constrained to the given \texttt{width} and stores the result in the \texttt{coffin}. The standard poles for the \texttt{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.

Sets the \texttt{pole} to run horizontally through the \texttt{coffin}. The \texttt{pole} is placed at the \texttt{offset} from the baseline of the \texttt{coffin}. The \texttt{offset} should be given as a dimension expression.
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.

Resets the poles of the ⟨coffin⟩ to the standard set, removing any custom or inherited poles. The poles will therefore be equal to those that would be obtained from \hcoffin\_set:Nn or similar; the bounding box of the coffin is not reset, so any material outside of the formal bounding box will not influence the poles.

### 36.3 Coffin affine transformations

Resized the ⟨coffin⟩ to ⟨width⟩ and ⟨total-height⟩, both of which should be given as dimension expressions.

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.

Scales the ⟨coffin⟩ by a factors ⟨x-scale⟩ and ⟨y-scale⟩ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.
36.4 Joining and using coffins

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

\[
\begin{align*}
\text{coffin} & \text{ attach: NNNNNNNN} \\
\text{coffin} & \text{ attach: (cnnNNNNNNnncnnnnnncnnnncnnnn)} \\
\text{coffin} & \text{ gattach: NNNNNNNN} \\
\text{coffin} & \text{ gattach: (cnnNNNNNNnncnnnnnnncnnnnnncnnnnn)}
\end{align*}
\]

This function joins \( \text{coffin}_2 \) to \( \text{coffin}_1 \) such that the bounding box of \( \text{coffin}_1 \) 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 \( \langle \text{handle}_1 \rangle \), the point of intersection of \( \langle \text{coffin}_2\text{-pole}_1 \rangle \) and \( \langle \text{coffin}_1\text{-pole}_2 \rangle \), and \( \langle \text{handle}_2 \rangle \), the point of intersection of \( \langle \text{coffin}_2\text{-pole}_2 \rangle \) and \( \langle \text{coffin}_1\text{-pole}_2 \rangle \). \( \text{coffin}_2 \) is then attached to \( \text{coffin}_1 \) such that the relationship between \( \langle \text{handle}_1 \rangle \) and \( \langle \text{handle}_2 \rangle \) is described by the \( \langle x\text{-offset} \rangle \) and \( \langle y\text{-offset} \rangle \). The two offsets should be given as dimension expressions.

\[
\begin{align*}
\text{coffin} & \text{ join: NNNNNNNN} \\
\text{coffin} & \text{ join: (cnnNNNNNNnncnnnnnncnnnnnncnnnnn)} \\
\text{coffin} & \text{ gjoin: NNNNNNNN} \\
\text{coffin} & \text{ gjoin: (cnnNNNNNNnncnnnnnncnnnnnncnnnnnncnnnnn)}
\end{align*}
\]

Typesetting is carried out by first calculating \( \langle \text{handle} \rangle \), the point of intersection of \( \langle \text{pole}_1 \rangle \) and \( \langle \text{pole}_2 \rangle \). The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the \( \langle \text{handle} \rangle \) is described by the \( \langle x\text{-offset} \rangle \) and \( \langle y\text{-offset} \rangle \). The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

36.5 Measuring coffins

\[
\begin{align*}
\text{coffin} & \text{ dp: N} \\
\text{coffin} & \text{ dp: C}
\end{align*}
\]

Calculates the depth (below the baseline) of the \( \langle \text{coffin} \rangle \) in a form suitable for use in a \( \langle \text{dim expr} \rangle \).
36.6 Coffin diagnostics

\coffin_ht:N  \coffin_ht:c
Calculates the height (above the baseline) of the (coffin) in a form suitable for use in a (dim expr).

\coffin_wd:N  \coffin_wd:c
Calculates the width of the (coffin) in a form suitable for use in a (dim expr).

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

\coffin_mark_handle:Nnnn  \coffin_mark_handle:cnnn
This function first calculates the (handle) for the (coffin) as defined by the intersection of (pole_1) and (pole_2). It then marks the position of the (handle) on the (coffin). The (handle) are labelled as part of this process: the location of the (handle) and the label are both printed in the (color) specified.

\coffin_show_structure:N  \coffin_show_structure:c
This function shows the structural information about the (coffin) in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.

Notice that the poles of a coffin are defined by four values: the x and y co-ordinates of a point that the pole passes through and the x- and y-components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

\coffin_log_structure:N  \coffin_log_structure:c
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_log:N
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

Shows poles and contents of the ⟨coffin⟩ in the terminal or log file, showing the first ⟨int expr⟩ items in the coffin, and descending into ⟨int expr⟩ group levels. See \coffin_-
show_structure:N and \box_show:Nnn to show separately the pole structure and the contents.

36.7 Constants and variables

\c_empty_coffin

A permanently empty coffin.

\l_tmpa_coffin \l_tmpb_coffin

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 37

The l3color module

Color support

37.1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

```
\color_group_begin: ...
\color_group_end:
```

Creates a color group: one used to “trap” color settings. This grouping is built in to for example `\hbox_set:Nn`.

```
\color_ensure_current:
```

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end: group`

37.2 Color models

A color model is a way to represent sets of colors. Different models are particularly suitable for different output methods, e.g. screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of axes which define a color space. In contrast, various proprietary models are available which define spot colors (more formally separations).

Core models are used to pass color information to output; these are “native” to l3color. Core models use real numbers in the range [0, 1] to represent values. The core models supported here are

- **gray** Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
- **rgb** Red-green-blue color, with three axes, one for each of the components
• **cmyk** Cyan-magenta-yellow-black color, with four axes, one for each of the components.

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based; see below for more detail. The supported interface models are

• **Gray** Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)

• **hsb** Hue-saturation-brightness color, with three axes, all real values in the range $[0, 1]$ for hue saturation and brightness

• **Hsb** Hue-saturation-brightness color, with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness

• **HSB** Hue-saturation-brightness color, with three axes, integers in the range $[0, 240]$ for hue, saturation and brightness

• **HTML** HTML format representation of RGB color given as a single six-digit hexadecimal number

• **RGB** Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255

• **wave** Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as **rgb**.

Finally, there are a small number of models which are parsed to allow data transfer from **xcolor** but which should not be used by end-users. These are

• **cmy** Cyan-magenta-yellow color with three axes, one for each of the components; converted to cmyk

• **tHsb** “Tuned” hue-saturation-brightness color with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness; converted to rgb using the standard tuning map defined by xcolor

• **&spot** Spot color tint with one value; treated as a gray tint as spot color data is not available for extraction

To allow parsing of data from **xcolor**, any leading model up the first : will be discarded; the approach of selecting an internal form for data is *not* used in **3color**.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 37.9 for more detail of color support for additional models.

When color is selected by model, the *values* given are specified as a comma-separated list. The length of the list will therefore be determined by the detail of the model involved.

Color models (and interconversion) are complex, and more details are given in the manual to the **\LaTeX** 2\epsilon **xcolor** package and in the *PostScript Language Reference Manual*, published by Addison-Wesley.
37.3 Color expressions

In addition to allowing specification of color by model and values, \texttt{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 \texttt{!} symbols in the expression. Thus for example

\begin{verbatim}
  red!50!green
\end{verbatim}

is a mixture of 50\% red and 50\% green. A trailing value is interpreted as implicitly followed by \texttt{!white}, and so

\begin{verbatim}
  red!25
\end{verbatim}

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

\begin{verbatim}
  red!50!cyan
\end{verbatim}

will result in a color specification using the \texttt{rgb} model, made up of 50\% red and 50\% of cyan \texttt{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 \texttt{gray}. In this case, the order of mixing is “swapped” internally, so that for example

\begin{verbatim}
  black!50!red
\end{verbatim}

has the same result as

\begin{verbatim}
  red!50!black
\end{verbatim}

(the predefined colors \texttt{black} and \texttt{white} use the \texttt{gray} model).

Where more than two colors are mixed in an expression, evaluation takes place in a stepwise fashion. Thus in

\begin{verbatim}
  cyan!50!magenta!10!yellow
\end{verbatim}

the sub-expression

\begin{verbatim}
  cyan!50!magenta
\end{verbatim}

is first evaluated to give an intermediate color specification, before the second step

\begin{verbatim}
  <intermediate>!10!yellow
\end{verbatim}

where \texttt{<intermediate>} represents this transitory calculated value.

Within a color expression, \texttt{.} may be used to represent the color active for typesetting (the current color). This allows for example

\begin{verbatim}
  .!50
\end{verbatim}

to mean a mixture of 50\% of current color with white.

(Color expressions supported here are a subset of those provided by the \texttt{\LaTeX} \texttt{2\epsilon xcolor} package. At present, only such features as are clearly useful have been added here.)

321
37.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names \texttt{black}, \texttt{white}, \texttt{red}, \texttt{green}, \texttt{blue}, \texttt{cyan}, \texttt{magenta} and \texttt{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, \texttt{.} represents the current color at the start of a color expression.

\begin{verbatim}
\color_set:nn\color_set:nn \{\texttt{name}\} \{\texttt{color expression}\}
\end{verbatim}

Evaluates the \texttt{color expression} and stores the resulting color specification as the \texttt{name}.

\begin{verbatim}
\color_set:nnn\color_set:nnn \{\texttt{name}\} \{\texttt{model(s)}\} \{\texttt{values}\}
\end{verbatim}

Stores the color specification equivalent to the \texttt{model(s)} and \texttt{values} as the \texttt{name}.

\begin{verbatim}
\color_set_eq:nn\color_set_eq:nn \{\texttt{name1}\} \{\texttt{name2}\}
\end{verbatim}

Copies the color specification in \texttt{name2} to \texttt{name1}. The special name \texttt{.} may be used to represent the current color, allowing it to be saved to a name.

\begin{verbatim}
\color_if_exist_p:n\color_if_exist:nn \{\texttt{name}\}
\color_if_exist:nTF \{\texttt{name}\} \{\texttt{true code}\} \{\texttt{false code}\}
\end{verbatim}

Tests whether \texttt{name} is currently defined to provide a color specification.

\begin{verbatim}
\color_show:n\color_log:n \{\texttt{name}\}
\end{verbatim}

Displays the color specification stored in the \texttt{name} on the terminal or log file.

37.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color \texttt{.}: other more specialised functions such as fill and stroke selectors do \texttt{not} adjust this value.

\begin{verbatim}
\color_select:n\color_select:nn \{\texttt{color expression}\}
\end{verbatim}

Parses the \texttt{color expression} and then activates the resulting color specification for typeset material.

\begin{verbatim}
\color_select:nn\color_select:nnn \{\texttt{model(s)}\} \{\texttt{values}\}
\end{verbatim}

Activates the color specification equivalent to the \texttt{model(s)} and \texttt{values} for typeset material.

When this is set to a non-empty value, colors will be converted to the specified model when they are selected. Note that included images and similar are not influenced by this setting.
37.6 Colors for fills and strokes

Colors for drawing operations and so forth are split into strokes and fills (the latter may also be referred to as non-stroke color). The fill color is used for text under normal circumstances. Depending on the backend, stroke color may use a stack, in which case it exhibits the same page breaking behavior as general color. However, dvips/dvisvgm do not support this, and so color will need to be contained within a scope, such as \draw_begin:/\draw_end:.

\color_fill:n \color_stroke:n

Parses the \textit{color expression} and then activates the resulting color specification for filling or stroking.

\color_fill:nn \color_stroke:nn

Activates the color specification equivalent to the \textit{model(s)} and \textit{value(s)} for filling or stroking.

\color.sc

When using dvips, this PostScript variables hold the stroke color.

37.6.1 Coloring math mode material

Coloring math mode material using \texttt{\color_select:nn(n)} has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating \texttt{\mathord} atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

\color_math:nn \color_math:nnn

Works as for \texttt{\color_select:n(n)} but applies color only to the math mode \textit{content}. The function does not generate a group and the \textit{content} therefore retains its math atom states. Sub/superscripts are also properly handled.

\l_color_math_active_tl

This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.

37.7 Multiple color models

When selecting or setting a color with an explicit model, it is possible to give values for more than one model at one time. This is particularly useful where automated conversion between models does not give the desired outcome. To do this, the list of models and list of values are both subdivided using / characters (as for the similar function in xcolor). For example, to save a color with explicit cmyk and rgb values, one could use

\color_set:nnn { foo } { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1 , 0.2 , 0.3 }
The manually-specified conversion will be used in preference to automated calculation whenever the model(s) listed are used: both in expressions and when a fixed model is active.

Similarly, the same syntax can be applied to directly selecting a color.

```
\color_select:nn { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

Again, this list is used when a fixed model is active: the first entry is used unless there is a fixed model matching one of the other entries.

### 37.8 Exporting color specifications

The major use of color expressions is in setting typesetting output, but there are other places in which some form of color information is required. These may need data in a different format or using a different model to the internal representation. Thus a set of functions are available to export colors in different formats.

Valid export targets are

- **backend** Two brace groups: the first containing the model, the second containing space-separated values appropriate for the model; this is the format required by backend functions of expl3
- **comma-sep-cmyk** Comma-separated cyan-magenta-yellow-black values
- **comma-sep-rgb** Comma-separated red-green-blue values suitable for use as a PDF annotation color
- **HTML** Uppercase two-digit hexadecimal values, expressing a red-green-blue color; the digits are not separated
- **space-sep-cmyk** Space-separated cyan-magenta-yellow-black values
- **space-sep-rgb** Space-separated red-green-blue values suitable for use as a PDF annotation color

```
\color_export:nnN {⟨color expression⟩} {⟨format⟩} {⟨tl⟩}
```

Parses the ⟨color expression⟩ as described earlier, then converts to the ⟨format⟩ specified and assigns the data to the ⟨tl⟩.

```
\color_export:nnnN {⟨model⟩} {⟨value(s)⟩} {⟨format⟩} {⟨tl⟩}
```

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⟩.
37.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see https://helpx.adobe.com/indesign/using/spot-process-colors.html for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (DeviceCMYK, DeviceRGB, DeviceGray), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that \texttt{l3color} uses the shorter names \texttt{cmyk}, etc.)

\begin{verbatim}
\color_model_new:nnn \{\langle model \rangle \} \{\langle family \rangle \} \{\langle params \rangle \}
\end{verbatim}

Creates a new \texttt{\langle model \rangle} which is derived from the color model \texttt{\langle family \rangle}. The latter should be one of

- DeviceN
- ICCBased
- Separation

(The \texttt{\langle family \rangle} may be given in mixed case as-in the PDF reference: internally, case of these strings is folded.) Depending on the \texttt{\langle family \rangle}, one or more \texttt{\langle params \rangle} 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 \texttt{\langle name \rangle} may contain spaces, etc., which are not permitted in the \texttt{\langle model \rangle}.
- **alternative-model** An alternative device colorspace, one of \texttt{cmyk}, \texttt{rgb}, \texttt{gray} or \texttt{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 \texttt{alternative-model}. This information is used by the PDF application if the Separation is not available.

CIELAB color separations are created using the \texttt{alternative-model = CIELAB} setting. These colors must also have an \texttt{illuminant} key, one of \texttt{a}, \texttt{c}, \texttt{e}, \texttt{d50}, \texttt{d55}, \texttt{d65} or \texttt{d75}. The \texttt{alternative-values} in this case are the three parameters \texttt{L*}, \texttt{a*} and \texttt{b*} of the CIELAB model. Full details of this device-independent color approach are given in the documentation to the \texttt{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, \texttt{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 \texttt{\langle name \rangle} of a Separation model, a process color name (cyan, etc.) or the special name \texttt{none}.

For an ICCBased space, there is one **compulsory** key.

- **file** The name of the file containing the profile.
37.9.1 Color profiles

Color profiles are used to ensure color accuracy by linking to collaboration. Applying a profile can be used to standardise color which is otherwise device-dependence.

\color_profile_apply\{\text{profile}\}\{\text{model}\}

This function applies a \text{profile} to one of the device \text{models}. The profile will then apply to all color of the selected \text{model}. The \text{profile} should specify an ICC profile file. The \text{model} has to be one the standard device models: \text{cmyk}, \text{gray} or \text{rgb}.
Chapter 38

The l3pdf module
Core PDF support

38.1 Objects

\pdf_object_new:n (object)

Declares \texttt{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:nnn {object} {type} {content}

Writes the \texttt{content} as content of the \texttt{object}. Depending on the \texttt{type} declared for the object, the format required for the \texttt{data} will vary

array A space-separated list of values

dict Key–value pairs in the form /\texttt{key} \texttt{value}

fstream Two brace groups: \texttt{file name} and \texttt{file content}

stream Two brace groups: \texttt{attributes (dictionary)} and \texttt{stream contents}

\pdf_object_ref:n \pdf_object_ref:n (object)

Inserts the appropriate information to reference the \texttt{object} in for example page resource allocation
\pdf_object_unnamed_write:nn \pdf_object_unnamed_write:nn \{type\} \{content\}
\pdf_object_unnamed_write:ne

New: 2021-02-10

Writes the \{content\} as content of an anonymous object. Depending on the \{type\}, the format required for the \{data\} will vary.

array A space-separated list of values

dict Key–value pairs in the form /\{key\} \{value\}

fstream Two value pairs: \{attributes (dictionary)\} and \{file name\}

stream Two brace groups: \{attributes (dictionary)\} and \{stream contents\}

\pdf_object_ref_last: * \pdf_object_ref_last:

New: 2021-02-10

Inserts the appropriate information to reference the last \{object\} created. This is particularly useful for anonymous objects.

\pdf_pageobject_ref:n * \pdf_pagobject_ref:n \{pageobject\}

New: 2021-02-10

Inserts the appropriate information to reference the \{pageobject\}.

\pdf_object_if_exist_p:n * \pdf_object_if_exist:n \{object\} \{object\}
\pdf_object_if_exist:nTF * \pdf_object_if_exist:nTF \{object\}

New: 2020-05-15

Tests whether an object with name \{object\} has been defined.

38.2 Version

\pdf_version_compare_p:Nn * \pdf_version_compare_p:Nn \{comparator\} \{version\}
\pdf_version_compare:NnTF * \pdf_version_compare:NnTF \{comparator\} \{version\} \{true code\} \{false code\}

New: 2021-02-10

Compares the version of the PDF being created with the \{version\} string specified, using the \{comparator\}. Either the \{true code\} or \{false code\} will be left in the output stream.

\pdf_version_gset:n \pdf_version_gset:n \{version\}

Sets the \{version\} of the PDF being created. The min version will not alter the output version unless it is currently lower than the \{version\} requested.

This function may only be used up to the point where the PDF file is initialised. With dvips it sets \pdf_version_major: and \pdf_version_minor:, and allows to compare the values with \pdf_version_compare:Nn, but the PDF version itself still has to be set with the command line option -dCompatibilityLevel of ps2pdf.

\pdf_version: * \pdf_version:
\pdf_version_major: * \pdf_version_major:
\pdf_version_minor: * \pdf_version_minor:

New: 2021-02-10

Expands to the currently-active PDF version.
38.3 Page (media) size

\pdf_pagesize_gset:nn \pdf_pagesize_gset:nn \{\langle width\rangle\} \{\langle height\rangle\}
Sets the page size (mediabox) of the PDF being created to the \langle width\rangle and \langle height\rangle,
both of which are \langle dimexpr\rangle.

38.4 Compression

\pdf_uncompress: \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 initialised.

38.5 Destinations

Destinations are the places a link jumped too. Unlike the name may suggest they don’t
described 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.
This creates a destination. \{\texttt{type or integer}\} can be one of \texttt{fit}, \texttt{fith}, \texttt{fitv}, \texttt{fitb}, \texttt{fitbh}, \texttt{fitbv}, \texttt{fitr}, \texttt{xyz} or an integer representing a scale factor in percent. \texttt{fitr} here gives only a lightweight version of /FitR: The backend code defines \texttt{fitr} so that it will with pdflTEX and LuaLTEX use the coordinates of the surrounding box, with \texttt{dvips} and \texttt{dvipdfmx} it falls back to \texttt{fit}. For full control use \texttt{pdf_destination:nnnn}.

The keywords match to the PDF names as described in the following tabular.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>PDF</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit</td>
<td>/Fit</td>
<td>Fits the page to the window</td>
</tr>
<tr>
<td>fith</td>
<td>/FitH top</td>
<td>Fits the width of the page to the window</td>
</tr>
<tr>
<td>fitv</td>
<td>/FitV left</td>
<td>Fits the height of the page to the window</td>
</tr>
<tr>
<td>fitb</td>
<td>/FitB</td>
<td>Fits the page bounding box to the window</td>
</tr>
<tr>
<td>fitbh</td>
<td>/FitBH top</td>
<td>Fits the width of the page bounding box to the window.</td>
</tr>
<tr>
<td>fitbv</td>
<td>/FitBV left</td>
<td>Fits the height of the page bounding box to the window.</td>
</tr>
<tr>
<td>fitr</td>
<td>/FitR left bottom right top</td>
<td>Fits the rectangle specified by the four coordinates to the window (see above for the restrictions)</td>
</tr>
<tr>
<td>xyz</td>
<td>/XYZ left top null</td>
<td>Sets a coordinate but doesn’t change the zoom.</td>
</tr>
</tbody>
</table>
| \{\texttt{integer}\} | /XYZ left top zoom | Sets a coordinate and a zoom meaning \{\texttt{integer}\}\%.

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
Additions and removals
Chapter 39

The \texttt{l3candidates} module
Experimental additions to \texttt{l3kernel}

39.1 Important notice

This module provides a space in which functions can be added to \texttt{l3kernel (expl3)} while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in \texttt{l3kernel} in the future.

In contrast to the material in \texttt{l3experimental}, the functions here are all \textit{small} additions to the kernel. We encourage programmers to test them out and report back on the \LaTeX-L mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the \LaTeX-L mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are \textbf{not necessarily stable} and we may have to adjust them!
39.2 Additions to \texttt{l3seq}

\begin{verbatim}
\seq_set_filter:NNn \seq_gset_filter:NNn
\seq_set_filter:NNn \langle sequence_1 \rangle \langle sequence_2 \rangle \{ \langle inline boolean \rangle \}
\end{verbatim}

Evaluates the \langle inline boolean \rangle for every \langle item \rangle stored within the \langle sequence_2 \rangle. The \langle inline boolean \rangle receives the \langle item \rangle as \#1. The sequence of all \langle items \rangle for which the \langle inline boolean \rangle evaluated to \texttt{true} is assigned to \langle sequence_1 \rangle.

\textbf{\TeXhacks} note: Contrarily to other mapping functions, \texttt{\seq_map_break}: cannot be used in this function, and would lead to low-level \TeX{} errors.
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.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>272</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>272</td>
</tr>
<tr>
<td>&amp;</td>
<td>271</td>
</tr>
<tr>
<td>*</td>
<td>272</td>
</tr>
<tr>
<td>**</td>
<td>272</td>
</tr>
<tr>
<td>+</td>
<td>272</td>
</tr>
<tr>
<td>–</td>
<td>272</td>
</tr>
<tr>
<td>/</td>
<td>272</td>
</tr>
<tr>
<td>\N</td>
<td>49</td>
</tr>
<tr>
<td>\V</td>
<td>49</td>
</tr>
<tr>
<td>\V_unbraced</td>
<td>49</td>
</tr>
<tr>
<td>\C</td>
<td>49</td>
</tr>
<tr>
<td>\e</td>
<td>49</td>
</tr>
<tr>
<td>\e_unbraced</td>
<td>49</td>
</tr>
<tr>
<td>\error</td>
<td>89</td>
</tr>
<tr>
<td>\f</td>
<td>49</td>
</tr>
<tr>
<td>\f_unbraced</td>
<td>49</td>
</tr>
<tr>
<td>\n</td>
<td>49</td>
</tr>
<tr>
<td>\o</td>
<td>49</td>
</tr>
<tr>
<td>\o_unbraced</td>
<td>49</td>
</tr>
<tr>
<td>\p</td>
<td>49</td>
</tr>
<tr>
<td>\V</td>
<td>49</td>
</tr>
<tr>
<td>\V_unbraced</td>
<td>49</td>
</tr>
<tr>
<td>\x</td>
<td>49</td>
</tr>
<tr>
<td>\x_unbraced</td>
<td>49</td>
</tr>
<tr>
<td>&lt;</td>
<td>272</td>
</tr>
<tr>
<td>=</td>
<td>272</td>
</tr>
<tr>
<td>&gt;</td>
<td>272</td>
</tr>
<tr>
<td>?</td>
<td>272</td>
</tr>
<tr>
<td>?:</td>
<td>271</td>
</tr>
<tr>
<td>⟨(scope)⟩ commands:</td>
<td></td>
</tr>
<tr>
<td>(\langle(scope)\rangle)</td>
<td>5</td>
</tr>
<tr>
<td>(\langle(scope)\rangle)</td>
<td>5</td>
</tr>
<tr>
<td>⟨(scope)⟩ commands:</td>
<td></td>
</tr>
<tr>
<td>(\langle(scope)\rangle)</td>
<td>5</td>
</tr>
</tbody>
</table>

**B**

bitset commands:

<table>
<thead>
<tr>
<th>Bitset commands:</th>
<th>281</th>
</tr>
</thead>
<tbody>
<tr>
<td>\bitset_addto_named_index:Nn</td>
<td>284</td>
</tr>
<tr>
<td>\bitset_clear:N</td>
<td>285</td>
</tr>
<tr>
<td>\bitset_gclear:N</td>
<td>285</td>
</tr>
<tr>
<td>\bitset_gset_false:N</td>
<td>285</td>
</tr>
<tr>
<td>\bitset_gset_true:N</td>
<td>285</td>
</tr>
<tr>
<td>\bitset_gset_if_exist:NTF</td>
<td>285</td>
</tr>
<tr>
<td>\bitset_gset_if_exist_p:N</td>
<td>285</td>
</tr>
</tbody>
</table>

**bool commands:**

<table>
<thead>
<tr>
<th>Bool commands:</th>
<th>72</th>
</tr>
</thead>
<tbody>
<tr>
<td>\bool_case:n</td>
<td>72</td>
</tr>
<tr>
<td>\bool_case:NTF</td>
<td>72</td>
</tr>
<tr>
<td>\bool_const:N</td>
<td>67</td>
</tr>
<tr>
<td>\bool_do_until:NN</td>
<td>71</td>
</tr>
<tr>
<td>\bool_do_until:nn</td>
<td>71</td>
</tr>
<tr>
<td>\bool_do_while:NN</td>
<td>71</td>
</tr>
<tr>
<td>\bool_do_while:nn</td>
<td>71</td>
</tr>
<tr>
<td>\bool_gset:N</td>
<td>240</td>
</tr>
<tr>
<td>\bool_gset:NN</td>
<td>67</td>
</tr>
<tr>
<td>\bool_gset_eq:NN</td>
<td>67</td>
</tr>
<tr>
<td>\bool_gset_false:N</td>
<td>67</td>
</tr>
<tr>
<td>\bool_gset_inverse:N</td>
<td>240</td>
</tr>
<tr>
<td>\bool_gset_inverse:NN</td>
<td>67</td>
</tr>
<tr>
<td>\bool_gset_true:N</td>
<td>67</td>
</tr>
<tr>
<td>\bool_gset_true:NTF</td>
<td>67</td>
</tr>
<tr>
<td>\bool_if:NTF</td>
<td>67</td>
</tr>
<tr>
<td>\bool_if:nTF</td>
<td>67-72</td>
</tr>
<tr>
<td>\bool_if_exist:NTF</td>
<td>68</td>
</tr>
<tr>
<td>\bool_if_exist_p:N</td>
<td>68</td>
</tr>
</tbody>
</table>

**A**

<table>
<thead>
<tr>
<th>Abs</th>
<th>272</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acos</td>
<td>275</td>
</tr>
<tr>
<td>Acosd</td>
<td>275</td>
</tr>
<tr>
<td>Acot</td>
<td>276</td>
</tr>
<tr>
<td>Acotd</td>
<td>276</td>
</tr>
<tr>
<td>Acsc</td>
<td>275</td>
</tr>
<tr>
<td>Acsce</td>
<td>275</td>
</tr>
<tr>
<td>Asec</td>
<td>275</td>
</tr>
</tbody>
</table>

334
debug commands:
\debug_off:n .......................... 30 \dim_to_decimal_in_mm:n ............ 229
\debug_on:n .......................... 30 \dim_to_decimal_in_pc:n ............ 229
\debug_resume: ......................... 30 \dim_to_decimal_in_sp:n ............ 230
\debug_suspend: ......................... 30 \dim_to_decimal_in_unit:mm ........ 230
\debug_suspend: ......................... 30 \dim_to_fp:n ..................... 230
default commands:
 .default:n ................................ 241 \dim_to_decimal_in_in:n ......... 229
 .. \default:n ................................ 241 \dim_to_decimal_in_cm:n ........ 229
... \debug: ................................ 241 \dim_to_decimal:n .................. 229
... \debug_suspend: ......................... 30 \dim_to_decimal_in_unit:nn ...... 229
... \debug_resume: ......................... 30 \dim_to_decimal_in_unit:mm ...... 230
... \debug_suspend: ......................... 30 \dim_to_decimal_in_unit:in ...... 230
... \debug_resume: ......................... 30 \dim_use:N ..................... 228
... \dim_zero:n ............................. 30 \dim_while_do:n ............ 227
... \dim_zero_new:n ....................... 30 \dim_until_do:n ............ 227
... \dim_max:nn ............................. 30 \dim_until_do:n ............ 227
... \dim_min:nn ............................. 30 \dim_until_do:n ............ 227
... \dim_zero:N ............................. 30 \dim_until_do:n ............ 227
dim commands:
 \dim_abs:n ................................ 223 \dim_zero:N ................................ 222
 \dim_add:N ................................ 223 \dim_zero_new:N ........................ 222
 \dim_case:nn ................................ 226 \c_max_dim ................................ 229, 231, 234
 \dim_case:nnT ................................ 226 \g_tmpa_dim ................................ 231
 \dim_case:nnTF ................................ 226 \l_tmpa_dim ................................ 231
 \dim_compare:nNnF ................................ 225 \g_tmpb_dim ................................ 231
 \dim_compare:nTF ................................ 224 \l_tmpb_dim ................................ 231
 \dim_compare_p:n ................................ 224 \c_zero_dim ................................ 231
draw commands:
 \dim_do_until:nn ........................... 227 \draw: ...................................... 323
 \dim_do_until:nNn ................................ 227 \draw_begin: ................................ 323
 \dim_do_until:nNn ................................ 226 \draw_end: .................................. 323
dim commands:
 \dim_eval:n ................................ 224, 225, 228 \draw: ...................................... 323
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
 \dim_abs:n ................................ 223 \dim_add:N ................................ 223
dim commands:
\file_get_mdfive_hash:nNTF 102  \fp_eval:n 259, 262, 266, 271-278, 286
\file_get_size:n 102  \fp_format:n 279
\file_get_size:nNTF 102  \fp_gadd:Nn 258
\file_get_timestamp:nN 102  .fp_gset:N 241
\file_get_timestamp:nNTF 102  \fp_gset:Nn 258
\file_hex_dump:n 101  \fp_gset_eq:NN 258
\file_hex_dump:nnn 101  \fp_gsub:Nn 259
\file_if_exist:nNTF 101, 103, 104  \fp_gzero:N 258
\file_if_exist_input:n 104  \fp_gzero_new:N 258
\file_if_exist_input:nNTF 104  \fp_if_exist:nNTF 260
\file_if_exist_p:n 101  \fp_if_exist_p:N 260
\file_input:n 104, 105  \fp_if_nan:nTF 262, 279
\file_input:raw:n 104  \fp_if_nan_p:n 262
\file_input:stop: 105  \fp_invalid_operation 268
\file_log_list: 105  \fp_log:N 269
\file_mdfive_hash:n 102  \fp_log:n 269
\file_parse_full_name:n 104  \fp_max:nn 278
\file_parse_full_name:nNNN 103, 104  \fp_min:nn 278
\file_parse_full_name_apply:n 104  \fp_new:N 258
\l_file_search_path_seq 101, 102, 104  \fp_new_function:n 266
\file_show_list: 105  \fp_new_variable:n 264-266
\file_size:n 102  \fp_overflow 268
\file_timestamp:n 102  \fp_set:Nn 241
\file_name 101  \fp_set:NNn 258, 264
flag commands:
\flag_clear:n 182, 183  \fp_set_eq:NN 258
\flag_clear_new:n 183  \fp_set_function:nnn 266
\flag_ensure_raised:n 183  \fp_set_variable:nn 264-266
\flag_if_exist:nNTF 183  \fp_show:N 264, 265, 269
\flag_if_exist:p:n 183  \fp_show:n 264-266, 269
\flag_if_exist_input:n 183  \fp_step_function:nnnn 264
\flag_if_exist_input:nNTF 183  \fp_step_inline:nnnn 264
\flag_if_exist_p:n 183  \fp_step_variable:nnnn 264
\flag_if_exist:NTF 183  \fp_sub:Nn 259
\flag_height:n 183  \fp_to_decimal:N 259, 260
\flag_if_exist_input:nNTF 183  \fp_to_decimal:n 259, 260
\flag_if_exist_input:n 183  \fp_to_decimal:N 259, 260
\flag_if_exist:NTF 183  \fp_to_decimal:n 259, 260
\flag_if_exist:p:n 183  \flag_show:n 259
\flag_if_raised:nNTF 183  \fp_to_dim:N 259
\flag_if_raised:p:n 183  \fp_to_dim:n 259, 268
\flag_log:n 183  \fp_to_int:N 259
\flag_new:n 182, 183  \fp_to_int:n 259
\flag_raise:n 183  \fp_to_int:nn 259
\flag_show:n 183  \fp_to_int:nnn 259

floor 274  \fp_to_int:nnn 259, 268
fp commands:
%c_e_fp 267, 270  \fp_to_int:n 259
\fp_abs:n 272, 278  \fp_to_scientific:N 260
\fp_add:Nn 258  \fp_to_scientific:n 260
\fp_clear_function:n 266  \fp_to_tl:N 260, 282
\fp_clear_variable:n 266  \fp_to_tl:nn 260
\fp_compare:nNNTF 261-263 \fp_trap:nn 268
\fp_compare:nTF 261-263, 272  \fp_underflow 268
\fp_compare_p:n 262  \fp_upto_do:n 263
\fp_compare_p:nNn 261  \fp_until_do:nn 263
\fp_compare_p:nNn 261  \fp_until_do:nNnn 263
\fp_const:Nn 258  \fp_use:N 260, 282
\fp_division_by_zero 268  \fp_while_do:nn 263
\fp_do_until:nNn 262  \fp_zero:N 258
\fp_do_until:nn 263  \fp_zero_new:N 258
\fp_do_while:nn 263  \c_inf_fp 267, 277

341
<table>
<thead>
<tr>
<th>Command</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ior_shell_open:Nn</td>
<td>92</td>
</tr>
<tr>
<td>\ior_show:NN</td>
<td>93</td>
</tr>
<tr>
<td>\ior_show_list:nn</td>
<td>93</td>
</tr>
<tr>
<td>\ior_str_get:NN</td>
<td>93, 94, 97</td>
</tr>
<tr>
<td>\ior_str_get:NNTF</td>
<td>94</td>
</tr>
<tr>
<td>\ior_str_get_term:nN</td>
<td>97</td>
</tr>
<tr>
<td>\ior_str_map_inline:Nn</td>
<td>95</td>
</tr>
<tr>
<td>\ior_str_map_variable:NNn</td>
<td>95</td>
</tr>
<tr>
<td>\keys_set_known:nnN</td>
<td>248</td>
</tr>
<tr>
<td>\keys_set_groups:nnn</td>
<td>248</td>
</tr>
<tr>
<td>\keys_set_filter:nnn</td>
<td>248</td>
</tr>
<tr>
<td>\keys_set:nn</td>
<td>100</td>
</tr>
<tr>
<td>\keys_precompile:nnN</td>
<td>100</td>
</tr>
<tr>
<td>\l_keys_path_str</td>
<td>100</td>
</tr>
<tr>
<td>\keys_log:NN</td>
<td>253</td>
</tr>
<tr>
<td>\keys_if_exist_p:nn</td>
<td>252, 253</td>
</tr>
<tr>
<td>\keys_if_choice_exist_p:nn</td>
<td>253</td>
</tr>
<tr>
<td>\keys_if_choice_exist:nnTF</td>
<td>253</td>
</tr>
<tr>
<td>\keys_define:nn</td>
<td>253</td>
</tr>
<tr>
<td>\l_keys_choice_tl</td>
<td>253</td>
</tr>
<tr>
<td>\keys_if_exist:nn</td>
<td>253</td>
</tr>
<tr>
<td>\keys_if_choice_exist:nnTF</td>
<td>253</td>
</tr>
<tr>
<td>\keys_if_exist_p:nn</td>
<td>253</td>
</tr>
<tr>
<td>\keys_set_known:nn</td>
<td>248</td>
</tr>
</tbody>
</table>

**Legacy commands:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\legacy_if:nTF</td>
<td>109</td>
</tr>
<tr>
<td>\legacy_if_gset:n</td>
<td>242</td>
</tr>
<tr>
<td>\legacy_if_gset:nn</td>
<td>109</td>
</tr>
<tr>
<td>\legacy_if_gset_false:n</td>
<td>109</td>
</tr>
<tr>
<td>\legacy_if_gset_inverse:n</td>
<td>242</td>
</tr>
<tr>
<td>\legacy_if_set:n</td>
<td>242</td>
</tr>
<tr>
<td>\legacy_if_set:nn</td>
<td>109</td>
</tr>
<tr>
<td>\legacy_if_set_false:n</td>
<td>109</td>
</tr>
<tr>
<td>\legacy_if_set_inverse:n</td>
<td>242</td>
</tr>
<tr>
<td>\legacy_if_true:n</td>
<td>109</td>
</tr>
</tbody>
</table>

**Lua commands:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lua_escape:n</td>
<td>107</td>
</tr>
<tr>
<td>\lua_load_module:n</td>
<td>107</td>
</tr>
<tr>
<td>\lua_now:n</td>
<td>106, 107</td>
</tr>
<tr>
<td>\lua_shipout:n</td>
<td>106</td>
</tr>
<tr>
<td>\lua_shipout_e:n</td>
<td>106</td>
</tr>
</tbody>
</table>

**Keys commands:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\l_keys_choice_int</td>
<td>240, 243, 245, 247</td>
</tr>
<tr>
<td>\keys_define:nn</td>
<td>239</td>
</tr>
<tr>
<td>\keys_if_choice_exist:nnTF</td>
<td>250</td>
</tr>
<tr>
<td>\keys_if_choice_exist_p:nn</td>
<td>250</td>
</tr>
<tr>
<td>\keys_if_exist:nnTF</td>
<td>250</td>
</tr>
<tr>
<td>\keys_if_exist_p:nn</td>
<td>250</td>
</tr>
<tr>
<td>\keys_set_key_str</td>
<td>248</td>
</tr>
<tr>
<td>\keys_log:nn</td>
<td>250</td>
</tr>
<tr>
<td>\keys_path_str</td>
<td>248</td>
</tr>
<tr>
<td>\keys_precompile:nnN</td>
<td>250</td>
</tr>
<tr>
<td>\keys_set:nn</td>
<td>239, 241, 242, 247-250</td>
</tr>
<tr>
<td>\keys_set_filter:nnn</td>
<td>249</td>
</tr>
<tr>
<td>\keys_set_filter:nnN</td>
<td>249</td>
</tr>
<tr>
<td>\keys_set_filter:nnnN</td>
<td>249</td>
</tr>
<tr>
<td>\keys_set_groups:nnn</td>
<td>250</td>
</tr>
<tr>
<td>\keys_set_known:nn</td>
<td>248</td>
</tr>
</tbody>
</table>

**Meta commands:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\meta:n</td>
<td>242</td>
</tr>
<tr>
<td>\meta:nn</td>
<td>243</td>
</tr>
</tbody>
</table>

**Mode commands:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\mode_if_horizontal:TF</td>
<td>72</td>
</tr>
<tr>
<td>\mode_if_horizontal:pp</td>
<td>72</td>
</tr>
<tr>
<td>\mode_if_inner:TF</td>
<td>73</td>
</tr>
<tr>
<td>\mode_if_inner:pp</td>
<td>73</td>
</tr>
<tr>
<td>\mode_if_math:TF</td>
<td>73</td>
</tr>
<tr>
<td>\mode_if_math:pp</td>
<td>73</td>
</tr>
<tr>
<td>\mode_if_vertical:TF</td>
<td>73</td>
</tr>
<tr>
<td>\mode_if_vertical:pp</td>
<td>73</td>
</tr>
</tbody>
</table>
\c_sys_minute_int .......... 75 \c_sys_month_int .......... 75 \c_sys_output_str .......... 77 \c_sys_platform_str ........ 77 \sys_rand_seed: ............ 77, 158, 277 \c_sys_shell_escape_int .... 78 \c_sys_shell_now:n .......... 79 \c_sys_shell_shipout:n .... 79 \c_sys_timer: ............... 76 \c_sys_timestamp_str ....... 75 \c_sys_year_int ............ 75

T
\tan ................. 274 \tand .................. 275

\LaTeX{} and \mbox{\LaTeX} 2\epsilon{} commands:

351