The \LaTeX3 Interfaces

The \LaTeX Project*

Released 2023-11-09

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

The expl3 modules are designed to be loaded on top of \LaTeX2ε. With an up-to-date \LaTeX2ε kernel, this material is loaded as part of the format. The fundamental programming code can also be loaded with other \TeX formats, subject to restrictions on the full range of functionality.

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

### I Introduction

1 Introduction to expl3 and this document ............................... 2  
   1.1 Naming functions and variables .................................. 2  
      1.1.1 Scratch variables ...................................... 5  
      1.1.2 Terminological inexactitude ................................ 5  
   1.2 Documentation conventions ....................................... 5  
   1.3 Formal language conventions which apply generally .......... 7  
   1.4 \TeX concepts not supported by \LaTeX3 ............................ 7

### II Bootstrapping

2 The \texttt{l3bootstrap} package: Bootstrap code ......................... 9  
   2.1 Using the \LaTeX3 modules ..................................... 9

3 The \texttt{l3names} package: Namespace for primitives ................ 11  
   3.1 Setting up the \LaTeX3 programming language ................. 11

### III Programming Flow

4 The \texttt{l3basics} package: Basic definitions ......................... 13  
   4.1 No operation functions ....................................... 13  
   4.2 Grouping material ............................................. 13  
   4.3 Control sequences and functions ............................... 14  
      4.3.1 Defining functions ...................................... 14  
      4.3.2 Defining new functions using parameter text .......... 15  
      4.3.3 Defining new functions using the signature .......... 17  
      4.3.4 Copying control sequences ............................... 19  
      4.3.5 Deleting control sequences ............................. 20  
      4.3.6 Showing control sequences ................................ 20  
      4.3.7 Converting to and from control sequences ............. 21  
   4.4 Analysing control sequences .................................. 22  
   4.5 Using or removing tokens and arguments .......................... 23  
      4.5.1 Selecting tokens from delimited arguments ............ 26  
   4.6 Predicates and conditionals ................................... 27  
      4.6.1 Tests on control sequences ............................. 28  
      4.6.2 Primitive conditionals .................................. 28  
   4.7 Starting a paragraph ........................................... 29  
   4.8 Debugging support .............................................. 30
# 10 The `l3sys` package: System/runtime functions

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 `l3msg` package: Messages

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 `l3file` package: File and I/O operations

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 operation functions .................................. 100

# 13 The `luatex` package: LuaTeX-specific functions

13.1 Breaking out to Lua ....................................... 106
13.2 Lua interfaces ............................................. 107

# 14 The `l3legacy` package: Interfaces to legacy concepts

# IV Data types

# 15 The `l3tl` package: Token lists

15.1 Creating and initialising token list variables .......... 111
15.2 Adding data to token list variables ..................... 112
15.3 Token list conditionals .................................... 113
15.3.1 Testing the first token ................................ 115
15.4 Working with token lists as a whole .................... 116
15.4.1 Using token lists ..................................... 116
15.4.2 Counting and reversing token lists .................. 117
15.4.3 Viewing token lists ................................... 118
15.5 Manipulating items in token lists ....................... 119
15.5.1 Mapping over token lists ............................ 119
# The \texttt{\l3token} package: Token manipulation

- 24.1 Creating character tokens ........................................ 195
- 24.2 Manipulating and interrogating character tokens ............ 196
- 24.3 Generic tokens .................................................. 199
- 24.4 Converting tokens ................................................ 200
- 24.5 Token conditionals ............................................... 200
- 24.6 Peeking ahead at the next token ................................. 204
- 24.7 Description of all possible tokens .............................. 209
- 24.8 Deprecated functions ............................................ 211

# The \texttt{\l3prop} package: Property lists

- 25.1 Creating and initialising property lists ....................... 212
- 25.2 Adding and updating property list entries .................... 214
- 25.3 Recovering values from property lists ....................... 215
- 25.4 Modifying property lists ...................................... 216
- 25.5 Property list conditionals .................................... 216
- 25.6 Recovering values from property lists with branching ....... 217
- 25.7 Mapping over property lists ................................... 218
- 25.8 Viewing property lists ........................................ 219
- 25.9 Scratch property lists ........................................ 220
- 25.10 Constants ..................................................... 220

# The \texttt{\l3skip} package: Dimensions and skips

- 26.1 Creating and initialising \texttt{dim} variables ............... 221
- 26.2 Setting \texttt{dim} variables .................................. 222
- 26.3 Utilities for dimension calculations .......................... 222
- 26.4 Dimension expression conditionals ............................ 223
- 26.5 Dimension expression loops .................................. 225
- 26.6 Dimension step functions ..................................... 226
- 26.7 Using \texttt{dim} expressions and variables .................. 227
- 26.8 Viewing \texttt{dim} variables .................................. 229
- 26.9 Constant dimensions ........................................... 230
- 26.10 Scratch dimensions ............................................ 230
- 26.11 Creating and initialising \texttt{skip} variables ............... 230
- 26.12 Setting \texttt{skip} variables .................................. 231
- 26.13 Skip expression conditionals ................................ 232
- 26.14 Using \texttt{skip} expressions and variables ................. 232
- 26.15 Viewing \texttt{skip} variables .................................. 232
- 26.16 Constant skips ................................................ 233
- 26.17 Scratch skips .................................................. 233
- 26.18 Inserting skips into the output ................................ 233
- 26.19 Creating and initialising \texttt{muskip} variables ............ 234
- 26.20 Setting \texttt{muskip} variables ................................ 234
- 26.21 Using \texttt{muskip} expressions and variables ............... 235
- 26.22 Viewing \texttt{muskip} variables ................................ 235
- 26.23 Constant muskips .............................................. 236
- 26.24 Scratch muskips ................................................ 236
- 26.25 Primitive conditional ......................................... 236
# The l3keys package: Key–value interfaces

27.1 Creating keys ............................................. 238
27.2 Sub-dividing keys ....................................... 243
27.3 Choice and multiple choice keys ....................... 244
27.4 Key usage scope ....................................... 246
27.5 Setting keys ........................................... 246
27.6 Handling of unknown keys .............................. 247
27.7 Selective key setting .................................. 248
27.8 Digesting keys ......................................... 249
27.9 Utility functions for keys ............................... 249
27.10 Low-level interface for parsing key–val lists ....... 250

# The l3intarray package: Fast global integer arrays

28.1 l3intarray documentation .............................. 253
   28.1.1 Implementation notes ............................. 254

# The l3fp package: Floating points

29.1 Creating and initialising floating point variables ... 257
29.2 Setting floating point variables ....................... 257
29.3 Using floating points ................................ 258
29.4 Floating point conditionals ............................ 259
29.5 Floating point expression loops ....................... 261
29.6 Symbolic expressions ................................ 263
29.7 User-defined functions ............................... 265
29.8 Some useful constants, and scratch variables ...... 266
29.9 Scratch variables ..................................... 266
29.10 Floating point exceptions ............................ 267
29.11 Viewing floating points ............................... 268
29.12 Floating point expressions .......................... 268
   29.12.1 Input of floating point numbers ............... 268
   29.12.2 Precedence of operators ....................... 269
   29.12.3 Operations ..................................... 270
29.13 Disclaimer and roadmap ............................. 277

# The l3fparray package: Fast global floating point arrays

30.1 l3fparray documentation .............................. 280

# The l3cctab package: Category code tables

31.1 Creating and initialising category code tables .... 281
31.2 Using category code tables ........................... 282
31.3 Category code table conditionals ..................... 282
31.4 Constant and scratch category code tables ......... 282

# Text manipulation

32.1 The l3unicode package: Unicode support functions ... 285
33 The \texttt{l3text} package: Text processing
  33.1 Expanding text ...................................... 288
  33.2 Case changing ....................................... 289
  33.3 Removing formatting from text ....................... 291
  33.4 Control variables .................................... 291
  33.5 Mapping to graphemes ................................ 292

VI Typesetting

34 The \texttt{l3box} package: Boxes ......................... 294
  34.1 Creating and initialising boxes ....................... 294
  34.2 Using boxes ......................................... 295
  34.3 Measuring and setting box dimensions ............... 296
  34.4 Box conditionals .................................... 297
  34.5 The last box inserted ................................ 297
  34.6 Constant boxes ..................................... 297
  34.7 Scratch boxes ...................................... 297
  34.8 Viewing box contents ................................ 298
  34.9 Boxes and color ..................................... 298
  34.10 Horizontal mode boxes ............................... 298
  34.11 Vertical mode boxes ................................ 299
  34.12 Using boxes efficiently ................................ 301
  34.13 Affine transformations .............................. 302
  34.14 Viewing part of a box ................................ 305
  34.15 Primitive box conditionals ......................... 306

35 The \texttt{l3coffins} package: Coffin code layer ......... 307
  35.1 Creating and initialising coffins .................... 307
  35.2 Setting coffin content and poles .................... 308
  35.3 Coffin affine transformations ....................... 309
  35.4 Joining and using coffins ............................ 310
  35.5 Measuring coffins ................................... 310
  35.6 Coffin diagnostics .................................. 311
  35.7 Constants and variables .............................. 312

36 The \texttt{l3color} package: Color support ............... 313
  36.1 Color in boxes ...................................... 313
  36.2 Color models ........................................ 313
  36.3 Color expressions .................................... 315
  36.4 Named colors ........................................ 316
  36.5 Selecting colors ..................................... 316
  36.6 Colors for fills and strokes ......................... 317
    36.6.1 Coloring math mode material .................... 317
  36.7 Multiple color models ................................ 317
  36.8 Exporting color specifications ...................... 318
  36.9 Creating new color models ......................... 319
    36.9.1 Color profiles .................................. 320
Part I
Introduction
Chapter 1

Introduction to expl3 and this document

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

1.1 Naming functions and variables

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

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

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

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

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

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

\begin{itemize}
\item This means \textit{expansion once}. In general, the \texttt{V} and \texttt{v} specifiers are favoured over \texttt{v} for recovering stored information. However, \texttt{v} 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{\textbackslash 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 \texttt{⟨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.
\end{itemize}

\texttt{\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.

\begin{itemize}
\item The letter \texttt{p} indicates \TeX\ \texttt{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{\texttt{D} stands for \texttt{Do not use}}. All of the \TeX\ primitives are initially \texttt{\textbackslash 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 \texttt{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:c} will take its argument, convert it to a control sequence and pass it to \texttt{\textbackslash foo:N}.

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

\footnote{If a primitive offers a functionality not yet in the kernel, programmers and users are encouraged to write to the \LaTeX-L mailing list (mailto:LATEX-L@listserv.uni-heidelberg.de) describing their use-case and intended behaviour, so that a possible interface can be discussed. Temporarily, while an interface is not provided, programmers may use the procedure described in the \texttt{\textbackslash lstyleguide.pdf}.}
c Constant: global parameters whose value should not be changed.

g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

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

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 \texttt{V}-type or \texttt{v}-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

bool Either true or false.

box Box register.

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

flag 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\_tmpa\_int} would be very unreadable.}
1.1.1 Scratch variables

Modules focussed on variable usage typically provide four scratch variables, two local and two global, with names of the form \(\langle\text{scope}\rangle_{\text{tmpa}}\langle\text{type}\rangle/\langle\text{scope}\rangle_{\text{tmpb}}\langle\text{type}\rangle\). 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. 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
   \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{\long} while token list variables are not.}
When a number of variants are described, the arguments are usually illustrated only for the base function. Here, (sequence) indicates that \seq_new:N expects the name of a sequence. From the argument specifier, \seq_new:c also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

**Fully expandable functions** Some functions are fully expandable, which allows them to be used within an \texttt{x}-type or \texttt{e}-type argument (in plain \TeX{} terms, inside an \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:

\texttt{\cs_to_str:N} \hspace{1em} \texttt{\cs_to_str:N (cs)}

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a \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:

\texttt{\seq_map_function:NN ⋆ \seq_map_function:NN (seq) (function)}

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

\texttt{\sys_if_engine_xetex:TF} \hspace{1em} \texttt{\sys_if_engine_xetex:TF {true code} {false code}}

The underlining and italic of 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 TF variant, and so both \texttt{(true code)} and \texttt{(false code)} will be shown. The two variant forms T and 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, all conditional functions in the expl3 modules should be defined in this way.

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

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

In some cases, the function is similar to one in \texttt{IfTeX} or plain \TeX{}. In these cases, the text will include an extra "\texttt{\TeX{}hackers note}" section:
The normal description text.

**TeX hackers note:** Detail for the experienced \TeX{} or \LaTeX{} \TeX2\ε programmer. In this case, it would point out that this function is the \TeX{} primitive `\string`.

**Changes to behaviour** When new functions are added to \exp3{}, 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 \exp3{} 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 \LaTeX{}3 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 \LaTeX{}3

The \TeX{} concept of an “\outer{}” macro is *not supported* at all by \LaTeX{}3. As such, the functions provided here may break when used on top of \LaTeX{}2\ε if \outer{} tokens are used in the arguments.
Part II
Bootstrapping
Chapter 2

The l3bootstrap package
Bootstrap code

2.1 Using the \LaTeX3 modules

The modules documented in this file (and source 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 \texttt{\usepackage{exp3}} or \texttt{\RequirePackage{exp3}} 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.

\begin{verbatim}
\ExplSyntaxOn
\ExplSyntaxOn \langle code \rangle \ExplSyntaxOff
\end{verbatim}

The \texttt{\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, \texttt{-} is used to input a space. The \texttt{\ExplSyntaxOff} reverts to the document category code regime.

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

\begin{verbatim}
\ProvidesExplPackage \ProvidesExplClass \ProvidesExplFile
\end{verbatim}

These functions act broadly in the same way as the corresponding \LaTeX2ε kernel functions \texttt{\ProvidesPackage}, \texttt{\ProvidesClass} and \texttt{\ProvidesFile}. However, they also implicitly switch \texttt{\ExplSyntaxOn} for the remainder of the code with the file. At the end of the file, \texttt{\ExplSyntaxOff} will be called to reverse this. (This is the same concept as \LaTeX2ε provides in turning on \texttt{\makeatletter} within package and class code.) The \texttt{\langle date \rangle} should be given in the format \texttt{(year)/(month)/(day)} or in the ISO date format \texttt{(year)-(month)-(day)}. If the \texttt{\langle version \rangle} is given then a leading \texttt{v} is optional: if given as a “pure” version string, a \texttt{v} will be prepended.
\setIdInfo $Id: \SVNinfo 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 \LaTeX2ε category codes and the \LaTeX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the \setIdInfo command you can use the information when loading a package with

\ProvidesExplPackage{\ExplFileName}
{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
Chapter 3

The l3names package
Namespace for primitives

3.1 Setting up the \LaTeX3 programming language

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

- defines new names for all \TeX{} primitives;
- emulate required primitives not provided by default in 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 \LaTeX3 code (outside of “kernel-level” code). As such, the primitives are not documented here: \TeXbook{}, \TeX{} by Topic and the manuals for pdf\TeX{}, \Xe\TeX{}, Lua\TeX{}, \p\TeX{} and up\TeX{} should be consulted for details of the primitives. These are named \texttt{\textbackslash tex\_\langle name\rangle} typically based on the primitive’s \texttt{\langle name\rangle} in pdf\TeX{} and omitting a leading \texttt{pdf} when the primitive is not related to pdf output.
Part III
Programming Flow
Chapter 4

The \texttt{l3basics} package

Basic definitions

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

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

4.1 No operation functions

\texttt{\prg_do_nothing}: \texttt{\prg_do_nothing:}

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

\texttt{\scan_stop}: \texttt{\scan_stop:}

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

4.2 Grouping material

\texttt{\group_begin}: \texttt{\group_end:}

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

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

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

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

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-\TeX primitive \showgroups.

\newpage

\section{Control sequences and functions}

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

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

\subsection{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, ...).

\texttt{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.

\texttt{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.

\texttt{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 \texttt{e}-type or \texttt{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:

- \texttt{N} and \texttt{n} No manipulation.
- \texttt{T} and \texttt{F} Functionally equivalent to \texttt{n} (you are actually encouraged to use the family of `\prg_new_conditional`: functions described in Section 9.1).
- \texttt{p} and \texttt{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

\begin{align*}
\texttt{\cs_new:Npn} & \texttt{\langle function \rangle \langle parameters \rangle \{\langle code \rangle\}} \\
\texttt{\cs_new:cpn} & \texttt{\cs_new:cpn} \\
\texttt{\cs_new:cpe} & \texttt{\cs_new:cpe} \\
\texttt{\cs_new:Npx} & \texttt{\cs_new:Npx} \\
\texttt{\cs_new:cpx} & \texttt{\cs_new:cpx} \\
\texttt{\cs_new_nopar:Npn} & \texttt{\cs_new_nopar:Npn} \texttt{\langle function \rangle \langle parameters \rangle \{\langle code \rangle\}} \\
\texttt{\cs_new_nopar:cpn} & \texttt{\cs_new_nopar:cpn} \\
\texttt{\cs_new_nopar:cpe} & \texttt{\cs_new_nopar:cpe} \\
\texttt{\cs_new_nopar:Npx} & \texttt{\cs_new_nopar:Npx} \\
\texttt{\cs_new_nopar:cpx} & \texttt{\cs_new_nopar:cpx} \\
\texttt{\cs_new_protected:Npn} & \texttt{\cs_new_protected:Npn} \texttt{\langle function \rangle \langle parameters \rangle \{\langle code \rangle\}} \\
\texttt{\cs_new_protected:cpn} & \texttt{\cs_new_protected:cpn} \\
\texttt{\cs_new_protected:cpe} & \texttt{\cs_new_protected:cpe} \\
\texttt{\cs_new_protected:Npx} & \texttt{\cs_new_protected:Npx} \\
\texttt{\cs_new_protected:cpx} & \texttt{\cs_new_protected:cpx}
\end{align*}

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. The definition is global and an error results if the \texttt{(function)} is already defined.
\cs_new_protected_nopar:Npn \cs_new_protected_nopar:Npn \langle \text{function} \rangle \langle \text{parameters} \rangle \{ \langle \text{code} \rangle \}

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

\cs_set:Npn \langle \text{function} \rangle \langle \text{parameters} \rangle \{ \langle \text{code} \rangle \}

Sets \langle \text{function} \rangle to expand to \langle \text{code} \rangle as replacement text. Within the \langle \text{code} \rangle, the \langle \text{parameters} \rangle (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \langle \text{function} \rangle is restricted to the current \LaTeX{} group level.

\cs_set_protected:Npn \langle \text{function} \rangle \langle \text{parameters} \rangle \{ \langle \text{code} \rangle \}

Sets \langle \text{function} \rangle to expand to \langle \text{code} \rangle as replacement text. Within the \langle \text{code} \rangle, the \langle \text{parameters} \rangle (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the \langle \text{function} \rangle is restricted to the current \LaTeX{} group level. The \langle \text{function} \rangle will not expand within an \texttt{e}-type or \texttt{x}-type argument.
4.3.3 Defining new functions using the signature

Globally sets \texttt{(function)} to expand to \texttt{(code)} as replacement text. Within the \texttt{(code)}, the \texttt{(parameters)} (#1, #2, \textit{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)} (#1, #2, \textit{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)} (#1, #2, \textit{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

\begin{verbatim}
\cs_new:Nn \cs_new: { (function) \{ (code) \}
\cs_new: { (parameters) \{ (code) \}
\end{verbatim}

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, \textit{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.

\begin{verbatim}
\cs_new: { (function) \( (parameters) \{ (code) \)
\end{verbatim}

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, \textit{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.
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.

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 definition is global and an error results if the \texttt{function} is already defined.

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.

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.
\cs_gset:Nn \cs_gset:(cn|Ne|ce) \cs_gset:Nn \langle function \rangle \{ (code) \}

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

\cs_gset_nopar:Nn \cs_gset_nopar:(cn|Ne|ce) \cs_gset_nopar:Nn \langle function \rangle \{ (code) \}

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

\cs_gset_protected:Nn \cs_gset_protected:(cn|Ne|ce) \cs_gset_protected:Nn \langle function \rangle \{ (code) \}

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

\cs_gset_protected_nopar:Nn \cs_gset_protected_nopar:(cn|Ne|ce) \cs_gset_protected_nopar:Nn \langle function \rangle \{ (code) \}

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

\cs_generate_from_arg_count:NNnn \cs_generate_from_arg_count:NNnn \langle function \rangle \langle creator \rangle \{ (number) \} \{ (code) \}

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

4.3.4 Copying control sequences

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

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

\textit{Updated: 2012-01-14}
\texttt{\cs_new_eq:NN} \langle \textit{cs} \rangle \langle \textit{cs}_1 \rangle \langle \textit{cs}_2 \rangle \langle \textit{token} \rangle

Globally creates \langle \textit{control sequence}_1 \rangle and sets it to have the same meaning as \langle \textit{control sequence}_2 \rangle or \langle \textit{token} \rangle. The second control sequence may subsequently be altered without affecting the copy.

\texttt{\cs_set_eq:NN} \langle \textit{cs} \rangle \langle \textit{cs}_1 \rangle \langle \textit{cs}_2 \rangle \langle \textit{token} \rangle

Sets \langle \textit{control sequence}_1 \rangle to have the same meaning as \langle \textit{control sequence}_2 \rangle (or \langle \textit{token} \rangle). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the \langle \textit{control sequence}_1 \rangle is restricted to the current \TeX{} group level.

\texttt{\cs_gset_eq:NN} \langle \textit{cs} \rangle \langle \textit{cs}_1 \rangle \langle \textit{cs}_2 \rangle \langle \textit{token} \rangle

Globally sets \langle \textit{control sequence}_1 \rangle to have the same meaning as \langle \textit{control sequence}_2 \rangle (or \langle \textit{token} \rangle). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the \langle \textit{control sequence}_1 \rangle 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.

\texttt{\cs_undefine:N} \langle \textit{control sequence} \rangle
\texttt{\cs_undefine:c}

Sets \langle \textit{control sequence} \rangle to be globally undefined.

4.3.6 Showing control sequences

\texttt{\cs_meaning:N} \langle \textit{control sequence} \rangle
\texttt{\cs_meaning:c}

This function expands to the meaning of the \langle \textit{control sequence} \rangle control sequence. For a macro, this includes the \langle \textit{replacement text} \rangle.

\textbf{TeX\textsc{hackers note}:} This is the \TeX{} primitive \texttt{\meaning} for tokens that are not control sequences, it is more logical to use \texttt{\token_to_meaning:N}. The \texttt{c} variant correctly reports undefined arguments.

\texttt{\cs_show:N} \langle \textit{control sequence} \rangle
\texttt{\cs_show:c}

Displays the definition of the \langle \textit{control sequence} \rangle on the terminal.

\textbf{TeX\textsc{hackers note}:} This is similar to the \TeX{} primitive \texttt{\show}, wrapped to a fixed number of characters per line.
4.3.7 Converting to and from control sequences

\use:c {⟨control sequence name⟩}

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

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

\use:c { a b c }

and

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

would be equivalent to

\abc

after two expansions of \use:c.

\cs_if_exist_use:N
\cs_if_exist_use:c
\cs_if_exist_use:NTF
\cs_if_exist_use:N

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

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

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

\TeXhackers note: These are the \TeX primitives \csname and \endcsname.

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

\cs:w a b c \cs_end:

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

would be equivalent to

\abc

after one expansion of \cs:w.

\cs_to_str:N \cs_to_str:N (control sequence)

Converts the given \textit{control sequence} into a series of characters with category code 12 (other), except spaces, of category code 10. The result does not include the current escape token, contrarily to \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 \textit{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 \texttt{function} into the \texttt{name} (i.e. the part before the colon) and the \texttt{signature} (i.e. after the colon). This information is then placed in the input stream in three parts: the \texttt{name}, the \texttt{signature} and a logic token indicating if a colon was found (to differentiate variables from function names). The \texttt{name} does not include the escape character, and both the \texttt{name} and \texttt{signature} are made up of tokens with category code 12 (other).

The next three functions decompose \LaTeX{} macros into their constituent parts: if the \texttt{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 \texttt{token} is a macro, this function leaves the applicable \LaTeX{} prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

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

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

\textbf{\TeX{} hackers note:} The prefix can be empty, \texttt{\long}, \texttt{\protected} or \texttt{\protected\long} with backslash replaced by the current escape character.
If the \langle 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}
\cs_set:Npn \next:nn \#1\#2 \{ x \#1 y \#2 \}
\cs_parameter_spec:N \next:nn
\end{verbatim}

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

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

---

If the \langle 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}
\cs_set:Npn \next:nn \#1\#2 \{ x \#1\_y \#2 \}
\cs_replacement_spec:N \next:nn
\end{verbatim}

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

\textbf{\TeX hackers 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
\begin{verbatim}
\use:n  \{ (group_1) \}
\use:nn \{ (group_1) \} \{ (group_2) \}
\use:nnn \{ (group_1) \} \{ (group_2) \} \{ (group_3) \}
\use:nnnn \{ (group_1) \} \{ (group_2) \} \{ (group_3) \} \{ (group_4) \}
\end{verbatim}
results in the input stream containing
\begin{verbatim}
\use:nn \{ abc \} \{ \{ def \} \}
\end{verbatim}

i.e. only the outer braces are removed.

\TeXhackers note: The \texttt{\use:n} function is equivalent to \LaTeX\texttt{2e}'s \texttt{@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, \(\text{use}_i:nn\) 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_ii:nnn { abc } { { def } } { ghi }
```

results in the input stream containing

```
abc { def }
```

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

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 \texttt{N} argument).

\textbf{\LaTeX}hackers note: These are equivalent to \LaTeX2e’s \texttt{@gobble}, \texttt{@gobbletwo}, etc.

\textbf{\LaTeX}hackers note: \texttt{\use:e} is a wrapper around the primitive \texttt{\expanded}. It requires two
expansions to complete its action.

### 4.5.1 Selecting tokens from delimited arguments

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

```
\use_none_delimit_by_q_nil:w * \use_none_delimit_by_q_nil:w \langle balanced text \rangle \texttt{\_q\_nil}
\use_none_delimit_by_q_stop:w * \use_none_delimit_by_q_stop:w \langle balanced text \rangle \texttt{\_q\_stop}
\use_none_delimit_by_q_recursion_stop:w * \use_none_delimit_by_q_recursion_stop:w \langle balanced text \rangle \texttt{\_q\_recursion\_stop}
```

Absorb the \texttt{\langle balanced text \rangle} 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

\textbf{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.

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

\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 \LaTeX. Their use is discouraged in expl3 (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

4.6.1 Tests on control sequences

\verb|\cs_if_eq_p:NN| \verb|\cs_if_eq_p:NN \langle cs_1 \rangle \langle cs_2 \rangle| \begin{verbatim} \cs_if_eq:NNTF \langle cs_1 \rangle \langle cs_2 \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \} \end{verbatim}

Compares the definition of two \emph{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 \verb|\cs_show:N|.

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

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

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

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

4.6.2 Primitive conditionals

The $\varepsilon$-\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 \verb|:w| part but higher level functions are often available. See for instance \verb|\int_compare_p:nNn| which is a wrapper for \verb|\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 \verb|\if_|, except for \verb|\if:w|.

\verb|\if_true:| \verb|\if_true: \{ \langle true code \rangle \} \else: \{ \langle false code \rangle \} \fi:|
\verb|\if_false:| \verb|\if_false: \{ \langle true code \rangle \} \else: \{ \langle false code \rangle \} \fi:|
\verb|\else:| \verb|\reverse_if:N \{ \langle primitive conditional \rangle \}|
\verb|\fi:| \verb|\if_true: \texttt{always executes \langle true code \rangle}, while \if_false: \texttt{always executes \langle false code \rangle}. \reverse_if:N \texttt{reverses any two-way primitive conditional. \else: and \fi: delimit the branches of the conditional. The function \verb|\or:| is documented in \verb|\l3int| and used in case switches.\TeXhackers note: \verb|\if_true:| and \verb|\if_false:| are equivalent to their corresponding \TeX primitive conditionals \texttt{iftrue} and \texttt{iffalse}; \verb|\else:| and \verb|\fi:| are the \TeX primitives \texttt{else} and \texttt{\fi}; \verb|\reverse_if:N| is the $\varepsilon$-\TeX primitive \texttt{unless}.
\if_meaning:w \if_meaning:w \langle arg_1 \rangle \langle arg_2 \rangle \langle true code \rangle \else: \langle false code \rangle \fi:
\if_meaning:w \if_meaning:w \langle token_1 \rangle \langle token_2 \rangle \langle true code \rangle \else: \langle false code \rangle \fi:
\if_catcode:w \if_catcode:w \langle token_1 \rangle \langle token_2 \rangle \langle true code \rangle \else: \langle false code \rangle \fi:
\if_mode_horizontal: \if_mode_horizontal: \langle true code \rangle \else: \langle false code \rangle \fi:
\if_mode_vertical: \if_mode_vertical: \langle true code \rangle \else: \langle false code \rangle \fi:
\if_mode_math: \if_mode_math: \langle true code \rangle \else: \langle false code \rangle \fi:
\if_mode_inner: \if_mode_inner: \langle true code \rangle \else: \langle false code \rangle \fi:

4.7 Starting a paragraph

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

\TeXhackers note: This results in the contents of the \everypar token register being inserted, after \mode_leave_vertical: is complete. Notice that in contrast to the \LaTeX{} approach, no box is used by the method implemented here.
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_suspend: ... \debug_resume: 
\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{13expan} package

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{}3 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\_}. 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.

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

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

- A few odd argument types remain: \textbf{T} and \textbf{F} for conditional processing, otherwise identical to \textbf{n}-type arguments, \textbf{p} for the parameter text in definitions, \textbf{w} for arguments with a specific syntax, and \textbf{D} to denote primitives that should not be used directly.
This function is used to define argument-specifier variants of the \langle parent control sequence \rangle for \LaTeX3 code-level macros. The \langle parent control sequence \rangle is first separated into the \langle base name \rangle and \langle original argument specifier \rangle. The comma-separated list of \langle variant argument specifiers \rangle is then used to define variants of the \langle original argument specifier \rangle if these are not already defined; entries which correspond to existing functions are silently ignored. For each \langle variant \rangle given, a function is created that expands its arguments as detailed and passes them to the \langle parent control sequence \rangle. 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 \foo:cn which expands its first argument into a control sequence name and passes the result to \foo:Nn. Similarly

\begin{verbatim}
\cs_generate_variant:Nn \foo:Nn { \foo:Nn \foo:Nn { \foo:Nn } \foo:Nn { \foo:Nn } }
\end{verbatim}

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

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

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

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

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

For backward compatibility it is currently possible to make \n, \o, \V, \v, \f, \e, or \x-type variants of an \N-type argument or \N or \c-type variants of an \n-type argument. Both are deprecated. The first because passing more than one token to an \N-type argument will typically break the parent function’s code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a \V-type or \v-type variant instead of \c-type. In those cases, using the lower-level \exp_args:No or \exp_args:Nc functions explicitly is preferred to defining confusing variants.
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 \texttt{#} 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 \texttt{#} 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 \texttt{3 + 4} and pass the result \texttt{7} as an argument to an expandable function \texttt{example:n}. For this, one should define a variant using \texttt{cs\_generate\_variant:NN} \texttt{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 \texttt{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: \{ 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 internal
functions for argument expansion come in two flavours, some faster than others.

• Arguments that might need expansion should come first in the list of arguments.

• Arguments that should consist of single tokens N, c, V, or v should come first among
these.

• Arguments that appear after the first multi-token argument n, f, e, or o require
slightly slower special processing to be expanded. Therefore it is best to use the
optimized functions, namely those that contain only N, c, V, and v, and, in the last
position, o, f, e, with possible trailing N or n or T or F, which are not expanded.
Any x-type argument causes slightly slower processing.

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

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

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

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

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

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

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.

5.6 Manipulating three arguments

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 and third arguments, etc.
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, \textit{etc.} These functions need slower processing.

\textbf{\LaTeX}hackers 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 \{ \} \q_stop} leads to an infinite loop, as the quark is \texttt{f}-expanded.

5.7 \textbf{Unbraced expansion}
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.

Carries out a single expansion of \langle token_2 \rangle (which may consume arguments) prior to the expansion of \langle token_1 \rangle. If \langle token_2 \rangle has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that \langle token_1 \rangle may be any single token, including group-opening and -closing tokens (\{ or \} assuming normal \LaTeX{} 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 function.

\TeX{}hackers note: This is the \LaTeX{} primitive \expandafter.

5.8 Preventing expansion

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

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.

\TeX{}hackers note: This is the \LaTeX{} 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 (\cs_new:Np), 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 \{\tokens\}

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

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

\exp_not:o = \exp_not:o \{\tokens\}

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

\exp_not:V = \exp_not:V \langle\text{variable}\rangle

Recovers the content of the \langle\text{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 \{\tokens\}

Expands the \tokens until only characters remains, and then converts this into a con-
trol sequence which should be a \langle\text{variable}\rangle name. The content of the \langle\text{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 \{\tokens\}

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

\exp_not:f = \exp_not:f \{\tokens\}

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

\exp_stop_f: = \foo_bar:f \{\tokens\} \exp_stop_f: \{\more\tokens\}

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

Updated: 2011-06-03
5.9 Controlled expansion

The \texttt{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 \texttt{\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 documented above should usually be sufficient.

While \texttt{\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 \texttt{⟨expandable-tokens⟩} as that will break badly if unexpandable tokens are encountered in that place!

\begin{verbatim}
\exp:w  \exp:w ⟨expandable tokens⟩ \exp_end:
\end{verbatim}

Expands \texttt{⟨expandable tokens⟩} until reaching \texttt{\exp_end:} at which point expansion stops. The full expansion of \texttt{⟨expandable tokens⟩} has to be empty. If any token in \texttt{⟨expandable tokens⟩} or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result \texttt{\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 \texttt{\exp_end:} is hidden somewhere in the replacement text of \texttt{⟨expandable-tokens⟩} rather than being on the same expansion level than \texttt{\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 \texttt{\@@_case:NnTF} the \texttt{\exp_end:} gets generated.

\texttt{\TeX} hackers note: The current implementation uses \texttt{\romannumeral} hence ignores space tokens and explicit signs + and – in the expansion of the \texttt{⟨expandable tokens⟩}, but this should not be relied upon.

\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!}
\exp:w * \exp:w \langle \text{expandable-tokens} \rangle \exp_end_continue_f:w \langle \text{further-tokens} \rangle

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

The full expansion of $\langle \text{expandable-tokens} \rangle$ 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.\footnote{In this particular case you may get a character into the output as well as an error message.}

In typical use cases $\langle \text{expandable-tokens} \rangle$ contains no tokens at all, e.g., you will see code such as

$$\exp_after:wN \{ \exp:w \exp_end_continue_f:w \texttt{#2} \}$$

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

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

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

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

$$\exp:w \exp_end_continue_f:w \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 \langle \text{expandable-tokens} \rangle \exp_end_continue_f:nw *

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

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 $f$-type expansion.
5.10 Internal functions

\cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }
\cs_new:Npn \exp_args:Nno { \::n \::o_unbraced \::: }

Internal forms for the base expansion types. These names do not conform to the general \TeX3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

\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 \TeX3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.
Chapter 6

The \texttt{l3sort} package

Sorting functions

6.1 Controlling sorting

\LaTeX{} 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 \#1 and \#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 \texttt{⟨comparison code⟩} 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 \texttt{⟨comparison code⟩} consisting only of \texttt{\sort_return_swapped}: reverses the list (in a fairly inefficient way).

\textbf{\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 \texttt{⟨comparison code⟩} 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.
\texttt{\texttt{\textbackslash sort\_return\_same}: \texttt{\textbackslash seq\_sort:nn (seq \texttt{var})}}
\texttt{\texttt{\textbackslash sort\_return\_swapped: \{ ... \texttt{\textbackslash sort\_return\_same: or \texttt{\textbackslash sort\_return\_swapped: ... \}}}}

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the \texttt{\textbackslash 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 \texttt{l3tl-analysis} package: Analysing token lists

This module provides functions that are particularly useful in the \texttt{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 \texttt{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}:

- \texttt{\langle tokens \rangle}, which both \texttt{o-expand} and \texttt{e/x-expand} to the \texttt{token}. The detailed form of \texttt{\langle tokens \rangle} may change in later releases.

- \texttt{\langle char code \rangle}, a decimal representation of the character code of the \texttt{token}, $-1$ if it is a control sequence.

- \texttt{\langle catcode \rangle}, 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 catcode \rangle}".

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 \langle token list \rangle
\tl_analysis_log:n \langle token list \rangle
\tl_analysis_log:N \langle token list \rangle
\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 \langle token list \rangle \langle inline function \rangle
\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.
Chapter 8

The \l3regex package: Regular expressions in \TeX

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

Let us give a few examples. After

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

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

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

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

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

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

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

8.1.1 Regular expression examples

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

- **Cat** matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use \bCat\b to match a complete word only.

- **[abc]** matches one letter among “a”, “b”, “c”; the pattern (a|b|c) matches the same three possible letters (but see the discussion of submatches below).

- **[A-Za-z]*** matches any number (due to the quantifier *) of Latin letters (not accented).

- \c[A-Za-z]*\ matches a control sequence made of Latin letters.

- \_\[\_\]*\_ matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to \_\.*?\_ where \_ matches arbitrary characters and the lazy quantifier *? means to match as few characters as possible, thus avoiding matching underscores.

- **\?[\+-\d+\-\*\d+\-\*\]*/\[\+-\d+\-\*\d+\-\*\]** 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\+\-\*\]** matches an explicit integer or decimal number; using \[\.,\] instead of \. would allow the comma as a decimal marker.

- **\?[\+-\d\+\-\*\d\+\-\*\]** matches an explicit dimension with any unit that \TeX\ knows, where \(?i\) means to treat lowercase and uppercase letters identically.

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

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

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

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

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

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

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

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

Characters.

 Benton.\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].
Any vertical space character, equivalent to \[\ ^{\text{\textbackslash \textkaux\textbackslash \textkim\textbackslash \textl\textbackslash \textm}}\]. Note that \textbackslash \textk is a vertical space, but not a space, for compatibility with Perl.

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

Any token not matched by \textbackslash d.

Any token not matched by \textbackslash h.

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

Any token not matched by \textbackslash s.

Any token not matched by \textbackslash v.

Any token not matched by \textbackslash w.

Of those, \texttt{, \textbackslash d, \textbackslash h, \textbackslash n, \textbackslash s, \textbackslash v, \textbackslash w} match arbitrary control sequences.

Character classes match exactly one token in the subject.

\texttt{[...]} Positive character class. Matches any of the specified tokens.

\texttt{[^...]} Negative character class. Matches any token other than the specified characters.

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

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

\texttt{[:^\textit{name}:]} Negative POSIX character class.

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

In character classes, only \texttt{[}, \texttt{\^}, \texttt{\_}, \texttt{\} 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 (\texttt{\either\textbackslash d, \textbackslash d, etc.}) is supported in character classes.

If the first character is \texttt{\^}, then the meaning of the character class is inverted; \texttt{\^} appearing anywhere else in the range is not special. If the first character (possibly following a leading \texttt{\^}) is \texttt{[} then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using \texttt{\textbackslash \textd0-5} and \texttt{[^6-9]} are equivalent.

\textbf{8.1.4 Structure: alternatives, groups, repetitions}

Quantifiers (repetition).

\texttt{?} 0 or 1, greedy.

\texttt{??} 0 or 1, lazy.

\texttt{*} 0 or more, greedy.

\texttt{??} 0 or more, lazy.

\texttt{+} 1 or more, greedy.
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.

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

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

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

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

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

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

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

8.1.5 Matching exact tokens

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

- C for control sequences;
- B for begin-group tokens;
- E for end-group tokens;
• M for math shift;
• T for alignment tab tokens;
• P for macro parameter tokens;
• U for superscript tokens (up);
• D for subscript tokens (down);
• S for spaces;
• L for letters;
• 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 CBEMTPUDSLOA). For instance, \cL[A-Z\d] matches uppercase letters and digits of category code letter, \cC. matches any control sequence, and \cO(abc) matches where each character has category other.6

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

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

The category code tests can be used inside classes; for instance, [\cO\d \c[L0][A-F]] matches what \pX considers as hexadecimal digits, namely digits with category other, or uppercase letters from A to F with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, \cL(ab\cO\*cd) matches ab\cd where all characters are of category letter, except * which has category other.

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

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

---

6This last example also captures “abc” as a regex group; to avoid this use a non-capturing group \cO(?:abc).
D separated by something that matches the regular expression \_tmpa_regex. This behaves as if a non-capturing group were surrounding \_tmpa_regex, and any group contained in \_tmpa_regex is converted to a non-capturing group. Quantifiers are supported.

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

\begin{verbatim}
\regex_show:n { A \ur{\_mymodule_BC_tl} D }
\regex_show:n { A B | C D }
\end{verbatim}

8.1.6 Miscellaneous

Anchors and simple assertions.

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

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

^ or \A Start of the subject token list.

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

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

The option \(?i) makes the match case insensitive (treating A–Z and a–z as equivalent, with no support yet for Unicode case changing). This applies until the end of the group in which it appears, and can be reverted using \(?-i). For instance, in \(?i)(a(?-i)b\{|c\}d, the letters a and d are affected by the i option. Characters within ranges and classes are affected individually: \(?i)\{\?-B\} is equivalent to \{\?-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:]].

8.2 Syntax of the replacement text

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

- \0 is the whole match;
- \1 is the submatch that was matched by the first (capturing) group (...); similarly for \2, ..., \9 and \g{number};
- \l inserts a space (spaces are ignored when not escaped);
• \a, \e, \f, \n, \r, \t, \xhh, \x{hhh} correspond to single characters as in regular expressions;
• \c{⟨cs name⟩} inserts a control sequence;
• \c{⟨category⟩}{⟨character⟩} (see below);
• \u{⟨tl var name⟩} inserts the contents of the ⟨tl var⟩ (see below).

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

For instance,
\begin{verbatim}
\tl_set:Nn \l_my_tl { Hello,-world! }
\regex_replace_all:nN { ([er]?l|o) . } { (\0--\1) } \l_my_tl
\end{verbatim}
results in \l_my_tl holding H(ell--el)(o,--o) w(or--o)(ld--l)!

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

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

• space characters (with character code 32) inserted with \x{20} 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.

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

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,
\texttt{\textbackslash{}l\textunderscore{}set:Nn \_l\_my\_one\_tl \{ \textit{first} \}}
\texttt{\textbackslash{}l\textunderscore{}set:Nn \_l\_my\_two\_tl \{ \textbf{second} \}}
\texttt{\textbackslash{}l\textunderscore{}set:Nn \_l\_my\_tl \{ one , two , one , one \}}
\texttt{\textbackslash{}regex\textunderscore{}replace\textunderscore{}all:nnN \{ \[^{,}]+ \} \{ \textup{u\{l\_my\_0\_tl\} } \} \_l\_my\_tl}

results in \_l\_my\_tl holding \textit{first},\textbf{second},first,first.

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

\texttt{\textbackslash{}l\textunderscore{}clear:N \l\_tmpa\_tl}
\texttt{\textbackslash{}regex\textunderscore{}replace\textunderscore{}all:nnN \{ \} \{ \textup{cU\% \textbackslash{}cA\~} \} \_l\_tmpa\_tl}

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

### 8.3 Pre-compiling regular expressions

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

\begin{itemize}
  \item \texttt{\textbackslash{}regex\textunderscore{}new:N} \langle \textbackslash{}regex \text{var} \rangle
    \begin{itemize}
      \item New: 2017-05-26
    \end{itemize}

  Creates a new \langle \textbackslash{}regex \text{var} \rangle or raises an error if the name is already taken. The declaration is global. The \langle \textbackslash{}regex \text{var} \rangle is initially such that it never matches.

  \item \texttt{\textbackslash{}regex\textunderscore{}set:Nn} \langle \textbackslash{}regex \text{var} \rangle \{ \langle \text{regex} \rangle \}
    \begin{itemize}
      \item New: 2017-05-26
    \end{itemize}

    Stores a compiled version of the \langle \text{regular expression} \rangle in the \langle \textbackslash{}regex \text{var} \rangle. The assignment is local for \texttt{\textbackslash{}regex\textunderscore{}set:Nn} and global for \texttt{\textbackslash{}regex\textunderscore{}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\ )? \textbackslash{}reg\{ex\|ular\ expression\} \}
    \end{verbatim}

  \item \texttt{\textbackslash{}regex\textunderscore{}const:Nn} \langle \textbackslash{}regex \text{var} \rangle \{ \langle \text{regex} \rangle \}
    \begin{itemize}
      \item New: 2017-05-26
    \end{itemize}

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

\begin{itemize}
  \item [+]-branch anchor at start (\A)
  \item char code 88 (X)
  \item [+]-branch char code 89 (Y)
\end{itemize}

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 \texttt{\regex_set:Nn}.

\begin{verbatim}
\regex_match:nnTF {⟨regex⟩} {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
\end{verbatim}

Tests whether the \texttt{(regular expression)} matches any part of the \texttt{(token list)}. For instance,

\begin{verbatim}
\regex_match:nnTF { b \[cde]* } { abecedcx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
\end{verbatim}

leaves \texttt{TRUE} then \texttt{FALSE} in the input stream.

\begin{verbatim}
\regex_count:nnN {⟨regex⟩} {⟨token list⟩} {⟨int var⟩}
\end{verbatim}

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

\begin{verbatim}
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcb } \l_foo_int
\end{verbatim}

results in \texttt{\l_foo_int} taking the value 5.
Determines which of the \langle regular expressions \rangle matches at the earliest point in the \langle token list \rangle, and leaves the corresponding \langle code \rangle followed by the \langle true code \rangle in the input stream. If several \langle regex \rangle match starting at the same point, then the first one in the list is selected and the others are discarded. If none of the \langle regex \rangle match, the \langle false code \rangle is left in the input stream. Each \langle regex \rangle can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the \langle token list \rangle, each of the \langle regex \rangle is searched in turn. If one of them matches then the corresponding \langle code \rangle is used and everything else is discarded, while if none of the \langle regex \rangle match at a given position then the next starting position is attempted. If none of the \langle regex \rangle match anywhere in the \langle token list \rangle then nothing is left in the input stream. Note that this differs from nested \regex_match_case:nnTF statements since all \langle regex \rangle are attempted at each position rather than attempting to match \langle regex1 \rangle at every position before moving on to \langle regex2 \rangle.

## 8.5 Submatch extraction

\regex_extract_once:nnN \regex_extract_once:nVN \regex_extract_once:nnNTF \regex_extract_once:nVN \regex_extract_once:NVN \regex_extract_once:NVN \regex_extract_once:NVN

<table>
<thead>
<tr>
<th>New: 2017-05-26</th>
</tr>
</thead>
</table>

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

For instance, assume that you type

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

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

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

\section*{8.6 Replacement}

\begin{Verbatim}
\regex_replace_once:nnN \{ \textit{regular expression} \} \{ \textit{replacement} \} \{ \textit{tl var} \}
\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}, \textbackslash 0 represents the full match, \textbackslash 1 represent the contents of the first capturing group, \textbackslash 2 of the second, \textit{etc}. The result is assigned locally to \textit{tl var}.  

New: 2017-05-26

New: 2017-05-26
\regex_replace_all:nnN \{\textit{regular expression}\} \{\textit{replacement}\} \{\textit{tl var}\}
\regex_replace_all:nVN \{\textit{regular expression}\} \{\textit{replacement}\} \{\textit{true code}\} \{\textit{false code}\}
\regex_replace_all:nN \{\textit{regular expression}\} \{\textit{replacement}\} \{\textit{true code}\} \{\textit{false code}\}

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

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

\regex_replace_case_once:nN \{\textit{regex}\} \{\textit{replacement}\} \{\textit{true code}\} \{\textit{false code}\}

Replaces the earliest match of the regular expression \texttt{(?|\textit{regex}|\ldots|\textit{regex})} in the \textit{token list variable} by the \textit{replacement} corresponding to which \textit{regex} matched, then leaves the \textit{true code} in the input stream. If none of the \textit{regex} match, then the \textit{tl var} is not modified, and 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 it is replaced by the corresponding \textit{replacement} as described for \texttt{\regex_replace_once:nnN}. This is equivalent to checking with \texttt{\regex_match_case:nn} which \textit{regex} matches, then performing the replacement with \texttt{\regex_replace_once:nnN}.
Replaces all occurrences of all regex in the token list by the corresponding replacement. Every match is treated independently, and matches cannot overlap. The result is assigned locally to tl var, and the true code or false code is left in the input stream depending on whether any replacement was made or not.

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

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

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

8.7 Scratch regular expressions

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

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

8.8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.
• Rewrite the documentation in a more ordered way, perhaps add a BNF?
  Additional error-checking to come.
• Clean up the use of messages.
• Cleaner error reporting in the replacement phase.
• Add tracing information.
• Detect attempts to use back-references and other non-implemented syntax.
• Test for the maximum register \c_max_register_int.
• Find out whether the fact that \W and friends match the end-marker leads to bugs.
  Possibly update \__regex_item_reverse:n.
• The empty cs should be matched by \c{}, not by \c{csname.?endcsname\s?}.
  Code improvements to come.
• Shift arrays so that the useful information starts at position 1.
• Only build \c{...} once.
• Use arrays for the left and right state stacks when compiling a regex.
• Should \__regex_action_free_group:n only be used for greedy \{n,\} quantifier?
  (I think not.)
• Quantifiers for \u and assertions.
• When matching, keep track of an explicit stack of curr_state and curr_-_submatches.
• If possible, when a state is reused by the same thread, kill other subthreads.
• Use an array rather than \g__regex_balance_tl to build the function \__regex_-_replacement_balance_one_match:n.
• Reduce the number of epsilon-transitions in alternatives.
• Optimize simple strings: use less states (abcade should give two states, for abc and ade). [Does that really make sense?]
• Optimize groups with no alternative.
• Optimize states with a single \__regex_action_free:n.
• Optimize the use of \__regex_action_success: by inserting it in state 2 directly instead of having an extra transition.
• Optimize the use of \int_step.... functions.
• Groups don’t capture within regexes for csnames; optimize and document.
• Better “show” for anchors, properties, and catcode tests.
• Does \K really need a new state for itself?
• When compiling, use a boolean `in_cs` and less magic numbers.
• 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 treeboxes.

The following features of PCRE or Perl may or may not be implemented.
• Callout with `(\C..)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break` and then of playing well with `\tl_map_break`: called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.
• Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn’t it?
• Named subpatterns: TEX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.
• Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
• Recursion: this is a non-regular feature.
• Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
• Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
• Backtracking control verbs: intrinsically tied to backtracking.
• \ddd, matching the character with octal code ddd: we already have \x{...} and
the syntax is confusingly close to what we could have used for backreferences (\1,
\2, ...), making it harder to produce useful error message.

• \cx, similar to \TeX’s own \^"x.

• Comments: \TeX already has its own system for comments.

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

• \C single byte in UTF-8 mode: Xe\TeX and Lua\TeX serve us characters directly,
and splitting those into bytes is tricky, encoding dependent, and most likely not
useful anyways.
Chapter 9

The l3prg package

Control structures

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

\LaTeX3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean \texttt{⟨true⟩} or \texttt{⟨false⟩}. 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 \texttt{⟨true⟩} or \texttt{⟨false⟩} 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 \texttt{N}) and then executes either \texttt{true} or \texttt{false} depending on the result.

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

9.1 Defining a set of conditional functions

\begin{verbatim}
\prg_new_conditional:Npnn \prg_set_conditional:Npnn \prg_gset_conditional:Npnn \prg_new_conditional:Nnn \prg_set_conditional:Nnn \prg_gset_conditional:Nnn
\end{verbatim}

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

The conditionals are defined by \texttt{\prg_new_protected_conditional:Npnn} and friends as:

- \langle\text{name}\rangle\_p: \langle\text{arg spec}\rangle — a predicate function which will supply either a logical \texttt{true} or logical \texttt{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.

- \langle\text{name}\rangle: \langle\text{arg spec}\rangle T — a function with one more argument than the original \langle\text{arg spec}\rangle demands. The \langle\text{true branch}\rangle code in this additional argument will be left on the input stream only if the test is \texttt{true}.

- \langle\text{name}\rangle: \langle\text{arg spec}\rangle F — a function with one more argument than the original \langle\text{arg spec}\rangle demands. The \langle\text{false branch}\rangle code in this additional argument will be left on the input stream only if the test is \texttt{false}.

- \langle\text{name}\rangle: \langle\text{arg spec}\rangle TF — a function with two more argument than the original \langle\text{arg spec}\rangle demands. The \langle\text{true branch}\rangle code in the first additional argument will be left on the input stream if the test is \texttt{true}, while the \langle\text{false branch}\rangle code in the second argument will be left on the input stream if the test is \texttt{false}.

The \langle\text{code}\rangle of the test may use \langle\text{parameters}\rangle as specified by the second argument to \texttt{\prg_set_protected_conditional:Npnn}; this should match the \langle\text{argument specification}\rangle but this is not enforced. The \texttt{Nnn} versions infer the number of arguments from the argument specification given (\emph{cf.} \texttt{\cs_new:Nn}, \emph{etc.}). Within the \langle\text{code}\rangle, the functions \texttt{\prg_return_true:} and \texttt{\prg_return_false:} are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

\verbatim
\prg_set_protected_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: 
}
This defines the function \texttt{\textbackslash foo\_if\_bar\_p:NN}, \texttt{\textbackslash foo\_if\_bar:NNTF} and \texttt{\textbackslash foo\_if\_bar:NNT} but not \texttt{\textbackslash foo\_if\_bar:NNF} (because F is missing from the \langle conditions \rangle 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{tabular}{l}
\texttt{\textbackslash prg\_new\_eq\_conditional:NNn}
\texttt{\langle name \rangle: \langle arg \ spec \rangle}
\texttt{\langle name \rangle: \langle arg \ spec \rangle}
\texttt{\langle conditions \rangle}
\end{tabular}

\begin{tabular}{l}
\texttt{\textbackslash prg\_set\_eq\_conditional:NNn}
\texttt{\langle conditions \rangle}
\end{tabular}

\begin{tabular}{l}
\texttt{\textbackslash prg\_gset\_eq\_conditional:NNn}
\end{tabular}

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

\begin{tabular}{l}
\texttt{\textbackslash prg\_return\_true: \star \textbackslash prg\_return\_true:}
\end{tabular}

\begin{tabular}{l}
\texttt{\textbackslash prg\_return\_false: \star \textbackslash prg\_return\_false:}
\end{tabular}

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

The return functions trigger what is internally an f-expansion process to complete the evaluation of the conditional. Therefore, after \texttt{\textbackslash prg\_return\_true:} or \texttt{\textbackslash 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{tabular}{l}
\texttt{\textbackslash prg\_generate\_conditional\_variant:Nmn}
\texttt{\langle name \rangle: \langle arg \ spec \rangle}
\texttt{\langle variant argument specifiers \rangle \langle condition specifiers \rangle}
\end{tabular}

Defines argument-specifier variants of conditionals. This is equivalent to running \texttt{\textbackslash cs\_generate\_variant:Nn \langle conditional \rangle \langle variant argument specifiers \rangle \langle condition specifiers \rangle} on each \langle conditional \rangle described by the \langle condition specifiers \rangle. These base-form \langle conditionals \rangle are obtained from the \langle name \rangle and \langle arg spec \rangle as described for \texttt{\textbackslash prg\_new\_conditional:Npnn}, and they should be defined.

\subsection{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{\textbackslash if\_predicate:w} test. The problem of the primitive \texttt{\textbackslash if\_false:} and \texttt{\textbackslash 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: \texttt{\_true\_bool} or \texttt{\_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 \texttt{\_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).

\textbf{\TeX} hackers note: The \texttt{bool} data type is not implemented using the \texttt{\_iffalse/\_iftrue} primitives, in contrast to \texttt{\_nevfif}, etc., in plain \TeX, \LaTeX and so on. Programmers should not base use of \texttt{bool} switches on any particular expectation of the implementation.

\begin{verbatim}
\bool_new:N \bool_new:c
\bool_const:Nn \bool_const:cn
\bool_set_false:N \bool_set_false:c \bool_gset_false:N \bool_gset_false:c
\bool_set_true:N \bool_set_true:c \bool_gset_true:N \bool_gset_true:c
\bool_set_eq:NN \bool_set_eq: (cN|Nc|cc)
\bool_gset_eq:NN \bool_gset_eq: (cN|Nc|cc)
\bool_set:Nn \bool_set:cn \bool_gset:Nn \bool_gset:cn
\bool_set_inverse:N \bool_set_inverse:c \bool_gset_inverse:N \bool_gset_inverse:c
\end{verbatim}

Updated: 2017-07-15

Updated: 2017-11-28

Updated: 2018-05-10

New: 2018-05-10

67
Tests the current truth of \texttt{boolean}, and continues expansion based on this result.

Expands to the letters \texttt{true} or \texttt{false} depending on the logical truth of the \texttt{boolean} or \texttt{boolean expression}.

Displays the logical truth of the \texttt{boolean} on the terminal.

Displays the logical truth of the \texttt{boolean expression} on the terminal.

Writes the logical truth of the \texttt{boolean} in the log file.

Writes the logical truth of the \texttt{boolean expression} in the log file.

Tests whether the \texttt{boolean} is currently defined. This does not check that the \texttt{boolean} really is a boolean variable.

\section*{9.2.1 Constant and scratch booleans}

\texttt{\ctruebool} \texttt{\cfalsebool}

\texttt{\ltmpabool} \texttt{\ltmpbbool}

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

9.3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean \( \langle \text{true} \rangle \) or \( \langle \text{false} \rangle \) values, it seems only fitting that we also provide a parser for \( \langle \text{boolean expressions} \rangle \).

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean \( \langle \text{true} \rangle \) or \( \langle \text{false} \rangle \). 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 \{ \text{1 = 1} \} \&\&
 ( \int_compare_p:n \{ \text{2 = 3} \} ||
  \int_compare_p:n \{ \text{4 <= 4} \} ||
  \str_if_eq_p:nn \{ \text{abc} \} \{ \text{def} \}
 ) \&\&
 ! \int_compare_p:n \{ \text{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 \texttt{\bool_lazy_...} functions.

\texttt{\TeX} hackers 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 \texttt{\bool_lazy_all:nTF}, \texttt{\bool_lazy_and:nnTF}, \texttt{\bool_lazy_any:nTF}, or \texttt{\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_all_p:nn
   
   \bool_lazy_any_p:n
   
   { \int_compare_p:n \{ \text{2 = 3} \} }
   
   { \int_compare_p:n \{ \text{4 <= 4} \} }
   
   { \int_compare_p:n \{ \text{1 = \text{error}} \} } % skipped
   
   ! \int_compare_p:n \{ \text{2 = 4} \}
 }
\end{verbatim}

69
the line marked with skipped is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically true. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

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

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

```latex
\bool_lazy_and_p:nn \bool_lazy_and:nnTF \...
```
Implements the "And" operation between two boolean expressions, hence is true if both are true. Contrarily to the infix operator \&\&, the \textit{boolean expressions} which are needed to determine the result of `\bool_lazy_and:nnTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are more than two \textit{boolean expressions}.

```latex
\bool_lazy_any_p:n \bool_lazy_any:nTF \...
```
Implements the "Or" operation on the \textit{boolean expressions}, hence is true if any of them is true and false if all of them are false. Contrarily to the infix operator ||, only the \textit{boolean expressions} which are needed to determine the result of `\bool_lazy_any:nTF` are evaluated. See also `\bool_lazy_any:nTF` when there are only two \textit{boolean expressions}.

```latex
\bool_lazy_or_p:nn \bool_lazy_or:nnTF \...
```
Implements the "Or" operation between two boolean expressions, hence is true if either one is true. Contrarily to the infix operator ||, the \textit{boolean expressions} which are needed to determine the result of `\bool_lazy_or:nnTF` are evaluated. See also `\bool_lazy_or:nnTF` when there are more than two \textit{boolean expressions}.

```latex
\bool_not_p:n \bool_not:p:n
```
Function version of !(\textit{boolean expression}) within a boolean expression.
\bool_xor_p:nn {\langle boolexpr \rangle} {\langle boolexpr \rangle}
\bool_xor:nnTF {\langle boolexpr \rangle} {\langle boolexpr \rangle} {\langle true \ code \rangle} {\langle false \ code \rangle}

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

9.4 Logical loops

Loops using either boolean expressions or stored boolean values.

\bool_do_until:Nn \bool_do_until:cn
\bool_do_while:Nn \bool_do_while:cn
\bool_until_do:Nn \bool_until_do:cn
\bool_while_do:Nn \bool_while_do:cn
\bool_do_until:nn \bool_do_until:cn
\bool_do_while:nn \bool_do_while:cn
\bool_until_do:nn \bool_until_do:cn
\bool_while_do:nn

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

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

This function first checks the logical value of the \langle boolean expression \rangle as described for \bool_if:nTF. If it is false then the \langle code \rangle is inserted into the input stream again and the process loops until the \langle boolean expression \rangle evaluates to true.

This function first checks the logical value of the \langle boolean expression \rangle as described for \bool_if:nTF. If it is true then the \langle code \rangle is inserted into the input stream again and the process loops until the \langle boolean expression \rangle evaluates to false.

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

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

\bool_case:n \bool_case:nTF
\bool_case:nTF *

\texttt{\{\texttt{expr case 1}\} \{\texttt{code case 1}\}}
\texttt{\{\texttt{expr case 2}\} \{\texttt{code case 2}\}}
\ldots
\texttt{\{\texttt{expr case n}\} \{\texttt{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 \textit{true}. The \textit{(code)} associated to this first case is left in the input stream, followed by the \textit{(true code)}, and other cases are discarded. If none of the cases match then only the \textit{(false code)} is inserted. The function \bool_case:n, which does nothing if there is no match, is also available. For example

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

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

\section{Producing multiple copies}

\prg_replicate:nn \prg_replicate:nn {(integer expression)} {(tokens)}

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

\section{Detecting \LaTeX’s mode}

\mode_if_horizontal_p:\mode_if_horizontal_p:
\mode_if_horizontal_p:
\mode_if_horizontal_p:
\mode_if_horizontal_p:TF \mode_if_horizontal_p:TF \{true code\} \{false code\}

Detects if \LaTeX{} is currently in horizontal mode.
9.7 Primitive conditionals

\if_predicate:w\langle predicate\rangle\langle true code\rangle\else:\langle false code\rangle\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 \langle predicate\rangle but
to make the coding clearer this should be done through \if_bool:N.)

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

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

9.8 Nestable recursions and mappings

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

\prg_break_point:Nn\langle type\rangle_map_break:\langle code\rangle

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

... \texttt{\textbackslash prg\_break\_point:Nn \langle type\rangle\_map\_break: \{(ending code)\}}

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

For types with mappings defined in the kernel, \texttt{\langle type\rangle\_map\_break:} and \texttt{\langle type\rangle\_map\_break:n} are defined as \texttt{\textbackslash 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.

\texttt{\textbackslash prg\_break\_point:} \texttt{*} \texttt{\textbackslash prg\_do\_nothing:} is used to mark the end of a fast short-term recursion: the function \texttt{\textbackslash prg\_break:n} uses this to break out of the loop.

\texttt{\textbackslash prg\_break:} \texttt{*} \texttt{\textbackslash prg\_break:n \{(code)\} ... \textbackslash prg\_break\_point:}

This copy of \texttt{\textbackslash prg\_do\_nothing:} is used to mark the end of a fast short-term recursion: the function \texttt{\textbackslash prg\_break:n} uses this to break out of the loop.

\texttt{\textbackslash prg\_break:n} \texttt{*} \texttt{\textbackslash prg\_break\_point:}

Breaks a recursion which has no \texttt{(ending code)} and which is not a user-breakable mapping (see for instance implementation of \texttt{\textbackslash int\_step\_function:nnnN}), and inserts the \texttt{(code)} in the input stream.

9.9 Internal programming functions

\texttt{\textbackslash group\_align\_safe\_begin:} \texttt{*} \texttt{\textbackslash 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 $k$ token inside \texttt{\textbackslash 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 \texttt{\textbackslash peek\_after:Nw} would result in a forbidden comparison of the internal \texttt{\textbackslash end\_template} token, yielding a fatal error. Each \texttt{\textbackslash group\_align\_safe\_begin:} must be matched by a \texttt{\textbackslash group\_align\_safe\_end:}, although this does not have to occur within the same function.
Chapter 10

The \texttt{l3sys} package: System/runtime functions

10.1 The name of the job

\texttt{\textbackslash c\_sys\_jobname\_str} Constant that gets the “job name” assigned when \TeX{} starts.

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

10.2 Date and time

\texttt{\textbackslash c\_sys\_minute\_int} \texttt{\textbackslash c\_sys\_hour\_int} \texttt{\textbackslash c\_sys\_day\_int} \texttt{\textbackslash c\_sys\_month\_int} \texttt{\textbackslash c\_sys\_year\_int} The date and time at which the current job was started: these are all reported as integers.

\textbf{\TeX{}hackers note}: Whilst the underlying \TeX{} primitives \texttt{\time{}}, \texttt{\day{}}, \texttt{\month{}}, and \texttt{\year{}} can be altered by the user, this interface to the time and date is intended to be the “real” values.

\texttt{\textbackslash c\_sys\_timestamp\_str} The timestamp for the current job: the format is as described for \texttt{\file\_timestamp\::\n}.
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 ε-pTEX and ε-upTEX as expl3 requires the ε-TEx extensions. Each conditional is true for \textit{exactly one} supported engine. In particular, \texttt{\sys_if_engine_ptex_p:} is true for ε-pTEX but false for ε-upTEX.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{\sys_if_engine_lua\_tex_p:} & \texttt{\sys_if_engine_pdf\_tex_p:} \\
\texttt{\sys_if_engine_lua\_tex:TF} & \texttt{\sys_if_engine_pdf\_tex:TF \{(true code\}\{(false code\)}} \\
\texttt{\sys_if_engine_pdf\_tex:TF} & \texttt{\sys_if_engine_ptex_p:} \\
\texttt{\sys_if_engine_ptex:TF} & \texttt{\sys_if_engine_uptex_p:} \\
\texttt{\sys_if_engine_uptex:TF} & \texttt{\sys_if_engine_xetex_p:} \\
\texttt{\sys_if_engine_xetex:TF} & \texttt{\sys_if_engine_xetex:TF} \\
\hline
\end{tabular}
\end{center}

New: 2015-09-07

\texttt{\c_sys\_engine\_str}

The current engine given as a lower case string: one of \texttt{lua\_tex}, \texttt{pdf\_tex}, \texttt{ptex}, \texttt{uptex} or \texttt{xetex}.

New: 2015-09-19

\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{lua\_tex}, \texttt{luah\_btex}, \texttt{pdf\_tex}, \texttt{eptex}, \texttt{euptex} or \texttt{xetex}.

New: 2020-08-20

\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{luatex} (or \texttt{dviluatex}), \texttt{pdflatex} (or \texttt{latex}), \texttt{platex}, \texttt{uplatex} or \texttt{xelatex} for \TeX, similar names for plain \TeX (except pdf\TeX{} in DVI mode yields \texttt{etex}), and \texttt{cont\_en} for Con\TeX{}t (i.e. the \texttt{\fmtname}).

New: 2020-08-20

\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 pdf\TeX{} and Lua\TeX{} this is of the form

\texttt{\langle major\rangle.\langle minor\rangle.\langle revision\rangle}

For Xe\TeX{} the form is

\texttt{\langle major\rangle.\langle minor\rangle}

For p\TeX{} and up\TeX{}, only releases since \TeX{} Live 2018 make the data available, and the form is more complex, as it comprises the p\TeX{} version, the up\TeX{} version and the e-p\TeX{} version.

\texttt{p\langle major\rangle.\langle minor\rangle.\langle revision\rangle-u\langle major\rangle.\langle minor\rangle-\langle eptex\rangle}

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

\texttt{\sys\_timer:}

New: 2021-05-12

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 (\texttt{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.

\texttt{\textbackslash c\_sys\_output\_str}

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 os.type() function, \textit{i.e.} all Unix-like systems are \texttt{unix} (including Linux and MacOS).

\texttt{\textbackslash c\_sys\_platform\_str}

The current platform given as a lower case string: one of \texttt{unix}, \texttt{windows} or \texttt{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.
Globally sets the seed for the engine’s pseudo-random number generator to the \textit{integer expression}. This random seed affects all \texttt{\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{T\(\text{eX}\)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 \([0, 2^{28} - 1]\).

\subsection*{10.7 Access to the shell}

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

\begin{itemize}
\item 0 \begin{flushright} Shell escape is disabled \end{flushright}
\item 1 \begin{flushright} Unrestricted shell escape is enabled \end{flushright}
\item 2 \begin{flushright} Restricted shell escape is enabled \end{flushright}
\end{itemize}

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.
\sys_if_shell_restricted_p: ★ \sys_if_shell_restricted_p:
\sys_if_shell_restricted:TF ★ \sys_if_shell_restricted:TF \{(true code)\} \{(false code)\}

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

\sys_shell_now:n \sys_shell_now:n \{(tokens)\}
\sys_shell_now:e Execute \textit{tokens} through shell escape immediately.

\sys_shell_shipout:n \sys_shell_shipout:n \{(tokens)\}
\sys_shell_shipout:e Execute \textit{tokens} through shell escape at shipout.

10.8 Loading configuration data

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

\sys_ensure_backend:
\sys_ensure_backend:
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
\begin{itemize}
  \item pdftex
  \item luatex
  \item xetex
  \item dvips
  \item dvipdfmx
  \item dvisvgm
\end{itemize}

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

\sys_finalise: Finalises all system-dependent functionality: required before loading a backend.

Rev: 2019-10-06
Chapter 11

The i3msg package

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The i3msg module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by i3msg to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message class has to be made, for example error, warning or info.

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

11.1 Creating new messages

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

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

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

will allow to filter out specifically messages from the submodule.

Some authors may find the need to include spaces as - characters tedious. This can be avoided by locally resetting the category code of _.
although in general this may be confusing; simply writing the messages using ~ characters is the method favored by the team.

\msg_new:nnn {⟨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:nnn {⟨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⟩}
\msg_if_exist:nnTF {⟨module⟩} {⟨message⟩} {⟨true code⟩} {⟨false code⟩}

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

## 11.2 Customizable information for message modules

\msg_module_name:n {⟨module⟩}
\g_msg_module_name_prop

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⟩}
\g_msg_module_type_prop

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
\g_msg_module_type_prop

Provides a mapping between the module name used for messages, and that for documentation.

Provides a mapping between the module name used for messages, and that type of module. For example, for \LaTeX3 core messages, an empty entry is set here meaning that they are not described using the standard Package text.
11.3 Contextual information for messages

\msg_line_context: ∗ \msg_line_context:
Prints the current line number when a message is given, and thus suitable for giving
context to messages. The number itself is proceeded by the text on line.

\msg_line_number: ∗ \msg_line_number:
Prints the current line number when a message is given.

\msg_fatal_text:n {⟨module⟩}
Produces the standard text
Fatal Package ⟨module⟩ Error
This function can be redefined to alter the language in which the message is given, using
#1 as the name of the ⟨module⟩ to be included.

\msg_critical_text:n {⟨module⟩}
Produces the standard text
Critical Package ⟨module⟩ Error
This function can be redefined to alter the language in which the message is given, using
#1 as the name of the ⟨module⟩ to be included.

\msg_error_text:n {⟨module⟩}
Produces the standard text
Package ⟨module⟩ Error
This function can be redefined to alter the language in which the message is given, using
#1 as the name of the ⟨module⟩ to be included.

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

\msg_info_text:n {⟨module⟩}
Produces the standard text:
Package ⟨module⟩ Info
This function can be redefined to alter the language in which the message is given, using
#1 as the name of the ⟨module⟩ to be included. The ⟨type⟩ of ⟨module⟩ may be adjusted:
Package is the standard outcome: see \msg_module_type:n.
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 T\TeX run;
- **critical**, ending the file being input;
- **error**, interrupting the T\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 \texttt{\langle module\rangle} error \texttt{\langle message\rangle}, passing \texttt{\langle arg\ one\rangle} to \texttt{\langle arg\ four\rangle} 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).

\TeX\texttt{\langle module\rangle} error \texttt{\langle message\rangle}, passing \texttt{\langle arg\ one\rangle} to \texttt{\langle arg\ four\rangle} to the text-creating functions. After issuing a critical error, \TeX stops reading the current input file. This may halt the \TeX run (if the current file is the main file) or may abort reading a sub-file.

\TeX\texttt{\langle module\rangle} error \texttt{\langle message\rangle}, passing \texttt{\langle arg\ one\rangle} to \texttt{\langle arg\ four\rangle} to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.
Issues \textit{(module)} warning \textit{(message)}, passing \texttt{arg one} to \texttt{arg four} to the text-creating functions. The warning text is added to the log file and the terminal, but the \TeX run is not interrupted.

Issues \textit{(module)} information \textit{(message)}, passing \texttt{arg one} to \texttt{arg four} to the text-creating functions. For the more common \texttt{msg_info:nnnnnn}, the information text is added to the log file only, while \texttt{msg_note:nnnnnn} adds the info text to both the log file and the terminal. The \TeX run is not interrupted.
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.

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 \{(module)\} \{(message)\} \{(arg one)\} \{(arg two)\} \{(arg three)\} \{(arg four)\}

Issues (module) information (message), passing (arg one) to (arg four) to the text-creating functions. The information text is shown on the terminal and the TeX run is interrupted in a manner similar to \tl_show:n. This is used in conjunction with \msg_show_item:n and similar functions to print complex variable contents completely. If the formatted text does not contain >~ at the start of a line, an additional line >~ will be put at the end. In addition, a final period is added if not present.

\msg_show_item:n * \seq_map_function:NN \{(seq)\} \msg_show_item:n
\msg_show_item_unbraced:n * \prop_map_function:NN \{(prop)\} \msg_show_item:nn
\msg_show_item:nn *
\msg_show_item_unbraced:nn *

Used in the text of messages for \msg_show:nnnnnn to show or log a list of items or key–value pairs. The output of \msg_show_item:n produces a newline, the prefix > , two spaces, then the braced string representation of its argument. The two-argument versions separates the key and value using \&\&=>\&\& , and the unbraced versions don’t print the surrounding braces.

These functions are suitable for usage with iterator functions like \seq_map_function:NN, \prop_map_function:NN, etc. For example, with a sequence \l_tmpa_seq containing a, {b} and \c ,

\seq_map_function:NN \l_tmpa_seq \msg_show_item:n

would expand to three lines:

>\&\&\{a\}
>\&\&\{b\}
>\&\&\{c\}

11.4.2 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using \msg_error:nnnnnn or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools
to print to the terminal or the log file are expandable. As a result, short-hands such as \\
 or \ do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

\msg_expandable_error:nnnnnn {module} {message} {arg one} {arg two} {arg three} {arg four}

Issues an “Undefined error” message from \TeX itself using the undefined control sequence \\error then prints “! \langle module\rangle: \langle error message\rangle”, 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.

\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:nn
\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:nn
\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 }
Chapter 12

The \texttt{l3file} package

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix \texttt{\textbackslash file...}, while others are used to work with files on a line by line basis and have prefix \texttt{\textbackslash ior...} (reading) or \texttt{\textbackslash 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 \LaTeX{}3. 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{\ior\_open:N} 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{\texttt{c\_term\_}}...

\ior\_open:N\{\texttt{file name}\}
\ior\_open:cn
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.

\ior\_shell\_open:N\{\texttt{shell command}\}
\ior\_shell\_open:cn
Opens the pseudo-file created by the output of the \texttt{shell command} for reading using \texttt{stream} as the control sequence for 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{shell command} until a \texttt{\ior\_close:N} instruction is given or the \TeX{} run ends. If piped system calls are disabled an error is raised. For details of handling of the \texttt{shell command}, see \texttt{\sys\_get\_shell:nnNTF}.

\ior\_shell\_open:NN
\ior\_shell\_open:NV\{\texttt{shell command}\}
\ior\_shell\_open:C
Opens the pseudo-file created by the output of the \texttt{shell command} for writing using \texttt{stream} as the control sequence for 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{shell command} until a \texttt{\ior\_close:N} instruction is given or the \TeX{} run ends. If piped system calls are disabled an error is raised. For details of handling of the \texttt{shell command}, see \texttt{\sys\_get\_shell:nnNTF}. 

\ior\_new:N\{\texttt{stream}\}
\ior\_new:c\{\texttt{stream}\}
\ior\_new:N\{\texttt{stream}\}
\ior\_new:c\{\texttt{stream}\}
\ior\_new:N\{\texttt{stream}\}
\ior\_new:c\{\texttt{stream}\}
\ior\_new:N\{\texttt{stream}\}
\ior\_new:c\{\texttt{stream}\}
\ior\_new:N\{\texttt{stream}\}
\ior\_new:c\{\texttt{stream}\}
\ior\_new:N\{\texttt{stream}\}
\ior\_new:c\{\texttt{stream}\}

\ior\_open:N\{\texttt{stream}\}\{\texttt{file name}\}
\ior\_open:cn
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. The \texttt{true code} is then inserted into the input stream. If the file is not found, no error is raised and the \texttt{false code} is inserted into the input stream.

\ior\_open:NNTF\{\texttt{file name}\}\{\texttt{true code}\}\{\texttt{false code}\}
\ior\_open:cn
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. The \texttt{true code} is then inserted into the input stream. If the file is not found, no error is raised and the \texttt{false code} is inserted into the input stream.

\ior\_open:N\{\texttt{stream}\}\{\texttt{file name}\}\{\texttt{true code}\}\{\texttt{false code}\}
\ior\_open:cn
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 piped system calls are disabled an error is raised. For details of handling of the \texttt{shell command}, see \texttt{\sys\_get\_shell:nnNTF}.

\ior\_shell\_open:N\{\texttt{shell command}\}
\ior\_shell\_open:cn
Opens the pseudo-file created by the output of the \texttt{shell command} for reading using \texttt{stream} as the control sequence for 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{shell command} until a \texttt{\ior\_close:N} instruction is given or the \TeX{} run ends. If piped system calls are disabled an error is raised. For details of handling of the \texttt{shell command}, see \texttt{\sys\_get\_shell:nnNTF}. 

\ior\_shell\_open:NN
\ior\_shell\_open:NV\{\texttt{shell command}\}
\ior\_shell\_open:C
Opens the pseudo-file created by the output of the \texttt{shell command} for writing using \texttt{stream} as the control sequence for 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{shell command} until a \texttt{\ior\_close:N} instruction is given or the \TeX{} run ends. If piped system calls are disabled an error is raised. For details of handling of the \texttt{shell command}, see \texttt{\sys\_get\_shell:nnNTF}. 

\ior\_shell\_open:NN
\ior\_shell\_open:NV\{\texttt{shell command}\}
\ior\_shell\_open:C
Opens the pseudo-file created by the output of the \texttt{shell command} for writing using \texttt{stream} as the control sequence for 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{shell command} until a \texttt{\ior\_close:N} instruction is given or the \TeX{} run ends. If piped system calls are disabled an error is raised. For details of handling of the \texttt{shell command}, see \texttt{\sys\_get\_shell:nnNTF}.
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.

12.1.1 Reading from files

Reading from files and reading from the terminal are separate processes in expl3. The functions \texttt{ior\_get:NN} and \texttt{ior\_str\_get:NN}, and their branching equivalents, are designed to work with \texttt{files}. 
Function that reads one or more lines (until an equal number of left and right braces are found) from the file input \langle \text{stream} \rangle and stores the result locally in the \langle \text{token list} \rangle variable. The material read from the \langle \text{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 \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 \par tokens. In the non-branching version, where the \langle \text{stream} \rangle is not open the \langle \text{tl var} \rangle is set to \texttt{\&\_no_value}.

TeXhackers note: This protected macro is a wrapper around the \TeX{} primitive \texttt{\readline}. 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.

Function that reads one line from the file input \langle \text{stream} \rangle and stores the result locally in the \langle \text{token list} \rangle variable. The material is read from the \langle \text{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 \text{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 \text{stream} \rangle is not open the \langle \text{tl var} \rangle is set to \texttt{\&\_no_value}.

TeXhackers note: This protected macro is a wrapper around the \texttt{\varepsilon-\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 (stream) \{ (inline function) \}

Applies the \textit{inline function} to each set of \textit{lines} obtained by calling \ior_get:NN until reaching the end of the file. \LaTeX{} ignores any trailing new-line marker from the file it reads. The \textit{inline function} should consist of code which receives the \textit{line} as \#1.

\ior_str_map_inline:Nn
\ior_str_map_inline:Nn (stream) \{ (inline function) \}

Applies the \textit{inline function} to every \textit{line} in the \textit{stream}. The material is read from the \textit{stream} as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The \textit{inline function} should consist of code which receives the \textit{line} as \#1. Note that \LaTeX{} removes trailing space and tab characters (character codes 32 and 9) from every line upon input. \LaTeX{} also ignores any trailing new-line marker from the file it reads.

\ior_map_variable:NNn
\ior_map_variable:NNn (stream) \{ (code) \}

For each set of \textit{lines} obtained by calling \ior_get:NN until reaching the end of the file, stores the \textit{lines} in the \textit{tl var} then 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 set of \textit{lines}, or its original value if the \textit{stream} is empty. \LaTeX{} ignores any trailing new-line marker from the file it reads. This function is typically faster than \ior_map_inline:Nn.

\ior_str_map_variable:NNn
\ior_str_map_variable:NNn (stream) \{ (variable) \} \{ (code) \}

For each \textit{line} in the \textit{stream}, stores the \textit{line} in the \textit{variable} then applies the \textit{code}. The material is read from the \textit{stream} as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The \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{line}, or its original value if the \textit{stream} is empty. Note that \LaTeX{} removes trailing space and tab characters (character codes 32 and 9) from every line upon input. \LaTeX{} also ignores any trailing new-line marker from the file it reads. This function is typically faster than \ior_str_map_inline:Nn.
\ior_map_break:

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

\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo } { \ior_map_break: }
  \%
  % Do something useful
}

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

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\ior_map_break:n

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

\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo } { \ior_map_break:n { ⟨code⟩ } }
  \%
  % Do something useful
}

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

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨code⟩ is inserted into the input stream. This depends on the design of the mapping function.

\ior_if_eof_p:N
\ior_if_eof: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.
12.1.2 Reading from the terminal

\ior_get_term:nN \ior_get_term:nN \ior_str_get_term:nN

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 \langle token list \rangle variable. Tokenization occurs as described for \ior_get:NN or \ior_str_get:NN, respectively. When the \langle prompt \rangle is empty, \TeX{} will wait for input without any other indication; typically the programmer will have provided a suitable text using e.g. \ior_term:n. Where the \langle prompt \rangle is given, it will appear in the terminal followed by an =, e.g.

\prompt=

12.1.3 Writing to files

\iow_now:Nn \iow_now:Nn \iow_now:(\langle stream \rangle \{\langle tokens \rangle\})

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

\iow_log:n \iow_log:e \iow_log:n \iow_log:e

This function writes the given \langle tokens \rangle to the log (transcript) file immediately: it is a dedicated version of \iow_now:Nn.

\iow_term:n \iow_term:e \iow_term:n \iow_term:e

This function writes the given \langle tokens \rangle to the terminal file immediately: it is a dedicated version of \iow_now:Nn.

\iow_shipout:Nn \iow_shipout:Nn \iow_shipout:(\langle stream \rangle \{\langle tokens \rangle\})

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

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

\TeX{}hackers note: When using expl3 with a format other than \LaTeX{}, new line characters inserted using \iow_newline: or using the line-wrapping code \iow_wrap:nnnN are not recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additional unwanted line-breaks.
\iow_shipout_e:Nn \iow_shipout_e:(Ne|cn|ce)

\iow_shipout_e:Nn \langle stream \rangle \{(tokens)\}

This function writes \langle tokens \rangle to the specified \langle stream \rangle when the current page is finalised (i.e. at shipout). The \langle 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{TeXhackers note:} This is a wrapper around the \TeX{} primitive \texttt{\write}. When using expl3 with a format other than \texttt{\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.

\iow_char:N \langle char \rangle

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

\iow_now:Ne \g_my_iow \{ \iow_char:N \langle text \rangle \iow_char:N \} \}

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

\iow_newline: \iow_newline:

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

\textbf{TeXhackers note:} When using expl3 with a format other than \texttt{\LaTeX{}}, the character inserted by \texttt{\iow_newline}: is not recognized by \texttt{\LaTeX{}}, 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

\iow_wrap:nnnN \iow_wrap:nnN \iow_wrap:nnN \iow_wrap:nnN \iow_wrap:nnnN \iow_wrap:nnN \iow_wrap:nnN \iow_wrap:nnN \iow_wrap:nnN

\iow_wrap:nnN \iow_wrap:nnN \iow_wrap:nenN

New: 2012-06-28
Updated: 2017-12-04

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:

- \textbackslash{} or \texttt{\iow_newline}: may be used to force a new line,
- \textbackslash{} 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 \texttt{\token_to_str:N}, \texttt{\tl_to_str:n}, \texttt{\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.

\textbf{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 \texttt{\exp_not:N} or \texttt{\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:

New: 2023-04-25

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{\texttt{\langle text \rangle}}\}

New: 2011-09-21

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 \texttt{\textbackslash{\textbackslash{\textbackslash}}} to force line breaks.

99
The maximum number of characters in a line to be written by the \texttt{\textbackslash iow\_wrap:nnnN} function. This value depends on the \TeX{} system in use: the standard value is 78, which is typically correct for unmodified \TeX{} Live and MiKTeX systems.

\section*{12.1.5 Constant input–output streams, and variables}

\texttt{\g\_tmpa\_ior} \newline \texttt{\g\_tmpb\_ior} \newline \texttt{Nov: 2017-12-11}

Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any \LaTeX{}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\texttt{\c\_log\_iow} \newline \texttt{\c\_term\_iow} \newline \texttt{Nov: 2017-12-11}

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

\texttt{\g\_tmpa\_iow} \newline \texttt{\g\_tmpb\_iow} \newline \texttt{Nov: 2017-12-11}

Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any \LaTeX{}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\section*{12.1.6 Primitive conditionals}

\texttt{\if\_eof:w} \newline \texttt{\if\_eof:w (stream)} \newline \texttt{⟨true code⟩} \newline \texttt{\else:} \newline \texttt{⟨false code⟩} \newline \texttt{\fi:}

Tests if the \texttt{⟨stream⟩} returns “end of file”, which is true for non-existent files. The \texttt{\else:} branch is optional.

\textbf{\TeX{}hackers note:} This is the \TeX{} primitive \texttt{\textbackslash ifeof}.

\section*{12.2 File operation functions}

\texttt{\g\_file\_curr\_dir\_str} \newline \texttt{\g\_file\_curr\_name\_str} \newline \texttt{\g\_file\_curr\_ext\_str} \newline \texttt{Nov: 2017-06-21}

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path \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{⟨name⟩} and \texttt{⟨ext⟩} parts together make up the file name, thus the \texttt{⟨name⟩} part may be thought of as the “job name” for the current file.

Note that \TeX{} does not provide information on the \texttt{⟨dir⟩} and \texttt{⟨ext⟩} part for the main (top level) file and that this file always has empty \texttt{⟨dir⟩} and \texttt{⟨ext⟩} components. Also, the \texttt{⟨name⟩} here will be equal to \texttt{\c\_sys\_jobname\_str}, which may be different from the real file name (if set using --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.

\textbf{\TeXhackers note:} When working as a package in L\LaTeX, \texttt{expl3} will automatically append the current \texttt{\input@path} to the set of values from \texttt{\l_file_search_path_seq}.

\begin{verbatim}
\file_if_exist_p:n \file_if_exist_p:V \file_if_exist:nTF \file_if_exist:V
\end{verbatim}

Searches for \langle file name \rangle using the current \TeX search path and the additional paths controlled by \texttt{\l_file_search_path_seq}.

\begin{verbatim}
\file_get:nnN \file_get:VnN \file_get:nnNTF \file_get:VnN
\end{verbatim}

Defines \langle tl \rangle to the contents of \langle filename \rangle. Category codes may need to be set appropriately via the \langle setup \rangle argument. The non-branching version sets the \langle tl \rangle to \texttt{\q_no_value} if the file is not found. The branching version runs the \langle true code \rangle after the assignment to \langle tl \rangle if the file is found, and \langle false code \rangle otherwise.

\begin{verbatim}
\file_get_full_name:nN \file_get_full_name:VN \file_get_full_name:nNTF \file_get_full_name:VN
\end{verbatim}

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

\begin{verbatim}
\file_full_name:n \file_full_name:V
\end{verbatim}

Searches for \langle file name \rangle 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 \langle file name \rangle has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.
\file_parse_full_name:nNNN \file_parse_full_name:VNNN

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

- The \texttt{\langle dir\rangle}: everything up to the last / (path separator) in the \texttt{\langle file path\rangle}. As with system PATH variables and related functions, the \texttt{\langle dir\rangle} 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{\langle name\rangle}: 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{\langle ext\rangle}: everything after the last . (including the dot). The \texttt{(ext)} is empty if there is no . after the last /.

Before parsing, the \texttt{(full name)} is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (”) are invalid in file names and are discarded from the input.

\file_hex_dump:n \file_hex_dump:V
\file_hex_dump:nnn \file_hex_dump:Vnn

Searches for \texttt{\langle file name\rangle} using the current \TeX{} search path and the additional paths controlled by \texttt{\l_file_search_path_seq}. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most \TeX{} behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The \texttt{(start index)} and \texttt{(end index)} values work as described for \texttt{\str_range:nnn}.
\file_get_hex_dump:n {⟨file name⟩} ⟨tl var⟩
\file_get_hex_dump:n {⟨file name⟩} ⟨{start index}⟩ ⟨{end index}⟩ ⟨tl var⟩
Sets the ⟨tl var⟩ to the result of applying \file_hex_dump:n/\file_hex_dump:nnn to the ⟨file⟩. If the file is not found, the ⟨tl var⟩ will be set to \q_no_value.

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

\file_get_size:n {⟨file name⟩} ⟨tl var⟩
Sets the ⟨tl var⟩ to the result of applying \file_mdfive_hash:n to the ⟨file⟩. If the file is not found, the ⟨tl var⟩ will be set to \q_no_value. This is not available in older versions of \TeX{}.

\file_timestamp:n {⟨file name⟩}
Searches for ⟨file name⟩ using the current \TeX{} search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form D:⟨year⟩⟨month⟩⟨day⟩⟨hour⟩⟨minute⟩⟨second⟩⟨offset⟩, where the latter may be Z (UTC) or ⟨plus-minus⟩⟨hours⟩'⟨minutes⟩'. When the file is not found, the result of expansion is empty. This is not available in older versions of \TeX{.}
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{file_get_timestamp:nN} \file{file_get_timestamp:VN} \file{file_get_timestamp:nN} \file{file_get_timestamp:VN}

Nov: 2017-07-09
Updated: 2019-02-16

Compares the file stamps on the two `{files}` as indicated by the `{comparator}`, and inserts either the `{true code}` or `{false case}` as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

\file{file_compare_timestamp:nNn} {source-file} > {derived-file} {
    % Code to regenerate derived file
}

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

\file{file_input:n} \file{file_input:V}

Updated: 2017-06-26

Searches for `{file name}` in the path as detailed for `{file_if_exist:nTF}`, and if found reads in the file as additional \LaTeX source as described for `{file_input:n}`, otherwise inserts the `{false case}`. Note that these functions do not raise an error if the file is not found, in contrast to `{file_input:n}`.

TEXhackers note: This function is intended only for contexts where files must be read purely by expansion, for example at the start of a table cell in an \halign.
\file_input_stop:  Ends the reading of a file started by \file_input:n or similar before the end of the file is reached. Where the file reading is being terminated due to an error, \msg_-critical:nn(nn) should be preferred.

\TeXhackers note: This function must be used on a line on its own: \TeX reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

\file_show_list: \file_log_list: These functions list all files loaded by \LaTeX\texttt{2e} commands that populate @filelist or by \file_input:n. While \file_show_list: displays the list in the terminal, \file_log_list: outputs it to the log file only.
Chapter 13

The \texttt{luatex} package: \LaTeX-TEX-specific functions

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

13.1 Breaking out to Lua

\begin{verbatim}
\lua_now:n \{⟨token list⟩}\end{verbatim}

The $⟨\text{token \ list}⟩$ 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 $⟨\text{Lua \ input}⟩$ 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 executes the $ ⟨\text{Lua \ input}⟩ $ immediately, and in an expandable manner.

\textbf{\LaTeX hackers 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{verbatim}
\lua_now:e \lua_now:n \{⟨token list⟩}\end{verbatim}

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

\textbf{\LaTeX hackers note:} At a \TeX level, the $⟨\text{Lua \ input}⟩$ is stored as a “whatsit”.

106
\lua_escape:n \lua_escape:e \n
\lua_escape:n \{(token list)\} \n\lua_escape:e \n
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.

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

\lua_load_module:n \lua_load_module:e \n
\lua_load_module:n \{(Lua module name)\}
\lua_load_module:e

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.

\textbf{\TeX} hackers note: This is a wrapper around the Lua call \texttt{require} \langle\texttt{module}\rangle.

13.2 Lua interfaces

As well as interfaces for \TeXLaTeX{}, there are a small number of Lua functions provided here.

\ltx_utils

Most public interfaces provided by the module are stored within the \ltx_utils table.

\ltx_utils.filedump

\ltx_utils.filedump \langle\texttt{dump}\rangle = \ltx_utils.filedump\langle\texttt{file}\rangle,\langle\texttt{offset}\rangle,\langle\texttt{length}\rangle

Returns the uppercase hexadecimal representation of the content of the \langle\texttt{file}\rangle read as bytes. If the \langle\texttt{length}\rangle is given, only this part of the file is returned; similarly, one may specify the \langle\texttt{offset}\rangle from the start of the file. If the \langle\texttt{length}\rangle is not given, the entire file is read starting at the \langle\texttt{offset}\rangle.

\ltx_utils.filemd5sum

\ltx_utils.filemd5sum \langle\texttt{hash}\rangle = \ltx_utils.filemd5sum\langle\texttt{file}\rangle

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 \TeXLaTeX{} behaviour. If the \langle\texttt{file}\rangle is not found, nothing is returned with no error raised.

\ltx_utils.filemoddate

\ltx_utils.filemoddate \langle\texttt{date}\rangle = \ltx_utils.filemoddate\langle\texttt{file}\rangle

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

\texttt{D:\langle\texttt{year}\rangle\langle\texttt{month}\rangle\langle\texttt{day}\rangle\langle\texttt{hour}\rangle\langle\texttt{minute}\rangle\langle\texttt{second}\rangle\langle\texttt{offset}\rangle}

where the latter may be \texttt{Z} (UTC) or \langle\texttt{plus-minus}\rangle\langle\texttt{hours}\rangle'\langle\texttt{minutes}\rangle'. If the \langle\texttt{file}\rangle is not found, nothing is returned with no error raised.
\texttt{ltx.utils.filesize} \texttt{size = 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}. 
There are a small number of \TeX{} or \LaTeX{} concepts which are not used in expl3 code but which need to be manipulated when working as a \LaTeX{}2ε package. To allow these to be integrated cleanly into expl3 code, a set of legacy interfaces are provided here.

\begin{verbatim}
\legacy_if_p:n {⟨name⟩}
\legacy_if:nTF {⟨name⟩} {⟨true code⟩} {⟨false code⟩}
Tests if the \LaTeX{}2ε/plain \TeX{} conditional (generated by \texttt{\newif}) is true or false and branches accordingly. The \texttt{⟨name⟩} of the conditional should omit the leading if.
\end{verbatim}

\begin{verbatim}
\legacy_if_set_true:n {⟨name⟩}
\legacy_if_set_false:n {⟨name⟩}
\legacy_if_gset_true:n
\legacy_if_gset_false:n
Sets the \LaTeX{}2ε/plain \TeX{} conditional \texttt{\if⟨name⟩} (generated by \texttt{\newif}) to be true or false.
\end{verbatim}

\begin{verbatim}
\legacy_if_set:nn {⟨name⟩} {⟨boolexpr⟩}
\legacy_if_set_g:nn
\legacy_if_gset:nn
Sets the \LaTeX{}2ε/plain \TeX{} conditional \texttt{\if⟨name⟩} (generated by \texttt{\newif}) to the result of evaluating the \texttt{⟨boolexpr⟩}.
\end{verbatim}
Part IV
Data types
Chapter 15

The \texttt{l3tl} package

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}
\texttt{\textbackslash 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}
\texttt{\textbackslash foo:N \textbackslash 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{\textbackslash}, \texttt{\{}, or \texttt{\}} (assuming normal \LaTeX{} category codes). Thus for example

\begin{verbatim}
\{ Hello \} - world
\end{verbatim}

contains six items (Hello, w, o, r, l and d), but thirteen tokens (\{, H, e, l, l, o, \}, \texttt{\textbackslash}, 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}
\texttt{\textbackslash tl\_new:N} \texttt{\textbackslash tl\_new:N (tl\_var)}
\end{verbatim}

\begin{verbatim}
\texttt{\textbackslash tl\_new:c}
\end{verbatim}

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

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

\tl_clear:N \tl_clear:c \tl_gclear:N
\tl_clear:N \tl_clear:c
\tl_gclear:N
\tl_clear:new:N \tl_clear:new:c \tl_gclear:new:N \tl_gclear:new:c

Clears all entries from the \langle tl\ var \rangle.

\tl_set_eq:NN \tl_set_eq:cc \tl_gset_eq:NN \tl_gset_eq:cc
\tl_concat:NNN \tl_concat:ccc \tl_gconcat:NNN \tl_gconcat:ccc

Sets \langle tl\ var \rangle equal to that of \langle tl\ var \rangle.

\tl_if_exist_p:N \tl_if_exist:NTF \tl_if_exist:cc \tl_if_exist:p:cc
\tl_if_exist:p:N \tl_if_exist:p:cc \tl_if_exist:NTF \tl_if_exist:cc

Tests whether the \langle tl\ var \rangle is currently defined. This does not check that the \langle tl\ var \rangle really is a token list variable.

\tl_concat:NNN \tlconcat:N \tlconcat:c \tl_gconcat:N \tlconcat:c
\tl_concat:NNN \tlconcat:cc \tl_gconcat:NNN \tlconcat:cc

Concatenates the content of \langle tl\ var \rangle and \langle tl\ var \rangle together and saves the result in \langle tl\ var \rangle. The \langle tl\ var \rangle is placed at the left side of the new token list.

\tl_set:Nn \tl_set:N \tl_gset:Nn \tl_gset:N
\tl_set:{\langle Nv|Nv|Ne|Ne|No|Ne|Nf|cn|cV|cv|co|ce|cf \rangle}
\tl_gset:{\langle Nv|Nv|Ne|Ne|No|Ne|Nf|cn|cV|cv|co|ce|cf \rangle}

Sets \langle tl\ var \rangle to contain \langle tokens \rangle, removing any previous content from the variable.

\tl_put_left:Nn \tl_put_left:N \tl_gput_left:Nn \tl_gput_left:N
\tl_put_left:{\langle Nv|Nv|Ne|Ne|No|Ne|Nf|cn|cV|cv|co|ce|cf \rangle}
\tl_gput_left:{\langle Nv|Nv|Ne|Ne|No|Ne|Nf|cn|cV|cv|co|ce|cf \rangle}

Appends \langle tokens \rangle to the left side of the current content of \langle tl\ var \rangle.

15.2 Adding data to token list variables

\tl_set:Nn \tl_set:N \tl_gset:Nn \tl_gset:N
\tl_set:{\langle Nv|Nv|Ne|Ne|No|Ne|Nf|cn|cV|cv|co|ce|cf \rangle}
\tl_gset:{\langle Nv|Nv|Ne|Ne|No|Ne|Nf|cn|cV|cv|co|ce|cf \rangle}

Sets \langle tl\ var \rangle to contain \langle tokens \rangle, removing any previous content from the variable.

\tl_put_left:Nn \tl_put_left:N \tl_gput_left:Nn \tl_gput_left:N
\tl_put_left:{\langle Nv|Nv|Ne|Ne|No|Ne|Nf|cn|cV|cv|co|ce|cf \rangle}
\tl_gput_left:{\langle Nv|Nv|Ne|Ne|No|Ne|Nf|cn|cV|cv|co|ce|cf \rangle}

Appends \langle tokens \rangle to the left side of the current content of \langle tl\ var \rangle.
\textbackslash tl\_put\_right:Nn \textbackslash tl\_put\_right:\{ NV|Nv|Ne|No|cn|cV|ce|co \}
\textbackslash tl\_gput\_right:Nn \textbackslash tl\_gput\_right:\{ NV|Nv|Ne|No|cn|cV|ce|co \}

Appends \textit{tokens} to the right side of the current content of \texttt{(tl\_var)}.

### 15.3 Token list conditionals

\texttt{\textbackslash tl\_if\_blank\_p:n \{token list\}}
\texttt{\textbackslash tl\_if\_blank:nTF \{token list\} \{true code\} \{false code\}}

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

\texttt{\textbackslash tl\_if\_empty\_p:N \textbackslash tl\_if\_empty:NTF \{token list\} \{true code\} \{false code\}}

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

\texttt{\textbackslash tl\_if\_eq\_p:NN \textbackslash tl\_if\_eq:NNTF \textbackslash tl\_if\_eq:NN \textbackslash tl\_if\_eq:cn}

Compares the content of two \textit{token list variables} 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

\texttt{\textbackslash tl\_set:Nn \_l\_tmpa\_tl \{ abc \}}
\texttt{\textbackslash tl\_set:Ne \_l\_tmpb\_tl \{ \texttt{\_l\_to\_str:n \{ abc \}} \}}
\texttt{\textbackslash tl\_if\_eq:NNTF \_l\_tmpa\_tl \_l\_tmpb\_tl \{ \texttt{\_true} \} \{ \texttt{\_false} \}}

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

\texttt{\textbackslash tl\_if\_eq:NnTF \textbackslash tl\_if\_eq:NnTF \{token list\2\{true code\2\{false code\}}

Tests if the \textit{token list variable} and the \textit{token list2} 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.
Tests if \(\text{token list}_1\) and \(\text{token list}_2\) contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see \texttt{\textbackslash tl\_if\_eq:nnTF} for an expandable version when token lists are stored in variables, or \texttt{\textbackslash str\_if\_eq:nnTF} if category codes are not important.

Tests if the \(\text{token list}\) is found in the content of the \(\text{tl var}\). The \(\text{token list}\) cannot contain the tokens \{,\} or \# (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

Tests if the \(\text{token list}\) consists of exactly one \(\langle\text{item}\rangle\), 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

\begin{verbatim}
\tl_if_head_eq_catcode_p:nN \tl_if_head_eq_catcode:nNTF \tl_if_head_eq_catcode_p:oN \tl_if_head_eq_catcode:nN
\tl_if_head_eq_catcode:nN \tl_if_head_eq_catcode:nNTF \tl_if_head_eq_catcode:p:oN
\tl_if_head_eq_catcode:p:nN \tl_if_head_eq_catcode:p:nNTF \tl_if_head_eq_catcode:p:oN
\tl_if_head_eq_catcode:p:oN
\end{verbatim}

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

\begin{verbatim}
\tl_if_head_eq_charcode_p:nN \tl_if_head_eq_charcode:nNTF \tl_if_head_eq_charcode_p:fN \tl_if_head_eq_charcode:p:fN
\tl_if_head_eq_charcode:p:nN \tl_if_head_eq_charcode:p:nNTF \tl_if_head_eq_charcode:p:oN
\tl_if_head_eq_charcode:p:oN
\end{verbatim}

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

\begin{verbatim}
\tl_if_head_eq_meaning_p:nN \tl_if_head_eq_meaning:nNTF \tl_if_head_eq_meaning_p:nN \tl_if_head_eq_meaning:nN
\tl_if_head_eq_meaning:nN \tl_if_head_eq_meaning:nNTF \tl_if_head_eq_meaning:p:nN
\tl_if_head_eq_meaning:p:nN
\end{verbatim}

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

\begin{verbatim}
\tl_if_head_is_group_p:n \tl_if_head_is_group:nTF \tl_if_head_is_group_p:n \tl_if_head_is_group:nTF
\tl_if_head_is_group:n \tl_if_head_is_group:nTF \tl_if_head_is_group:p:n \tl_if_head_is_group:p:nTF
\tl_if_head_is_N_type_p:n \tl_if_head_is_N_type:nTF \tl_if_head_is_N_type_p:n \tl_if_head_is_N_type:nTF
\tl_if_head_is_N_type:n \tl_if_head_is_N_type:nTF \tl_if_head_is_N_type:p:n \tl_if_head_is_N_type:p:nTF
\end{verbatim}

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

Tests if the first \textit{token} in the \textit{token list} is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields \textit{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.
\texttt{tl\_if\_head\_is\_space\_p:n} \star \texttt{tl\_if\_head\_is\_space\_p:n \{\langle token\ list\ \rangle\}}
\texttt{tl\_if\_head\_is\_space:ntF} \star \texttt{tl\_if\_head\_is\_space:ntF \{\langle token\ list\ \rangle\\} \{\langle true\ code\ \rangle\} \{\langle false\ code\ \rangle\}}

Tests if the first \langle token \rangle in the \langle token\ list\rangle is an explicit space character (explicit token with character code 32 and category code 10). In particular, the test is \texttt{false} if the \langle token\ list\rangle starts with an implicit token such as \texttt{c\_space\_token}, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

15.4 Working with token lists as a whole

15.4.1 Using token lists

\texttt{tl\_to\_str:n} \star \texttt{tl\_to\_str:n \{\langle token\ list\ \rangle\}}
\texttt{tl\_to\_str:cn\|n\|v\|e} \star \texttt{tl\_to\_str:cn\|n\|v\|e\} \star

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

\texttt{\textsc{\textup{\textbf{T\textsc{e}X}hacker\textsc{s}\ note:}} This is the e-\textsc{\textup{T\textsc{e}X}} primitive \texttt{\textbackslash detokenize}. Converting a \langle token\ list\rangle to a \langle string \rangle yields a concatenation of the string representations of every token in the \langle token\ list\rangle. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter \texttt{\textbackslash escapechar}, absent if the \texttt{\textbackslash escapechar} is negative or greater than the largest character code;
- the control sequence name, as defined by \texttt{\textbackslash 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{\textbackslash 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\_to\_str:N} \star \texttt{tl\_to\_str:N \{tl\ var\}}
\texttt{tl\_to\_str:cn \star}

Converts the content of the \langle tl\ var\rangle into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This \langle string \rangle is then left in the input stream. For low-level details, see the notes given for \texttt{tl\_to\_str:n}.

\texttt{tl\_use:N} \star \texttt{tl\_use:N \{tl\ var\}}
\texttt{tl\_use:cn} \star

Recovers the content of a \langle tl\ 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 tl\ var\rangle directly without an accessor function.

116
15.4.2 Counting and reversing token lists

\texttt{\textbackslash tl\_count:n} \texttt{\{\texttt{tokens}\}}

Counts the number of \texttt{\{items\}} in \texttt{\{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 \texttt{\{tokens\}}. See also \texttt{\textbackslash tl\_count:N}. This function requires three expansions, giving an \textit{integer denotation}.

\texttt{\textbackslash tl\_count:N} \texttt{\{\texttt{tl\ var}\}}

Counts the number of \texttt{\{items\}} in the \texttt{\{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 \texttt{\{tl\ var\}}. See also \texttt{\textbackslash tl\_count:n}. This function requires three expansions, giving an \textit{integer denotation}.

\texttt{\textbackslash tl\_count\_tokens:n} \texttt{\{\texttt{tokens}\}}

Counts the number of \texttt{\TeX} tokens in the \texttt{\{tokens\}} and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of \texttt{(abc)} is 6.

\texttt{\textbackslash tl\_reverse:n} \texttt{\{\texttt{token\ list\}}}

Reverses the order of the \texttt{\{items\}} in the \texttt{\{token\ list\}}, so that \texttt{\{item\}_1 \{item\}_2 \{item\}_3} \ldots \texttt{\{item\}_n} becomes \texttt{\{item\}_n} \ldots \texttt{\{item\}_2 \{item\}_1}. This process preserves unprotected space within the \texttt{\{token\ list\}}. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider \texttt{\textbackslash tl\_reverse\_items:n}. See also \texttt{\textbackslash tl\_reverse:N}. \texttt{\TeX} hackers note: The result is returned within \texttt{\unexpanded}, which means that the token list does not expand further when appearing in an \texttt{\textbf{e}}-type or \texttt{\textit{x}}-type argument expansion.

\texttt{\textbackslash tl\_reverse:N} \texttt{\{\texttt{\textbf{tl\ var}\}}}

Sets the \texttt{\{\textbf{tl\ var}\}} to contain the result of reversing the order of its \texttt{\{items\}}, so that \texttt{\{item\}_1 \{item\}_2 \{item\}_3} \ldots \texttt{\{item\}_n} becomes \texttt{\{item\}_n} \ldots \texttt{\{item\}_2 \{item\}_1}. This process preserves unprotected spaces within the \texttt{\{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 \texttt{\textbackslash tl\_reverse:\texttt{\textbf{V}}}. See also \texttt{\textbackslash tl\_reverse\_items:n} for improved performance.

\texttt{\textbackslash tl\_reverse\_items:n} \texttt{\{\texttt{token\ list\}}}

Reverses the order of the \texttt{\{items\}} in the \texttt{\{token\ list\}}, so that \texttt{\{item\}_1 \{item\}_2 \{item\}_3} \ldots \texttt{\{item\}_n} becomes \texttt{\{item\}_n} \ldots \texttt{\{item\}_2 \{item\}_1}. This process removes any unprotected space within the \texttt{\{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 \texttt{\textbackslash tl\_reverse:n}. \texttt{\TeX} hackers note: The result is returned within \texttt{\unexpanded}, which means that the token list does not expand further when appearing in an \texttt{\textbf{e}}-type or\texttt{\textit{x}}-type argument expansion.

117
\tl_trim_spaces:n * \tl_trim_spaces:n \{\token\ list\}\*

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the \{\token\ list\} and leaves the result in the input stream.

\TeX\ hackers note: The result is returned within \unexpanded, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

\tl_trim_spaces_apply:nN * \tl_trim_spaces_apply:nN \{\token\ list\} \{\function\}

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the \{\token\ list\} and passes the result to the \{\function\} as an n-type argument.

\tl_trim_spaces:N \tl_trim_spaces:c \tl_gtrim_spaces:N \tl_gtrim_spaces:c

New: 2011-07-09

Sets the \{\tl\ var\} to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.

15.4.3 Viewing token lists

\tl_show:N \tl_show:c

Updated: 2021-04-29

Displays the content of the \{\tl\ var\} on the terminal.

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

\tl_show:n \tl_show:e

Updated: 2015-08-07

Displays the \{\token\ list\} on the terminal.

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

\tl_log:N \tl_log:c

Rewrites the content of the \{\tl\ var\} in the log file. See also \tl_show:N which displays the result in the terminal.

\tl_log:n \tl_log:e[x]

Updated: 2015-08-07

Rewrites the \{\token\ list\} in the log file. See also \tl_show:n which displays the result in the terminal.
15.5 Manipulating items in token lists

15.5.1 Mapping over token lists

All mappings are done at the current group level, i.e. any local assignments made by the \textlangle function\textrangle or \textlangle code\textrangle discussed below remain in effect after the loop.

\tl_map_function:NN \textlangle tl var\textrangle \textlangle function\textrangle
Applies \textlangle function\textrangle to every \textlangle item\textrangle in the \textlangle tl var\textrangle. The \textlangle function\textrangle receives one argument for each iteration. This may be a number of tokens if the \textlangle item\textrangle was stored within braces. Hence the \textlangle function\textrangle should anticipate receiving n-type arguments. See also \tl_map_function:nN.

\tl_map_function:nN \textlangle token list\textrangle \textlangle function\textrangle
Applies \textlangle function\textrangle to every \textlangle item\textrangle in the \textlangle token list\textrangle. The \textlangle function\textrangle receives one argument for each iteration. This may be a number of tokens if the \textlangle item\textrangle was stored within braces. Hence the \textlangle function\textrangle should anticipate receiving n-type arguments. See also \tl_map_function:nN.

\tl_map_inline:Nn \textlangle tl var\textrangle \{\textlangle inline function\textrangle\}
Applies the \textlangle inline function\textrangle to every \textlangle item\textrangle stored within the \textlangle tl var\textrangle. The \textlangle inline function\textrangle should consist of code which receives the \textlangle item\textrangle as \textlangle #1\textrangle. See also \tl_map_function:NN.

\tl_map_inline:nn \textlangle token list\textrangle \{\textlangle inline function\textrangle\}
Applies the \textlangle inline function\textrangle to every \textlangle item\textrangle stored within the \textlangle token list\textrangle. The \textlangle inline function\textrangle should consist of code which receives the \textlangle item\textrangle as \textlangle #1\textrangle. See also \tl_map_function:nN.

\tl_map_tokens:Nn \textlangle tl var\textrangle \{\textlangle code\textrangle\}
Analogue of \tl_map_function:NN which maps several tokens instead of a single function. The \textlangle code\textrangle receives each \textlangle item\textrangle in the \textlangle tl var\textrangle or in \langle tokens\rangle as a trailing brace group. For instance,

\tl_map_tokens:Nn \tl_my_tl \{ \prg_replicate:nn \{ 2 \} \}

expands to twice each \textlangle item\textrangle in the \textlangle tl var\textrangle; for each \textlangle item\textrangle in \tl_my_tl the function \prg_replicate:nn receives 2 and \textlangle item\textrangle as its two arguments. The function \tl_map_inline:Nn is typically faster but is not expandable.

\tl_map_variable:NNn \textlangle tl var\textrangle \langle variable\rangle \{\textlangle code\textrangle\}
Stores each \textlangle item\textrangle of the \textlangle tl var\textrangle in turn in the \langle token list\rangle \langle variable\rangle and applies the \textlangle code\textrangle. The \textlangle code\textrangle will usually make use of the \langle variable\rangle, but this is not enforced. The assignments to the \langle variable\rangle are local. Its value after the loop is the last \textlangle item\textrangle in the \textlangle tl var\textrangle, or its original value if the \textlangle tl var\textrangle is blank. See also \tl_map_inline:Nn.
\tl_map_variable:nNn \tl_map_variable:nNn \{(token list)\} \{variable\} \{(code)\}

Stores each \textit{(item)} of the \textit{(token list)} in turn in the \textit{(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{(tl var)}, or its original value if the \textit{(tl var)} is blank. See also \texttt{\tl_map_inline:nn}.

\tl_map_break: ✡ \tl_map_break:

\begin{verbatim}
\tl_map_break:n \{(code)\}
\end{verbatim}

Used to terminate a \texttt{\tl_map...} function before all entries in the \textit{(token list variable)} have been processed. insert the \textit{(code)} after the mapping has ended. This normally takes place within a conditional statement, for example

\begin{verbatim}
\str_if_eq:nnT { #1 } { bingo } { \tl_map_break:n { \ldots } }
\end{verbatim}

See also \texttt{\tl_map_break:n}. Use outside of a \texttt{\tl_map...} scenario leads to low level \TeX{} errors.

\textbf{\TeX{}hackers note:} When the mapping is broken, additional tokens may be inserted before the \textit{(tokens)} are inserted into the input stream. This depends on the design of the mapping function.

15.5.2 Head and tail of token lists

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.
\tl_head:n \{ \langle \text{token list} \rangle \}
\tl_head:n \{ \langle \text{item} \rangle \} in the \langle \text{token list} \rangle, discarding the rest of the \langle \text{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 a in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so
\tl_head:n \{ ~ \ { ~ \ ab } \ c \}
yields \ab. A blank \langle \text{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.

\tl_tail:n \{ \langle \text{token list} \rangle \}
\tl_tail:n \{ \langle \text{item} \rangle \} in the \langle \text{token list} \rangle, discarding the rest of the \langle \text{token list} \rangle. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank \langle \text{token list} \rangle (which consists only of space characters) results in a low-level \TeX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, \tl_if_blank:nF may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an \texttt{e}-type expansion. In general, \tl_head:n should be preferred if the number of expansions is not critical.

\tl_tail:n \{ \langle \text{token list} \rangle \}
\tl_tail:n \{ \langle \text{item} \rangle \} in the \langle \text{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 \( \{bc\}d \) in the input stream. A blank \langle \text{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{\LaTeX}hackers note: The result is returned within the \texttt{\exp_not:n} primitive, which means that the ⟨item⟩ does not expand further when appearing in an \texttt{e}-type or \texttt{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
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 \LaTeX.\textdagger

\textbf{\LaTeX}hackers note: The result is returned within the \texttt{\exp_not:n} primitive, which means that the ⟨item⟩ does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion.

122
Leaves in the input stream the items from the \texttt{(start index)} to the \texttt{(end index)} inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here \texttt{(start index)} and \texttt{(end index)} should be \texttt{(integer expressions)}. For describing in detail the functions' behavior, let \texttt{m} and \texttt{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 \texttt{l} be the count of the token list.

The \textit{actual start point} is determined as \texttt{M} = \texttt{m} if \texttt{m} > 0 and as \texttt{M} = \texttt{l} + \texttt{m} + 1 if \texttt{m} < 0. Similarly the \textit{actual end point} is \texttt{N} = \texttt{n} if \texttt{n} > 0 and \texttt{N} = \texttt{l} + \texttt{n} + 1 if \texttt{n} < 0. If \texttt{M} > \texttt{N}, the result is empty. Otherwise it consists of all items from position \texttt{M} to position \texttt{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 \texttt{s} for \texttt{s} ≤ 0 or \texttt{s} > \texttt{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 \texttt{l} = 7 as in the examples below, all of the following are equivalent and result in the whole token list

\begin{verbatim}
\tl_range:nnn \{ abcd-{e{}}fg \} \{ 1 \} \{ 7 \}
\tl_range:nnn \{ abcd-{e{}}fg \} \{ 1 \} \{ 12 \}
\tl_range:nnn \{ abcd-{e{}}fg \} \{ -7 \} \{ 7 \}
\tl_range:nnn \{ abcd-{e{}}fg \} \{ -12 \} \{ 7 \}
\end{verbatim}

Here are some more interesting examples. The calls

\begin{verbatim}
\iow_term:e \{ \tl_range:nnn \{ abcd{e{}}fg \} \{ 2 \} \{ 5 \} \}
\iow_term:e \{ \tl_range:nnn \{ abcd{e{}}fg \} \{ 2 \} \{ -3 \} \}
\iow_term:e \{ \tl_range:nnn \{ abcd{e{}}fg \} \{ -6 \} \{ 5 \} \}
\iow_term:e \{ \tl_range:nnn \{ abcd{e{}}fg \} \{ -6 \} \{ -3 \} \}
\end{verbatim}

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

\begin{verbatim}
\iow_term:e \{ \tl_range:nnn \{ abcd-{e{}}fg \} \{ 2 \} \{ 5 \} \}
\iow_term:e \{ \tl_range:nnn \{ abcd-{e{}}fg \} \{ 2 \} \{ -3 \} \}
\iow_term:e \{ \tl_range:nnn \{ abcd-{e{}}fg \} \{ -6 \} \{ 5 \} \}
\iow_term:e \{ \tl_range:nnn \{ abcd-{e{}}fg \} \{ -6 \} \{ -3 \} \}
\end{verbatim}

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

\begin{verbatim}
\tl_range:nnn \{ abcd-{e{}}f \} \{ 2 \} \{ 4 \}
\end{verbatim}

will discard the space after ‘d’.

If we want to get the items from, say, the third to the last in a token list <tl>, the call is \texttt{\tl_range:nnn \{ <tl> \} \{ 3 \} \{ -1 \}}. Similarly, for discarding the last item, we can do \texttt{\tl_range:nnn \{ <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 \(\text{tl var}\) according to the \(\text{comparison code}\), and assigns the result to \(\text{tl var}\). 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 \(\langle\text{token list}\rangle\), using the \(\langle\text{conditional}\rangle\) to compare items, and leaves the result in the input stream. The \(\langle\text{conditional}\rangle\) should have signature :nnTF, and return \text{true} if the two items being compared should be left in the same order, and \text{false} if the items should be swapped. The details of sorting comparison are described in Section 6.1.

\text{Text Hackers note:} The result is returned within \text{\exp_not:n}, which means that the token list does not expand further when appearing in an \text{e-type} or \text{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:cn
\tl_replace_once:NVn | \tl_replace_once:NnV | \tl_replace_once:Nen | \tl_replace_once:Nne | \tl_replace_once:Nee | \tl_greplace_once:cnVn | \tl_greplace_once:cnV | \tl_greplace_once:cen
\tl_greplace_once:cee

Replaces the first (leftmost) occurrence of \(\langle\text{old tokens}\rangle\) in the \(\langle\text{tl var}\rangle\) with \(\langle\text{new tokens}\rangle\). \(\langle\text{Old tokens}\rangle\) 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 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). As this function operates from left to right, the pattern \langle old tokens \rangle may remain after the replacement (see \texttt{\tl_remove_all:Nn} for an example).

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

Removes all occurrences of \langle tokens \rangle from the \langle tl var \rangle. \langle 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). As this function operates from left to right, the pattern \langle tokens \rangle may remain after the removal, for instance, \texttt{\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl \{bc\}} results in \texttt{\l_tmpa_tl} containing abcd.

### 15.6.2 Reassigning category codes

These functions allow the rescanning of tokens: re-apply \TeX{}’s tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using \texttt{\char_generate:nn}).
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\}}\texttt{\{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.

\texttt{\{tl_rescan\}nn} {\texttt{\{setup\}}} \texttt{\{\{tokens\}\}}

Recans \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 regime rather than one token at a time, so that directives such as \texttt{\{verb\}} that rely on changing category codes will not function properly.

\section{15.7 Constant token lists}

\begin{itemize}
  \item \texttt{\{c_empty_tl\}} Constant that is always empty.
\end{itemize}
A marker for the absence of an argument. This constant \texttt{tl} can safely be typeset (cf. \texttt{\textbackslash q-nil}), with the result being \texttt{-NoValue-}. It is important to note that \texttt{\c_novalue_tl} is constructed such that it will not match the simple text input \texttt{-NoValue-}, \textit{i.e.} that
\begin{verbatim}
\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }
\end{verbatim}
is logically \texttt{false}. The \texttt{\c_novalue_tl} marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty \texttt{tl}.

An explicit space character contained in a token list (compare this with \texttt{\c_space_token}). For use where an explicit space is required.

15.8Scratch token lists

\texttt{\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.

\texttt{\l_tmpb_tl}

\texttt{\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.

\texttt{\g_tmpb_tl}
Chapter 16

The l3tl-build package
Piecewise t1 constructions

16.1 Constructing ⟨t1 var⟩ by accumulation

When creating a ⟨t1 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 ⟨t1 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 ⟨t1 var⟩
Clears the ⟨t1 var⟩ and sets it up to support other \tl_build_... functions. Until \tl_build_end:N ⟨t1 var⟩ is called, applying any function from l3tl other than \tl_... will lead to incorrect results. The \begin and gbegin functions must be used for local and global ⟨t1 var⟩ respectively.

\tl_build_put_left:Nn ⟨t1 var⟩ {⟨tokens⟩}
\tl_build_put_left:Ne ⟨t1 var⟩ {⟨tokens⟩}
\tl_build_gput_left:Nn ⟨t1 var⟩ {⟨tokens⟩}
\tl_build_gput_left:Ne ⟨t1 var⟩ {⟨tokens⟩}
Adds ⟨tokens⟩ to the left or right side of the current contents of ⟨t1 var⟩. The ⟨t1 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 ⟨t1 var⟩ respectively. The right functions are about twice faster than the left functions.

\tl_build_end:N ⟨t1 var⟩
\tl_build_gend:N ⟨t1 var⟩
Gets the contents of ⟨t1 var⟩ and stores that into the ⟨t1 var⟩ using \tl_set:Nn or \tl_gset:Nn. The ⟨t1 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 ⟨t1 var⟩ respectively. These functions completely remove the setup code that enabled ⟨t1 var⟩ to be used for other \tl_build_... functions. After the action of end/gend, the ⟨t1 var⟩ may be manipulated using standard t1 functions.
Chapter 17

The \l3str package: Strings

\TeX{} associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a \TeX{} sense.

A \TeX{} string (and thus an \expB{} string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a \TeX{} string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix \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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\str_new:N</code> (<code>\str_new:c</code>)</td>
<td>Creates a new <code>str var</code> or raises an error if the name is already taken. The declaration is global. The <code>str var</code> is initially empty.</td>
</tr>
<tr>
<td><code>\str_const:Nn</code> (`\str_const:Ncn</td>
<td>cV</td>
</tr>
<tr>
<td><code>\str_clear:N</code> (<code>\str_clear:c</code>)</td>
<td>Clears the content of the <code>str var</code>.</td>
</tr>
<tr>
<td><code>\str_clear_new:N</code> (<code>\str_clear_new:c</code>)</td>
<td>Ensures that the <code>str var</code> exists globally by applying <code>\str_new:N</code> if necessary, then applies <code>\str_(g)clear:N</code> to leave the <code>str var</code> empty.</td>
</tr>
<tr>
<td><code>\str_set_eq:NN</code> (`\str_set_eq:cn</td>
<td>Nc</td>
</tr>
<tr>
<td><code>\str_concat:NNN</code> (<code>\str_concat:ccc</code>)</td>
<td>Concatenates the content of <code>str var2</code> and <code>str var3</code> together and saves the result in <code>str var1</code>. The <code>str var2</code> is placed at the left side of the new string variable. The <code>str var2</code> and <code>str var3</code> must indeed be strings, as this function does not convert their contents to a string.</td>
</tr>
<tr>
<td><code>\str_if_exist_p:N</code> (<code>\str_if_exist_p:c</code>)</td>
<td>Tests whether the <code>str var</code> is currently defined. This does not check that the <code>str var</code> really is a string.</td>
</tr>
<tr>
<td><code>\str_if_exist:NTF</code> (<code>\str_if_exist:c</code>)</td>
<td>Tests whether the <code>str var</code> is currently defined. This does not check that the <code>str var</code> really is a string.</td>
</tr>
</tbody>
</table>
17.2 Adding data to string variables

\str_set:Nn
\str_set:(NV|Ne|cn|cv|ce)
\str_set:Nn
\str_set:(NV|Ne|cn|cv|ce)

Converts the \textit{token list} to a \textit{string}, and stores the result in \textit{str var}.

\str_put_left:Nn
\str_put_left:(NV|Ne|cn|cv|ce)
\str_gput_left:Nn
\str_gput_left:(NV|Ne|cn|cv|ce)

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

\str_put_right:Nn
\str_put_right:(NV|Ne|cn|cv|ce)
\str_gput_right:Nn
\str_gput_right:(NV|Ne|cn|cv|ce)

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

17.3 String conditionals

\str_if_empty_p:N
\str_if_empty_p:NTF
\str_if_empty:N
\str_if_empty:c
\str_if_empty:n
\str_if_empty:n

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

\str_if_eq_p:NN
\str_if_eq:NNTF
\str_if_eq:NN
\str_if_eq:NTF

Compares the content of two \textit{str variables} and is logically \textit{true} if the two contain the same characters in the same order. See \texttt{tl_if_eq:NNTF} to compare tokens (including their category codes) rather than characters.
\str_if_eq_p:nn * \str_if_eq_p:nn \{tl_1\} \{tl_2\}
\str_if_eq_p:\{Vn|on|no|nV|VV|vn|nv|ee\} * \str_if_eq:nnTF \{tl_1\} \{tl_2\} \{\{true code\}\} \{\{false code\}\}
\str_if_eq:nnTF *
\str_if_eq:\{Vn|on|no|nV|VV|vn|nv|ee\}TF *

Compares the two \textit{token lists} on a character by character basis (namely after converting them to strings), and is \texttt{true} if the two \textit{strings} contain the same characters in the same order. Thus for example

\[
\str_if_eq_p:no \{ abc \} \{ \tl_to_str:n \{ abc \} \}
\]

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

\str_if_in:NnTF
\str_if_in:cn
\str_if_in:nnTF

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

\str_if_in:nn

Converts both \textit{token lists} to \textit{strings} and tests whether \textit{string}_2 is found inside \textit{string}_1.

\str_case:nn *
\str_case:\{Vn|on|en|nv|nV\} *
\str_case:\{Vn|on|en|nv\}TF *
\str_case:nn *
\str_case:nnTF *
\str_case:NN *
\str_case:NNTF *

Compares the \textit{test string} in turn with each of the \textit{string case}s (all token lists are converted to strings). If the two are equal (as described for \str_if_eq:nnTF) 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 function \str_case:nn, which does nothing if there is no match, is also available.

This set of functions performs no expansion on each \textit{string case} argument, so any variable in there will be compared as a string. If expansion is needed in the \textit{string case}s, then \str_case_e:nn(TF) should be used instead.
Compares the full expansion of the \textit{test string} in turn with the full expansion of the \textit{string case}s (all token lists are converted to strings). If the two full expansions are equal (as described for \texttt{str_if_eq:eeTF}) 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. In \texttt{str_case_e:nn(TF)}, the \textit{test string} is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

\begin{verbatim}
\str_case_e:nn \str_case_e:en \str_case_e:nn TF \str_case_e:en \str_map_function:nN \str_map_function:NN \str_map_function:cN
\end{verbatim}

\newdate{2018-06-19}

\str_compare_p:nNn \str_compare_p:eNe \str_compare:nNnTF \str_compare:eNe

\newdate{2021-05-17}

\begin{verbatim}
17.4 Mapping over strings

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}
\str_map_function:nN \str_map_function:NN \str_map_function:cN
\end{verbatim}

\newdate{2017-11-14}

\TeXhackersnote{This is a wrapper around the \TeX primitive \texttt{(pdf)strcmp}. It is meant for programming and not for sorting textual contents, as it simply considers character codes and not more elaborate considerations of grapheme clusters, locale, etc.}
\str_map_inline:nn \str_map_inline:nn \{\textit{token list}\} \{\textit{inline function}\}
\str_map_inline:Nn \str_map_inline:Nn \textit{str var} \{\textit{inline function}\}
\textbf{Converts the} \textit{token list} \textbf{to a} \textit{string} \textbf{then applies the} \textit{inline function} \textbf{to every} \textit{character} \textbf{in the} \textit{str var} \textbf{including spaces. The} \textit{inline function} \textbf{should consist of code which receives the} \textit{character} \textbf{as \#1.}

\str_map_tokens:nn \str_map_tokens:nn \{\textit{token list}\} \{\textit{code}\}
\str_map_tokens:Nn \str_map_tokens:Nn \textit{str var} \{\textit{code}\}
\textbf{Converts the} \textit{token list} \textbf{to a} \textit{string} \textbf{then applies} \textit{code} \textbf{to every} \textit{character} \textbf{in the} \textit{str var} \textbf{including spaces. The} \textit{code} \textbf{receives each character as a trailing brace group. This is equivalent to} \texttt{\str_map_function:nN} \textbf{if the} \textit{code} \textbf{consists of a single function.}

\str_map_variable:nNn \str_map_variable:nNn \{\textit{token list}\} \{\textit{variable}\} \{\textit{code}\}
\str_map_variable:NNn \str_map_variable:NNn \textit{str var} \{\textit{variable}\} \{\textit{code}\}
\str_map_variable:cNn \str_map_variable:cNn
\textbf{Converts the} \textit{token list} \textbf{to a} \textit{string} \textbf{then stores each} \textit{character} \textbf{in the} \textit{string} \textbf{(including spaces) in turn in the} \textit{string} \textbf{or token list} \textit{variable} \textbf{and applies the} \textit{code}. \textbf{The} \textit{code} \textbf{will usually make use of the} \textit{variable}, \textbf{but this is not enforced. The assignments to the} \textit{variable} \textbf{are local. Its value after the loop is the last} \textit{character} \textbf{in the} \textit{string}, \textbf{or its original value if the} \textit{string} \textbf{is empty. See also} \texttt{\str_map_inline:Nn}. 

\str_map_break: \str_map_break:
\textbf{Used to terminate a} \texttt{\str_map...} \textbf{function before all characters in the} \textit{string} \textbf{have been processed. This normally takes place within a conditional statement, for example}

\begin{verbatim}
\str_map_inline:Nn \l_my_str
{ 
  \str_if_eq:nnT { #1 } { bingo } { \str_map_break: } 
  \% Do something useful 
}
\end{verbatim}

\textbf{See also} \texttt{\str_map_break:n}. \textbf{Use outside of a} \texttt{\str_map...} \textbf{scenario leads to low level \LaTeX errors.}

\textbf{\LaTeX hackers note:} \textbf{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.}

\textbf{134}
\str_map_break:n \str_map_break:n \langle code \rangle

Used to terminate a \str_map... 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 \langle code \rangle \}}
% Do something useful
\}

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

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before the \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 str var \rangle
\str_use:c \str_use:c

Recovers the content of a \langle 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 str \rangle directly without an accessor function.

\str_count:n \str_count:n \langle token list \rangle
\str_count:N \str_count:N
\str_count:c \str_count:c
\str_count_spaces:n \str_count_spaces:n
\str_count_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 token list \rangle
\str_count_spaces:c \str_count_spaces:c
\str_count_spaces:n \str_count_spaces:n

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, −1 is the right-most character, etc.
\str_range:nnn * \str_range:nnn \{token list\} \{start index\} \{end index\}  
\str_range:nnn *  
\str_range:nnn *  
\str_range_ignore_spaces:nnn *  

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,

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

prints bcde, cdef, ef, and an empty line to the terminal. The \{start index\} must always be smaller than or equal to the \{end index\}: if this is not the case then no output is generated. Thus

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

both yield empty strings.

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

\iow_term:e { \str_range:nnn { abcdefg } { 2 } { 5 } }  
\iow_term:e { \str_range:nnn { abcdefg } { 2 } { -3 } }  
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { 5 } }  
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { -3 } }  
\iow_term:e { \str_range_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 } }  

137
will print four instances of `bcde`, four instances of `bc e` and eight instances of `bcde`.

### 17.6 Modifying string variables

\(\texttt{\textbackslash str\_replace\_once:NNn}\) \(\texttt{\textbackslash str\_replace\_once:cn}\) \(\texttt{\textbackslash str\_greplace\_once:NNn}\) \(\texttt{\textbackslash str\_greplace\_once:cn}\)

```
\textbackslash str\_replace\_once:NNn \langle str\ var\ \rangle \{\langle old\ \rangle\} \{\langle new\ \rangle\}
```

Converts the \(\langle\text{old}\rangle\) and \(\langle\text{new}\rangle\) token lists to strings, then replaces the first (leftmost) occurrence of \(\langle\text{old\ string}\rangle\) in the \(\langle\text{str\ var}\rangle\) with \(\langle\text{new\ string}\rangle\).

```
\textbackslash str\_replace\_all:NNn \langle str\ var\ \rangle \{\langle old\ \rangle\} \{\langle new\ \rangle\}
```

Converts the \(\langle\text{old}\rangle\) and \(\langle\text{new}\rangle\) token lists to strings, then replaces all occurrences of \(\langle\text{old\ string}\rangle\) in the \(\langle\text{str\ var}\rangle\) with \(\langle\text{new\ string}\rangle\). As this function operates from left to right, the pattern \(\langle\text{old\ string}\rangle\) may remain after the replacement (see \(\texttt{\textbackslash str\_remove\_all:Nn}\) for an example).

```
\textbackslash str\_remove\_once:NNn \langle str\ var\ \rangle \{\langle token\ list\rangle\}
```

Converts the \(\langle\text{token\ list}\rangle\) to a \(\langle\text{string}\rangle\) then removes the first (leftmost) occurrence of \(\langle\text{string}\rangle\) from the \(\langle\text{str\ var}\rangle\).

```
\textbackslash str\_remove\_all:Nn \langle str\ var\ \rangle \{\langle token\ list\rangle\}
```

Converts the \(\langle\text{token\ list}\rangle\) to a \(\langle\text{string}\rangle\) then removes all occurrences of \(\langle\text{string}\rangle\) from the \(\langle\text{str\ var}\rangle\). As this function operates from left to right, the pattern \(\langle\text{string}\rangle\) may remain after the removal, for instance,

```
\texttt{\textbackslash str\_set:Nn \l_tmpa\_str \{abccdc\} \texttt{\textbackslash 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 \textit{tokens} to their string representation, as described for \texttt{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 \texttt{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

\texttt{
\cs_new_protected:Npn \myfunc:nn #1#2
{\cs_set_protected:cpn
{user
\str_uppercase:f { \tl_head:n {#1} }
\str_lowercase:f { \tl_tail:n {#1} }
}
{ #2 }
}}

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should \textit{not} be used for

- Caseless comparisons: use \texttt{str_casefold:n} for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the \texttt{text_lowercase:n(n)}, \texttt{text_uppercase:n(n)} and \texttt{text_titlecase_(all|once):n(n)} functions which correctly deal with context-dependence and other factors appropriate to text case changing.

New: 2019-11-26
\str_casefold:n \str_casefold:V

Converts the input \tokens\ to their string representation, as described for \tl_to_str:n, and then folds the case of the resulting \string\ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by \str_casefold:n follows the mappings provided by the Unicode Consortium, who state:

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by \str_casefold:n follows the “full” scheme defined by the Unicode Consortium (e.g. SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (i.e. İ always folds to i and not to i).

\str_mdfive_hash:n \str_mdfive_hash:e

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

17.8 Viewing strings

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

Displays the content of the \str on the terminal.

\str_log:N \str_log:c \str_log:n

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

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

\c_underscore_str
\c_underscore_str
\c_circumflex_str
\c_colon_str
\c_dollar_str
\c_hash_str
\c_percent_str
\c_tilde_str
\c_zero_str

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

17.10 Scratch strings

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.

\l_tmpa_str
\l_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.

\g_tmpa_str
\g_tmpb_str

17.11 Deprecated functions

\str_foldcase:n \star \str_foldcase:n \{\text{tokens}\}
\str_foldcase:V \star

A previous name for the functionally-identical \str_casefold:n.

New: 2019-11-26
Chapter 18

The l3str-convert package:
String encoding conversions

18.1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (e.g., PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be UTF-16, ISO 8859-1, etc. See Table 1 for a list of supported encodings.7

- Bytes are translated to \texttt{T\LaTeX} tokens through a given “escaping”. Those are defined for the most part by the pdf file format. See Table 2 for a list of escaping methods supported.7

7Encodings 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>name</td>
<td>\pdffile \pdfescapefilename</td>
</tr>
<tr>
<td>string</td>
<td>\pdffile \pdfescapestring</td>
</tr>
<tr>
<td>url</td>
<td>Encoding used in URLs</td>
</tr>
</tbody>
</table>
18.2 Conversion functions

\str_set_convert:Nnnn \str_set_convert:Nnnn \str_gset_convert:Nnnn

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

For example,

\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }

results in the variable \l_foo_str holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark `"FEFF, which can be avoided by specifying the encoding utf16be/hex.

An error is raised if the \textlangle string\textrangle is not valid according to the \textlangle escaping 1\textrangle and \textlangle encoding 1\textrangle, or if it cannot be reencoded in the \textlangle encoding 2\textrangle and \textlangle escaping 2\textrangle (for instance, if a character does not exist in the \textlangle encoding 2\textrangle). Erroneous input is replaced by the Unicode replacement character `*FFFD, and characters which cannot be reencoded are replaced by either the replacement character `*FFFD if it exists in the \textlangle encoding 2\textrangle, or an encoding-specific replacement character, or the question mark character.

\str_set_convert:NnnnTF \str_set_convert:NnnnTF \str_gset_convert:NnnnTF

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

18.3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These assume UTF-8 and no escaping in the input.

\str_convert_pdfname:n  \str_convert_pdfname:n

As \str_convert_pdfname:n, converts the \textlangle string\textrangle 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 Xe\TeX/Lua\TeX, 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 \scantokens.

- Change \texttt{\str_set_convert:Nnnn} to expand its last two arguments.

- Describe the internal format in the code comments. Refuse code points in \texttt{[“D800,“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{uri} escaping. Perhaps the characters \texttt{!’()*/\_} 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: ascii85, shell escapes, lua escapes, \texttt{etc}.?
Chapter 19

The \texttt{l3quark} package

Quarks and scan marks

Two special types of constants in \LaTeX{} 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 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_new:N \quark (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
\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\{\true list\}
\quark_if_nil:nTF  \quark_if_nil:nTF \token\{\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
\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\{\true list\}
\quark_if_no_value:nTF \quark_if_no_value:nTF \token\{\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 \q_recursion_tail \q_recursion_stop
Tests if \token contains only the marker \q_recursion_tail, and if so uses \use_none_delimit_by_q_recursion_stop:w to terminate the recursion that this belongs to. The recursion input must include the marker tokens \q_recursion_tail and \q_recursion_stop as the last two items.

\quark_if_recursion_tail_stop:n \q_recursion_tail \q_recursion_stop:n \{\token list\}
\quark_if_recursion_tail_stop:o \q_recursion_tail \q_recursion_stop:o
Tests if the \token list contains only \q_recursion_tail, and if so uses \use_i_delimit_by_q_recursion_stop:w to terminate the recursion that this belongs to. The recursion input must include the marker tokens \q_recursion_tail and \q_recursion_stop as the last two items. The \insertion code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_stop_do:Nn \q_recursion_tail \q_recursion_stop:do:Nn \{\token\} \{\insertion\}
\quark_if_recursion_tail_stop_do:nn \q_recursion_tail \q_recursion_stop:do:nn \{\token list\} \{\insertion\}
Tests if the \token list contains only \q_recursion_tail, and if so uses \use_i_delimit_by_q_recursion_stop:w to terminate the recursion that this belongs to. The recursion input must include the marker tokens \q_recursion_tail and \q_recursion_stop as the last two items. The \insertion code is then added to the input stream after the recursion has ended.
Tests if \(\langle\text{token list}\rangle\) contains only \(\texttt{\_map\_break}\), and if so terminates the recursion using \(\langle\text{type}\rangle\texttt{\_map\_break}\). The recursion end should be marked by \(\texttt{prg\_break-}\texttt{point}N\langle\text{type}\rangle\texttt{\_map\_break}\).

### 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{\_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{\my\_map\_dbl:nn} which takes a token list and applies an operation to every pair of tokens. For example, \texttt{\my\_map\_dbl:nn {abcd} \{{--\#1--\#2--}\}} would produce “\(\text{[-a-b-]}\ [-c-d-]\)”.

Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here’s the definition of \texttt{\my\_map\_dbl:nn}. First of all, define the function that does the processing based on the inline function argument \#2. Then initiate the recursion using an internal function. The token list \#1 is terminated using \(\texttt{\_recursion\_tail}\), with delimiters according to the type of recursion (here a pair of \(\texttt{\_recursion\_tail}\)), concluding with \(\texttt{\_recursion\_stop}\). 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 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).

\begin{verbatim}
\scan_new:N \scan_new:N \scan_mark\scan_stop:
\scan_new:N \use_none_delimit_by_s_stop:w
\end{verbatim}

Creates a new \langle \textit{scan mark} \rangle which is set equal to \\scan_stop:. The \langle \textit{scan mark} \rangle is defined globally, and an error message is raised if the name was already taken by another scan mark.

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

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.

\begin{verbatim}
\use_none_delimit_by_s_stop:w * \use_none_delimit_by_s_stop:w \tokens \s_stop
\end{verbatim}

Removes the \langle \textit{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} package
Sequences and stacks

\LaTeX{}X3 implements a “sequence” data type, which contain an ordered list of entries which may contain any \texttt{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 \LaTeX{}X3. 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_clear:N \seq_clear:C \seq_gclear:N \seq_gclear:C
\seq_set_eq:NN \seq_set_eq:N \seq_set_eq:C \seq_gset_eq:NN \seq_gset_eq:C
\end{verbatim}

\verb+\seq_new:N \seq_new:N \seq_new:C+  \texttt{\seq_new:N \seq \seq \seq}  \texttt{\seq \seq \seq}

Creates a new \texttt{(seq var)} or raises an error if the name is already taken. The declaration is global. The \texttt{(seq var)} initially contains no items.

\verb+\seq_clear:N \seq_clear:C \seq_gclear:N \seq_gclear:C+  \texttt{\seq \seq \seq \seq}

Clears all items from the \texttt{(seq var)}.

\verb+\seq_clear_new:N \seq_clear_new:C \seq_gclear_new:N \seq_gclear_new:C+  \texttt{\seq \seq \seq \seq}

Ensures that the \texttt{(seq var)} exists globally by applying \texttt{\seq_new:N} if necessary, then applies \texttt{\seq\textunderscore (g)clear:N} to leave the \texttt{(seq var)} empty.

\verb+\seq_set_eq:NN \seq_set_eq:N \seq_set_eq:C \seq_gset_eq:NN \seq_gset_eq:C+  \texttt{\seq \seq \seq \seq}

Sets the content of \texttt{(seq var\textsubscript{1})} equal to that of \texttt{(seq var\textsubscript{2})}.  

151
Converts the data in the ⟨comma list⟩ into a ⟨seq var⟩: the original ⟨comma list⟩ is unchanged.

Creates a new constant ⟨seq var⟩ or raises an error if the name is already taken. The ⟨seq var⟩ is set globally to contain the items in the ⟨comma list⟩.

Splits the ⟨token list⟩ into ⟨items⟩ separated by ⟨delimiter⟩, and assigns the result to the ⟨seq var⟩. Spaces on both sides of each ⟨item⟩ are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of \texttt{l3clist} functions. Empty ⟨items⟩ are preserved by \seq_set_split:Nnn, and can be removed afterwards using \seq_remove_all:Nn ⟨seq var⟩ \{\}. The ⟨delimiter⟩ may not contain {, } or # (assuming \TeX’s normal category code régime). If the ⟨delimiter⟩ is empty, the ⟨token list⟩ is split into ⟨items⟩ as a ⟨token list⟩. See also \seq_set_split_keep_spaces:Nnn, which omits space stripping.

Splits the ⟨token list⟩ into ⟨items⟩ separated by ⟨delimiter⟩, and assigns the result to the ⟨seq var⟩. One set of outer braces is removed (if any) but any surrounding spaces are retained: any braces inside one or more spaces are therefore kept. Empty ⟨items⟩ are preserved by \seq_set_split_keep_spaces:Nnn, and can be removed afterwards using \seq_remove_all:Nn ⟨seq var⟩ \{\}. The ⟨delimiter⟩ may not contain {, } or # (assuming \TeX’s normal category code régime). If the ⟨delimiter⟩ is empty, the ⟨token list⟩ is split into ⟨items⟩ as a ⟨token list⟩. See also \seq_set_split:Nnn, which removes spaces around the delimiters.
20.2 Appending data to sequences

\seq_concat:NNN \seq_concat:ccc \seq_gconcat:NNN \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_if_exist_p:N \seq_if_exist_p:c \seq_if_exist:NTF \seq_if_exist:c \*
Tests whether the \seq var is currently defined. This does not check that the \seq var really is a sequence variable.

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 \seq_get_left:cN \seq_get_right:NN \seq_get_right:cN
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_left:NN \seq_get_left:cN \seq_get_right:NN \seq_get_right:cN
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:NN \seq_pop_left:CN
Pops the left-most item from a \seq{} into the \tokenlist{}, \ie removes the item from the sequence and stores it in the \tokenlist{}. Both of the variables are assigned locally. If \seq{} is empty the \tokenlist{} is set to the special marker \q_no_value.

\seq_gpop_left:NN \seq_gpop_left:CN
Pops the left-most item from a \seq{} into the \tokenlist{}, \ie removes the item from the sequence and stores it in the \tokenlist{}. The \seq{} is modified globally, while the assignment of the \tokenlist{} is local. If \seq{} is empty the \tokenlist{} is set to the special marker \q_no_value.

\seq_pop_right:NN \seq_pop_right:CN
Pops the right-most item from a \seq{} into the \tokenlist{}, \ie removes the item from the sequence and stores it in the \tokenlist{}. Both of the variables are assigned locally. If \seq{} is empty the \tokenlist{} is set to the special marker \q_no_value.

\seq_gpop_right:NN \seq_gpop_right:CN
Pops the right-most item from a \seq{} into the \tokenlist{}, \ie removes the item from the sequence and stores it in the \tokenlist{}. The \seq{} is modified globally, while the assignment of the \tokenlist{} is local. If \seq{} is empty the \tokenlist{} is set to the special marker \q_no_value.

\seq_item:Nn
Indexing items in the \seq{} from 1 at the top (left), this function evaluates the \integer{} and leaves the appropriate item from the sequence in the input stream. If the \integer{} is negative, indexing occurs from the bottom (right) of the sequence. If the \integer{} is larger than the number of items in the \seq{} (as calculated by \seq_count:N) then the function expands to nothing.

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

\seq_rand_item:N \seq_rand_item:CN
Selects a pseudo-random item of the \seq{}. If the \seq{} is empty the result is empty. This is not available in older versions of \TeX{}.

\TeXhackers{} note: The result is returned within the \unexpanded{} primitive (\exp_not:n), which means that the \item{} does not expand further when appearing in an e-type or x-type argument expansion.
20.4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

\seq_get_left:NNTF
\seq_get_left: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 left-most item from the \seq var in the \token list variable without removing it from the \seq var, then leaves the \true code in the input stream. The \token list variable is assigned locally.

\seq_get_right:NNTF
\seq_get_right: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 right-most item from the \seq var in the \token list variable without removing it from the \seq var, then leaves the \true code in the input stream. The \token list variable is assigned locally.

\seq_pop_left:NNTF
\seq_pop_left: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, pops the left-most item from the \seq var in the \token list variable, \ie removes the item from the \seq var, then leaves the \true code in the input stream. Both the \seq var and the \token list variable are assigned locally.

\seq_gpop_left:NNTF
\seq_gpop_left: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, pops the left-most item from the \seq var in the \token list variable, \ie 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.

\seq_pop_right:NNTF
\seq_pop_right: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, pops the right-most item from the \seq var in the \token list variable, \ie removes the item from the \seq var, then leaves the \true code in the input stream. Both the \seq var and the \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.

Removing duplicate items from the ⟨seq var⟩, leaving the left most copy of each item in the ⟨seq var⟩. The ⟨item⟩ comparison takes place on a token basis, as for \tl_if_eq:nnTF.

\TeXhackers note: This function iterates through every item in the ⟨seq var⟩ and does a comparison with the ⟨items⟩ already checked. It is therefore relatively slow with large sequences.

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

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.

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:c \seq_if_in:NnTF \seq_if_in:N \seq_if_in:NTF \seq_if_in:NV \seq_if_in:Nv \seq_if_in:Ne \seq_if_in:No \seq_if_in:cn \seq_if_in:cV \seq_if_in:cv \seq_if_in:ce \seq_if_in:co
Tests if the \item is present in the \seq var.

\seq_map_function:NN \seq_map_function: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, \ie any local assignments made by the \function or \code discussed below remain in effect after the loop.

\seq_map_function:NN \seq_map_function:cn
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.
Analogue of \seq_map_function:NN which maps several tokens instead of a single function. The \langle code \rangle receives each item in the \langle seq var \rangle 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 \langle seq var \rangle: for each item in \l_my_seq the function \prg_replicate:nn receives 2 and \langle item \rangle as its two arguments. The function \seq_map_inline:Nn is typically faster but it is not expandable.

\seq_map_variable:NNn \langle seq var \rangle \langle variable \rangle \{ \langle code \rangle \}

Stores each \langle item \rangle of the \langle seq var \rangle in turn in the (token list) \langle variable \rangle and applies the \langle code \rangle. The \langle code \rangle will usually make use of the \langle variable \rangle, but this is not enforced. The assignments to the \langle variable \rangle are local. Its value after the loop is the last \langle item \rangle in the \langle seq var \rangle, or its original value if the \langle seq var \rangle is empty. The \langle items \rangle are returned from left to right.

\seq_map_indexed_function:NN \seq_map_indexed_function:NN \langle seq var \rangle \{ \langle function \rangle \}

Applies \langle function \rangle to every entry in the \langle sequence variable \rangle. The \langle function \rangle should have signature :nn. It receives two arguments for each iteration: the \langle index \rangle (namely 1 for the first entry, then 2 and so on) and the \langle item \rangle.

\seq_map_indexed_inline:Nn \seq_map_indexed_inline:Nn \langle seq var \rangle \{ \langle inline function \rangle \}

Applies \langle inline function \rangle to every entry in the \langle sequence variable \rangle. The \langle inline function \rangle should consist of code which receives the \langle index \rangle (namely 1 for the first entry, then 2 and so on) as #1 and the \langle item \rangle as #2.

\seq_map_pairwise_function:NNN \seq_map_pairwise_function:NNN \langle seq_1 \rangle \langle seq_2 \rangle \{ \langle function \rangle \}

Applies \langle function \rangle to every pair of items \langle seq_1-item \rangle–\langle seq_2-item \rangle from the two sequences, returning items from both sequences from left to right. The \langle function \rangle receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (i.e. whichever sequence has fewer items determines how many iterations occur).
\seq_map_break: \cleanlist
\seq_map_break:

Used to terminate a \seq_map... function before all entries in the \seqvar{seq var} have been processed. This normally takes place within a conditional statement, for example

\seq_map_inline:Nn \l_my_seq
{\str_if_eq:nnTF { #1 } { bingo } { \seq_map_break: }
{ % Do something useful }
}

Use outside of a \seq_map... scenario leads to low level \TeX errors.

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\seq_map_break:n ⟨code⟩
\seq_map_break:n

Used to terminate a \seq_map... function before all entries in the \seqvar{seq var} have been processed, inserting the \langle code⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

\seq_map_inline:Nn \l_my_seq
{\str_if_eq:nnTF { #1 } { bingo } { \seq_map_break:n { ⟨code⟩ } }
{ % Do something useful }
}

Use outside of a \seq_map... scenario leads to low level \TeX errors.

\TeXhackers note: When the mapping is broken, additional tokens may be inserted before the \langle code⟩ is inserted into the input stream. This depends on the design of the mapping function.

\seq_set_map:NNn \seq_gset_map:NNn
\seq_set_map:NNn \seq_gset_map:NNn
\Rev: 2011-12-22
\Updated: 2020-07-16

Applies \langle inline function⟩ to every \langle item⟩ stored within the \seqvar{seq var2}. The \langle inline function⟩ should consist of code which will receive the \langle item⟩ as #1. The sequence resulting applying \langle inline function⟩ to each \langle item⟩ is assigned to \seqvar{seq var1}.

\TeXhackers note: Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level \TeX errors.
\seq_set_map_e:NNn \seq_set_gmap_e:NNn
\seq_count:N \seq_count:c
\seq_use:Nnnn \seq_use:cnnn
\seq_set_split:Nnn

**20.8 Using the content of sequences directly**

By default, sequences are stored in the TeX kernel as \texttt{seq} variables. To access the contents of a sequence, the \texttt{\seq_count:N} primitive is defined, which returns the number of items in the sequence as an integer denotation. The total number of items in a sequence includes those which are empty and duplicates, i.e. every item in a sequence is unique.

For example,

\begin{verbatim}
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { \textbf{and} } { ,~ } { ,\textbf{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.

**\TeXhackers\ note:** The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the items do not expand further when appearing in an e-type or x-type argument expansion.

---

**\TeXhackers\ note:** Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level \TeX\ errors.
Places the contents of the \seq var in the input stream, with the \langle separator\rangle between the items. If the sequence has a single item, it is placed in the input stream with no \langle separator\rangle, 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.

\TeXHackersNote: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the \langle items\rangle 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
\UseTextCommand:NN \UseTextCommand:cN
Reads the top item from a \seq var into the \langle token list variable\rangle without removing it from the \seq var. The \langle token list variable\rangle is assigned locally. If \seq var is empty the \langle token list variable\rangle is set to the special marker \q_no_value.

\seq_pop:NN \seq_pop:cN
\UseTextCommand:NN \UseTextCommand:cN
Pops the top item from a \seq var into the \langle token list variable\rangle. Both of the variables are assigned locally. If \seq var is empty the \langle token list variable\rangle is set to the special marker \q_no_value.

\seq_gpop:NN \seq_gpop:cN
\UseTextCommand:NN \UseTextCommand:cN
Pops the top item from a \seq var into the \langle token list variable\rangle. The \seq var is modified globally, while the \langle token list variable\rangle is assigned locally. If \seq var is empty the \langle token list variable\rangle is set to the special marker \q_no_value.

\seq_get:NNTF \seq_get:cN
\UseTextCommand:NN \UseTextCommand:cN
If the \seq var is empty, 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 \seq var is non-empty, stores the top item from a \seq var in the \langle token list variable\rangle without removing it from the \seq var. The \langle token list variable\rangle is assigned locally.
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 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{sequence variable} are straightforward. For instance, \texttt{\seq_count:N} \texttt{sequence variable} expands to the number of items, while \texttt{\seq_if_in:NnTF} \texttt{sequence variable} \texttt{item} tests if the \texttt{item} is in the set.

Adding an \texttt{item} to a set \texttt{sequence variable} can be done by appending it to the \texttt{sequence variable} if it is not already in the \texttt{sequence variable}:

\begin{verbatim}
\seq_if_in:NnF \seq \{item\}
{ \seq_put_right:Nn \seq \{item\} }
\end{verbatim}

Removing an \texttt{item} from a set \texttt{sequence variable} can be done using \texttt{\seq_remove_all:N},

\begin{verbatim}
\seq_remove_all:N \seq \{item\}
\end{verbatim}

The intersection of two sets \texttt{sequence variable} and \texttt{sequence variable} can be stored into \texttt{sequence variable} by collecting items of \texttt{sequence variable} which are in \texttt{sequence variable}.

\begin{verbatim}
\seq_push:Nn \seq \{item\}
\seq_gpush:Nn \seq \{item\}
\end{verbatim}
The code as written here only works if \texttt{\seq var 3} is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence \texttt{\l_\langle pkg\rangle_internal_seq}, then \texttt{\seq var 3} should be set equal to this internal sequence.

The same remark applies to other set functions.

The union of two sets \texttt{\seq var 1} and \texttt{\seq var 2} can be stored into \texttt{\seq var 3} through

\begin{verbatim}
\seq_concat:NNN \seq var 3 \seq var 1 \seq var 2
\seq_remove_duplicates:N \seq var 3
\end{verbatim}

or by adding items to (a copy of) \texttt{\seq var 3} one by one

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

The second approach is faster than the first when the \texttt{\seq var 2} is short compared to \texttt{\seq var 1}.

The difference of two sets \texttt{\seq var 1} and \texttt{\seq var 2} can be stored into \texttt{\seq var 3} by removing items of the \texttt{\seq var 2} from (a copy of) the \texttt{\seq var 1} one by one.

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

The symmetric difference of two sets \texttt{\seq var 1} and \texttt{\seq var 2} can be stored into \texttt{\seq var 3} by computing the difference between \texttt{\seq var 1} and \texttt{\seq var 2} and storing the result as \texttt{\l_\langle pkg\rangle_internal_seq}, then the difference between \texttt{\seq var 2} and \texttt{\seq var 1}, and finally concatenating the two differences to get the symmetric differences.

\begin{verbatim}
\seq_set_eq:NN \l_\langle pkg\rangle_internal_seq \seq var 1
\seq_map_inline:Nn \seq var 2
{ \seq_remove_all:Nn \l_\langle pkg\rangle_internal_seq \{#1\} }
\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\} }
\seq_concat:NNN \seq var 3 \seq var 3 \l_\langle pkg\rangle_internal_seq
\end{verbatim}

\section{Constant and scratch sequences}

\texttt{\c_empty_seq} Constant that is always empty.

Rev: 2012-07-02
Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any \texttt{\LaTeX{}3}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\begin{verbatim}
\l_tmpa_seq \l_tmpb_seq
\end{verbatim}

Scratch sequences for global assignment. These are never used by the kernel code, and so are safe for use with any \texttt{\LaTeX{}3}-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_seq \g_tmpb_seq
\end{verbatim}

\section*{20.12 Viewing sequences}

\begin{verbatim}
\seq_show:N \seq_show:c
\end{verbatim}

\texttt{\seq_show:N \seq var} Displays the entries in the \texttt{\seq var} in the terminal.

\begin{verbatim}
\seq_log:N \seq_log:c
\end{verbatim}

\texttt{\seq_log:N \seq var} Writes the entries in the \texttt{\seq var} in the log file.
Chapter 21

The \texttt{l3int} package

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 \( +, - / \) and \( \ast \) and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on \emph{integer expressions} \( \langle \texttt{int expr} \rangle \).

21.1 Integer expressions

Throughout this module, \( \text{(almost)} \) all \texttt{n}-type argument allow for a \( \langle \texttt{int expr} \rangle \) argument with the following syntax. The \emph{integer expression} should consist, after expansion, of \( +, -, \ast, /, (, ) \) 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 \( a \ast b/c \), for which \( a \ast b \) may be arbitrarily large (but the operands \( a, b, c \) are still constrained to an absolute value at most \( 2^{31} - 1 \));
- parentheses may not appear after unary \( + \) or \( - \), namely placing \( + ( \) or \( - ( \) at the start of an expression or after \( +, -, \ast, /, ( \) 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}
\texttt{\int_show:n \{ 5 + 4 \ast 3 - ( 3 + 4 \ast 5 ) \}}
\end{verbatim}

and

\begin{verbatim}
\texttt{\tl_new:N \l_my_tl }  \\
\texttt{\tl_set:Nn \l_my_tl \{ 5 \}}  \\
\texttt{\int_new:N \l_my_int}  \\
\texttt{\int_set:Nn \l_my_int \{ 4 \}}  \\
\texttt{\int_show:n \{ \l_my_tl + \l_my_int \ast 3 - ( 3 + 4 \ast 5 ) \}}
\end{verbatim}

165
show the same result −6 because \_my
tl expands to the integer denotation 5 while the integer variable \_my
tl_int takes the value 4. As the \textit{integer expression} is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and \exp_not:n and its variants have no effect while \exp_not:N may incorrectly interrupt the expression.

\begin{itemize}
  \item \textbf{\LaTeX}hackers note: Exactly two expansions are needed to evaluate \int_eval:n. The result is \textit{not} an \textit{internal integer}, and therefore should be terminated by a space if used in \int_value:w or in a \LaTeX\-style integer assignment.
  \item As all \TeX integers, integer operands can also be: \texttt{\textbackslash value\{\textbackslash L\textbackslash A\textbackslash TEX2\textbackslash \varepsilon\textbackslash counter\}}; dimension or skip variables, converted to integers in sp; the character code of some character given as ‘\texttt{char}’ or ‘\texttt{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.
\end{itemize}
\int_eval:n \{\langle\text{int expr}\rangle\}

Evaluates the \langle\text{int expr}\rangle 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 \langle\text{int expr}\rangle should consist, after expansion, of +, -, *, /, ( ) and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- / denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling operations $a*b/c$, for which $a*b$ may be arbitrarily large;
- parentheses may not appear after unary + or -, namely placing +( or -( at the start of an expression or after +, -, *, / or ( leads to an error.

Each integer operand can be either an integer variable (with no need for \int_use:N) or an integer denotation. For example both

\int_eval:n \{ 5 + 4 * 3 - ( 3 + 4 * 5 ) \}

and

\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl \{ 5 \}
\int_new:N \l_my_int
\int_set:Nn \l_my_int \{ 4 \}
\int_eval:n \{ \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) \}

evaluate to $-6$ because $\l_my_tl$ expands to the integer denotation 5. As the \langle\text{int expr}\rangle is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and \exp_not:n and its variants have no effect while \exp_not:N may incorrectly interrupt the expression.

\TeXhackers note: Exactly two expansions are needed to evaluate \int_eval:n. The result is not an \langle\text{internal integer}\rangle, and therefore requires suitable termination if used in a \TeX-style integer assignment.

As all \TeX integers, integer operands can also be dimension or skip variables, converted to integers in sp, or octal numbers given as ' followed by digits other than 8 and 9, or hexadecimal numbers given as " followed by digits or upper case letters from A to F, or the character code of some character or one-character control sequence, given as \langle\text{char}\rangle.

\int_eval:w \{\langle\text{int expr}\rangle\}

Nov: 2018-03-30

Evaluates the \langle\text{int expr}\rangle as described for \int_eval:n. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is \scan_stop: it is removed, otherwise not. Spaces do not terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, \int_eval:w 1_0*1_0 9 (with explicit space tokens inserted using \~ in a code setting) expands to 29 since the digit 9 is not part of the expression. Expansion details, etc., are as given for \int_eval:n.
\int_sign:n \{\text{int expr}\}

Evaluates the \text{(int expr)} then leaves 1 or 0 or \(-1\) in the input stream according to the sign of the result.

\int_abs:n \{\text{int expr}\}

Evaluates the \text{(int expr)} as described for \int_eval:n and leaves the absolute value of the result in the input stream as an \text{integer denotation} after two expansions.

\int_div_round:nn \{\text{int expr}_1\} \{\text{int expr}_2\}

Evaluates the two \text{(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 \text{(int expr)}. The result is left in the input stream as an \text{integer denotation} after two expansions.

\int_div_truncate:nn \{\text{int expr}_1\} \{\text{int expr}_2\}

Evaluates the two \text{(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.

\int_max:nn \{\text{int expr}_1\} \{\text{int expr}_2\}
\int_min:nn \{\text{int expr}_1\} \{\text{int expr}_2\}

Evaluates the \text{(int expr)s} as described for \int_eval:n and leaves either the larger or smaller value in the input stream as an \text{integer denotation} after two expansions.

\int_mod:nn \{\text{int expr}_1\} \{\text{int expr}_2\}

Evaluates the two \text{(int expr)s} as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting \int_div_truncate:nn \{\text{int expr}_1\} \{\text{int expr}_2\} times \text{int expr}_2 from \text{int expr}_1. Thus, the result has the same sign as \text{int expr}_1 and its absolute value is strictly less than that of \text{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

\int_new:N \{\text{integer}\}

\int_new:N \{\text{integer}\}

\int_new:c \{\text{integer}\}

\int_new:N\int_new:c

\int_const:Nn \{\text{integer}\} \{\text{int expr}\}
\int_const:cn \{\text{integer}\} \{\text{int expr}\}

\int_const:Nn \int_const:cn

Updated: 2011-10-22

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.

\int_const:Nn \int_const:cn

Updated: 2011-10-22

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 \text{int expr}.
\int_zero:N \int_zero:c \int_gzero:N \int_gzero:c

Sets \langle\text{integer}\rangle to 0.

\int_zero_new:N \int_zero_new:c \int_gzero_new:N \int_gzero_new:c

Ensures that the \langle\text{integer}\rangle exists globally by applying \int_new:N if necessary, then applies \int_(g)zero:N to leave the \langle\text{integer}\rangle set to zero.

\int_set_eq:NN \int_set_eq:cn \int_gset_eq:NN \int_gset_eq:cn

Sets the content of \langle\text{integer}_1\rangle equal to that of \langle\text{integer}_2\rangle.

\int_if_exist_p:N \int_if_exist:NTF \int_if_exist:c \int_if_exist_p:c \int_if_exist:NTF cN \int_if_exist:c

Tests whether the \langle\text{int}\rangle is currently defined. This does not check that the \langle\text{int}\rangle really is an integer variable.

21.3 Setting and incrementing integers

\int_add:Nn \int_add:cn \int_gadd:Nn \int_gadd:cn

Adds the result of the \langle\text{int expr}\rangle to the current content of the \langle\text{integer}\rangle.

\int_decr:N \int_decr:c \int_gdecr:N \int_gdecr:c

Decreases the value stored in \langle\text{integer}\rangle by 1.

\int_incr:N \int_incr:c \int_gincr:N \int_gincr:c

Increases the value stored in \langle\text{integer}\rangle by 1.

\int_set:Nn \int_set:cn \int_gset:Nn \int_gset:cn

Sets \langle\text{integer}\rangle to the value of \langle\text{int expr}\rangle, which must evaluate to an integer (as described for \int_eval:n).
\int_{sub}:Nn \int_{sub}:cn \int_{gsub}:Nn \int_{gsub}:cn

Updated: 2011-10-22

21.4 Using integers

\int_{use}:N \int_{use}:c

Updated: 2011-10-22

21.5 Integer expression conditionals

\int_{compare}:p:nNn \int_{compare}:nNnTF

This function first evaluates each of the \int exprs as described for \int eval:n. The two results are then compared using the \relation:

Equal =
Greater than >
Less than <

This function is less flexible than \int_{compare}:nTF but around 5 times faster.
\texttt{\textbackslash int\_compare\_p:n} * \texttt{\textbackslash int\_compare\_nTF} * \\
{ 
  \texttt{(int expr\_1)} \texttt{\textlangle relation\_1\textrangle} \\
  \ldots \\
  \texttt{(int expr\_N)} \texttt{\textlangle relation\_N\textrangle} \\
  \texttt{(int expr\_N+1)} 
}

\texttt{\textbackslash int\_compare\_nTF} \\
{ 
  \texttt{(int expr\_1)} \texttt{\textlangle relation\_1\textrangle} \\
  \ldots \\
  \texttt{(int expr\_N)} \texttt{\textlangle relation\_N\textrangle} \\
  \texttt{(int expr\_N+1)} 
}

\texttt{((true code))} \texttt{((false code))}

This function evaluates the \texttt{(int expr)}s as described for \texttt{\textbackslash int\_eval:n} and compares consecutive result using the corresponding \texttt{\textlangle relation\textrangle}, namely it compares \texttt{(int expr\_1)} and \texttt{(int expr\_2)} using the \texttt{\textlangle relation\_1\textrangle}, then \texttt{(int expr\_2)} and \texttt{(int expr\_3)} using the \texttt{\textlangle relation\_2\textrangle}, until finally comparing \texttt{(int expr\_N)} and \texttt{(int expr\_N+1)} using the \texttt{\textlangle relation\_N\textrangle}. The test yields \texttt{true} if all comparisons are \texttt{true}. Each \texttt{(int expr)} is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is \texttt{false}, then no other \texttt{(integer expression)} is evaluated and no other comparison is performed. The \texttt{\textlangle relations\textrangle} can be any of the following:

\begin{itemize}
  \item Equal \quad = \text{ or } ==
  \item Greater than or equal to \quad \geq
  \item Greater than \quad >
  \item Less than or equal to \quad \leq
  \item Less than \quad <
  \item Not equal \quad !=
\end{itemize}

This function is more flexible than \texttt{\textbackslash int\_compare:nNnTF} but around 5 times slower.
This function evaluates the \textit{test int expr} and compares this in turn to each of the \textit{int 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{\textbackslash int_case:nn}, which does nothing if there is no match, is also available. For example

\texttt{\textbackslash 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 \textit{int expr} as described for \texttt{\textbackslash int_eval:n}. It then evaluates if this is odd or even, as appropriate.

This function first evaluates the \textit{int expr} as described for \texttt{\textbackslash int_eval:n}. It then evaluates if this is zero or not.

\subsection{21.6 Integer expression loops}

\texttt{\textbackslash int_do_until:nNnn}\{ \textit{int expr} \}\{ \textit{relation} \}\{ \textit{int expr2} \}\{ \textit{code} \}

Places the \textit{code} in the input stream for \LaTeX{} 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{false} then the \textit{code} is inserted into the input stream again and a loop occurs until the \textit{relation} is \texttt{true}. 172
\int_do_while:nNnn \{\text{expr}_1\} \{\text{relation}\} \{\text{expr}_2\} \{\text{code}\}

Places the \{\text{code}\} in the input stream for \TeX to process, and then evaluates the relationship between the two \{\text{expr}\}s as described for \\texttt{\int_compare:nNnTF}. If the test is \texttt{true} then the \{\text{code}\} is inserted into the input stream again and a loop occurs until the \{\text{relation}\} is \texttt{false}.

\int_until_do:nNnn \{\text{expr}_1\} \{\text{relation}\} \{\text{expr}_2\} \{\text{code}\}

Evaluates the relationship between the two \{\text{expr}\}s as described for \\texttt{\int_compare:nNnTF}, and then places the \{\text{code}\} in the input stream if the \{\text{relation}\} is \texttt{false}. After the \{\text{code}\} has been processed by \TeX the test is repeated, and a loop occurs until the test is \texttt{true}.

\int_while_do:nNnn \{\text{expr}_1\} \{\text{relation}\} \{\text{expr}_2\} \{\text{code}\}

Evaluates the relationship between the two \{\text{expr}\}s as described for \\texttt{\int_compare:nNnTF}, and then places the \{\text{code}\} in the input stream if the \{\text{relation}\} is \texttt{true}. After the \{\text{code}\} has been processed by \TeX the test is repeated, and a loop occurs until the test is \texttt{false}.

\int_do_until:nn \{\text{integer relation}\} \{\text{code}\}

Placed the \{\text{code}\} in the input stream for \TeX to process, and then evaluates the \{\text{integer relation}\} as described for \\texttt{\int_compare:nNnTF}. If the test is \texttt{false} then the \{\text{code}\} is inserted into the input stream again and a loop occurs until the \{\text{relation}\} is \texttt{true}.

\int_until_do:nn \{\text{integer relation}\} \{\text{code}\}

Placed the \{\text{code}\} in the input stream for \TeX to process, and then evaluates the \{\text{integer relation}\} as described for \\texttt{\int_compare:nNnTF}. If the test is \texttt{true} then the \{\text{code}\} is inserted into the input stream again and a loop occurs until the \{\text{relation}\} is \texttt{false}.

\int_do_until:nn \{\text{integer relation}\} \{\text{code}\}

Evaluates the \{\text{integer relation}\} as described for \\texttt{\int_compare:nNnTF}, and then places the \{\text{code}\} in the input stream if the \{\text{relation}\} is \texttt{false}. After the \{\text{code}\} has been processed by \TeX the test is repeated, and a loop occurs until the test is \texttt{true}.

\int_while_do:nn \{\text{integer relation}\} \{\text{code}\}

Evaluates the \{\text{integer relation}\} as described for \\texttt{\int_compare:nNnTF}, and then places the \{\text{code}\} in the input stream if the \{\text{relation}\} is \texttt{true}. After the \{\text{code}\} has been processed by \TeX the test is repeated, and a loop occurs until the test is \texttt{false}.

Updated: 2013-01-13
21.7 Integer step functions

\int_step_function:nN \int_step_function:nnN \int_step_function:nN

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

\cs_set:Npn \my_func:n #1 \{ \text{[I saw #1]} \quad \}
\int_step_function:nnN \{ 1 \} \{ 1 \} \{ 5 \} \my_func:n

would print

[I saw 1]  [I saw 2]  [I saw 3]  [I saw 4]  [I saw 5]

The functions \int_step_function:nN and \int_step_function:nnN both use a fixed \langle step \rangle of 1, and in the case of \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.

\int_step_inline:nn \int_step_inline:nn \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 \int_step_inline:nn and \int_step_inline:nnn both use a fixed \langle step \rangle of 1, and in the case of \int_step_inline:nn the \langle initial value \rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\int_step_variable:nN \int_step_variable:nnN \int_step_variable:nnN \int_step_variable: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 the \langle t1 var \rangle defined as the current \langle value \rangle. Thus the \langle code \rangle should make use of the \langle t1 var \rangle.

The functions \int_step_variable:nN and \int_step_variable:nnN both use a fixed \langle step \rangle of 1, and in the case of \int_step_variable:nN 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.

\int_to_arabic:n \{ \langle \text{int expr} \rangle \}

Places the value of the \langle \text{int expr} \rangle in the input stream as digits, with category code 12 (other).

\int_to_arabic:v \star \int_to_arabic:n \star

\int_to_alph:n \{ \langle \text{int expr} \rangle \}

Evaluates the \langle \text{int expr} \rangle and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

\int_to_alph:n \{ 1 \}

places a in the input stream,

\int_to_alph:n \{ 26 \}

is represented as z and

\int_to_alph:n \{ 27 \}

is converted to aa. For conversions using other alphabets, use \int_to_symbols:nnn to define an alphabet-specific function. The basic \int_to_alph:n and \int_to_Alph:n functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_symbols:nnn \star \int_to_symbols:nnn \star

\cs_new:Npn \int_to_alph:n #1

{ \int_to_symbols:nnn \{ \langle \text{int expr} \rangle \} \{ \langle \text{total symbols} \rangle \} \{ \langle \text{value to symbol mapping} \rangle \} }

This is the low-level function for conversion of an \langle \text{int expr} \rangle into a symbolic form (often letters). The \langle \text{total symbols} \rangle available should be given as an integer expression. Values are actually converted to symbols according to the \langle \text{value to symbol mapping} \rangle. This should be given as \langle \text{total symbols} \rangle pairs of entries, a number and the appropriate symbol. Thus the \int_to_alph:n function is defined as

\cs_new:Npn \int_to_alph:n #1

{ \int_to_symbols:nnn \{ #1 \} \{ 26 \} 
  { 1 } \{ a \} 
  { 2 } \{ b \} 
    ... 
  { 26 } \{ z \} 
}

175
\int_to_bin:n \{ \text{int expr} \}

Calculates the value of the \text{int expr} and places the binary representation of the result in the input stream.

\int_to_hex:n \int_to_Hex:n

\text{New: 2014-02-11}

Calculates the value of the \text{int expr} and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for \text{int_to_hex:n} and upper case ones for \text{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

\text{New: 2014-02-11}

Calculates the value of the \text{int expr} and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_base:nn \int_to_Base:nn

\text{Updated: 2014-02-11}

Calculates the value of the \text{int expr} and converts it into the appropriate representation in the \text{base}; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for \text{int_to_base:n} and upper case ones for \text{int_to_Base:n}. The maximum \text{base} value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\text{Texpackers note: This is a generic version of \text{int_to_bin:n}, etc.}

\int_to_roman:n \int_to_Roman:n

\text{Updated: 2011-10-22}

Places the value of the \text{int expr} in the input stream as Roman numerals, either lower case \text{int_to_roman:n} or upper case \text{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 \text{mdclxvi}, repeated as needed: the notation with bars (such as \bar{v} for 5000) is not used. For instance \text{int_to_roman:n \{ 8249 \}} expands to \text{mmmmmmmccxlix}.

21.9 Converting from other formats to integers

\int_from_alph:n \int_from_alph:n \{ \text{letters} \}

\text{Updated: 2014-08-26}

Converts the \text{letters} into the integer (base 10) representation and leaves this in the input stream. The \text{letters} are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of + and -. This is the inverse function of \text{int_to_alph:n} and \text{int_to_Alph:n}.

\int_from_bin:n \int_from_bin:n \{ \text{binary number} \}

\text{New: 2014-02-11 \ Updated: 2014-08-26}

Converts the \text{binary number} into the integer (base 10) representation and leaves this in the input stream. The \text{binary number} is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by binary digits. This is the inverse function of \text{int_to_bin:n}. 176
\int_from_hex:n \{(hexadecimal number)\}

Converts the \textit{hexadecimal number} into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the \textit{hexadecimal number} by upper or lower case letters. The \textit{hexadecimal number} is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and -. This is the inverse function of \int_to_hex:n and \int_to_Hex:n.

\int_from_oct:n \{(octal number)\}

Converts the \textit{octal number} into the integer (base 10) representation and leaves this in the input stream. The \textit{octal number} is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by octal digits. This is the inverse function of \int_to_oct:n.

\int_from_roman:n \{(roman numeral)\}

Converts the \textit{roman numeral} into the integer (base 10) representation and leaves this in the input stream. The \textit{roman numeral} is first converted to a string, with no expansion. The \textit{roman numeral} may be in upper or lower case; if the numeral contains characters besides \textit{mdclxvi} or \textit{MDCLXVI} then the resulting value is \textit{-1}. This is the inverse function of \int_to_roman:n and \int_to_Roman:n.

\int_from_base:nn \{\textit{number}\} \{\textit{base}\}

Converts the \textit{number} expressed in \textit{base} into the appropriate value in base 10. The \textit{number} is first converted to a string, with no expansion. The \textit{number} should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum \textit{base} value is 36. This is the inverse function of \int_to_base:nn and \int_to_Base:nn.

### 21.10 Random integers

\int_rand:nn \{(int expr1)\} \{(int expr2)\}

Evaluates the two \textit{int expr}s and produces a pseudo-random number between the two (with bounds included). This is not available in older versions of X\LaTeX.X.

\int_rand:n \{(int expr)\}

Evaluates the \textit{int expr} then produces a pseudo-random number between 1 and the \textit{int expr} (included). This is not available in older versions of X\LaTeX.X.

### 21.11 Viewing integers

\int_show:N \textit{integer}
\int_show:c

Displays the value of the \textit{integer} on the terminal.
\int_show:n \{\int expr\}
Displays the result of evaluating the \{\int expr\} on the terminal.
New: 2011-11-22
Updated: 2015-08-07

\int_log:N \int_log:c
\int_log:n
\{\int expr\}
Writes the value of the \{integer\} in the log file.
New: 2014-08-22
Updated: 2015-08-03

21.12 Constant integers

\c_zero_int \c_one_int
New: 2018-05-07
Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.

\c_max_int
The maximum value that can be stored as an integer.

\c_max_register_int
Maximum number of registers.

\c_max_char_int
Maximum character code completely supported by the engine.

21.13 Scratch integers

\l_tmpa_int \l_tmpb_int
Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any \LaTeXe-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 \LaTeXe-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

\texttt{\textbackslash int\_value:w}\texttt{(integer)}
\texttt{\textbackslash int\_value:w\ (integer\ denomination)\ (optional\ space)}

Expands the following tokens until an \texttt{(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 \texttt{(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 '\texttt{(octal digits)} or "\texttt{(hexadecimal digits)} or '{\texttt{(character)}}).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in \texttt{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

\texttt{\textbackslash if\_int\_compare:w}\texttt{\ (integer\_1)\ (relation)\ (integer\_2)}
\texttt{\ (true\ code)}
\texttt{\ else:}
\texttt{\ (false\ code)}
\texttt{\fi:}

Compare two integers using \texttt{(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}.

\texttt{\textbackslash if\_case:w}\texttt{\ (integer)}\texttt{\ (case\_0)}
\texttt{\ (case\_1)}
\texttt{\ ...}
\texttt{\ else: (default)}
\texttt{\fi:}

Selects a case to execute based on the value of the \texttt{(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 \texttt{(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 \{tokens\} \{optional space\}
\{true code\}
\else:
  \{true code\}
\fi:

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

\TeXhackers note: This is the \TeX primitive \ifodd.
Chapter 22

The l3flag package: 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 \textit{height}. In expansion-only contexts, a flag can only be “raised”: this increases the \textit{height} by 1. The \textit{height} 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 \textit{(flag name)} such as \texttt{str_missing}. The \textit{(flag name)} is used as part of \texttt{\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 \{\flag name\}
\end{verbatim}

Creates a new flag with a name given by \texttt{(flag name)}, or raises an error if the name is already taken. The \texttt{(flag name)} may not contain spaces. The declaration is global, but flags are always local variables. The \texttt{(flag)} initially has zero height.

\begin{verbatim}
\flag_clear:n \flag_clear:n \{\flag name\}
\end{verbatim}

The \texttt{(flag)}’s height is set to zero. The assignment is local.
\flag_clear_new:n \flag_clear_new:n \{\text{flag name}\}

Ensures that the \(\text{flag}\) exists globally by applying \flag_new:n if necessary, then applies \flag_clear:n, setting the height to zero locally.

\flag_show:n \flag_show:n \{(\text{flag name})\}

Displays the \(\text{flag}\)'s height in the terminal.

\flag_log:n \flag_log:n \{(\text{flag name})\}

Writes the \(\text{flag}\)'s height to the log file.

### 22.2 Expandable flag commands

\flag_if_exist_p:n \flag_if_exist_p:n \{\text{flag name}\}

\flag_if_exist:pTF \flag_if_exist:pTF \{\text{flag name}\} \{(\text{true code})\} \{(\text{false code})\}

This function returns \texttt{true} if the \texttt{\{flag name\}} references a flag that has been defined previously, and \texttt{false} otherwise.

\flag_if_raised_p:n \flag_if_raised_p:n \{\text{flag name}\}

\flag_if_raised:pTF \flag_if_raised:pTF \{\text{flag name}\} \{(\text{true code})\} \{(\text{false code})\}

This function returns \texttt{true} if the \texttt{\{flag\}} has non-zero height, and \texttt{false} if the \texttt{\{flag\}} has zero height.

\flag_height:n \flag_height:n \{\text{flag name}\}

Expands to the height of the \texttt{\{flag\}} as an integer denotation.

\flag_raise:n \flag_raise:n \{\text{flag name}\}

The \texttt{\{flag\}}'s height is increased by 1 locally.

\flag_ensure_raised:n \flag_ensure_raised:n \{(\text{flag name})\}

Ensures the \texttt{\{flag\}} is raised by making its height at least 1, locally.
Chapter 23

The l3clist package
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
occurred. Because comma list variables are token list variables, expanding them once yields their items separated by commas, and \l\_list\_functions such as \_tl\_show:N can be applied to them. (These functions often have \l\_list\_analogue, 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 \l\_seq\) may be advisable if speed is important. The exception is that \_clist\_if\_in:NnTF and \_clist\_remove\_duplicates:N may be faster than their sequence analogues for large lists. However, these functions work slowly for "unsafe" items that must be braced, and may produce errors when their argument contains \{,\} or \# (assuming the usual \TeX\ category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

### 23.1 Creating and initialising comma lists

```
\clist\_new:N \clist\_new:N \langle\list\ var\rangle
\clist\_new:c

Creates a new \langle\list\ var\rangle or raises an error if the name is already taken. The declaration is global. The \langle\list\ var\rangle initially contains no items.
```

```
\clist\_const:Nn \clist\_const:(N|e|c|n|ce)

\clist\_const:Nn \langle\list\ var\rangle \{\langle\comma\ list\}\}

Creates a new constant \langle\list\ var\rangle or raises an error if the name is already taken. The value of the \langle\list\ var\rangle is set globally to the \langle\comma\ list\⟩.
```

```
\clist\_clear:N \clist\_clear:c
\clist\_gclear:N \clist\_gclear:c

\clist\_clear:N \langle\list\ var\rangle
\clist\_clear:c
\clist\_gclear:N \langle\list\ var\rangle
\clist\_gclear:c

Clears all items from the \langle\list\ var\rangle.
```

```
\clist\_clear\_new:N \clist\_clear\_new:c
\clist\_gclear\_new:N \clist\_gclear\_new:c

\clist\_clear\_new:N \langle\list\ var\rangle
\clist\_clear\_new:c
\clist\_gclear\_new:N \langle\list\ var\rangle
\clist\_gclear\_new:c

Ensures that the \langle\list\ var\rangle exists globally by applying \clist\_new:N if necessary, then applies \clist\_(g)clear:N to leave the list empty.
```

```
\clist\_set\_eq:NN \clist\_set\_eq:(cN|Nc|cc)
\clist\_gset\_eq:NN \clist\_gset\_eq:(cN|Nc|cc)

\clist\_set\_eq:NN \langle\comma\ list\_1\rangle \langle\comma\ list\_2\rangle
\clist\_gset\_eq:NN \langle\comma\ list\_1\rangle \langle\comma\ list\_2\rangle

Sets the content of \langle\comma\ list\_1\rangle equal to that of \langle\comma\ list\_2\rangle. To set a token list variable equal to a comma list variable, use \_tl\_set\_eq:NN. Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.
```

```
\clist\_set\_from\_seq:NN \clist\_set\_from\_seq:(cN|Nc|cc)
\clist\_gset\_from\_seq:NN \clist\_gset\_from\_seq:(cN|Nc|cc)

\clist\_set\_from\_seq:NN \langle\list\ var\rangle \langle\sequence\rangle
\clist\_gset\_from\_seq:NN \langle\list\ var\rangle \langle\sequence\rangle

Converts the data in the \langle\sequence\rangle into a \langle\list\ var\rangle; the original \langle\sequence\rangle is unchanged. Items which contain either spaces or commas are surrounded by braces.
```

New: 2014-07-05

New: 2014-07-17
\clist_concat:NNN \clist_concat:ccc \clist_gconcat:NNN \clist_gconcat:ccc
Concatenates the content of (\textit{comma list}_2) and (\textit{comma list}_3) together and saves the result in (\textit{comma list}_1). The items in (\textit{comma list}_2) are placed at the left side of the new comma list.

\clist_if_exist_p:N \clist_if_exist:NTF \clist_gconcat:ccc
Tests whether the (\textit{clist var}) is currently defined. This does not check that the (\textit{clist var}) really is a comma list.

New: 2012-03-03

23.2 Adding data to comma lists

\clist_set:Nn \clist_set:Nn \clist_set:NV \clist_set:Ne \clist_set:No \clist_set:cn \clist_set:cV \clist_set:ce \clist_set:co
Sets (\textit{clist var}) to contain the (\textit{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 (\textit{tokens}) as a single (\textit{item}) even if the (\textit{tokens}) contain commas or spaces, add a set of braces: \clist_set:Nn (\textit{clist var}) \{(\textit{tokens})\}.

\clist_put_left:Nn \clist_put_left:Nn \clist_put_left:NV \clist_put_left:Ne \clist_put_left:No \clist_put_left:cn \clist_put_left:cV \clist_put_left:ce \clist_put_left:co
Updated: 2011-09-05
Appends the (\textit{items}) to the left of the (\textit{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 (\textit{tokens}) as a single (\textit{item}) even if the (\textit{tokens}) contain commas or spaces, add a set of braces: \clist_put_left:Nn (\textit{clist var}) \{(\textit{tokens})\}.

\clist_put_right:Nn \clist_put_right:Nn \clist_put_right:NV \clist_put_right:Ne \clist_put_right:No \clist_put_right:cn \clist_put_right:cV \clist_put_right:ce \clist_put_right:co
Updated: 2011-09-05
Appends the (\textit{items}) to the right of the (\textit{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 (\textit{tokens}) as a single (\textit{item}) even if the (\textit{tokens}) contain commas or spaces, add a set of braces: \clist_put_right:Nn (\textit{clist var}) \{(\textit{tokens})\}. 

185
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_remove_duplicates:c \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_remove_all:(cnNVcV) \clist_gremove_all:Nn \clist_gremove_all:(cnNVcV)

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: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 {\langle comma list \rangle}

New: 2014-07-18

Leaves the items in the \langle comma list \rangle in the input stream in reverse order. Contrarily to other what is done for other n-type \langle comma list \rangle arguments, braces and spaces are preserved by this process.

\TeXhackers note: The result is returned within \unexpanded, which means that the comma list does not expand further when appearing in an e-type or x-type argument expansion.
\clist_sort:Nn \clist_sort:cn \clist_gsort:Nn \clist_gsort:cn
\flushright Rev: 2017-02-06

\clist_sort:Nn \clist_sort:cn \clist_gsort:Nn \clist_gsort:cn
\flushright New: 2017-02-06

23.4 Comma list conditionals

\clist_if_empty_p:N \clist_if_empty:p:c \clist_if_empty:NTF \clist_if_empty:c
\flushright Rev: 2014-07-05

\clist_if_empty_p:n \clist_if_empty:n \clist_if_empty:nTF \clist_if_empty:n c
\flushright New: 2011-09-06

\clist_if_in:NnTF \clist_if_in:NV \clist_if_in:nnTF
\flushright Updated: 2011-09-06

23.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, i.e. any local assignments made by the (function) or (code) discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is \{a,\{b\},c\}, then the arguments passed to the mapped function are 'a', 'b', \{c\}, an empty argument, and 'c'.

TEXhackers note: The function may fail if the (item) contains {, }, or # (assuming the usual \TeX{} category codes apply).
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.

\begin{verbatim}
\clist_map_function:NN \clist_map_function:nN \clist_map_function:en \clist_map_function:cn \clist_map_function:nn \clist_map_function:ch
\end{verbatim}

\texttt{\clist_map_function:NN} \texttt{\clist_map_function:nN} \texttt{\clist_map_function:en} \texttt{\clist_map_function:cn} \texttt{\clist_map_function:nn}

\begin{verbatim}
\clist_map_function:NN \clist_map_function:nN \clist_map_function:en \clist_map_function:cn \clist_map_function:nn
\end{verbatim}

\texttt{\clist_map_function:NN} \texttt{\clist_map_function:nN} \texttt{\clist_map_function:en} \texttt{\clist_map_function:cn} \texttt{\clist_map_function:nn}

\begin{verbatim}
\clist_map_variable:NNn \clist_map_variable:cn \clist_map_variable:nn \clist_map_variable:nNn \clist_map_variable:cn \clist_map_variable:nNn
\end{verbatim}

\texttt{\clist_map_variable:NNn} \texttt{\clist_map_variable:cn} \texttt{\clist_map_variable:nn} \texttt{\clist_map_variable:nNn} \texttt{\clist_map_variable:cn} \texttt{\clist_map_variable:nNn}

\begin{verbatim}
\clist_map_tokens:Nn \clist_map_tokens:nn \clist_map_tokens:cn \clist_map_tokens:nn
\end{verbatim}

\texttt{\clist_map_tokens:Nn} \texttt{\clist_map_tokens:nn} \texttt{\clist_map_tokens:cn} \texttt{\clist_map_tokens:nn}

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

\texttt{\clist_map_break:} \texttt{\clist_map_break:}

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

\texttt{\clist_map_break:}

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

\textbf{\TeXhackers note:} When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.
\clist_map_break:n \clist_map_break:n \{\text{code}\}

Updated: 2012-06-29

Used to terminate a \clist_map\ldots function before all entries in the \textlangle\text{comma list}\textrangle have been processed, inserting the \textlangle\text{code}\textrangle 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 \{ \text{code} \} }{ % Do something useful \}}

Use outside of a \clist_map\ldots scenario leads to low level TeX errors.

\textsc{Texhacker note:} When the mapping is broken, additional tokens may be inserted before the \textlangle\text{code}\textrangle is inserted into the input stream. This depends on the design of the mapping function.

\clist_count:N \clist_count:N \clist_count:c \clist_count:n \clist_count:e

\textsc{New: 2012-07-13}

Leaves the number of items in the \textlangle\text{clist var}\textrangle in the input stream as an \textlangle\text{integer denotation}\textrangle. The total number of items in a \textlangle\text{clist var}\textrangle includes those which are duplicates, \textit{i.e.} every item in a \textlangle\text{clist var}\textrangle is counted.
23.6 Using the content of comma lists directly

\clist_use:Nnnn \clist_use:Nnnn \clist var \{\langle separator between two\rangle\}
\clist_use:cn \clist_use:cn \{\langle separator between more than two\rangle\} \{\langle separator between final two\rangle\}

Places the contents of the \clist var in the input stream, with the appropriate \langle separator \rangle between the items. Namely, if the comma list has more than two items, the \langle separator between more than two\rangle is placed between each pair of items except the last, for which the \langle separator between final two\rangle is used. If the comma list has exactly two items, then they are placed in the input stream separated by the \langle separator between two\rangle. 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.

\texttt{\TeXhackers note:} The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \langle items \rangle do not expand further when appearing in an e-type or x-type argument expansion.

\clist_use:Nn \clist_use:Nn \clist var \{\langle separator\rangle\}
\clist_use:cn \clist_use:cn

Places the contents of the \clist var in the input stream, with the \langle separator \rangle 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- } { ,- } { ,-and- }

inserts “a and b and c and de and f” in the input stream.

\texttt{\TeXhackers note:} The result is returned within the \texttt{\unexpanded} primitive (\texttt{\exp_not:n}), which means that the \langle items \rangle do not expand further when appearing in an e-type or x-type argument expansion.
\list_use:nnnn \list_use:nn {(comma list) \{\textit{separator between two}\}}
\list_use:nn {(comma list) \{\textit{separator between more than two}\}}
\list_use:nn {(comma list) \{\textit{separator between final two}\}}
\list_use:nn \{\textit{comma list}\} \{\textit{separator}\}

Places the contents of the \textit{(comma list)} in the input stream, with the appropriate \textit{(separator)} between the items. As for \clist_set:Nn, blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. The \textit{\{separators\}} are then inserted in the same way as for \clist_use:Nnnn and \clist_use:Nn, respectively.

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

### 23.7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

\clist_get:NN \clist_get:cN \clist_get:NN \clist_get:cN
\clist_pop:NN \clist_pop:cN \clist_gpop:NN \clist_gpop:cN
\clist_pop:NNTF

Stores the left-most item from a \textit{(clist var)} in the \textit{(token list variable)} without removing it from the \textit{(clist var)}. The \textit{(token list variable)} is assigned locally. In the non-branching version, if the \textit{(clist var)} is empty the \textit{(token list variable)} is set to the marker value \q_no_value.

Pops the left-most item from a \textit{(clist var)} into the \textit{(token list variable)}, \textit{i.e.} removes the item from the comma list and stores it in the \textit{(token list variable)}. Both of the variables are assigned locally.

Pops the left-most item from a \textit{(clist var)} into the \textit{(token list variable)}, \textit{i.e.} removes the item from the comma list and stores it in the \textit{(token list variable)}. The \textit{(clist var)} is modified globally, while the assignment of the \textit{(token list variable)} is local.

If the \textit{(clist var)} is empty, 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{(clist var)} is non-empty, pops the top item from the \textit{(clist var)} in the \textit{(token list variable)}, \textit{i.e.} removes the item from the \textit{(clist var)}. Both the \textit{(clist var)} and the \textit{(token list variable)} are assigned locally.

191
If the \clist 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 \clist var is non-empty, pops the top item from the \clist var in the \token list variable, i.e. removes the item from the \clist var. The \clist var 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)

Adds the \{items\} to the top of the \clist var. Spaces are removed from both sides of each item as for any n-type comma list.

\section{23.8 Using a single item}

\clist_item:Nn * \clist_item:N (clist var) \{int expr\}
\clist_item:cn * \clist_item:cn (clist var) \{int expr\}
\clist_item:nn * \clist_item:nn (clist var) \{int expr\}
\clist_item:en * \clist_item:en (clist var) \{int expr\}

Indexing items in the \clist var from 1 at the top (left), this function evaluates the \int expr and leaves the appropriate item from the comma list in the input stream. If the \int expr is negative, indexing occurs from the bottom (right) of the comma list. When the \int expr is larger than the number of items in the \clist var (as calculated by \clist_count:N) then the function expands to nothing.

\TeXhacks{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:N (clist var) \clist_rand_item:c (clist var) \clist_rand_item:n (clist var) \clist_rand_item:cn (clist var) \clist_rand_item:nn (clist var) \clist_rand_item:en (clist var)

Selects a pseudo-random item of the \clist var/\comma list. If the \comma list has no item, the result is empty.

\TeXhacks{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.}

\section{23.9 Viewing comma lists}

\clist_show:N \clist_show:N (clist var)
\clist_show:n \clist_show:n (clist var)
\clist_show:c \clist_show:c (clist var)

Displays the entries in the \clist var in the terminal.

Updated: 2021-04-29
\clist_show:n  \clist_show:n \{\text{tokens}\}
Updated: 2013-08-03
Displays the entries in the comma list in the terminal.

\clist_log:N  \clist_log:N \{\text{clist var}\}
\clist_log:c
New: 2014-08-22
Updated: 2021-04-29
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}\}
New: 2014-08-22
Writes the entries in the comma list in the log file. See also \clist_show:n which displays the result in the terminal.

23.10 Constant and scratch comma lists

\c_empty_clist
New: 2012-07-02
Constant that is always empty.

\l_tmpa_clist
\l_tmpb_clist
New: 2011-09-06
Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_clist
\g_tmpb_clist
New: 2011-09-06
Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
Chapter 24

The \texttt{l3token} package

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let’s try with a better description: When programming in \TeX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: \texttt{\texttt{token} } for anything that deals with tokens and \texttt{\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, \if_charcode:w, and \tex_if:D} are three names for the same internal operation of \TeX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, \TeX distinguishes them when searching for a delimited argument. Namely, the example function \texttt{\show_until_if: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.

\begin{verbatim}
cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
\end{verbatim}

A list of all possible shapes and a list of all possible meanings are given in section \ref{sec: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:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc

\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\. Another way to build token lists with unusual category codes is \regex_replace:nnN \{.*\} \{\{replacement\}\} \{tl var\}.

\textbf{\TeXhacks not}: 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).
Token list containing one character with category code 12, ("other"), and character code 32 (space).

### 24.2 Manipulating and interrogating character tokens

Sets the category code of the `⟨character⟩` to that indicated in the function name. Depending on the current category code of the `⟨token⟩` the escape token may also be needed:

\[
\text{\textbackslash char_set_catcode_other:N} \ %
\]

The assignment is local.
Sets the category code of the `<character>` which has character code as given by the `⟨integer expression⟩`. This version can be used to set up characters which cannot otherwise be given (cf. the N-type variants). The assignment is local.

These functions set the category code of the `<character>` which has character code as given by the `⟨integer expression⟩`. The first `⟨integer expression⟩` is the character code and the second is the category code to apply. The setting applies within the current TeX group. In general, the symbolic functions `\char_set_catcode_⟨type⟩` should be preferred, but there are cases where these lower-level functions may be useful.

Expands to the current category code of the `<character>` with character code given by the `⟨integer expression⟩`.

Displays the current category code of the `<character>` with character code given by the `⟨integer expression⟩` on the terminal.

Sets up the behaviour of the `<character>` when found inside `\text_lowercase:n`, such that `<character₁>` will be converted into `<character₂>`. The two `<characters>` may be specified using an `⟨integer expression⟩` for the character code concerned. This may include the TeX `⟨character⟩` method for converting a single character into its character code:

\begin{verbatim}
\char_set_lccode:nn { \textbackslash A } { \textbackslash a } \ % Standard behaviour
\char_set_lccode:nn { \textbackslash A } { \textbackslash a + 32 }
\char_set_lccode:nn { 50 } { 60 }
\end{verbatim}

The setting applies within the current TeX group.
\char_value_lccode:n \{integer expression\}

Expands to the current lower case code of the \textit{character} with character code given by the \textit{integer expression}.

\char_show_value_lccode:n \{integer expression\}

Displays the current lower case code of the \textit{character} with character code given by the \textit{integer expression} on the terminal.

\char_set_uccode:nn \{int expr\} \{int expr\}

Sets up the behaviour of the \textit{character} when found inside \texttt{text.uppercase:n}, such that \texttt{\textbackslash character}_1 will be converted into \texttt{\textbackslash character}_2. The two \texttt{\textbackslash character}s may be specified using an \textit{integer expression} for the character code concerned. This may include the \TeX\ '\texttt{\textbackslash character}' method for converting a single character into its character code:

\begin{verbatim}
\char_set_uccode:nn { \texttt{\char\char} } { \texttt{\char\char - 32} }
\char_set_uccode:nn { 60 } { 50 }
\end{verbatim}

The setting applies within the current \TeX\ group.

\char_value_uccode:n \{integer expression\}

Expands to the current upper case code of the \textit{character} with character code given by the \textit{integer expression}.

\char_show_value_uccode:n \{integer expression\}

Displays the current upper case code of the \textit{character} with character code given by the \textit{integer expression} on the terminal.

\char_set_mathcode:nn \{int expr\} \{int expr\}

This function sets up the math code of \textit{character}. The \texttt{\textbackslash character} is specified as an \textit{integer expression} which will be used as the character code of the relevant character. The setting applies within the current \TeX\ group.

\char_value_mathcode:n \{integer expression\}

Expands to the current math code of the \textit{character} with character code given by the \textit{integer expression}.

\char_show_value_mathcode:n \{integer expression\}

Displays the current math code of the \textit{character} with character code given by the \textit{integer expression} on the terminal.

\char_set_sfcode:nn \{int expr\} \{int expr\}

This function sets up the space factor for the \textit{character}. The \textit{character} is specified as an \textit{integer expression} which will be used as the character code of the relevant character. The setting applies within the current \TeX\ group.

198
\char_value_sfcode:n \char_value_sfcode:n \{\textit{integer expression}\}

Expands to the current space factor for the \textit{character} with character code given by the \textit{integer expression}.

\char_show_value_sfcode:n \char_show_value_sfcode:n \{\textit{integer expression}\}

Displays the current space factor for the \textit{character} with character code given by the \textit{integer expression} on the terminal.

\l_char_active_seq

Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category \textit{active} (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

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 \textit{letter} (catcode 11) or \textit{other} (catcode 12). Each entry in the sequence consists of a single escaped token, for example \textbackslash{} for the backslash or \{ for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

### 24.3 Generic tokens

\c_group_begin_token
\c_group_end_token
\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

\c_catcode_letter_token
\c_catcode_other_token

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

\c_catcode_active_tl

A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.
24.4 Converting tokens

\token_to_meaning:N \token_to_meaning:N \token\token
Inserts the current meaning of the \token into the input stream as a series of characters of category code 12 (other). This is the primitive \TeX description of the \token, thus for example both functions defined by \cs_set_nopar:Npn and token list variables defined using \tl_new:N are described as macros.

\TeXhackers\note: This is the \TeX primitive \meaning. The \token can thus be an explicit space token or an explicit begin-group or end-group character token (\{ or \} when normal \TeX category codes apply) even though these are not valid \hookleft\-type arguments.

\token_to_str:N \token_to_str:N \token\token
Converts the given \token into a series of characters with category code 12 (other). If the \token is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the \token). This function requires only a single expansion.

\TeXhackers\note: \token_to_str:N is the \TeX primitive \string. The \token can thus be an explicit space tokens or an explicit begin-group or end-group character token (\{ or \} when normal \TeX category codes apply) even though these are not valid \hookleft\-type arguments.

\token_to_catcode:N \token_to_catcode:N \token\token
Converts the given \token into a number describing its category code. If \token is a control sequence this expands to 16. This can’t detect the categories 0 (escape character), 5 (end of line), 9 (ignored character), 14 (comment character), or 15 (invalid character). Control sequences or active characters let to a token of one of the detectable category codes will yield that category.

24.5 Token conditionals

\token_if_group_begin_p:N \token_if_group_begin_p:N \token\token
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 \hookleft\-type argument.

\token_if_group_end_p:N \token_if_group_end_p:N \token\token
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 \hookleft\-type argument.
\texttt{\textbackslash token\_if\_math\_toggle\_p:N = \textbackslash token\_if\_math\_toggle\_p:N (token)}
\texttt{\textbackslash token\_if\_math\_toggle\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of a math shift token ($ when normal \TeX{} category codes are in force).

\texttt{\textbackslash token\_if\_alignment\_p:N = \textbackslash token\_if\_alignment\_p:N (token)}
\texttt{\textbackslash token\_if\_alignment\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of an alignment token (& when normal \TeX{} category codes are in force).

\texttt{\textbackslash token\_if\_parameter\_p:N = \textbackslash token\_if\_parameter\_p:N (token)}
\texttt{\textbackslash token\_if\_parameter\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of a macro parameter token (# when normal \TeX{} category codes are in force).

\texttt{\textbackslash token\_if\_math\_superscript\_p:N = \textbackslash token\_if\_math\_superscript\_p:N (token)}
\texttt{\textbackslash token\_if\_math\_superscript\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of a superscript token (^ when normal \TeX{} category codes are in force).

\texttt{\textbackslash token\_if\_math\_subscript\_p:N = \textbackslash token\_if\_math\_subscript\_p:N (token)}
\texttt{\textbackslash token\_if\_math\_subscript\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of a subscript token (\_ when normal \TeX{} category codes are in force).

\texttt{\textbackslash token\_if\_space\_p:N = \textbackslash token\_if\_space\_p:N (token)}
\texttt{\textbackslash token\_if\_space\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid \texttt{\textit{N}}-type argument.

\texttt{\textbackslash token\_if\_letter\_p:N = \textbackslash token\_if\_letter\_p:N (token)}
\texttt{\textbackslash token\_if\_letter\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of a letter token.

\texttt{\textbackslash token\_if\_other\_p:N = \textbackslash token\_if\_other\_p:N (token)}
\texttt{\textbackslash token\_if\_other\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of an “other” token.

\texttt{\textbackslash token\_if\_active\_p:N = \textbackslash token\_if\_active\_p:N (token)}
\texttt{\textbackslash token\_if\_active\_p:N (token) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if \texttt{(token)} has the category code of an active character.

\texttt{\textbackslash token\_if\_eq\_catcode\_p:NN = \textbackslash token\_if\_eq\_catcode\_p:NN (token\textsubscript{1}) (token\textsubscript{2})}
\texttt{\textbackslash token\_if\_eq\_catcode\_p:NN (token\textsubscript{1}) (token\textsubscript{2}) \{\textbackslash\textit{true code}\} \{\textbackslash\textit{false code}\}}

Tests if the two \texttt{(tokens)} have the same category code.
\token_if_eq_charcode_p:NN \token_if_eq_charcode_p:NN \token_if_eq_charcode:p\token_if_eq_charcode:NNTF \token_if_eq_charcode:p\token_if_eq_charcode:NNTF \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the two \textit{tokens} have the same character code.

\token_if_eq_meaning_p:NN \token_if_eq_meaning_p:NN \token_if_eq_meaning:p\token_if_eq_meaning:NNTF \token_if_eq_meaning:p\token_if_eq_meaning:NNTF \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the two \textit{tokens} have the same meaning when expanded.

\token_if_macro_p:N \token_if_macro_p:N \token_if_macro:p\token_if_macro:NTF \token_if_macro:p\token_if_macro:NTF \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the \texttt{token} is a \TeX{} macro.

Updated: 2011-05-23

\token_if_cs_p:N \token_if_cs_p:N \token_if_cs:p\token_if_cs:NTF \token_if_cs:p\token_if_cs:NTF \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the \texttt{token} is a control sequence.

Updated: 2012-01-20

\token_if-expandable_p:N \token_if-expandable_p:N \token_if-expandable:p\token_if-expandable:NTF \token_if-expandable:p\token_if-expandable:NTF \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the \texttt{token} is expandable. This test returns \texttt{false} for an undefined token.

Updated: 2012-01-20

\token_if_long_macro_p:N \token_if_long_macro_p:N \token_if_long_macro:p\token_if_long_macro:NTF \token_if_long_macro:p\token_if_long_macro:NTF \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the \texttt{token} is a long macro.

Updated: 2012-01-20

\token_if_protected_macro_p:N \token_if_protected_macro_p:N \token_if_protected_macro:p\token_if_protected_macro:NTF \token_if_protected_macro:p\token_if_protected_macro:NTF \{\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}.

Updated: 2012-01-20

\token_if_protected_long_macro_p:N \token_if_protected_long_macro_p:N \token_if_protected_long_macro:p\token_if_protected_long_macro:NTF \token_if_protected_long_macro:p\token_if_protected_long_macro:NTF \{\texttt{true code}\} \{\texttt{false code}\}

Tests if the \texttt{token} is a protected long macro.

Updated: 2012-01-20

\token_if_chardef_p:N \token_if_chardef_p:N \token_if_chardef:p\token_if_chardef:NTF \token_if_chardef:p\token_if_chardef:NTF \{\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 \verb|\chardefs|.
Tests if the \textit{token} is defined to be a \texttt{mathchardef}.

 Tests if the \textit{token} is defined to be a \texttt{font selection} command.

 Tests if the \textit{token} is defined to be a \texttt{dim} register.

 Tests if the \textit{token} is defined to be a \texttt{int} register.

 TexHackers note: Constant integers may be implemented as integer registers, \texttt{chardefs}, or \texttt{mathchardefs} depending on their value.

 Tests if the \textit{token} is defined to be a \texttt{muskip} register.

 Tests if the \textit{token} is defined to be a \texttt{skip} register.

 Tests if the \textit{token} is defined to be a \texttt{toks} register (not used by \LaTeXX).

 Tests if the \textit{token} is an engine primitive. In \LaTeX this includes primitive-like commands defined using \texttt{token.set_lua}.
This function compares the ⟨test token⟩ in turn with each of the ⟨token cases⟩. If the two are equal (as described for \token_if_eq_catcode:NNTF, \token_if_eq_charcode:NNTF and \token_if_eq_meaning:NNTF, respectively) then the associated ⟨code⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨true code⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨false code⟩ is inserted. The functions \token_case_catcode:Nn, \token_case_charcode:Nn, and \token_case_meaning:Nn, which do nothing if there is no match, are also available.

### 24.6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic \peek_after:Nw is provided along with a family of predefined tests for common cases. As peeking ahead does not skip spaces the predefined tests include both a space-respecting and space-skipping version. In addition, using \peek_analysis_map_inline:n, one can map through the following tokens in the input stream and repeatedly perform some tests.

\peek_after:Nw \peek_after:Nw \token (function) \token

Locally sets the test variable \l_peek_token equal to \token (as an implicit token, not as a token list), and then expands the \token. The \token remains in the input stream as the next item after the \token. The \token here may be \, \{ or \} (assuming normal \TeX\ category codes), i.e. it is not necessarily the next argument which would be grabbed by a normal function.

\peek_gafter:Nw \peek_gafter:Nw \token (function) \token

Globally sets the test variable \g_peek_token equal to \token (as an implicit token, not as a token list), and then expands the \token. The \token remains in the input stream as the next item after the \token. The \token here may be \, \{ or \} (assuming normal \TeX\ category codes), i.e. it is not necessarily the next argument which would be grabbed by a normal function.

\l_peek_token Token set by \peek_after:Nw and available for testing as described above.

\g_peek_token Token set by \peek_gafter:Nw and available for testing as described above.
\peek_catcode:N \peek_catcode:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}
Updated: 2012-12-20
Tests if the next ⟨token⟩ in the input stream has the same category code as the ⟨test token⟩ (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the ⟨token⟩ is left in the input stream after the ⟨true code⟩ or ⟨false code⟩ (as appropriate to the result of the test).

\peek_catcode_remove:N \peek_catcode_remove:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}
Updated: 2012-12-20
Tests if the next ⟨token⟩ in the input stream has the same category code as the ⟨test token⟩ (as defined by the test \token_if_eq_catcode:NNTF). Spaces are respected by the test and the ⟨token⟩ is removed from the input stream if the test is true. The function then places either the ⟨true code⟩ or ⟨false code⟩ in the input stream (as appropriate to the result of the test).

\peek_charcode:N \peek_charcode:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}
Updated: 2012-12-20
Tests if the next ⟨token⟩ in the input stream has the same character code as the ⟨test token⟩ (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the ⟨token⟩ is left in the input stream after the ⟨true code⟩ or ⟨false code⟩ (as appropriate to the result of the test).

\peek_charcode_remove:N \peek_charcode_remove:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}
Updated: 2012-12-20
Tests if the next ⟨token⟩ in the input stream has the same character code as the ⟨test token⟩ (as defined by the test \token_if_eq_charcode:NNTF). Spaces are respected by the test and the ⟨token⟩ is removed from the input stream if the test is true. The function then places either the ⟨true code⟩ or ⟨false code⟩ in the input stream (as appropriate to the result of the test).

\peek_meaning:N \peek_meaning:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}
Updated: 2011-07-02
Tests if the next ⟨token⟩ in the input stream has the same meaning as the ⟨test token⟩ (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the ⟨token⟩ is left in the input stream after the ⟨true code⟩ or ⟨false code⟩ (as appropriate to the result of the test).

\peek_meaning_remove:N \peek_meaning_remove:NTF ⟨test token⟩ {⟨true code⟩} {⟨false code⟩}
Updated: 2011-07-02
Tests if the next ⟨token⟩ in the input stream has the same meaning as the ⟨test token⟩ (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the ⟨token⟩ is removed from the input stream if the test is true. The function then places either the ⟨true code⟩ or ⟨false code⟩ in the input stream (as appropriate to the result of the test).

\peek_remove_spaces:n \peek_remove_spaces:n ⟨code⟩
New: 2018-10-01
Peeks ahead and detect if the following token is a space (category code 10 and character code 32). If so, removes the token and checks the next token. Once a non-space token is found, the ⟨code⟩ will be inserted into the input stream. Typically this will contain a peek operation, but this is not required.
\peek_remove_filler:n \peek_remove_filler:n \{\langle\text{code}\rangle\}

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to \texttt{\textbackslash scan_stop}. If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using \texttt{f-type} expansion, then checks the resulting leading token in the same way. If after expansion the next token is neither of the two test cases, the \langle\text{code}\rangle will be inserted into the input stream. Typically this will contain a \texttt{\textbackslash peek} operation, but this is not required.

\textbf{\TeX\ hackers note:} This is essentially a macro-based implementation of how \TeX\ handles the search for a left brace after for example \texttt{\textbackslash everypar}, except that any non-expandable token cleanly ends the \langle\text{filler}\rangle (i.e. it does not lead to a \TeX\ error).

In contrast to \TeX\ ’s filler removal, a construct \texttt{\textbackslash exp_not:N \textbackslash foo} will be treated in the same way as \texttt{\textbackslash 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 \texttt{\LaTeX3}) 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{\textbackslash 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).
\peek_analysis_map_inline:n \peek_analysis_map_inline:n {\langle inline function\rangle}

Repeatedly removes one \langle token\rangle from the input stream and applies the \langle inline function\rangle to it, until \peek_analysis_map_break: is called. The \langle inline function\rangle receives three arguments for each \langle token\rangle in the input stream:

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

These arguments are the same as for \tl_analysis_map_inline:nn defined in \l3tl-analysis. The \langle char code\rangle and \langle catcode\rangle do not take the meaning of a control sequence or active character into account: for instance, upon encountering the token \texttt{c_group_-begin_token} in the input stream, \peek_analysis_map_inline:n calls the \langle inline function\rangle with \#1 being \exp_not:n { \texttt{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 \langle inline function\rangle is called with arguments \exp_after:wN { \texttt{\if\false: } } \texttt{\fi}: \texttt{123} and \texttt{1}.

The mapping is done at the current group level, \textit{i.e.} any local assignments made by the \langle inline function\rangle remain in effect after the loop. Within the code, \tl\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 {\langle code\rangle} }

Stops the \peek_analysis_map_inline:n loop from seeking more tokens, and inserts \langle code\rangle in the input stream (empty for \peek_analysis_map_break:).
\peek_regex:nTF \peek_regex:nTF {⟨regex⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨tokens⟩ that follow in the input stream match the ⟨regular expression⟩. Any ⟨tokens⟩ that have been read are left in the input stream after the ⟨true code⟩ or ⟨false code⟩ (as appropriate to the result of the test). See 3\texttt{regex} for documentation of the syntax of regular expressions. The ⟨regular expression⟩ is implicitly anchored at the start, so for instance \peek_regex:nTF { a } is essentially equivalent to \peek_\texttt{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_\texttt{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 {⟨regex⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨tokens⟩ that follow in the input stream match the ⟨regex⟩. If the test is true, the ⟨tokens⟩ are removed from the input stream and the ⟨true code⟩ is inserted, while if the test is false, the ⟨false code⟩ is inserted followed by the ⟨tokens⟩ that were originally in the input stream. See 3\texttt{regex} for documentation of the syntax of regular expressions. The ⟨regular expression⟩ is implicitly anchored at the start, so for instance \peek_regex_remove_once:nTF { a } is essentially equivalent to \peek_\texttt{charcode}: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_\texttt{charcode}:NTF) only take into account their meaning.
If the \langle tokens \rangle that follow in the input stream match the \langle regex \rangle, replaces them according to the \langle replacement \rangle as for \regex_replace_once:nnN, and leaves the result in the input stream, after the \{true code\}. Otherwise, leaves \{false code\} followed by the \langle tokens \rangle that were originally in the input stream, with no modifications. See \l3regex for documentation of the syntax of regular expressions and of the \langle replacement \rangle: for instance \0 in the \langle replacement \rangle is replaced by the tokens that were matched in the input stream. The \langle regular expression \rangle is implicitly anchored at the start. In contrast to \regex_replace_once:nnN, no error arises if the \langle replacement \rangle leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

\TeX hacklers note: Implicit character tokens are correctly considered by \peek_regex_replace_once:nnTF as control sequences, while functions that inspect individual tokens (for instance \peek_charcode:NTF) only take into account their meaning.

24.7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on \TeX’s state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, \use:n is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for Lua\TeX and X\TeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

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

- Expanding \the:font results in a token that looks identical to the command that was used to select the current font (such as \tenrm) but it differs from it in shape.
- A “frozen” \relax, which differs from the primitive in shape (but has the same meaning), is inserted when the closing \fi of a conditional is encountered before the conditional is evaluated.
- Expanding \noexpand (\token) (when the \langle\token\rangle is expandable) results in an internal token, displayed (temporarily) as \notexpanded: (\token), whose shape coincides with the \langle\token\rangle and whose meaning differs from \relax.
• An \texttt{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 \texttt{endwrite}.

• Some frozen tokens can only be accessed in interactive sessions: \texttt{\textbackslash cr, \textbackslash right, \textbackslash endgroup, \textbackslash fi, \textbackslash inaccessible}.

• In Lua\TeX, there is also the strange case of “bytes” \texttt{\textbackslash{1100\textbackslash{xy}} where\textbackslash{xy}} are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from \texttt{\textbackslash{1100\textbackslash{00}}=\textbackslash{1114112}} to \texttt{\textbackslash{1100\textbackslash{ff}} \textbackslash{1114367}}. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (\textit{not characters}) whose meaning is the character followed by the given byte. If this byte is in the range 80–ff this gives an “invalid utf-8 sequence” error: applying \texttt{\textbackslash{token_to_str:N}} or \texttt{\textbackslash{token_to_meaning:N}} to these tokens is unsafe. Unfortunately, they don’t seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens \textit{explicit} character tokens. Category codes that a character token can have are listed below by giving a sample output of the \TeX{} primitive \texttt{\meaning{}}, together with their \LaTeX{} 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 several other category codes which do not lead to character tokens for later processing: 0 (escape), 5 (end line), 9 (ignore), 14 (comment), and 15 (invalid).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens \textit{implicit} character tokens. The meaning is otherwise in the following list:

• a macro, used in \LaTeX{} for most functions and some variables (\texttt{tl, fp, seq, ...}),
• a primitive such as \texttt{\textbackslash{def}} or \texttt{\topmark}, used in \LaTeX{} for some functions,
• a register such as \texttt{\textbackslash{count123}}, used in \LaTeX{} for the implementation of some variables (\texttt{int, dim, ...}).

210
• a constant integer such as \char^{56} or \mathchar^{121},
• a font selection command,
• undefined.

Macros can be \texttt{protected} or not, \texttt{long} or not (the opposite of what \LaTeX{}3 calls \texttt{nopar}), and \texttt{outer} or not (unused in \LaTeX{}3). Their \texttt{meaning} takes the form

\begin{center}
\texttt{\langle prefix \rangle \macro: \langle argument \rangle -> \langle replacement \rangle}
\end{center}

where \texttt{\langle prefix \rangle} is among \texttt{\protected\long\outer}, \texttt{\langle argument \rangle} describes parameters that the macro expects, such as \texttt{#1#2#3}, and \texttt{\langle replacement \rangle} describes how the parameters are manipulated, such as \texttt{\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).

\section{24.8 Deprecated functions}

\begin{itemize}
  \item \char_lowercase:N
  \item \char_uppercase:N
  \item \char_titlecase:N
  \item \char_foldcase:N
  \item \char_str_lowercase:N
  \item \char_str_uppercase:N
  \item \char_str_titlecase:N
  \item \char_str_foldcase:N
\end{itemize}

\begin{itemize}
  \item Converts the \texttt{\langle char \rangle} to the equivalent case-changed character as detailed by the function name (see \texttt{\str_foldcase:n} and \texttt{\text_titlecase_all:n} for details of these terms).
  \item The case mapping is carried out with no context-dependence (cf. \texttt{\text_uppercase:n}, \texttt{\text_compare:n}, \texttt{\text_uppercase:n}, etc.) The \texttt{\str} versions always generate “other” (category code 12) characters, whilst the standard versions generate characters with the category code of the \texttt{\langle char \rangle} (i.e. only the character code changes).
\end{itemize}

Rev: 2020-01-09
Chapter 25

The \l3prop package

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_=nnTF}.

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{itemize}
  \item \prop_new:N \prop_new:c \textit{property list}
    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 \prop_clear:N \prop_clear:c \prop_gclear:N \prop_gclear:c
    Clears all entries from the \textit{property list}.
  \item \prop_clear_new:N \prop_clear_new:c \prop_gclear_new:N \prop_gclear_new:c
    Ensures that the \textit{property list} exists globally by applying \prop_new:N if necessary, then applies \prop_(g)clear:N to leave the list empty.
\end{itemize}
\prop_set_eq:NN \prop_set_eq:(cN|N|cc) \prop_gset_eq:NN \prop_gset_eq:(cN|N|cc)

Sets the content of \textit{property list} 1 equal to that of \textit{property list} 2.

\prop_set_from_keyval:Nn \prop_set_from_keyval:cn \prop_gset_from_keyval:Nn \prop_gset_from_keyval:cn

\texttt{\prop_set_from_keyval:(property list)} \{ \langle \texttt{key1} \rangle = \langle \texttt{value1} \rangle, \langle \texttt{key2} \rangle = \langle \texttt{value2} \rangle, \ldots \} \texttt{\prop_set_from_keyval} 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 \texttt{key} and every \texttt{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 \texttt{key} and the \texttt{value} to contain spaces, commas or equal signs. The \texttt{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 (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{\prop_const_from_keyval:(property list)} \{ \langle \texttt{key1} \rangle = \langle \texttt{value1} \rangle, \langle \texttt{key2} \rangle = \langle \texttt{value2} \rangle, \ldots \} \texttt{\prop_const_from_keyval} creates a new constant \textit{property list} or raises an error if the name is already taken. The \textit{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 (e.g. those in \texttt{l3keys}), each key here \textit{must} be followed with an = sign.
25.2 Adding and updating property list entries

\prop_put:Nnn \prop_put: \prop_gput:Nnn

\prop_put_if_new:Nnn \prop_get:Nnn
\prop_gput_if_new:Nnn

\prop_concat:NNN \prop_gconcat:NNN

Adds an entry to the \texttt{property list} which may be accessed using the \texttt{key} and which has \texttt{value}. If the \texttt{key} is already present in the \texttt{property list}, the existing entry is overwritten by the new \texttt{value}. Both the \texttt{key} and \texttt{value} may contain any \texttt{balanced text}. The \texttt{key} is stored after processing with \texttt{\tl_to_str:n}, meaning that category codes are ignored.

\prop_concat:ccc \prop_gconcat:ccc

If the \texttt{key} is present in the \texttt{property list} then no action is taken. Otherwise, a new entry is added as described for \texttt{\prop_put:Nnn}.

Combines the key–value pairs of \texttt{property list} and \texttt{property list} and saves the result in \texttt{property list}. If a key appears in both \texttt{property list} and \texttt{property list} then the last value, namely the value in \texttt{property list} is kept.
Updates the \( \text{property list} \) by adding entries for each key–value pair given in the second argument. The addition is done through \prop_put:Nnn, hence if the \( \text{property list} \) 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 \( \text{property list} \) with the temporary variable using \prop_concat:NNN. In particular, the \( \text{keys} \) and \( \text{values} \) 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

Recovers the \( \text{value} \) stored with \( \text{key} \) from the \( \text{property list} \), and places this in the \( \text{token list variable} \). If the \( \text{key} \) is not found in the \( \text{property list} \) then the \( \text{token list variable} \) is set to the special marker \q_no_value. The \( \text{token list variable} \) is set within the current \TeX{} group. See also \prop_get:NnNTF.

Recovers the \( \text{value} \) stored with \( \text{key} \) from the \( \text{property list} \), and places this in the \( \text{token list variable} \). If the \( \text{key} \) is not found in the \( \text{property list} \) then the \( \text{token list variable} \) is set to the special marker \q_no_value. The \( \text{key} \) and \( \text{value} \) are then deleted from the property list. Both assignments are local. See also \prop_pop:NnNTF.

Recovers the \( \text{value} \) stored with \( \text{key} \) from the \( \text{property list} \), and places this in the \( \text{token list variable} \). If the \( \text{key} \) is not found in the \( \text{property list} \) then the \( \text{token list variable} \) is set to the special marker \q_no_value. The \( \text{key} \) and \( \text{value} \) are then deleted from the property list. The \( \text{property list} \) is modified globally, while the assignment of the \( \text{token list variable} \) is local. See also \prop_gpop:NnNTF.
\prop_item:Nn   \prop_item: (property list) \{ (key) \}
\prop_item: (NV|Ne|No|cn|cV|ce|co) *

Expects to find the \langle value\rangle corresponding to the \langle key\rangle in the \langle property list\rangle. If the \langle key\rangle is missing, this has an empty expansion.

\textbf{\LaTeX{}}hackers note: This function is slower than the non-expandable analogue \prop_get:N. The result is returned within the \unexpanded primitive (\exp_not:n), which means that the \langle value\rangle does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion.

\prop_count:N   \prop_count: (property list)\n\prop_count:c *

Leaves the number of key–value pairs in the \langle property list\rangle in the input stream as an \langle integer denotation\rangle.

\prop_to_keyval:N   \prop_to_keyval: (property list)\n\prop_to_keyval:c *

Expands to the \langle property list\rangle in a key–value notation. Keep in mind that a \langle property list\rangle is unordered, while key–value interfaces don’t necessarily are, so this can’t be used for arbitrary interfaces.

\textbf{\LaTeX{}}hackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the key–value list does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion. It also needs exactly two steps of expansion.

\section*{25.4 Modifying property lists}

\prop_remove:Nn   \prop_remove: (property list) \{ (key) \}
\prop_remove: (NV|Ne|cn|cV|ce)\n\prop_gremove:Nn\n\prop_gremove: (NV|Ne|cn|cV|ce) \hfill \texttt{Nov: 2012-05-12}

Removes the entry listed under \langle key\rangle from the \langle property list\rangle. If the \langle key\rangle is not found in the \langle property list\rangle no change occurs, \textit{i.e.} there is no need to test for the existence of a key before deleting it.

\section*{25.5 Property list conditionals}

\prop_if_exist_p:N   \prop_if_exist_p: (property list)\n\prop_if_exist_p:c * \prop_if_exist_p: (false code)\n\prop_if_exist:NTF   \prop_if_exist:NTF: (true code) \{ (false code) \}
\prop_if_exist:c * \hfill \texttt{Nov: 2012-03-03}

Tests whether the \langle property list\rangle is currently defined. This does not check that the \langle property list\rangle really is a property list variable.
Tests if the \textit{property list} is empty (containing no entries).

\begin{verbatim}
\prop_if_empty_p:N \prop_if_empty_p:c \prop_if_empty:N \prop_if_empty:c \prop_if_empty_p:N \prop_if_empty_p:c \prop_if_empty:N \prop_if_empty:c
\end{verbatim}

Tests if the \textit{key} is present in the \textit{property list}, making the comparison using the method described by \texttt{\str_if_eq:nnTF}.

\textbf{\LaTeX Xhacker note:} This function iterates through every key–value pair in the \textit{property list} and is therefore slower than using the non-expandable \texttt{\prop_get:NnNTF}.

\subsection*{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.

\begin{verbatim}
\prop_get:NnNTF \prop_get:NnN \prop_pop:NnNTF \prop_pop:NnN
\end{verbatim}

If the \textit{key} is not present in the \textit{property list}, leaves the \texttt{(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}, stores the corresponding \textit{value} in the \textit{token list variable} without removing it from the \textit{property list}, then leaves the \texttt{(true code)} in the input stream. The \textit{token list variable} is assigned locally.

\begin{verbatim}
\prop_get:NnN \prop_pop:NnN
\end{verbatim}

If the \textit{key} is not present in the \textit{property list}, leaves the \texttt{(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}, \textit{i.e.} removes the item from the \textit{property list}. Both the \textit{property list} and the \textit{token list variable} are assigned locally.

\clearpage

217
If the \texttt{key} is not present in the \texttt{property list}, 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{key} is present in the \texttt{property list}, pops the corresponding \texttt{value} in the \texttt{token list variable}, i.e. removes the item from the \texttt{property list}. The \texttt{property list} is modified globally, while the \texttt{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 \texttt{function} or \texttt{code} discussed below remain in effect after the loop.

\begin{verbatim}
\prop_map_function:NN \prop_map_function:cn
\end{verbatim}

\texttt{\prop_map_function:NN} Applies \texttt{\prop_map_function:NN} to every \texttt{entry} stored in the \texttt{property list}. The \texttt{function} receives two arguments for each iteration: the \texttt{key} and associated \texttt{value}. The order in which \texttt{entries} are returned is not defined and should not be relied upon. To pass further arguments to the \texttt{function}, see \texttt{\prop_map_tokens:Nn}.

\begin{verbatim}
\prop_map_inline:Nn \prop_map_inline:cn
\end{verbatim}

\texttt{\prop_map_inline:Nn} Applies \texttt{\prop_map_inline:Nn} to every \texttt{entry} stored within the \texttt{property list}. The \texttt{inline function} should consist of code which receives the \texttt{key} as \#1 and the \texttt{value} as \#2. The order in which \texttt{entries} are returned is not defined and should not be relied upon.

\begin{verbatim}
\prop_map_tokens:Nn \prop_map_tokens:cn
\end{verbatim}

\texttt{\prop_map_tokens:Nn} Analogue of \texttt{\prop_map_function:NN} which maps several tokens instead of a single function. The \texttt{code} receives each key–value pair in the \texttt{property list} as two trailing brace groups. For instance,

\begin{verbatim}
\prop_map_tokens:Nn \l_my_prop \str_if_eq:nnT { mykey }
\end{verbatim}

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 \texttt{key} and the \texttt{value} as its three arguments. For that specific task, \texttt{\prop_item:Nn} is faster.
\prop_map_break:

Used to terminate a \prop_map... function before all entries in the \langle property list \rangle have been processed. This normally takes place within a conditional statement, for example:

\prop_map_inline:Nn \l_my_prop
{\str_if_eq:nnTF { #1 } { bingo }
 { \prop_map_break: }
{ % Do something useful }
}

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

\textbf{\TeXhackers note}: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\prop_map_break:n

Used to terminate a \prop_map... function before all entries in the \langle property list \rangle have been processed, inserting the \langle code \rangle 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 code \rangle } }
{ % Do something useful }
}

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

\textbf{\TeXhackers 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.

### 25.8 Viewing property lists

\prop_show:N
\prop_show:c

Displays the entries in the \langle property list \rangle in the terminal.

\textbf{Updated: 2021-04-29}
25.9 Scratch property lists

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.

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

A permanently-empty property list used for internal comparisons.
Chapter 26

The \texttt{l3skip} package

Dimensions and skips

\LaTeX{} 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} ("\langle \texttt{dim expr} \rangle") or \textit{skip expressions} ("\langle \texttt{skip expr} \rangle") as arguments.

26.1 Creating and initialising \texttt{dim} variables

\begin{verbatim}
\dim_new:N \dim_new:N \langle \texttt{dimension} \rangle
\dim_new:c \langle \texttt{dimension} \rangle
\dim_const:Nn \dim_const:Nn \langle \texttt{dimension} \rangle \{ \langle \texttt{dim expr} \rangle \}
\dim_const:cn \dim_const:cn \langle \texttt{dimension} \rangle \{ \langle \texttt{dim expr} \rangle \}
\dim_zero:N \dim_zero:N \langle \texttt{dimension} \rangle
\dim_zero:c \dim_zero:c \langle \texttt{dimension} \rangle
\dim_gzero:N \dim_gzero:N \langle \texttt{dimension} \rangle
\dim_gzero:c \dim_gzero:c \langle \texttt{dimension} \rangle
\end{verbatim}

\texttt{\dim_new:N \dim_new:N \langle \texttt{dimension} \rangle}

Creates a new \langle \texttt{dimension} \rangle or raises an error if the name is already taken. The declaration is global. The \langle \texttt{dimension} \rangle is initially equal to 0 pt.

\texttt{\dim_new:c \langle \texttt{dimension} \rangle}

Creates a new \langle \texttt{dimension} \rangle or raises an error if the name is already taken. The value of the \langle \texttt{dimension} \rangle is set globally to \langle \texttt{dim expr} \rangle.

\texttt{\dim_const:Nn \dim_const:Nn \langle \texttt{dimension} \rangle \{ \langle \texttt{dim expr} \rangle \}}

Sets \langle \texttt{dimension} \rangle to 0 pt.

\texttt{\dim_zero:N \dim_zero:N \langle \texttt{dimension} \rangle}

Ensures that the \langle \texttt{dimension} \rangle exists globally by applying \texttt{\dim_new:N} if necessary, then applies \texttt{\dim\_gzero:N} to leave the \langle \texttt{dimension} \rangle set to zero.
\textbf{26.2 Setting dim variables}

\texttt{\dim_add:Nn} \quad \texttt{\dim_add:Nn \{\texttt{dim expr}\}}
\begin{itemize}
  \item \texttt{\dim_add:cn}
  \item \texttt{\dim_gadd:Nn}
  \item \texttt{\dim_gadd:cn}
\end{itemize}

Adds the result of the \{\texttt{dim expr}\} to the current content of the \{\texttt{dimension}\}.

\texttt{\dim_set:Nn} \quad \texttt{\dim_set:Nn \{\texttt{dimension}\} \{\texttt{dim expr}\}}
\begin{itemize}
  \item \texttt{\dim_set:cn}
  \item \texttt{\dim_gset:Nn}
  \item \texttt{\dim_gset:cn}
\end{itemize}

Sets \{\texttt{dimension}\} to the value of \{\texttt{dim expr}\}, which must evaluate to a length with units.

\texttt{\dim_set_eq:NN} \quad \texttt{\dim_set_eq:NN \{\texttt{dimension}_1\} \{\texttt{dimension}_2\}}
\begin{itemize}
  \item \texttt{\dim_set_eq:cn\texttt{|Ncc}}
  \item \texttt{\dim_gset_eq:NN\texttt{|Ncc}}
\end{itemize}

Sets the content of \{\texttt{dimension}_1\} equal to that of \{\texttt{dimension}_2\}.

\texttt{\dim_sub:Nn} \quad \texttt{\dim_sub:Nn \{\texttt{dimension}\} \{\texttt{dim expr}\}}
\begin{itemize}
  \item \texttt{\dim_sub:cn}
  \item \texttt{\dim_gsub:Nn}
  \item \texttt{\dim_gsub:cn}
\end{itemize}

Subtracts the result of the \{\texttt{dim expr}\} from the current content of the \{\texttt{dimension}\}.

\textbf{26.3 Utilities for dimension calculations}

\texttt{\dim_abs:n} \quad \texttt{\dim_abs:n \{\texttt{dim expr}\}}
\begin{itemize}
  \item \texttt{\dim_abs:cn}
\end{itemize}

Converts the \{\texttt{dim expr}\} to its absolute value, leaving the result in the input stream as a \{\texttt{dimension denotation}\}.

\texttt{\dim_max:nn} \quad \texttt{\dim_max:nn \{\texttt{dim expr}_1\} \{\texttt{dim expr}_2\}}
\begin{itemize}
  \item \texttt{\dim_max:cn\texttt{|Ncc}}
  \item \texttt{\dim_gmax:nn\texttt{|Ncc}}
\end{itemize}

Evaluates the two \{\texttt{dim exprs}\} and leaves either the maximum or minimum value in the input stream as appropriate, as a \{\texttt{dimension denotation}\}.
\dim_ratio:nn \(\dim_ratio:nn \{ (\dim expr_1) \} \{ (\dim expr_2) \}\)

 Parses the two \(\dim\ exprs\) and converts the ratio of the two to a form suitable for use inside a \(\dim\ expr\). This ratio is then left in the input stream, allowing syntax such as

\dim_set:Nn \l_my_dim
\{ 10 \text{ pt} \times \dim_ratio:nn \{ 5 \text{ pt} \} \{ 10 \text{ pt} \} \}

The output of \(\dim_ratio:nn\) on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

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

displays \(327680/655360\) on the terminal.

### 26.4 Dimension expression conditionals

\dim_compare_p:nNn \(\dim_compare_p:nNn \{ (\dim expr_1) \} \{ (\dim expr_2) \}\)
\dim_compare:nNn\*TF \(\dim_compare:nNn\*TF \{ (\dim expr_1) \} \{ (\dim expr_2) \} \{ (true\ code) \} \{ (false\ code) \}\)

This function first evaluates each of the \(\dim\ exprs\) as described for \(\dim_eval:n\). The two results are then compared using the \(\langle\ relation\rangle\):

- Equal \(=\)
- Greater than \(>\)
- Less than \(<\)

This function is less flexible than \(\dim_compare:nTF\) but around 5 times faster.
This function evaluates the \texttt{dim exprs} as described for \texttt{dim_eval:n} and compares consecutive result using the corresponding \texttt{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 \texttt{true} if all comparisons are \texttt{true}. Each \texttt{dim expr} is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is \texttt{false}, then no other \texttt{dim expr} is evaluated and no other comparison is performed. The \texttt{relations} can be any of the following:

- Equal \quad = \quad or \quad ==
- Greater than or equal to \quad >=
- Greater than \quad >
- Less than or equal to \quad <=
- Less than \quad <
- Not equal \quad !=

This function is more flexible than \texttt{dim_compare:nNnTF} but around 5 times slower.
\dim_case:nn \dim_case:nnTF {⟨test dim expr⟩}
\dim_case:nnTF {
 {⟨dim expr case_1⟩} {⟨code case_1⟩}
 {⟨dim expr case_2⟩} {⟨code case_2⟩}
...
 {⟨dim expr case_n⟩} {⟨code case_n⟩}
}
{⟨true code⟩}
{⟨false code⟩}
This function evaluates the ⟨test dim expr⟩ and compares this in turn to each of the ⟨dim expr cases⟩. If the two are equal then the associated ⟨code⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨true code⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨false code⟩ is inserted. The function \dim_case:nn, which does nothing if there is no match, is also available. For example
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
 { 2 \l_tmpa_dim }
{
 { 5 pt } { Small }
 { 4 pt + 6 pt } { Medium }
 { - 10 pt } { Negative }
}
{ No idea! }
leaves “Medium” in the input stream.

26.5 Dimension expression loops

\dim_do_until:nNnn \dim_do_until:nNnn {⟨dim expr_1⟩} ⟨relation⟩ {⟨dim expr_2⟩} {⟨code⟩}
Places the ⟨code⟩ in the input stream for \TeX to process, and then evaluates the relationship between the two ⟨dim exprs⟩ as described for \dim_compare:nNnTF. If the test is false then the ⟨code⟩ is inserted into the input stream again and a loop occurs until the ⟨relation⟩ is true.

\dim_do_while:nNnn \dim_do_while:nNnn {⟨dim expr_1⟩} ⟨relation⟩ {⟨dim expr_2⟩} {⟨code⟩}
Places the ⟨code⟩ in the input stream for \TeX to process, and then evaluates the relationship between the two ⟨dim exprs⟩ as described for \dim_compare:nNnTF. If the test is true then the ⟨code⟩ is inserted into the input stream again and a loop occurs until the ⟨relation⟩ is false.

\dim_until_do:nNnn \dim_until_do:nNnn {⟨dim expr_1⟩} ⟨relation⟩ {⟨dim expr_2⟩} {⟨code⟩}
Evaluates the relationship between the two ⟨dim exprs⟩ as described for \dim_compare:nNnTF, and then places the ⟨code⟩ in the input stream if the ⟨relation⟩ is false. After the ⟨code⟩ has been processed by \TeX the test is repeated, and a loop occurs until the test is true.
\texttt{\textbackslash dim\_while\_do\textcolon{}nn} \quad \texttt{\textbackslash dim\_while\_do\textcolon{}nn \{(dim\ expr_1)\} \{relation\} \{(dim\ expr_2)\} \{(code)\}}

Evaluates the relationship between the two \texttt{(dim exprs)} as described for \texttt{\textbackslash dim\_compare\textcolon{}nNF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{true}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.

\texttt{\textbackslash dim\_do\_until\textcolon{}nn} \quad \texttt{\textbackslash dim\_do\_until\textcolon{}nn \{(dimension\ relation)\} \{(code)\}}

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the \texttt{(dimension relation)} as described for \texttt{\textbackslash dim\_compare\textcolon{}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}.

\texttt{\textbackslash dim\_while\_do\textcolon{}nn} \quad \texttt{\textbackslash dim\_while\_do\textcolon{}nn \{(dimension\ relation)\} \{(code)\}}

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the \texttt{(dimension relation)} as described for \texttt{\textbackslash dim\_compare\textcolon{}nTF}. If the test is \texttt{true} then the \texttt{(code)} is inserted into the input stream again and a loop occurs until the \texttt{(relation)} is \texttt{false}.

\texttt{\textbackslash dim\_until\_do\textcolon{}nn} \quad \texttt{\textbackslash dim\_until\_do\textcolon{}nn \{(dimension\ relation)\} \{(code)\}}

Evaluates the \texttt{(dimension relation)} as described for \texttt{\textbackslash dim\_compare\textcolon{}nTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{false}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\texttt{\textbackslash dim\_while\_do\textcolon{}nn} \quad \texttt{\textbackslash dim\_while\_do\textcolon{}nn \{(dimension\ relation)\} \{(code)\}}

Evaluates the \texttt{(dimension relation)} as described for \texttt{\textbackslash dim\_compare\textcolon{}nTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{true}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.

### 26.6 Dimension step functions

\texttt{\textbackslash dim\_step\_function\colon{}nnnN} \quad \texttt{\textbackslash dim\_step\_function\colon{}nnnN \{(initial value)\} \{(step)\} \{(final value)\} \{function\}}

This function first evaluates the \texttt{(initial value)}, \texttt{(step)} and \texttt{(final value)}, all of which should be dimension expressions. The \texttt{(function)} is then placed in front of each \texttt{(value)} from the \texttt{(initial value)} to the \texttt{(final value)} in turn (using \texttt{(step)} between each \texttt{(value)}). The \texttt{(step)} must be non-zero. If the \texttt{(step)} is positive, the loop stops when the \texttt{(value)} becomes larger than the \texttt{(final value)}. If the \texttt{(step)} is negative, the loop stops when the \texttt{(value)} becomes smaller than the \texttt{(final value)}. The \texttt{(function)} should absorb one argument.

\texttt{\textbackslash dim\_step\_inline\colon{}nn} \quad \texttt{\textbackslash dim\_step\_inline\colon{}nn \{(initial value)\} \{(step)\} \{(final value)\} \{(code)\}}

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 \texttt{\#1} replaced by the current \texttt{(value)}. Thus the \texttt{(code)} should define a function of one argument (\texttt{\#1}).
26.7 Using dim expressions and variables

\dim_step_variable:nnn

\dim_step_variable:nnn
{(initial value)} \{(step)\} \{(final value)\} \{tl \ var\} \{(code)\}

This function first evaluates the \textit{initial value}, \textit{step} and \textit{final value}, all of which should be dimension expressions. Then for each \textit{value} from the \textit{initial value} to the \textit{final value} in turn (using \textit{step} between each \textit{value}), the \textit{code} is inserted into the input stream, with the \textit{tl \ var} defined as the current \textit{value}. Thus the \textit{code} should make use of the \textit{tl \ var}.

\dim_eval:n

\dim_eval:n \{\textit{dim expr}\}

Evaluates the \textit{dim expr}, expanding any dimensions and token list variables within the \textit{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 \textit{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 \textit{internal dimension}.

\dim_sign:n

\dim_sign:n \{\textit{dim expr}\}

Evaluates the \textit{dim expr} then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

\dim_use:N

\dim_use:N \textit{dimension}

Recovers the content of a \textit{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 \textit{dimension} is required (such as in the argument of \texttt{\dim_eval:n}).

\textbf{\TeX\-hackers note:} \texttt{\dim_use:N} is the \TeX\ primitive \texttt{\the}: this is one of several \TeX3 names for this primitive.

\dim_to_decimal:n

\dim_to_decimal:n \{\textit{dim expr}\}

Evaluates the \textit{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

\begin{verbatim}
\dim_to_decimal:n \{ 1bp \}
\end{verbatim}

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 \text{dim expr} \rangle \}

Evaluates the \( \langle \text{dim expr} \rangle \), and leaves the result, expressed in big points (bp) 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

\begin{verbatim}
\dim_to_decimal_in_bp:n \{ 1pt \}
\end{verbatim}

leaves 0.99628 in the input stream, \textit{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

\begin{verbatim}
\dim_compare:nNnTF\{ <x>bp \} = \{ \dim_to_decimal_in_bp:n \{ <x>bp \} \bp \}
\end{verbatim}

will be logically \textit{true}. The decimal representations may differ provided they produce the same \TeX{} dimension.

\dim_to_decimal_in_cm:n \{ \langle \text{dim expr} \rangle \}

Evaluates the \( \langle \text{dim expr} \rangle \), and leaves the result, expressed with the appropriate scaling in the input stream, with \textit{no units}. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precision of the result is limited to a maximum of five decimal places with trailing zeros omitted.

The maximum \TeX{} allowable dimension value (available as \texttt{\maxdimen} in plain \TeX{} and \LaTeX{} and \texttt{\c_max_dim} in expl3) can only be expressed exactly in the units \texttt{pt}, \texttt{bp} and \texttt{sp}. The maximum allowable input values to five decimal places are

\begin{verbatim}
1276.00215 cc
575.83174 cm
15312.02584 dd
226.70540 in
5758.31742 mm
1365.33333 pc
\end{verbatim}

(Note that these are not all equal, but rather any larger value will overflow due to the way \TeX{} converts to \texttt{sp}.) Values given to five decimal places larger than these will result in \TeX{} errors; the behavior if additional decimal places are given depends on the \TeX{} internals and thus larger values are \textit{not} supported by expl3.

\TeX{}hackers note: The implementation of these functions is re-entrant: the result of

\begin{verbatim}
\dim_compare:nNnTF\{ <x><unit> \} = \{ \dim_to_decimal_in_<unit>:n \{ <x><unit> \} <unit> \}
\end{verbatim}

will be logically \textit{true}. The decimal representations may differ provided they produce the same \TeX{} dimension.
\texttt{\textbackslash dim\_to\_decimal\_in\_sp:n} \hspace{1em} \texttt{\textbackslash 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.

\texttt{\textbackslash dim\_to\_decimal\_in\_unit:nn} \hspace{1em} \texttt{\textbackslash 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
\begin{verbatim}
\texttt{\textbackslash dim\_to\_decimal\_in\_unit:nn \{ 1bp \} \{ 1mm \}}
\end{verbatim}
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
\begin{verbatim}
\texttt{\textbackslash dim\_compare:nNnTF}
\{ 1bp \} =
\{ \texttt{\textbackslash dim\_to\_decimal\_in\_unit:nn \{ 1bp \} \{ 1mm \} mm} \}
\end{verbatim}
will take the false branch.

\texttt{\textbackslash dim\_to\_fp:n} \hspace{1em} \texttt{\textbackslash 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, \texttt{\textbackslash 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

\texttt{\textbackslash dim\_show:N} \hspace{1em} \texttt{\textbackslash dim\_show:N \{dimension\}}

Displays the value of the \langle dimension\rangle on the terminal.

\texttt{\textbackslash dim\_show:n} \hspace{1em} \texttt{\textbackslash dim\_show:n \{\langle dim\ expr\rangle\}}

Displays the result of evaluating the \langle dim expr\rangle on the terminal.

\texttt{\textbackslash dim\_log:N} \hspace{1em} \texttt{\textbackslash dim\_log:N \{dimension\}}

\texttt{\textbackslash dim\_log:n} \hspace{1em} \texttt{\textbackslash dim\_log:n \{\langle dim\ expr\rangle\}}

Writes the value of the \langle dimension\rangle in the log file.
\dim_log:n \{\dim expr\}

Writers the result of evaluating the \texttt{\dim expr} in the log file.

\textbf{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.

\textbf{26.10 Scratch dimensions}

\l_tmpa_dim \l_tmpb_dim

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

\g_tmpa_dim \g_tmpb_dim

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

\textbf{26.11 Creating and initialising skip variables}

\skip_new:N \skip_new:c

Creates a new \texttt{\skip} or raises an error if the name is already taken. The declaration is global. The \texttt{\skip} is initially equal to 0 pt.

\skip_const:Nn \skip_const:cn

Creates a new constant \texttt{\skip} or raises an error if the name is already taken. The value of the \texttt{\skip} is set globally to the \texttt{\skip expr}.

\skip_zero:N \skip_zero:c \skip_gzero:N \skip_gzero:c

Sets \texttt{\skip} to 0 pt.
\skip_zero_new:N \skip_zero_new:c \skip_gzero_new:N \skip_gzero_new:c

Ensures that the \texttt{(skip)} exists globally by applying \texttt{\skip_new:N} if necessary, then applies \texttt{\skip_(g)zero:N} to leave the \texttt{(skip)} set to zero.

\skip_if_exist_p:N \skip_if_exist:NTF \skip_if_exist:c \star

Tests whether the \texttt{(skip)} is currently defined. This does not check that the \texttt{(skip)} really is a skip variable.

\skip_add:Nn \skip_add:cn \skip_gadd:Nn \skip_gadd:cn

Updated: 2011-10-22

\skip_set:Nn \skip_set:cn \skip_gset:Nn \skip_gset:cn

Updated: 2011-10-22

\skip_set_eq:NN \skip_set_eq:(cN|Nc|cc) \skip_gset_eq:NN \skip_gset_eq:(cN|Nc|cc)

Sets the content of \texttt{(skip\textsubscript{1})} equal to that of \texttt{(skip\textsubscript{2})}.

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

Updated: 2011-10-22

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

Subtracts the result of the \texttt{(skip expr)} from the current content of the \texttt{(skip)}.

\textbf{26.12 Setting skip variables}

\skip_add:Nn \texttt{(skip \{skip expr\})}

Adds the result of the \texttt{(skip expr)} to the current content of the \texttt{(skip)}.

\skip_set:Nn \texttt{(skip \{skip expr\})}

Sets \texttt{(skip)} to the value of \texttt{(skip expr)}, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).
26.13 Skip expression conditionals

\skip_if_eq_p:nn \{⟨\text{skip expr}_1⟩\} \{⟨\text{skip expr}_2⟩\}
\skip_if_eq:nnTF \{⟨\text{skip expr}_1⟩\} \{⟨\text{skip expr}_2⟩\} \{⟨\text{true code}⟩\} \{⟨\text{false code}⟩\}

This function first evaluates each of the ⟨\text{skip expr}s⟩ 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.

\skip_if_finite_p:n \{⟨\text{skip expr}⟩\}
\skip_if_finite:nTF \{⟨\text{skip expr}⟩\} \{⟨\text{true code}⟩\} \{⟨\text{false code}⟩\}

Evaluates the ⟨\text{skip expr}⟩ as described for \texttt{\skip_eval:n}, and then tests if all of its components are finite.

26.14 Using skip expressions and variables

\skip_eval:n \{⟨\text{skip expr}⟩\}

Updated: 2011-10-22

Evaluates the ⟨\text{skip expr}⟩, expanding any skips and token list variables within the ⟨\text{expression}⟩ to their content (without requiring \texttt{\skip_use:N/\tl_use:N}) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨\text{glue denotation}⟩ after two expansions. This is expressed in points (pt), and requires suitable termination if used in a \LaTeX-style assignment as it is not an ⟨\text{internal glue}⟩.

\skip_use:N \{⟨\text{skip}⟩\}
\skip_use:c \{⟨\text{skip}⟩\}

Recovers the content of a ⟨\text{skip}⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨\text{dimension}⟩ or ⟨\text{skip}⟩ is required (such as in the argument of \texttt{\skip_eval:n}).

\TeXhackers note: \texttt{\skip_use:N} is the \TeX primitive \texttt{\the}: this is one of several \LaTeX names for this primitive.

26.15 Viewing skip variables

\skip_show:N \{⟨\text{skip}⟩\}
\skip_show:c \{⟨\text{skip}⟩\}

Displays the value of the ⟨\text{skip}⟩ on the terminal.

\skip_show:n \{⟨\text{skip expr}⟩\}

Updated: 2015-08-03

Updated: 2015-08-07

Nov: 2011-11-22

Displays the result of evaluating the ⟨\text{skip expr}⟩ on the terminal.
26.16 Constant skips

\c_max_skip
Updated: 2012-11-02

The maximum value that can be stored as a skip (equal to \c_max_dim in length), with no stretch nor shrink component.

\c_zero_skip
Updated: 2012-11-01

A zero length as a skip, with no stretch nor shrink component.

26.17 Scratch skips

\l_tmpa_skip
\l_tmpb_skip

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

\g_tmpa_skip
\g_tmpb_skip

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

26.18 Inserting skips into the output

\skip_horizontal:N\skip_horizontal:N\skip_horizontal:C
\skip_horizontal:n {\skip expr}
\skip_horizontal:n

Inserts a horizontal \(<\text{skip}>\) into the current list. The argument can also be a \(<\text{dim}>\).

\TeX\text{hackers note}: \skip_horizontal:N is the \TeX primitive \hskip.
\skip_vertical:N \skip_vertical:c \skip_vertical:n

Updated: 2011-10-22

\Texhackers note: \skip_vertical:N is the \TeX primitive \vskip.

26.19 Creating and initialising muskip variables

\muskip_new:N \muskip_new:c \muskip_new:Nn \muskip_new:cn

\muskip_new:N (\muskip)

Creates a new \muskip or raises an error if the name is already taken. The declaration is global. The \muskip is initially equal to 0\mu.

\muskip_const:Nn \muskip_const:cn

\muskip_const:Nn (\muskip) {\langle \muskip expr \rangle}

\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 \langle \muskip expr \rangle.

\muskip_zero:N \muskip_zero:c \muskip_zero:Nn \muskip_zero:cn

\muskip_zero:N (\muskip)

Sets \muskip to 0\mu.

\muskip_zero_new:N \muskip_zero_new:c \muskip_gzero:N \muskip_gzero:Nn \muskip_gzero:cn

\muskip_zero_new:N (\muskip)

Ensures that the \muskip exists globally by applying \muskip_new:N if necessary, then applies \muskip_(g)zero:N to leave the \muskip set to zero.

\muskip_zero_new:N \muskip_zero_new:cn \muskip_gzero_new:N \muskip_gzero_new:cn

New: 2012-01-07

\muskip_if_exist_p:N \muskip_if_exist_p:c \muskip_if_exist:NTF \muskip_if_exist:c

\muskip_if_exist_p:N (\muskip)

\muskip_if_exist:NTF (\muskip) {\langle true code \rangle} {\langle false code \rangle}

\muskip_if_exist_p:c

\muskip_if_exist:NTF (\muskip) {\langle true code \rangle} {\langle false code \rangle}

\muskip_if_exist:c

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

New: 2012-03-03

26.20 Setting muskip variables

\muskip_add:N \muskip_add:cn \muskip_add:Nn \muskip_add:cn

\muskip_add:N (\muskip) {\langle \muskip expr \rangle}

\muskip_add:cn

\muskip_add:N (\muskip) {\langle \muskip expr \rangle}

Addsextraretult of the \muskip expr to the current content of the \muskip.

Updated: 2011-10-22
\verb|\muskip_set:Nn| \verb|\muskip_set:cn| \verb|\muskip_gset:Nn| \verb|\muskip_gset:cn|

Updated: 2011-10-22

\verb|\muskip_set_eq:NN| \verb|\muskip_set_eq:cn| \verb|\muskip_gset_eq:NN|

Updated: 2011-10-22

\verb|\muskip_sub:Nn| \verb|\muskip_sub:cn| \verb|\muskip_gsub:Nn| \verb|\muskip_gsub:cn|

Updated: 2011-10-22

---

26.21 Using \texttt{muskip} expressions and variables

\verb|\muskip_eval:n| \verb|\muskip_eval:n| \verb|\muskip_use:N| \verb|\muskip_use:c|

Updated: 2011-10-22

26.22 Viewing \texttt{muskip} variables

\verb|\muskip_show:N| \verb|\muskip_show:c|

Updated: 2015-08-03

235
26.23 Constant muskips

\c_max_muskip
The maximum value that can be stored as a muskip, with no stretch nor shrink component.

\c_zero_muskip
A zero length as a muskip, with no stretch nor shrink component.

26.24 Scratch muskips

\l_tmpa_muskip
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.

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

26.25 Primitive conditional

\if_dim:w
Compare two dimensions. The \langle relation \rangle is one of \langle, = or > with category code 12.

\TeXhackers note: This is the \TeX primitive \texttt{\ifdim}.  

236
Chapter 27

The l3keys package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

\MyModuleSetup{
  key-one = value one,
  key-two = value two
}

or

\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more properties of the key:

\keys_define:nn { mymodule }
{
  key-one .code:n = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}

These values can then be set as with other key–value approaches:

\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
As illustrated, keys are created inside a \texttt{⟨module⟩}: a set of related keys, typically those for a single \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 \textit{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}.

### 27.1 Creating keys

\texttt{\keys_define:nn}{\langle module⟩}{\langle keyval list⟩}

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 \textit{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 \textit{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}
\texts_define:nn { mymodule }
{ 
  keyname .code:n = Some-code-using-#1,
  keyname .value_required:n = true
}
\texts_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 \texttt{undefines} the key within the current \TeX group.

\begin{verbatim}
\texttt{(key) .bool_set:N = (boolean variable)}
\end{verbatim}

Defines \texttt{(key)} to set \texttt{(boolean variable)} to \texttt{(value)} (which must be either \texttt{"true"} or \texttt{"false"}). If the variable does not exist, it will be created globally at the point that the key is set up.

\begin{verbatim}
\texttt{(key) .bool_set_inverse:N = (boolean variable)}
\end{verbatim}

Defines \texttt{(key)} to set \texttt{(boolean variable)} to the logical inverse of \texttt{(value)} (which must be either \texttt{"true"} or \texttt{"false"}). If the \texttt{(boolean variable)} does not exist, it will be created globally at the point that the key is set up.

\begin{verbatim}
\texttt{(key) .choice:}
\end{verbatim}

Sets \texttt{(key)} to act as a choice key. Each valid choice for \texttt{(key)} must then be created, as discussed in section 27.3.

\begin{verbatim}
\texttt{(key) .choices:nn = \{ \texttt{(choices)} \}\{\texttt{(code)}\}}
\end{verbatim}

Sets \texttt{(key)} to act as a choice key, and defines a series \texttt{(choices)} which are implemented using the \texttt{(code)}. Inside \texttt{(code)}, \texttt{\l_keys_choice_tl} will be the name of the choice made, and \texttt{\l_keys_choice_int} will be the position of the choice in the list of \texttt{(choices)} (indexed from 1). Choices are discussed in detail in section 27.3.

\begin{verbatim}
\texttt{(key) .clist_set:N = (comma list variable)}
\end{verbatim}

Defines \texttt{(key)} to set \texttt{(comma list variable)} to \texttt{(value)}. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.
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:Np} \texttt{.cs_set:cp} \texttt{.cs_set_protected:Np} \texttt{.cs_set_protected:cp} \texttt{.cs_gset:Np} \texttt{.cs_gset:cp} \texttt{.cs_gset_protected:Np} \texttt{.cs_gset_protected:cp}

Defines \texttt{(key)} to set \texttt{(control sequence)} to have \texttt{(arg. spec.)} and replacement text \texttt{(value)}.

\texttt{.default:n} \texttt{.default:(V|e|o)}

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} \texttt{.dim_set:c} \texttt{.dim_gset:N} \texttt{.dim_gset:c}

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} \texttt{.fp_set:c} \texttt{.fp_gset:N} \texttt{.fp_gset:c}

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 = {\text{groups}}

New: 2013-07-14

 Defines \key as belonging to the \text{groups} declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 27.7.

.inherit:n \key \inherit:n = {\text{parents}}

New: 2016-11-22

 Specifies that the \key path should inherit the keys listed as any of the \text{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 \text{parent} after the use of \text{inherit:n} and will be active. If more than one \text{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 = {\text{value}}

Initialises the \key with the \text{value}, equivalent to

\begin{verbatim}
\keys_set:nn {\text{module}} { \key = \text{value} }
\end{verbatim}

Updated: 2013-07-09

.int_set:N \key \int_set:N = \text{integer}

Defines \key to set \text{integer} to \text{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.

Updated: 2020-01-17

.legacy_if_set:n \key .legacy_if_set:n = \text{switch}

Defines \key to set legacy \text{if} \text{switch} to \text{value} (which must be either “true” or “false”). The \text{switch} is the name of the switch without the leading \text{if}. The \text{inverse} versions will set the \text{switch} to the logical opposite of the \text{value}.

Updated: 2022-01-15

.meta:n \key .meta:n = {\text{keyval list}}

Makes \key a meta-key, which will set \text{keyval list} in one go. The \text{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).

Updated: 2013-07-10
Makes `<key>` a meta-key, which will set `<keyval list>` in one go using the `<path>` in place of the current one. The `<keyval list>` can refer as `#1` to the value given at the time the `<key>` is used (or, if no value is given, the `<key>`'s default value).

Sets `<key>` to act as a multiple choice key. Each valid choice for `<key>` must then be created, as discussed in section 27.3.

Sets `<key>` to act as a multiple choice key, and defines a series of `<choices>` which are implemented using the `<code>` inline. `<code>` will be the name of the choice made, and `<l_keys_choice_int>` will be the position of the choice in the list of `<choices>` (indexed from 1). Choices are discussed in detail in section 27.3.

Defines `<key>` to set `<muskip>` to `<value>` (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

Defines `<key>` to set `<string>` to `<value>` (which must be a string variable). If the variable does not exist, it is created globally at the point that the key is set up.
\texttt{.tl_set:N} \texttt{(key)} \texttt{.tl_set:N = (token list variable)}

Defines \texttt{(key)} to set \texttt{(token list variable)} to \texttt{(value)}. If the variable does not exist, it is created globally at the point that the key is set up.

\texttt{.tl_set_e:N} \texttt{(key)} \texttt{.tl_set_e:N = (token list variable)}

Defines \texttt{(key)} to set \texttt{(token list variable)} to \texttt{(value)}, which will be subjected to an e-type expansion (i.e. using \texttt{\tl_set:Ne}). If the variable does not exist, it is created globally at the point that the key is set up.

\texttt{.undefine:} \texttt{(key)} \texttt{.undefine:}

Removes the definition of the \texttt{(key)} within the current \TeX{} group.

\texttt{.value_forbidden:n} \texttt{(key)} \texttt{.value_forbidden:n = true|false}

Specifies that \texttt{(key)} cannot receive a \texttt{(value)} when used. If a \texttt{(value)} is given then an error will be issued. Setting the property “false” cancels the restriction.

\texttt{.value_required:n} \texttt{(key)} \texttt{.value_required:n = true|false}

Specifies that \texttt{(key)} must receive a \texttt{(value)} when used. If a \texttt{(value)} is not given then an error will be issued. Setting the property “false” cancels the restriction.

### 27.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

\begin{verbatim}
\keys_define:nn { mymodule / subgroup }
  { key .code:n = code }
\end{verbatim}

or to the key name:

\begin{verbatim}
\keys_define:nn { mymodule }
  { subgroup / key .code:n = code }
\end{verbatim}

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 \texttt{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 l3keys system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the .choice: property:

\keys_define:nn { mymodule }
{ key .choice: }

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the .choices:nn property.

\keys_define:nn { mymodule }
{ key .choices:nn =
  { choice-a, choice-b, choice-c }
  { You\-\ gave\-\ choice-\'\tl_use:N \l_keys_choice_tl\',~
    which\-is\-in\-position\-\int_use:N \l_keys_choice_int \c_space_tl
    in\-the\-list. }
}

The index \l_keys_choice_int in the list of choices starts at 1.

\l_keys_choice_int \l_keys_choice_tl

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

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

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

It is possible to mix the two methods, but manually-created choices should not use \l_keys_choice_tl or \l_keys_choice_int. These variables do not have defined
behaviour when used outside of code created using \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
    { choice-a , choice-b , choice-c } % Valid choices
    { \exp_not:n {#1} } % Invalid choice given
  }
\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 define 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.

```
\keys_set:nn
  \keys_set:nn { module } { keyval list }
```

Parses the \texttt{keyval list}, and sets those keys which are defined for \texttt{module}. The behaviour on finding an unknown key can be set by defining a special \texttt{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.

\keys_define:nn \{ mymodule \}
{ unknown .code:n = You~tried~to~set~key~'\l_keys_key_str'~to~'#1'. }

These functions set keys which are known for the \langle module \rangle, and simply ignore other keys. The \keys_set_known:nn function parses the \langle keyval list \rangle, and sets those keys which are defined for \langle module \rangle. Any keys which are unknown are not processed further by the parser. In addition, \keys_set_known:nn stores the key–value pairs in the \langle tl \rangle in comma-separated form \langle i.e. an edited version of the \langle keyval list \rangle \rangle. When a \langle root \rangle is given \langle \keys_set_known:nnn \rangle, 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

```
\keys_define:nn { mymodule }
  {
    key-one .code:n = { \my_func:n {#1} } ,
    key-two .tl_set:N = \l_my_a_tl ,
    key-three .tl_set:N = \l_my_b_tl ,
    key-four .fp_set:N = \l_my_a_fp ,
  }
```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to `groups`: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys_define:nn { mymodule }
  {
    key-one .code:n = { \my_func:n {#1} } ,
    key-one .groups:n = { first } ,
    key-two .tl_set:N = \l_my_a_tl ,
    key-two .groups:n = { first } ,
    key-three .tl_set:N = \l_my_b_tl ,
    key-three .groups:n = { second } ,
    key-four .fp_set:N = \l_my_a_fp ,
  }
```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

```
\keys_set_filter:nnn { \langle module \rangle } { \langle groups \rangle } { \langle keyval list \rangle }
\keys_set_filter:nnnN { \langle module \rangle } { \langle groups \rangle } { \langle keyval list \rangle } { \langle tl \rangle }
\keys_set_filter:nnnnN { \langle module \rangle } { \langle groups \rangle } { \langle keyval list \rangle } { \langle root \rangle } { \langle tl \rangle }
```

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 (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.

New: 2013-07-14
Updated: 2019-01-29
27.8 Digesting keys

 Parses the \texttt{\keysval} as for \keysset, placing the resulting code for those which set variables or functions into the \texttt{\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 is defined for the \module, \ie if any code has been defined for \key. 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 \keysshow 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

\[
\begin{align*}
\text{KeyOne} &= \text{ValueOne} , \\
\text{KeyTwo} &= \text{ValueTwo} , \\
\text{KeyThree} &
\end{align*}
\]

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 \(\langle\text{key–value list}\rangle\) into \(\langle\text{keys}\rangle\) and associated \(\langle\text{values}\rangle\).

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

\[
\begin{align*}
\text{key} &= \{\text{value here}\}, \\
\text{key} &= \text{value here},
\end{align*}
\]

are treated identically.
\keyval_parse:nnn \keyval_parse:(nnVnnv) \keyval_parse:nnn

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:nnn { code 2 } }
    { key1 = value1 , key2 = value2, key3 = , key4 }
\end{verbatim}

is converted into an input stream

\begin{verbatim}
\use_none:nnn { code 2 } { key1 } { value1 }
\use_none:nnn { code 2 } { key2 } { value2 }
\use_none:nnn { code 2 } { key3 } { }
\use_none:nn { code 1 } { key4 }
\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{\TeX\hackers 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 \texttt{e}-type or \texttt{x}-type argument expansion.
\keyval_parse:NNn \keyval_parse:(NNV|NNv) \keyval_parse:NNn (function1) (function2) \keyval_parse:NNn (key–value list)}

Parses the \keyval_parse:NNn (key–value list) into a series of \keyval_parse:NNn (keys) and associated \keyval_parse:NNn (values), or \keyval_parse:NNn keys alone (if no \keyval_parse:NNn value was given). \keyval_parse:NNn (function1) should take one argument, while \keyval_parse:NNn (function2) should absorb two arguments. After \keyval_parse:NNn has parsed \keyval_parse:NNn the \keyval_parse:NNn (key–value list), \keyval_parse:NNn (function1) is used to process keys given with no value and \keyval_parse:NNn (function2) is used to process keys given with a value. The order of the \keyval_parse:NNn (keys) in the \keyval_parse:NNn (key–value list) is preserved. Thus

\keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn { key1 = value1 , key2 = value2, key3 = , key4 }

is converted into an input stream

\keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn { key1 } { value1 }
\keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn { key2 } { value2 }
\keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn { key3 } {}
\keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn \keyval_parse:NNn { 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 \keyval_parse:NNn (key) and \keyval_parse:NNn (value), then one outer set of braces is removed from the \keyval_parse:NNn (key) and \keyval_parse:NNn (value) as part of the processing.

This shares the implementation of \keyval_parse:NNn, the difference is only semantically.

\TeXhacker note: The result is returned within \exp_not:n, which means that the converted input stream does not expand further when appearing in an e-type or x-type argument expansion.
Chapter 28

The \texttt{l3intarray} package: 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}
\intarray_new:Nn \intarray_new:cn
\intarray_count:N
\intarray_gset:Nnn \intarray_gset:cnn
\end{verbatim}

\texttt{\intarray_new:Nn} \langle \text{intarray var} \rangle \{ \langle \text{size} \rangle \}

Evaluates the integer expression \langle \text{size} \rangle and allocates an \langle \text{integer array variable} \rangle with that number of (zero) entries. The variable name should start with \texttt{\g_} because assignments are always global.

\texttt{\intarray_count:N} \langle \text{intarray var} \rangle

Expands to the number of entries in the \langle \text{integer array variable} \rangle. Contrarily to \texttt{\seq_count:N} this is performed in constant time.

\texttt{\intarray_gset:Nnn} \langle \text{intarray var} \rangle \{ \langle \text{position} \rangle \} \{ \langle \text{value} \rangle \}

Stores the result of evaluating the integer expression \langle \text{value} \rangle into the \langle \text{integer array variable} \rangle at the (integer expression) \langle \text{position} \rangle. If the \langle \text{position} \rangle is not between 1 and the \texttt{\intarray_count:N}, or the \langle \text{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 \intarray_var \{ \text{int expr clist}\}
\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:c
New: 2018-05-04

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

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:c
New: 2018-05-05

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:c \intarray_log:N \intarray_log:c
New: 2018-05-04

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} package 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 \texttt{l3fp} package: Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions ("\texttt{fp expr}"") support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x \ast y$, division $x/y$, square root $\sqrt{x}$, and parentheses.
- Comparison operators: $x < y$, $x <= y$, $x > y$, $x ! = y$ etc.
- Boolean logic: sign $\text{sign} x$, negation $\text{!} x$, conjunction $x \&\& y$, disjunction $x || y$, ternary operator $x ? y : z$.
- Exponentials: $\exp x$, $\ln x$, $x^y$, $\logb x$.
- Integer factorial: $\text{fact} x$.
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\text{sind} x$, $\text{cosd} x$, $\text{tand} x$, $\text{cotd} x$, $\text{secd} x$, $\text{csed} x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\text{asin} x$, $\text{acos} x$, $\text{atan} x$, $\text{acot} x$, $\text{asec} x$, $\text{acsc} x$ giving a result in radians, and $\text{asind} x$, $\text{acosd} x$, $\text{atand} x$, $\text{acotd} x$, $\text{asecd} x$, $\text{acsed} x$ giving a result in degrees.

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

- Extrema: $\text{max}(x_1, x_2, \ldots)$, $\text{min}(x_1, x_2, \ldots)$, $\text{abs}(x)$.
- Rounding functions, controlled by two optional values, $n$ (number of places, 0 by default) and $t$ (behavior on a tie, \texttt{nan} by default):
  - $\text{trunc}(x, n)$ rounds towards zero,
  - $\text{floor}(x, n)$ rounds towards $-\infty$, 

255
– ceil(x, n) rounds towards +∞,
– round(x, n, t) rounds to the closest value, with ties rounded to an even value by default, towards zero if \( t = 0 \), towards +∞ if \( t > 0 \) and towards −∞ if \( t < 0 \).

And (not yet) modulo, and “quantize”.

• Random numbers: \( \text{rand}() \), \( \text{randint}(m,n) \).
• Constants: \( \pi \), \( \deg \) (one degree in radians).
• Dimensions, automatically expressed in points, e.g., \( \text{pc} \) is 12.
• Automatic conversion (no need for \( \langle \text{type} \rangle \_\text{use:N} \)) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
• Tuples: \((x_1, \ldots, x_n)\) that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as 1.234e-34, or −.0001), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. See section 29.12.1 for a description of what a floating point is, section 29.12.2 for details about how an expression is parsed, and section 29.12.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 29.10.

An example of use could be the following.

\[ \LaTeX{} \text{ can now compute: } \frac{\sin (3.5)}{2} + 2 \cdot 10^{-3} = \text{\ExplSyntaxOn \fp_to_decimal:n {sin(3.5)/2 + 2e-3}} \].

The operation round can be used to limit the result’s precision. Adding +0 avoids the possibly undesirable output −0, replacing it by +0. However, the l3fp module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

\texttt{\documentclass{article}}
\texttt{\usepackage{siunitx}}
\texttt{\ExplSyntaxOn}
\texttt{\NewDocumentCommand { \calcnum } { \m }}
\texttt{ { \num { \fp_to_scientific:n \{#1\} } }}
\texttt{\ExplSyntaxOff}
\texttt{\begin{document}}
\texttt{\calcnum { 2 \pi \ast \sin ( 2.3 ^ 5 ) }}
\texttt{\end{document}}

See the documentation of \texttt{siunitx} for various options of \texttt{\num}.
29.1 Creating and initialising floating point variables

\texttt{\fp_new:N} \hspace{1em} \texttt{\fp_new:c}

Creates a new \texttt{\fp var} or raises an error if the name is already taken. The declaration is global. The \texttt{\fp var} is initially $+0$.

\texttt{\fp_const:Nn} \hspace{1em} \texttt{\fp_const:cn}

Creates a new constant \texttt{\fp var} or raises an error if the name is already taken. The \texttt{\fp var} is set globally equal to the result of evaluating the \texttt{\fp expr}.

\texttt{\fp_zero:N} \hspace{1em} \texttt{\fp_zero:c} \hspace{1em} \texttt{\fp_gzero:N} \hspace{1em} \texttt{\fp_gzero:c}

Sets the \texttt{\fp var} to $+0$.

\texttt{\fp_zero_new:N} \hspace{1em} \texttt{\fp_zero_new:c} \hspace{1em} \texttt{\fp_gzero_new:N} \hspace{1em} \texttt{\fp_gzero_new:c}

Ensures that the \texttt{\fp var} exists globally by applying \texttt{\fp_new:N} if necessary, then applies \texttt{\fp_(g)zero:N} to leave the \texttt{\fp var} set to $+0$.

29.2 Setting floating point variables

\texttt{\fp_set:Nn} \hspace{1em} \texttt{\fp_set:cn} \hspace{1em} \texttt{\fp_gset:Nn} \hspace{1em} \texttt{\fp_gset:cn}

Sets \texttt{\fp var} equal to the result of computing the \texttt{\fp expr}.

\texttt{\fp_set_eq:NN} \hspace{1em} \texttt{\fp_set_eq:(cN|Nc|cc)} \hspace{1em} \texttt{\fp_gset_eq:NN} \hspace{1em} \texttt{\fp_gset_eq:(cN|Nc|cc)}

Sets the floating point variable \texttt{\fp var$_1$} equal to the current value of \texttt{\fp var$_2$}.

\texttt{\fp_add:Nn} \hspace{1em} \texttt{\fp_add:cn} \hspace{1em} \texttt{\fp_gadd:Nn} \hspace{1em} \texttt{\fp_gadd:cn}

Adds the result of computing the \texttt{\fp expr} to the \texttt{\fp var}. This also applies if \texttt{\fp var} and \texttt{\floating point expression} evaluate to tuples of the same size.
29.3 Using floating points

\fp_to_decimal:n \fp_to_decimal:n {\langle \text{fp expr} \rangle}
\fp_to_dim:n \fp_to_dim:n {\langle \text{fp expr} \rangle}
\fp_to_int:n \fp_to_int:n {\langle \text{fp expr} \rangle}
\fp_eval:n \fp_eval:n {\langle \text{fp expr} \rangle}
\fp_sign:n \fp_sign:n {\langle \text{fp expr} \rangle}
\fp_sub:Nn \fp_sub:Nn {\langle \text{fp expr} \rangle}
\fp_gsub:Nn \fp_gsub:cn

Subtracts the result of computing the \textit{floating point expression} from the \textit{fp var}. This also applies if \textit{fp var} and \textit{floating point expression} evaluate to tuples of the same size.

\fp_gsub:cn

Updated: 2012-05-08
\texttt{\textbackslash fp\_to\_scientific:N} * \texttt{\textbackslash fp\_to\_scientific:c} * \texttt{\textbackslash fp\_to\_scientific:n} \{\textit{fp expr}\}  

Evaluates the \textit{fp expr} and expresses the result in scientific notation: 

\(\text{(optional -)(digit).(15 digits)e(optional sign)(exponent)}\)

The leading \textit{digit} is non-zero except in the case of \(\pm 0\). The values \(\pm \infty\) and \texttt{nan} trigger an “invalid operation” exception. Normal category codes apply: thus the \texttt{e} is category code 11 (a letter). For a tuple, each item is converted using \texttt{\textbackslash fp\_to\_scientific: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.

\texttt{\textbackslash fp\_to\_tl:N} * \texttt{\textbackslash fp\_to\_tl:c} * \texttt{\textbackslash fp\_to\_tl:n} \{\textit{fp expr}\}  

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 \texttt{\textbackslash 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 \texttt{\textbackslash fp\_to\_decimal:n}). Negative numbers start with -. The special values \(\pm 0, \pm \infty\) and \texttt{nan} are rendered as 0, -0, \texttt{inf}, -\texttt{inf}, and \texttt{nan} respectively. Normal category codes apply and thus \texttt{inf} or \texttt{nan}, if produced, are made up of letters. For a tuple, each item is converted using \texttt{\textbackslash fp\_to\_tl: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 \texttt{\textbackslash fp\_to\_decimal:N}.

\texttt{\textbackslash fp\_use:N} * \texttt{\textbackslash fp\_use:c} * \texttt{\textbackslash fp\_use:n} \{\textit{fp var}\}  

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 \(\pm \infty\) and \texttt{nan} trigger an “invalid operation” exception. For a tuple, each item is converted using \texttt{\textbackslash fp\_to\_decimal: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 \texttt{\textbackslash fp\_to\_decimal:N}.

### 29.4 Floating point conditionals

\texttt{\textbackslash fp\_if\_exist_p:N} * \texttt{\textbackslash fp\_if\_exist_p:c} * \texttt{\textbackslash fp\_if\_exist:NTF} \{\textit{fp var}\} \{\langle true code\rangle \} \{\langle false code\rangle\}

Tests whether the \textit{fp var} is currently defined. This does not check that the \textit{fp var} really is a floating point variable.
\fp_compare_p:nNn \fp_compare_p:nNn \{\textit{fp expr}_1\} \{\textit{relation}\} \{\{\textit{fp expr}_2\}\}
\fp_compare:nNnTF \fp_compare:nNnTF \{\textit{fp expr}_1\} \{\textit{relation}\} \{\{\textit{fp expr}_2\}\} \{\{\textit{true code}\} \{\{\textit{false code}\}\}

Updated: 2012-05-08

Comparing the \textit{fp expr}_1 and the \textit{fp expr}_2, and returns \textit{true} if the \textit{relation} is obeyed. Two floating points \(x\) and \(y\) may obey four mutually exclusive relations: \(x < y\), \(x = y\), \(x > y\), or \(x \neq y\) (“not ordered”). The last case occurs exactly if one or both operands is \texttt{nan} or is a tuple, unless they are equal tuples. Note that a \texttt{nan} is distinct from any value, even another \texttt{nan}, hence \(x = x\) is not true for a \texttt{nan}. To test if a value is \texttt{nan}, compare it to an arbitrary number with the “not ordered” relation.

\begin{verbatim}
\fp_compare:nNnTF { <value> } ? { 0 }
 { } \% <value> is nan
 { } \% <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 \texttt{nan}). At present any other comparison with tuples yields \texttt{?} (not ordered). This is experimental.

This function is less flexible than \texttt{\fp_compare:nTF} but slightly faster. It is provided for consistency with \texttt{\int_compare:nNnTF} and \texttt{\dim_compare:nNnTF}.
Evaluates the \( \langle \text{fp exprs} \rangle \) as described for \( \langle \text{fp_eval:n} \rangle \) and compares consecutive result using the corresponding \( \langle \text{relation} \rangle \), namely it compares \( \langle \text{fp expr} 1 \rangle \) and \( \langle \text{fp expr} 2 \rangle \) using the \( \langle \text{relation} 1 \rangle \), then \( \langle \text{fp expr} 2 \rangle \) and \( \langle \text{fp expr} 3 \rangle \) using the \( \langle \text{relation} 2 \rangle \), until finally comparing \( \langle \text{fp expr} N \rangle \) and \( \langle \text{fp expr} N+1 \rangle \) using the \( \langle \text{relation} N \rangle \). The test yields true if all comparisons are true. Each floating point expression is evaluated only once. Contrarily to \( \langle \text{int_compare:nTF} \rangle \), all \( \langle \text{fp exprs} \rangle \) are computed, even if one comparison is false. Two floating points \( x \) and \( y \) may obey four mutually exclusive relations: \( x < y \), \( x = y \), \( x > y \), or \( x ? y \) (“not ordered”). The last case occurs exactly if one or both operands is nan or is a tuple, unless they are equal tuples. Each \( \langle \text{relation} \rangle \) can be any (non-empty) combination of \(<, \leq, \geq \), and \( ? \), plus an optional leading ! which negates the \( \langle \text{relation} \rangle \), with the restriction that the \( \langle \text{relation} \rangle \) may not start with \( ? \), as this symbol has a different meaning (in combination with : ) within floating point expressions. The comparison \( x \langle \text{relation} \rangle y \) is then true if the \( \langle \text{relation} \rangle \) does not start with ! and the actual relation (\(<, \leq, \geq \), or \( ? \)) between \( x \) and \( y \) appears within the \( \langle \text{relation} \rangle \), or on the contrary if the \( \langle \text{relation} \rangle \) starts with ! and the relation between \( x \) and \( y \) does not appear within the \( \langle \text{relation} \rangle \). Common choices of \( \langle \text{relation} \rangle \) include \( \geq \) (greater or equal), \( != \) (not equal), \( !? \) or \( <=> \) (comparable).

This function is more flexible than \( \langle \text{fp_compare:nNnTF} \rangle \) and only slightly slower.

Evaluates the \( \langle \text{fp expr} \rangle \) and tests whether the result is exactly nan. The test returns false for any other result, even a tuple containing nan.

### 29.5 Floating point expression loops

Places the \( \langle \text{code} \rangle \) in the input stream for \( \text{TeX} \) to process, and then evaluates the relationship between the two \( \langle \text{floating point expressions} \rangle \) as described for \( \langle \text{fp_compare:nNnTF} \rangle \). If the test is false then the \( \langle \text{code} \rangle \) is inserted into the input stream again and a loop occurs until the \( \langle \text{relation} \rangle \) is true.
\fp_do_while:nNnn \{ \text{\texttt{fp expr}}_{1} \} \{ \text{\texttt{relation}} \} \{ \text{\texttt{fp expr}}_{2} \} \{ \text{\texttt{code}} \}

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:nNnTF}. If the test is \texttt{true} then the \texttt{(code)} is inserted into the input stream again and a loop occurs until the \texttt{(relation)} is \texttt{false}.

\fp_until_do:nNnn \{ \text{\texttt{fp expr}}_{1} \} \{ \text{\texttt{relation}} \} \{ \text{\texttt{fp expr}}_{2} \} \{ \text{\texttt{code}} \}

Evaluates the relationship between the two \texttt{(floating point expressions)} as described for \texttt{\fp_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{false}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\fp_do_until:nn \{ \text{\texttt{fp expr}}_{1} \} \{ \text{\texttt{relation}} \} \{ \text{\texttt{fp expr}}_{2} \} \{ \text{\texttt{code}} \}

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:nNnTF}. 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}.

\fp_until_do:nn \{ \text{\texttt{fp expr}}_{1} \} \{ \text{\texttt{relation}} \} \{ \text{\texttt{fp expr}}_{2} \} \{ \text{\texttt{code}} \}

Evaluates the relationship between the two \texttt{(floating point expressions)} as described for \texttt{\fp_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{false}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\fp_while_do:nn \{ \text{\texttt{fp expr}}_{1} \} \{ \text{\texttt{relation}} \} \{ \text{\texttt{fp expr}}_{2} \} \{ \text{\texttt{code}} \}

Evaluates the relationship between the two \texttt{(floating point expressions)} as described for \texttt{\fp_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{true}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.

\fp_do_while:nn \{ \text{\texttt{fp expr}}_{1} \} \{ \text{\texttt{relation}} \} \{ \text{\texttt{fp expr}}_{2} \} \{ \text{\texttt{code}} \}

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:nNnTF}. If the test is \texttt{true} then the \texttt{(code)} is inserted into the input stream again and a loop occurs until the \texttt{(relation)} is \texttt{false}.

\fp_until_do:nn \{ \text{\texttt{fp expr}}_{1} \} \{ \text{\texttt{relation}} \} \{ \text{\texttt{fp expr}}_{2} \} \{ \text{\texttt{code}} \}

Evaluates the relationship between the two \texttt{(floating point expressions)} as described for \texttt{\fp_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{false}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{true}.

\fp_while_do:nn \{ \text{\texttt{fp expr}}_{1} \} \{ \text{\texttt{relation}} \} \{ \text{\texttt{fp expr}}_{2} \} \{ \text{\texttt{code}} \}

Evaluates the relationship between the two \texttt{(floating point expressions)} as described for \texttt{\fp_compare:nNnTF}, and then places the \texttt{(code)} in the input stream if the \texttt{(relation)} is \texttt{true}. After the \texttt{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \texttt{false}.
\texttt{\textbackslash  fp\_step\_function:nnn}\texttt{N} \texttt{\{initial value\}} \texttt{\{step\}} \texttt{\{final value\}} \texttt{\{function\}}  

This function first evaluates the \texttt{\{initial value\}}, \texttt{\{step\}} and \texttt{\{final value\}}, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The \texttt{\{function\}} is then placed in front of each \texttt{\{value\}} from the \texttt{\{initial value\}} to the \texttt{\{final value\}} in turn (using \texttt{\{step\}} between each \texttt{\{value\}}). The \texttt{\{step\}} must be non-zero. If the \texttt{\{step\}} is positive, the loop stops when the \texttt{\{value\}} becomes larger than the \texttt{\{final value\}}. If the \texttt{\{step\}} is negative, the loop stops when the \texttt{\{value\}} becomes smaller than the \texttt{\{final value\}}. The \texttt{\{function\}} should absorb one numerical argument. For example

\texttt{\textbackslash cs\_set:Npn \textbackslash my\_func:n \textbackslash #1 \{ [I saw \textbackslash #1] \textbackslash quad \} \textbackslash \textbackslash \textbackslash \textbackslash fp\_step\_function:nnn \textbackslash N \{ 1.0 \} \{ 0.1 \} \{ 1.5 \} \textbackslash \textbackslash my\_func:n}

would print

\begin{itemize}
  \item [I saw 1.0]
  \item [I saw 1.1]
  \item [I saw 1.2]
  \item [I saw 1.3]
  \item [I saw 1.4]
  \item [I saw 1.5]
\end{itemize}

\texttt{\TeX}Hacks note: Due to rounding, it may happen that adding the \texttt{\{step\}} to the \texttt{\{value\}} does not change the \texttt{\{value\}}; such cases give an error, as they would otherwise lead to an infinite loop.

\texttt{\textbackslash fp\_step\_inline:nnn}

This function first evaluates the \texttt{\{initial value\}}, \texttt{\{step\}} and \texttt{\{final value\}}, all of which should be floating point expressions evaluating to a floating point number, not a tuple. 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 \#1 replaced by the current \texttt{\{value\}}. Thus the \texttt{\{code\}} should define a function of one argument (\#1).

\texttt{\textbackslash fp\_step\_variable:nnn}\texttt{Nn}

This function first evaluates the \texttt{\{initial value\}}, \texttt{\{step\}} and \texttt{\{final value\}}, all of which should be floating point expressions evaluating to a floating point number, not a tuple. 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\}}.

29.6 Symbolic expressions

Floating point expressions support variables: these can only be set locally, so act like standard \LaTeX variables.

\texttt{\textbackslash fp\_new\_variable:n \{ A \}}
\texttt{\textbackslash \textbackslash \textbackslash fp\_set:Nn \l\_tmpb\_fp \{ 1 * \textbackslash sin(A) + 3**2 \}}
\texttt{\textbackslash \textbackslash \textbackslash fp\_show:n \{ \l\_tmpb\_fp \}}
\texttt{\textbackslash \textbackslash \textbackslash fp\_show:N \l\_tmpb\_fp}
\texttt{\textbackslash \textbackslash \textbackslash fp\_set\_variable:nn \{ A \} \{ \textbackslash pi/2 \}}
defines $A$ to be a variable, then defines $\l\_tmpb\_fp$ to stand for $1\cdot\sin(A)+9$ (note that $3\cdot2$ is evaluated, but the $1\cdot$ product is not simplified away). Until $\l\_tmpb\_fp$ is changed, $\fp\_show:N \l\_tmpb\_fp$ 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 $\fp\_show:n \{ \l\_tmpb\_fp \}$ will evaluate $\l\_tmpb\_fp$ and show 10. We then redefine $A$ to be 0: since $\l\_tmpb\_fp$ still stands for $1\cdot\sin(A)+9$, the value shown is then 9. Variables can be set with $\fp\_set\_variable:nn$ to arbitrary floating point expressions including other variables.

\begin{verbatim}
\fp\_new\_variable:n { \langle identifier \rangle }
\end{verbatim}

Declares the $\langle identifier \rangle$ as a variable, which allows it to be used in floating point expressions. For instance,

\begin{verbatim}
\fp\_new\_variable:n { A }
\fp\_show:n { A**2 - A + 1 }
\end{verbatim}

shows $((A^2)-A)+1)$. If the declaration was missing, the parser would complain about an "Unknown fp word 'A'". The $\langle identifier \rangle$ must consist entirely of Latin letters among [a-zA-Z].

\begin{verbatim}
\fp\_set\_variable:nn { \langle identifier \rangle } { \langle fp expr \rangle }
\end{verbatim}

Defines the $\langle identifier \rangle$ to stand in any further expression for the result of evaluating the $\langle floating point expression \rangle$ as much as possible. The result may contain other variables, which are then replaced by their values if they have any. For instance,

\begin{verbatim}
\fp\_new\_variable:n { A }
\fp\_new\_variable:n { B }
\fp\_new\_variable:n { C }
\fp\_set\_variable:nn { A } { 3 }
\fp\_set\_variable:nn { C } { A ** 2 + B * 1 }
\fp\_show:n { C + 4 }
\fp\_set\_variable:nn { A } { 4 }
\fp\_show:n { C + 4 }
\end{verbatim}

shows $((9+(B\cdot1))+4)$ twice: changing the value of $A$ to 4 does not alter $C$ because $A$ was replaced by its value 3 when evaluating $A\cdot2+B\cdot1$.
\texttt{\textbackslash fp\_clear\_variable:n} \quad \texttt{\textbackslash fp\_clear\_variable:n \{(identifier)\}}

Removes any value given by \texttt{\textbackslash fp\_set\_variable:nn} to the variable with this \texttt{(identifier)}.

For instance,

\begin{verbatim}
\texttt{\textbackslash fp\_new\_variable:n \{ A \}}
\texttt{\textbackslash fp\_set\_variable:nn \{ A \} \{ 3 \}}
\texttt{\textbackslash fp\_show:n \{ A ^ 2 \}}
\texttt{\textbackslash fp\_clear\_variable:n \{ A \}}
\texttt{\textbackslash fp\_show:n \{ A ^ 2 \}}
\end{verbatim}

shows 9, then \((A^2)\).

### 29.7 User-defined functions

It is possible to define new user functions which can be used inside the argument to \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} \quad \texttt{\textbackslash fp\_new\_function:n \{(identifier)\}}

Declares the \texttt{(identifier)} as a function, which allows it to be used in floating point expressions. For instance,

\begin{verbatim}
\texttt{\textbackslash fp\_new\_function:n \{ foo \}}
\texttt{\textbackslash fp\_show:n \{ foo ( 1 + 2 , foo(3), A ) ** 2 \}}
\end{verbatim}

shows \((foo(3, foo(3), A))^{(2)}\). 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} \quad \texttt{\textbackslash fp\_set\_function:nnn \{(identifier)\} \{(vars)\} \{(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)} \texttt{(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}
\texttt{\textbackslash fp\_new\_function:n \{ foo \}}
\texttt{\textbackslash fp\_set\_function:nnn \{ npow \} \{ a,b \} \{ a**b \}}
\texttt{\textbackslash 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} \quad \texttt{\textbackslash fp\_clear\_function:n \{(identifier)\}}

Removes any definition given by \texttt{\textbackslash fp\_set\_function:nnn} to the function with this \texttt{(identifier)}. 265
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 p1.

Updated: 2013-11-17

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

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

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 ±∞.
- **Underflow** occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ±0.
- **Invalid operation** occurs for operations with no defined outcome, for instance 0/0 or sin(∞), 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 ±∞.
- **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.

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

All occurrences of the \{exception\} (overflow, underflow, invalid_operation or division_by_zero) within the current group are treated as \{trap type\}, which can be

- **none**: the \{exception\} will be entirely ignored, and leave no trace;
- **flag**: the \{exception\} will turn the corresponding flag on when it occurs;
- **error**: additionally, the \{exception\} will halt the \TeX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.
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.

New: 2012-05-08
Updated: 2021-04-29

\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.

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

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 (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 sign \rangle: a possibly empty string of + and - characters;
- \langle significand \rangle: a non-empty string of digits together with zero or one dot;
- \langle exponent \rangle optionally: the character e 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 sign \rangle contains an even number of -, and - otherwise, hence, an empty \langle sign \rangle denotes a non-negative input. The stored significand is obtained from \langle significand \rangle by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input \langle significand \rangle has at most 16 digits. The stored \langle exponent \rangle is obtained by combining the input \langle exponent \rangle (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting \langle exponent \rangle is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by \(\pm \infty\)), or an underflow (resulting in \(\pm 0\)).
The result is thus ±0 if and only if \langle \textit{significand} \rangle contains no non-zero digit (\textit{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 \textit{significand} \rangle must be non-empty, so e1 and e-1 are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as e and should be input as exp(1) or \c_e_fp (which is faster).

Special numbers are input as follows:

- \texttt{inf} represents +∞, and can be preceded by any \langle \textit{sign} \rangle, yielding ±∞ as appropriate.
- \texttt{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 \texttt{nan}.
- Note that commands such as \texttt{\textbackslash infty}, \texttt{\pi}, or \texttt{\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 (\texttt{sin}, \texttt{ln}, \texttt{etc}).
- Binary ** and ^ (right associative).
- Unary +, −, !.
- Implicit multiplication by juxtaposition (2pi) when neither factor is in parentheses.
- Binary * and /, implicit multiplication by juxtaposition with parentheses (for instance 3(4+5)).
- Binary + and −.
- Comparisons >=, !=, <?, etc.
- Logical and, denoted by &&.
- Logical or, denoted by ||.
- Ternary operator ?: (right associative).
- Comma (to build tuples).
The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

\[
\frac{1}{2\pi} = \frac{1}{(2\pi)}, \\
\frac{1}{2\pi}(\pi + \pi) = (2\pi)^{-1}(\pi + \pi) \approx 1, \\
\sin 2\pi = \sin(2\pi) \neq 0, \\
2^{-2} \max(3, 5) = 2^2 \max(3, 5) = 20, \\
1\text{in}/1\text{cm} = (1\text{in})/(1\text{cm}) = 2.54.
\]

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 `false` if it is \(\pm 0\), and `true` otherwise, including when it is `nan` or a tuple such as \((0, 0)\). Tuples are only supported to some extent by operations that work with truth values (?:, ||, &&, !), by comparisons (\(!=\?), and by +, -, *, /. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a `nan` result.

\[
\texttt{\fp_eval:n \{ (operand_1) ? (operand_2) : (operand_3) \}}
\]

The ternary operator `?:` results in `⟨operand_2⟩` if `⟨operand_1⟩` is true (not `±0`), and `⟨operand_3⟩` if `⟨operand_1⟩` is false (`±0`). All three `⟨operands⟩` are evaluated in all cases; they may be tuples. The operator is right associative, hence

\[
\texttt{\fp_eval:n \{ 1 + 3 > 4 ? 1 : 2 + 4 > 5 ? 2 : 3 + 5 > 6 ? 3 : 4 \}}
\]

first tests whether \(1 + 3 > 4\); since this isn’t true, the branch following `:` is taken, and \(2 + 4 > 5\) is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

\[
\texttt{\fp_eval:n \{ (operand_1) || (operand_2) \}}
\]

If `⟨operand_1⟩` is true (not `±0`), use that value, otherwise the value of `⟨operand_2⟩`. Both `⟨operands⟩` are evaluated in all cases; they may be tuples. In `⟨operand_1⟩ || ⟨operand_2⟩ || ... || ⟨operands_n⟩`, the first true (nonzero) `⟨operand⟩` is used and if all are zero the last one (`±0`) is used.

\[
\texttt{\fp_eval:n \{ (operand_1) && (operand_2) \}}
\]

If `⟨operand_1⟩` is false (equal to `±0`), use that value, otherwise the value of `⟨operand_2⟩`. Both `⟨operands⟩` are evaluated in all cases; they may be tuples. In `⟨operand_1⟩ && ⟨operand_2⟩ && ... && ⟨operands_n⟩`, the first false (`±0`) `⟨operand⟩` is used and if none is zero the last one is used.
Each \( \text{relation} \) consists of a non-empty string of \(<\), \(=\), \(>\), and \(?\), optionally preceded by \(!\), and may not start with \(?\). This evaluates to +1 if all comparisons \( \langle \text{operand}_1 \rangle \text{ relation}_1 \langle \text{operand}_{i+1} \rangle \) are true, and +0 otherwise. All \( \text{operands} \) are evaluated (once) in all cases. See \texttt{fp_compare:nTF} for details.

- \( \text{fp_eval:n} \{ \langle \text{operand}_1 \rangle + \langle \text{operand}_2 \rangle \} \)
- \( \text{fp_eval:n} \{ \langle \text{operand}_1 \rangle - \langle \text{operand}_2 \rangle \} \)
- Computes the sum or the difference of its two \( \text{operands} \). The “invalid operation” exception occurs for \( \infty - \infty \). “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is \texttt{nan}.

- \( \text{fp_eval:n} \{ \langle \text{operand}_1 \rangle \times \langle \text{operand}_2 \rangle \} \)
- \( \text{fp_eval:n} \{ \langle \text{operand}_1 \rangle \div \langle \text{operand}_2 \rangle \} \)
- Computes the product or the ratio of its two \( \text{operands} \). The “invalid operation” exception occurs for \( \infty / \infty \), \( 0/0 \), or \( 0 \times \infty \). “Division by zero” occurs when dividing a finite non-zero number by \( \pm 0 \). “Underflow” and “overflow” occur when appropriate. When \( \langle \text{operand}_1 \rangle \) is a tuple and \( \langle \text{operand}_2 \rangle \) is a floating point number, each item of \( \langle \text{operand}_1 \rangle \) is multiplied or divided by \( \langle \text{operand}_2 \rangle \). Multiplication also supports the case where \( \langle \text{operand}_1 \rangle \) is a floating point number and \( \langle \text{operand}_2 \rangle \) a tuple. Other combinations yield an “invalid operation” exception and a \texttt{nan} result.

- \( \text{fp_eval:n} \{ + \langle \text{operand} \rangle \} \)
- \( \text{fp_eval:n} \{ - \langle \text{operand} \rangle \} \)
- \( \text{fp_eval:n} \{ ! \langle \text{operand} \rangle \} \)
- The unary \( + \) does nothing, the unary \( - \) changes the sign of the \( \langle \text{operand} \rangle \) (for a tuple, of all its components), and \( ! \langle \text{operand} \rangle \) evaluates to 1 if \( \langle \text{operand} \rangle \) is false (is \( \pm 0 \)) and 0 otherwise (this is the \texttt{not} boolean function). Those operations never raise exceptions.

- \( \text{fp_eval:n} \{ \langle \text{operand}_1 \rangle ^{\langle \text{operand}_2 \rangle} \} \)
- Raises \( \langle \text{operand}_1 \rangle \) to the power \( \langle \text{operand}_2 \rangle \). This operation is right associative, hence \( 2 \times 2 \times 3 \) equals \( 2^3 = 256 \). If \( \langle \text{operand}_1 \rangle \) is negative or \( \pm 0 \) then: the result’s sign is + if the \( \langle \text{operand}_2 \rangle \) is infinite and \( (-1)^p \) if the \( \langle \text{operand}_2 \rangle \) is \( p/5^q \) with \( p, q \) integers; the result is +0 if abs(\( \langle \text{operand}_1 \rangle \))\( ^{\langle \text{operand}_2 \rangle} \) evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising \( \pm 0 \) to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.

\texttt{abs} \( \text{fp_eval:n} \{ \text{abs} \langle \text{fp expr} \rangle \} \)

Computes the absolute value of the \( \langle \text{fp expr} \rangle \). If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also \texttt{fp_abs:n}.
\textbf{exp} \begin{verbatim} \texttt{exp(\langle fp expr\rangle)} \end{verbatim}
Computes the exponential of the \langle fp expr\rangle. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

\textbf{fact} \begin{verbatim} \texttt{fact(\langle fp expr\rangle)} \end{verbatim}
Computes the factorial of the \langle fp expr\rangle. If the \langle 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.

\textbf{ln} \begin{verbatim} \texttt{ln(\langle fp expr\rangle)} \end{verbatim}
Computes the natural logarithm of the \langle fp expr\rangle. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for \texttt{ln(-0)}.
“Division by zero” occurs when evaluating \texttt{ln(+0)} = $-\infty$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

\textbf{logb} \begin{verbatim} \texttt{logb(\langle fp expr\rangle)} \end{verbatim}
Determines the exponent of the \langle fp expr\rangle, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating \texttt{logb(\pm0)} = $-\infty$. Other special values are \texttt{logb(\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}.

\textbf{max} \begin{verbatim} \texttt{max(\langle fp expr1\rangle, \langle fp expr2\rangle, \ldots)} \end{verbatim}
\textbf{min} \begin{verbatim} \texttt{min(\langle fp expr1\rangle, \langle fp expr2\rangle, \ldots)} \end{verbatim}
Evaluates each \langle fp expr\rangle and computes the largest (smallest) of those. If any of the \langle 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.
round \( \textsf{round} \) \( \{ \textsf{round} \ ( \langle \textsf{fp} \ \textsf{expr} \rangle ) \} \)

\( \textsf{trunc} \) \( \{ \textsf{round} \ ( \langle \textsf{fp} \ \textsf{expr}_1 \rangle , \langle \textsf{fp} \ \textsf{expr}_2 \rangle ) \} \)

\( \textsf{ceil} \) \( \{ \textsf{round} \ ( \langle \textsf{fp} \ \textsf{expr}_1 \rangle , \langle \textsf{fp} \ \textsf{expr}_2 \rangle , \langle \textsf{fp} \ \textsf{expr}_3 \rangle ) \} \)

Only \( \textsf{round} \) accepts a third argument. Evaluates \( \langle \textsf{fp} \ \textsf{expr}_1 \rangle = x \) and \( \langle \textsf{fp} \ \textsf{expr}_2 \rangle = n \) and \( \langle \textsf{fp} \ \textsf{expr}_3 \rangle = t \) 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 \( \textsf{nan} \); if \( n = \textsf{nan} \), this yields \( \textsf{nan} \); if \( n \) is neither \( \pm \infty \) nor an integer, then an “invalid operation” exception is raised. When \( \langle \textsf{fp} \ \textsf{expr}_2 \rangle \) is omitted, \( n = 0 \), i.e., \( \langle \textsf{fp} \ \textsf{expr}_1 \rangle \) is rounded to an integer. The rounding direction depends on the function.

- \( \textsf{round} \) yields the multiple of \( 10^{-n} \) closest to \( x \), with ties (\( x \) half-way between two such multiples) rounded as follows. If \( t \) is \( \textsf{nan} \) (or not given) the even multiple is chosen (“ties to even”), if \( t = \pm 0 \) the multiple closest to 0 is chosen (“ties to zero”), if \( t \) is positive/negative the multiple closest to \( \infty/-\infty \) is chosen (“ties towards positive/negative infinity”).

- \( \textsf{floor} \) yields the largest multiple of \( 10^{-n} \) smaller or equal to \( x \) (“round towards negative infinity”);

- \( \textsf{ceil} \) yields the smallest multiple of \( 10^{-n} \) greater or equal to \( x \) (“round towards positive infinity”);

- \( \textsf{trunc} \) 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 \( \langle \textsf{fp} \ \textsf{expr}_2 \rangle < -9984 \)). If any operand is a tuple, “invalid operation” occurs.

\( \textsf{sign} \) \( \{ \textsf{sign} \ ( \langle \textsf{fp} \ \textsf{expr} \rangle ) \} \)

Evaluates the \( \langle \textsf{fp} \ \textsf{expr} \rangle \) and determines its sign: +1 for positive numbers and for \( +\infty \), −1 for negative numbers and for \( -\infty \), ±0 for ±0, and \( \textsf{nan} \) for \( \textsf{nan} \). If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

\( \textsf{sin} \) \( \{ \textsf{sin} \ ( \langle \textsf{fp} \ \textsf{expr} \rangle ) \} \)

\( \textsf{cos} \) \( \{ \textsf{cos} \ ( \langle \textsf{fp} \ \textsf{expr} \rangle ) \} \)

\( \textsf{tan} \) \( \{ \textsf{tan} \ ( \langle \textsf{fp} \ \textsf{expr} \rangle ) \} \)

\( \textsf{cot} \) \( \{ \textsf{cot} \ ( \langle \textsf{fp} \ \textsf{expr} \rangle ) \} \)

\( \textsf{csc} \) \( \{ \textsf{csc} \ ( \langle \textsf{fp} \ \textsf{expr} \rangle ) \} \)

\( \textsf{sec} \) \( \{ \textsf{sec} \ ( \langle \textsf{fp} \ \textsf{expr} \rangle ) \} \)

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the \( \langle \textsf{fp} \ \textsf{expr} \rangle \) given in radians. For arguments given in degrees, see \textsf{sind}, \textsf{cosd}, 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 sine, cosine, tangent, cotangent, cosecant, or secant of the \texttt{fp expr} given in degrees. For arguments given in radians, see \texttt{sin}, \texttt{cos}, \texttt{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 \texttt{fp expr} and returns the result in radians, in the range \([-\pi/2, \pi/2]\) for \texttt{asin} and \texttt{acsc} and \([0, \pi]\) for \texttt{acos} and \texttt{asec}. For a result in degrees, use \texttt{asind}, \texttt{acosd}, \texttt{acscd}, or \texttt{asecd}. If the argument of \texttt{asin} or \texttt{acos} lies outside the range \([-1, 1]\), or the argument of \texttt{acsc} or \texttt{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 \texttt{fp expr} and returns the result in degrees, in the range \([-\pi/2, \pi/2]\) for \texttt{asin} and \texttt{acsc} and \([0, \pi]\) for \texttt{acos} and \texttt{asec}. For a result in radians, use \texttt{asind}, \texttt{acosd}, \texttt{acscd}, or \texttt{asecd}. If the argument of \texttt{asin} or \texttt{acos} lies outside the range \([-1, 1]\), or the argument of \texttt{acsc} or \texttt{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.
\texttt{atan} \fp_eval:n \{ atan( \fp expr ) \} \\
\texttt{acot} \fp_eval:n \{ atan( \fp expr_1, \fp expr_2 ) \} \\
\texttt{sqrt} \fp_eval:n \{ sqrt( \fp expr ) \} \\
\texttt{atand} \fp_eval:n \{ atand( \fp expr_1, \fp expr_2 ) \} \\
\texttt{acotd} \fp_eval:n \{ acotd( \fp expr_1, \fp expr_2 ) \} \\
\texttt{atan} \fp_eval:n \{ atan( \fp expr ) \} \\
\texttt{acot} \fp_eval:n \{ acot( \fp expr ) \} \\
\texttt{sqrt} \fp_eval:n \{ sqrt( \fp expr ) \}

Those functions yield an angle in radians: \texttt{atand} and \texttt{acotd} are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the \texttt{(fp expr)}: 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 \((\fp expr_2, \fp expr_1)\): this is the arctangent of \(\fp expr_1 / \fp expr_2\), possibly shifted by \(\pi\) depending on the signs of \(\fp expr_1\) and \(\fp expr_2\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\fp expr_1, \fp expr_2)\), equal to the arccotangent of \(\fp expr_1 / \fp expr_2\), possibly shifted by \(\pi\). Both two-argument functions take values in the wider range \([-\pi, \pi]\). The ratio \(\fp expr_1 / \fp expr_2\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm 0\), 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.

\texttt{atan} \fp_eval:n \{ atand( \fp expr ) \} \\
\texttt{acot} \fp_eval:n \{ atand( \fp expr_1, \fp expr_2 ) \} \\
\texttt{atan} \fp_eval:n \{ acotd( \fp expr ) \} \\
\texttt{acot} \fp_eval:n \{ acotd( \fp expr_1, \fp expr_2 ) \} \\
\texttt{sqrt} \fp_eval:n \{ sqrt( \fp expr ) \}

Those functions yield an angle in degrees: \texttt{atan} and \texttt{acot} are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the \texttt{(fp expr)}: arctangent takes values in the range \([-90, 90]\), and arccotangent in the range \([0, 180]\). The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates \((\fp expr_2, \fp expr_1)\): this is the arctangent of \(\fp expr_1 / \fp expr_2\), possibly shifted by \(180\) depending on the signs of \(\fp expr_1\) and \(\fp expr_2\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\fp expr_1, \fp expr_2)\), equal to the arccotangent of \(\fp expr_1 / \fp expr_2\), possibly shifted by \(180\). Both two-argument functions take values in the wider range \([-180, 180]\). The ratio \(\fp expr_1 / \fp expr_2\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm 0\), 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.

\texttt{sqrt} \fp_eval:n \{ sqrt( \fp expr ) \}

Computes the square root of the \texttt{(fp expr)}. The “invalid operation” is raised when the \texttt{(fp expr)} is negative or is a tuple: no other exception can occur. Special values yield \(\sqrt{-0} = -0, \sqrt{+0} = +0, \sqrt{+\infty} = +\infty\) and \(\sqrt{\text{nan}} = \text{nan}\).
rand \[
\texttt{\fp_eval:n \{ rand() \}}
\]

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 \(\LaTeX\). The random seed can be queried using \texttt{\sys_rand_seed:} and set using \texttt{\sys_gset_rand_seed:n}.

\textbf{\TeX\ hackers note}: This is based on pseudo-random numbers provided by the engine's primitive \texttt{\pdfuniformdeviate} in pdf\TeX, p\TeX, up\TeX\ and \texttt{\uniformdeviate} in Lua\TeX\ and X\LaTeX. 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 \texttt{\uniformdeviate} to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

randint

\[
\texttt{\fp_eval:n \{ randint( \langle fp expr \rangle ) \}}
\]

\[
\texttt{\fp_eval:n \{ randint( \langle fp expr_1 \rangle, \langle fp expr_2 \rangle ) \}}
\]

New: 2016-12-05

Produces a pseudo-random integer between 1 and \(\langle fp\ \text{expr} \rangle\) or between \(\langle fp\ \text{expr}_1 \rangle\) and \(\langle fp\ \text{expr}_2 \rangle\) 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 \texttt{rand} for important comments on how these pseudo-random numbers are generated.

\[
\texttt{inf}
\]

The special values \(+\infty\), \(-\infty\), and \texttt{nan} are represented as \texttt{inf}, \texttt{-inf} and \texttt{nan} (see \texttt{\c_-inf_fp}, \texttt{\c_minus_inf_fp} and \texttt{\c_nan_fp}).

\[
\texttt{\Pi}
\]

The value of \(\pi\) (see \texttt{\c_pi_fp}).

\[
\texttt{deg}
\]

The value of 1\(^{\circ}\) in radians (see \texttt{\c_one_degree_fp}).
Those units of measurement are equal to their values in pt, namely

\[ 1 \text{ in} = 72.27 \text{ pt} \]
\[ 1 \text{ pt} = 1 \text{ pt} \]
\[ 1 \text{ pc} = 12 \text{ pt} \]
\[ 1 \text{ cm} = \frac{1}{2.54} \text{ in} = 28.45275590551181 \text{ pt} \]
\[ 1 \text{ mm} = \frac{1}{25.4} \text{ in} = 2.845275590551181 \text{ pt} \]
\[ 1 \text{ dd} = 0.376065 \text{ mm} = 1.0700956496063 \text{ pt} \]
\[ 1 \text{ cc} = 12 \text{ dd} = 12.84010277952756 \text{ pt} \]
\[ 1 \text{ nd} = 0.375 \text{ mm} = 1.06697834646693 \text{ pt} \]
\[ 1 \text{ nc} = 12 \text{ nd} = 12.80374015748031 \text{ pt} \]
\[ 1 \text{ bp} = \frac{1}{72} \text{ in} = 1.00375 \text{ pt} \]
\[ 1 \text{ sp} = 2^{-16} \text{ pt} = 1.52587890625 \times 10^{-5} \text{ pt}. \]

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

\begin{verbatim}
\texttt{true}
\texttt{false}
\end{verbatim}

\begin{verbatim}
\texttt{\fp_abs:n} \{\texttt{\langle fp expr\rangle}\}
Evaluates the \texttt{\langle fp expr\rangle} as described for \texttt{\fp_eval:n} and leaves the absolute value of the result in the input stream. If the argument is $\pm\infty$, \texttt{nan} or a tuple, “invalid operation” occurs. Within floating point expressions, \texttt{abs()} can be used; it accepts $\pm\infty$ and \texttt{nan} as arguments.
\end{verbatim}

\begin{verbatim}
\texttt{\fp_max:nn} \{\texttt{\langle fp expr\rangle}\} \{\texttt{\langle fp expr2\rangle}\}
\texttt{\fp_min:nn} \{\texttt{\langle fp expr\rangle}\} \{\texttt{\langle fp expr2\rangle}\}
Evaluates the \texttt{\langle fp expr\rangle} as described for \texttt{\fp_eval:n} and leaves the resulting larger (max) or smaller (min) value in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, \texttt{max()} and \texttt{min()} can be used.
\end{verbatim}

### 29.13 Disclaimer and roadmap

The package may break down if the escape character is among 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.

277
• Support signalling \texttt{nan}.
• Modulo and remainder, and rounding function \texttt{quantize} (and its friends analogous to \texttt{trunc}, \texttt{ceil}, \texttt{floor}).
\verb|\fp_format:nn {⟨fp expr⟩} ⟨⟨format⟩⟩|, but what should ⟨⟨format⟩⟩ be? More general pretty printing?
• Add \texttt{and}, \texttt{or}, \texttt{xor}? Perhaps under the names \texttt{all}, \texttt{any}, and \texttt{xor}?
• Add log_{x,b} for logarithm of \texttt{x} in base \texttt{b}.
• \texttt{hypot} (Euclidean length). Cartesian-to-polar transform.
• Hyperbolic functions \texttt{cosh}, \texttt{sinh}, \texttt{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{\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 \texttt{?:}, \texttt{!<>?}, \texttt{&&}, \texttt{||}, and \texttt{!}.
• Logarithms of numbers very close to 1 are inaccurate.
• When rounding towards \texttt{−∞}, \texttt{\dim_to_fp:n} {\texttt{0pt}} should return \texttt{−0}, not \texttt{+0}.
• The result of \texttt{(±0) + (±0)}, of \texttt{x + (−x)}, and of \texttt{(−x) + x} should depend on the rounding mode.
• \texttt{0e9999999999} gives a \texttt{TeX} “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 \texttt{29.12.1}, write a grammar.
• It would be nice if the parse auxiliaries for each operation were set up in the corresponding module, rather than centralizing in l3fp-parse.

• Some functions should get an _o ending to indicate that they expand after their result.

• More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.

• The code for the ternary set of functions is ugly.

• There are many - missing in the doc to avoid bad line-breaks.

• The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/([200x]+1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of $t$. However, we would then have to hard-code the logarithms of 44 small integers instead of 9.

• Improve notations in the explanations of the division algorithm (l3fp-basics).

• Understand and document \_\_fp\_basics\_pack\_weird\_low:NNNNw and \_\_fp\_-basics\_pack\_weird\_high:NNNNNNNNw better. Move the other basics_pack auxiliaries to l3fp-aux under a better name.

• Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.

• Add bibliography. Some of Kahan’s articles, some previous T\TeX\ fp packages, the international standards,…

• Also take into account the “inexact” exception?

• Support multi-character prefix operators (e.g., @/ or whatever)?
Chapter 30

The \texttt{l3fparray} package: 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 \texttt{fparray var} \{\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 \texttt{fparray var}
\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 \texttt{fparray var} \{\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 \texttt{fparray var}
\end{verbatim}

Sets all entries of the \texttt{floating point array variable} to +0. Assignments are always global.

\begin{verbatim}
\fparray_item:Nn \fparray_item:Nn \texttt{fparray var} \{\langle position\rangle\}
\fparray_item_to_tl:Nn \fparray_item_to_tl:Nn \texttt{fparray var} \{\langle position\rangle\}
\end{verbatim}

Applies \texttt{\fp_use:N} or \texttt{\fp_to_tl:N} (respectively) to the floating point entry stored at the (integer expression) \texttt{\langle position\rangle} in the \texttt{floating point array variable}. If the \texttt{\langle position\rangle} is not between 1 and the \texttt{\fparray_count:N}, an error occurs.
Chapter 31

The l3cctab package
Category code tables

A category code table enables rapid switching of all category codes in one operation. For LuaTeX, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables. The implementation of category code tables in expl3 also saves and restores the TeX \endlinechar primitive value, meaning they could be used for example to implement \ExplSyntaxOn.

31.1 Creating and initialising category code tables

\cctab_new:N \cctab_new:N \langle category code table \rangle
Creates a new \langle category code table \rangle variable or raises an error if the name is already taken. The declaration is global. The \langle category code table \rangle is initialised with the codes as used by iniTeX.

\cctab_const:Nn \cctab_const:Nn \langle category code table \rangle \{ \langle category code set up \rangle \}
Creates a new \langle category code table \rangle, applies (in a group) the \langle category code set up \rangle on top of iniTeX settings, then saves them globally as a constant table. The \langle category code set up \rangle can include a call to \cctab_select:N.

\cctab_gset:Nn \cctab_gset:Nn \langle category code table \rangle \{ \langle category code set up \rangle \}
Starting from the iniTeX category codes, applies (in a group) the \langle category code set up \rangle, then saves them globally in the \langle category code table \rangle. The \langle category code set up \rangle can include a call to \cctab_select:N.

\cctab_gsave_current:N \cctab_gsave_current:N \langle category code table \rangle
Saves the current prevailing category codes in the \langle category code table \rangle.
31.2 Using category code tables

\cctab_begin:N (category code table)
\cctab_begin: (category code table)
Switches locally the category codes in force to those stored in the \category code table. The prevailing codes before the function is called are added to a stack, for use with \cctab_end:. This function does not start a \TeX group.

\cctab_end:
Ends the scope of a \category code table started using \cctab_begin:N, returning the codes to those in force before the matching \cctab_begin:N was used. This must be used within the same \TeX group (and at the same \TeX group level) as the matching \cctab_begin:N.

\cctab_select:N (category code table)
\cctab_select: (category code table)
Selects the \category code table for the scope of the current group. This is in particular useful in the \setup arguments of \tl_set_rescan:Nnn, \tl_rescan:nn, \cctab_const:Nn, and \cctab_gset:Nn.

\cctab_item:Nn (category code table) {\int expr}
\cctab_item:cn {\int expr}
Determines the \character with character code given by the \int expr and expands to its category code specified by the \category code table.

31.3 Category code table conditionals

\cctab_if_exist_p:N (category code table)
\cctab_if_exist_p: (category code table)
\cctab_if_exist:NTF (category code table) {(true code)} {(false code)}
\cctab_if_exist:TNF (category code table) {(false code)} {(true code)}
\cctab_if_exist:NC (category code table)
\cctab_if_exist:N (category code table)
Tests whether the \category code table is currently defined. This does not check that the \category code table really is a category code table.

31.4 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 $\Theta$, which is retained as an “other” character. Sets the \endlinechar value to 32 (a space).

\c_document_cctab
Updated: 2020-07-08
Category code table for a standard \LaTeX document, as set by the \LaTeX kernel. In particular, the upper-half of the 8-bit range will be set to “active” with pdf\TeX only. No babel shorthands will be activated. Sets the \endlinechar value to 13 (normal line ending).
\c_initex_cctab

Category code table as set up by \LaTeX{X}.

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 32

The l3unicode package: Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. Most of the code here is internal, but there are a small set of public functions. These work with Unicode (codepoints) and are designed to give usable results with both Unicode-aware and 8-bit engines.
\codepoint_generate:nn \codepoint_generate:nn \{(codepoint)\} \{(catcode)\}

Generates one or more character tokens representing the \codepoint. With Unicode engines, exactly one character token will be generated, and this will have the \catcode specified as the second argument:

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the \codepoint. For all codepoints outside of the classical ASCII range, the generated character tokens will be active (category code 13); for codepoints in the ASCII range, the given \catcode will be used. To allow the result of this function to be used inside a expansion context, the result is protected by \exp_not:n.

\TeXhackers note: Users of (u)p\TeX note that these engines are treated as 8-bit in this context. In particular, for up\TeX, irrespective of the \catcode of the \codepoint, any value outside the ASCII range will result in a series of active bytes being generated.

\codepoint_str_generate:n \codepoint_str_generate:n \{(codepoint)\}

Generates one or more character tokens representing the \codepoint. With Unicode engines, exactly one character token will be generated. For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the \codepoint. All of the generated character tokens will be of category code 12, except any spaces (codepoint 32), which will be category code 10.
\codepoint_to_category:n \{\codepoint\}  
Expands to the Unicode general category identifier of the \codepoint. The general category identifier is a string made up of two letter characters, the first uppercase and the second lowercase. The uppercase letters divide codepoints into broader groups, which are then refined by the lowercase letter. For example, codepoints representing letters all have identifiers starting L, for example Lu (uppercase letter), Lt (titlecase letter), etc. Full details are available in the documentation provided by the Unicode Consortium: see https://www.unicode.org/reports/tr44/#General_Category_Values

\codepoint_to_nfd:n \{\codepoint\}  
Converts the \codepoint to the Unicode Normalization Form Canonical Decomposition. The generated character(s) will have the current category code as they would if typed in directly for Unicode engines; for 8-bit engines, active characters are used for all codepoints outside of the ASCII range.
Chapter 33

The \textbf{l3text} package: 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.

33.1 Expanding text

\begin{verbatim}
\textdeclareexpand{cmd}{replacement}
\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}
\textdeclareexpand{\'}{\exp_not:n { \'} }
\end{verbatim}
33.2 Case changing

\textlowercase{n} \textuppercase{n} \texttitlecase{all:n} \texttitlecase{first:n} \textlowercase{nn} \textuppercase{nn} \texttitlecase{all:nn} \texttitlecase{first:nn}

* Takes user input (text) first applies \textexpand{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 ij in Dutch which becomes IJ. There are two functions available for titlecasing: one which applies the change to each “word” and a second which only applies at the start of the input. (Here, “word” boundaries are spaces: at present, full Unicode word breaking is not attempted.)

Importantly, notice that these functions are intended for working with user text for typesetting. For case changing programmatic data see the l3str module and discussion there of \strlowercase{n}, \strupercase{n} and \strcasefold{n}.

Case changing does not take place within math mode material so for example

\textuppercase{n} \{ Some-text-$y = mx + c$-with-{Braces} \}

becomes

SOME TEXT $y = mx + c$ WITH {BRACES}

The first mandatory argument of commands listed in \l_text_case_exclude_arg-t1 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 PDfX, only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Locale-sensitive conversions are enabled using the ⟨BCP-47⟩ argument, and follow Unicode Consortium guidelines. Currently, the locale strings recognized for special handling are as follows.

- Armenian (hy and hy-x-yiwn) The setting hy maps the codepoint U+0587, the ligature of letters ech and yiwn, to the codepoints for capital ech and vew when uppercasing: this follows the spelling reform which is used in Armenia. The alternative hy-x-yiwn maps U+0587 to capital ech and yiwn on uppercasing (also the output if Armenian is not selected at all).

- Azeri and Turkish (az and tr). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lowercasing I-dot and introduced when upper casing i-dotless.

- German (de-x-eszett). An alternative mapping for German in which the lowercase Eszett maps to a großes Eszett.
- Greek (el). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. A variant el-x-iota is available which converts the ypos-grammeni (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-mediieval). The characters u and V are interchanged on case changing.

- Dutch (nl). Capitalisation of ij at the beginning of titlecased input produces IJ rather than Ij.

Determining whether non-letter characters at the start of text should count as the uppercase element is controllable. When \l_text_titlecase_check_letter_bool is true, codepoints which are not letters (Unicode general category L) are not changed, and only the first letter is uppercased. When \l_text_titlecase_check_letter_bool is false, the first codepoint is uppercased, irrespective of the general code of the character.

\text_declare_case_equivalent:Nn \text_declare_case_equivalent:cn

New: 2022-07-04

Declares that the (replacement) tokens should be used whenever the (cmd) (a single token) is encountered during case changing.

\text_declare_lowercase_mapping:nn \text_declare_lowercase_mapping:nn \text_declare_lowercase_mapping:nn \text_declare_lowercase_mapping:nn \text_declare_lowercase_mapping:nn \text_declare_lowercase_mapping:nn \text_declare_lowercase_mapping:nn

New: 2023-04-11
Updated: 2023-04-20

Declares that the (replacement) tokens should be used when case mapping the (codepoint), rather than the standard mapping given in the Unicode data files. The nnn version takes a BCP-47 tag, which can be used to specify that the customisation only applies to that locale.

\text_case_switch:nnnn \text_case_switch:nnnn \text_case_switch:nnnn \text_case_switch:nnnn

New: 2022-07-04

Context-sensitive function which will expand to one of the (normal), (upper), (lower) or (title) tokens depending on the current case changing operation. Outside of case changing, the (normal) tokens are produced. Within case changing, the appropriate mapping tokens are inserted.

290
33.3 Removing formatting from text

\text_purify:n \{\text\}
\text_purify:n ⋆

Takes user input \text and expands as described for \text_expand:n, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in \l_text_math_delims_tl or as the argument to commands listed in \l_text_math_arg_tl) is left contained in a pair of $ delimiters. Non-expandable functions present in the \text must either have a defined equivalent (see \text_declare_purify_equivalent:Nn) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

\text_declare_purify_equivalent:Nn \text_declare_purify_equivalent:Nn \text_declare_purify_equivalent:Nn \text_declare_purify_equivalent:Nn \text_declare_purify_equivalent:Nn \text_declare_purify_equivalent:Nn
\text_declare_purify_equivalent:Ne
\text_declare_purify_equivalent:Ne

Declares that the \text_replace tokens should be used whenever the \text_replace (a single token) is encountered. The \text_replace tokens should be expandable.

33.4 Control variables

\l_text_math_arg_tl

Lists commands present in the \text where the argument of the command should be treated as math mode material. The treatment here is similar to \l_text_math_delims_tl but for a command rather than paired delimiters.

\l_text_math_delims_tl

Lists pairs of tokens which delimit (in-line) math mode content; such content may be excluded from processing.

\l_text_case_exclude_arg_tl

Lists commands where the first mandatory argument is excluded from case changing.

\l_text_expand_exclude_tl

Lists commands which are excluded from expansion. This protection includes everything up to and including their first braced argument.

\l_text_titlecase_check_letter_bool

Controls how the start of titlecasing is handled: when \text true, the first letter in text is considered. The standard setting is \text true.
33.5 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 \langle text \rangle \{\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_function:nN✩

\text_map_inline:nn \langle text \rangle \{\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: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 34

The \texttt{l3box} package

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 \texttt{\hbox\_}, vertical mode with prefix \texttt{\vbox\_}, and the generic operations working in both modes with prefix \texttt{\box\_}. For instance, a new box variable containing the words “Hello, world!” (in a horizontal box) can be obtained by the following code.

\begin{verbatim}
\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, - world! }
\end{verbatim}

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 \texttt{l3box} are compatible with those of \LaTeX{} and plain \TeX{} and can be used interchangeably. The \texttt{l3box} commands to construct boxes, such as \texttt{\hbox:n} or \texttt{\hbox_set:Nn}, are “color-safe”, meaning that

\begin{verbatim}
\hbox:n { \color_select:n { blue } Hello, } - world!
\end{verbatim}

will result in “Hello,” taking the color blue, but “world!” remaining with the prevailing color outside the box.

34.1 Creating and initialising boxes

\begin{verbatim}
\box_new:N \box_new:N \box_new:c
\box_new:N \box_new:c
\box_clear:N \box_clear:c
\box_clear:N \box_clear:N \box_clear:N \box_clear:c
\box_gclear:N \box_gclear:N \box_gclear:N
\box_gclear:N \box_gclear:c
\end{verbatim}

Creates a new \texttt{\box} or raises an error if the name is already taken. The declaration is global. The \texttt{\box} is initially void.

Clears the content of the \texttt{\box} by setting the box equal to \texttt{\c_empty_box}. 

294
\texttt{\textbackslash box\_clear\_new:N}
\texttt{\textbackslash box\_clear\_new:c}
\texttt{\textbackslash box\_gclear\_new:N}
\texttt{\textbackslash box\_gclear\_new:c}

Ensures that the \texttt{box} exists globally by applying \texttt{\textbackslash box\_new:N} if necessary, then applies \texttt{\textbackslash box\_\_\_(g)\_clear\_N} to leave the \texttt{box} empty.

\texttt{\textbackslash box\_set\_eq:NN}
\texttt{\textbackslash box\_set\_eq:(cN|Nc|cc)}
\texttt{\textbackslash box\_gset\_eq:NN}
\texttt{\textbackslash box\_gset\_eq:(cN|Nc|cc)}

Sets the content of \texttt{box1} equal to that of \texttt{box2}.

\texttt{\textbackslash box\_if\_exist\_p:N}
\texttt{\textbackslash box\_if\_exist\_p:c}
\texttt{\textbackslash box\_if\_exist\_NTF}
\texttt{\textbackslash box\_if\_exist\_c}

Tests whether the \texttt{box} is currently defined. This does not check that the \texttt{box} really is a box.

\texttt{\textbackslash box\_use:N}
\texttt{\textbackslash 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{\textbackslash box\_move\_right:nn}
\texttt{\textbackslash 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{\textbackslash box\_use:N \textbackslash box} or a "raw" box specification such as \texttt{\textbackslash vbox:n \{ xyz \}}.

\texttt{\textbackslash box\_move\_up:nn}
\texttt{\textbackslash 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{\textbackslash box\_use:N \textbackslash box} or a "raw" box specification such as \texttt{\textbackslash vbox:n \{ xyz \}}.

\textbf{New: 2012-03-03}

\section*{34.2 Using boxes}

\texttt{\textbackslash box\_use:N}
\texttt{\textbackslash box\_use:c}

\textbf{TeXhackers note:} This is the TeX primitive \texttt{\textbackslash copy}.
34.3 Measuring and setting box dimensions

\texttt{\box_dp:N} \texttt{\box_dp:c}
Calculates the depth (below the baseline) of the \texttt{\langle box\rangle} in a form suitable for use in a \texttt{\langle dim expr\rangle}.

\textbf{TEXhackers note:} This is the \TeX{} primitive \texttt{\dp{}}.

\texttt{\box_ht:N} \texttt{\box_ht:c}
Calculates the height (above the baseline) of the \texttt{\langle box\rangle} in a form suitable for use in a \texttt{\langle dim expr\rangle}.

\textbf{TEXhackers note:} This is the \TeX{} primitive \texttt{\ht{}}.

\texttt{\box_wd:N} \texttt{\box_wd:c}
Calculates the width of the \texttt{\langle box\rangle} in a form suitable for use in a \texttt{\langle dim expr\rangle}.

\textbf{TEXhackers note:} This is the \TeX{} primitive \texttt{\wd{}}.

\texttt{\box_ht_plus_dp:N} \texttt{\box_ht_plus_dp:c}
Calculates the total vertical size (height plus depth) of the \texttt{\langle box\rangle} in a form suitable for use in a \texttt{\langle dim expr\rangle}.

\texttt{\box_set_dp:Nn} \texttt{\box_set_dp:cn} \texttt{\box_gset_dp:Nn} \texttt{\box_gset_dp:cn}
Set the depth (below the baseline) of the \texttt{\langle box\rangle} to the value of the \texttt{\langle dim expr\rangle}.

\textbf{Updated: 2019-01-22}

\texttt{\box_set_ht:Nn} \texttt{\box_set_ht:cn} \texttt{\box_gset_ht:Nn} \texttt{\box_gset_ht:cn}
Set the height (above the baseline) of the \texttt{\langle box\rangle} to the value of the \texttt{\langle dim expr\rangle}.

\textbf{Updated: 2019-01-22}

\texttt{\box_set_wd:Nn} \texttt{\box_set_wd:cn} \texttt{\box_gset_wd:Nn} \texttt{\box_gset_wd:cn}
Set the width of the \texttt{\langle box\rangle} to the value of the \texttt{\langle dim expr\rangle}.

\textbf{Updated: 2019-01-22}
34.4 Box conditionals

\box_if_empty_p:N \box_if_empty_p:c \box_if_empty:NTF \box_if_empty:c
Tests if ⟨box⟩ is an empty (equal to \c_empty_box).

\box_if_horizontal_p:N \box_if_horizontal_p:c \box_if_horizontal:NTF \box_if_horizontal:c
Tests if ⟨box⟩ is a horizontal box.

\box_if_vertical_p:N \box_if_vertical_p:c \box_if_vertical:NTF \box_if_vertical:c
Tests if ⟨box⟩ is a vertical box.

34.5 The last box inserted

\box_set_to_last:N \box_set_to_last:c \box_gset_to_last:N \box_gset_to_last:c
Sets the ⟨box⟩ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the ⟨box⟩ is always void as it is not possible to recover the last added item.

34.6 Constant boxes

\c_empty_box
This is a permanently empty box, which is neither set as horizontal nor vertical.

\TeXhackers note: At the \TeX level this is a void box.

34.7 Scratch boxes

\l_tmpa_box \l_tmpb_box
Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any \E\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_box \g_tmpb_box
Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any \E\TeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
34.8 Viewing box contents

\texttt{\textbackslash box\_show: N} \texttt{(box)} \texttt{\textbackslash box\_show: N} \\
\texttt{Updated: 2012-05-11}

Shows full details of the content of the (box) in the terminal.

\texttt{\textbackslash box\_show: Nnn} \texttt{(box)} \texttt{\{int expr\_1\}} \texttt{\{int expr\_2\}} \\
\texttt{\textbackslash box\_show: cnn}

New: 2012-05-11

Display the contents of (box) in the terminal, showing the first \texttt{int expr\_1} items of the box, and descending into \texttt{int expr\_2} group levels.

\texttt{\textbackslash box\_log: N} \texttt{(box)} \texttt{\textbackslash box\_log: N}

Write full details of the content of the (box) to the log.

\texttt{\textbackslash box\_log: Nnn} \texttt{(box)} \texttt{\{int expr\_1\}} \texttt{\{int expr\_2\}} \\
\texttt{\textbackslash box\_log: cnn}

New: 2012-05-11

Writes the contents of (box) to the log, showing the first \texttt{int expr\_1} items of the box, and descending into \texttt{int expr\_2} group levels.

34.9 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.

34.10 Horizontal mode boxes

\texttt{\textbackslash hbox: n} \texttt{(contents)} \\
\texttt{Updated: 2017-04-05}

Typesets the (contents) into a horizontal box of natural width and then includes this box in the current list for typesetting.

\texttt{\textbackslash hbox\_to\_wd: nn} \texttt{(dim expr)} \texttt{(contents)} \\
\texttt{Updated: 2017-04-05}

Typesets the (contents) into a horizontal box of width (dim expr) and then includes this box in the current list for typesetting.

\texttt{\textbackslash hbox\_to\_zero: n} \texttt{(contents)} \\
\texttt{Updated: 2017-04-05}

Typesets the (contents) into a horizontal box of zero width and then includes this box in the current list for typesetting.

\texttt{\textbackslash hbox\_set: Nn} \texttt{(box)} \texttt{(contents)} \\
\texttt{\textbackslash hbox\_set: cn} \\
\texttt{\textbackslash hbox\_gset: Nn} \\
\texttt{\textbackslash hbox\_gset: cn}

Updated: 2017-04-05

Typesets the (contents) at natural width and then stores the result inside the (box).
\hbox_set_to_wd:Nnn \hbox_set_to_wd:cnm \hbox_gset_to_wd:Nnn \hbox_gset_to_wd:cnm

Typesets the \texttt{\langle contents\rangle} to the width given by the \texttt{\langle dim expr\rangle} and then stores the result inside the \texttt{\langle box\rangle}.

\hbox_overlap_center:n \hbox_overlap_left:n \hbox_overlap_right:n

Updated: 2017-04-05

\hbox_set:Nw \hbox_set:Nn \hbox_set_end: \hbox_gset:Nw \hbox_gset:Nn \hbox_gset_end:

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

\hbox_set_to_wd:Nnw \hbox_set_to_wd:cnw \hbox_gset_to_wd:Nnw \hbox_gset_to_wd:cnw

Updated: 2017-06-08

\hbox_unpack:N \hbox_unpack:Nn \hbox_unpack:cn

Unpacks the content of the horizontal \texttt{\langle box\rangle}, retaining any stretching or shrinking applied when the \texttt{\langle box\rangle} was set.

\textbf{\TeX hackers note:} This is the \TeX\ primitive \texttt{\unhcopy}.

34.11 \textbf{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
\[ \text{top} \] 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.

\texttt{vbox:nn} \{\texttt{contents}\}

Typesets the \texttt{⟨contents⟩} into a vertical box of natural height and includes this box in the current list for typesetting.

\texttt{vbox_top:nn} \{\texttt{contents}\}

Typesets the \texttt{⟨contents⟩} into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the first item added to the box.

\texttt{vbox_to_ht:nn} \{(\texttt{dim expr})\} \{\texttt{contents}\}

Typesets the \texttt{⟨contents⟩} into a vertical box of height \texttt{⟨dim expr⟩} and then includes this box in the current list for typesetting.

\texttt{vbox_to_zero:nn} \{\texttt{contents}\}

Typesets the \texttt{⟨contents⟩} into a vertical box of zero height and then includes this box in the current list for typesetting.

\texttt{vbox_set:Nn} \texttt{⟨box⟩} \{\texttt{contents}\}

Typesets the \texttt{⟨contents⟩} at natural height and then stores the result inside the \texttt{⟨box⟩}.

\texttt{vbox_set_top:Nn} \texttt{⟨box⟩} \{\texttt{contents}\}

Typesets the \texttt{⟨contents⟩} at natural height and then stores the result inside the \texttt{⟨box⟩}. The baseline of the box is equal to that of the first item added to the box.

\texttt{vbox_set_to_ht:Nnn} \texttt{⟨box⟩} \{(\texttt{dim expr})\} \{\texttt{contents}\}

Typesets the \texttt{⟨contents⟩} to the height given by the \texttt{⟨dim expr⟩} and then stores the result inside the \texttt{⟨box⟩}.

\texttt{vbox_set:Nw} \{\texttt{contents}\} \texttt{vbox_set_end:}

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

\vbox_set_split_to_ht:NNn \vbox_set_split_to_ht:NNn \langle box_1 \rangle \langle box_2 \rangle \{\langle dim expr \rangle\}
\vbox_set_split_to_ht:hNn \vbox_set_split_to_ht:hNn \vbox_set_split_to_ht:cNn \vbox_set_split_to_ht:cNn

Sets \langle box_1 \rangle to contain material to the height given by the \langle dim expr \rangle by removing content from the top of \langle box_2 \rangle (which must be a vertical box).

\vbox_unpack:N \vbox_unpack:N \langle box \rangle
\vbox_unpack:c

Unpacks the content of the vertical \langle box \rangle, retaining any stretching or shrinking applied when the \langle box \rangle was set.

\TeX{}hackers note: This is the \TeX{} primitive \texttt{unvcopy}.

### 34.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other expl3 variable. However, for efficiency reasons, it is also useful to have functions which drop box contents on use. When a box is dropped, the box becomes empty at the group level where the box was originally set rather than necessarily at the current group level. For example, with

\hbox_set:Nn \l_tmpa_box \{ A \}
\group_begin:
\hbox_set:Nn \l_tmpa_box \{ B \}
\group_begin:
\box_use_drop:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box

the first use of \texttt{\box_show:N} will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of \texttt{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.
\texttt{\textbackslash box\_use\_drop:N} \texttt{\textbackslash box\_use\_drop:N} \langle box \rangle
\texttt{\textbackslash box\_use\_drop:c}

Inserts the current content of the \langle box \rangle 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\hackers\ note:} This is the \TeX\ primitive \texttt{\textbackslash box}.

\texttt{\textbackslash box\_set\_eq\_drop:NN} \texttt{\textbackslash box\_set\_eq\_drop:NN} \langle box_1 \rangle \langle box_2 \rangle
\texttt{\textbackslash box\_set\_eq\_drop:(cN\textbackslash N|c|cc)}

Sets the content of \langle box_1 \rangle equal to that of \langle box_2 \rangle, then drops \langle box_2 \rangle.

\texttt{\textbackslash box\_gset\_eq\_drop:NN} \texttt{\textbackslash box\_gset\_eq\_drop:NN} \langle box_1 \rangle \langle box_2 \rangle
\texttt{\textbackslash box\_gset\_eq\_drop:(cN\textbackslash N|c|cc)}

Sets the content of \langle box_1 \rangle globally equal to that of \langle box_2 \rangle, then drops \langle box_2 \rangle.

\texttt{\textbackslash hbox\_unpack\_drop:N} \texttt{\textbackslash hbox\_unpack\_drop:N} \langle box \rangle
\texttt{\textbackslash hbox\_unpack\_drop:c}
\texttt{\textbackslash new: 2019-01-17}

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\hackers\ note:} This is the \TeX\ primitive \texttt{\unhbox}.

\texttt{\textbackslash vbox\_unpack\_drop:N} \texttt{\textbackslash vbox\_unpack\_drop:N} \langle box \rangle
\texttt{\textbackslash vbox\_unpack\_drop:c}
\texttt{\textbackslash new: 2019-01-17}

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\hackers\ note:} This is the \TeX\ primitive \texttt{\unvbox}.

### 34.13 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 \texttt{\{box\}} to fit within the given \texttt{(x-size)} (horizontally) and \texttt{(y-size)} (vertically); both of the sizes are dimension expressions. The \texttt{(y-size)} is the height only: it does not include any depth. The updated \texttt{\{box\}} is an hbox, irrespective of the nature of the \texttt{\{box\}} before the resizing is applied. The final size of the \texttt{\{box\}} is the smaller of \texttt{(x-size)} and \texttt{(y-size)}, i.e. the result fits within the dimensions specified. Negative sizes cause the material in the \texttt{\{box\}} to be reversed in direction, but the reference point of the \texttt{\{box\}} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\{box\}} having a depth dependent on the height of the original and \textit{vice versa}.

Resizes the \texttt{\{box\}} to fit within the given \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{\{box\}} is an hbox, irrespective of the nature of the \texttt{\{box\}} before the resizing is applied. The final size of the \texttt{\{box\}} is the smaller of \texttt{(x-size)} and \texttt{(y-size)}, i.e. the result fits within the dimensions specified. Negative sizes cause the material in the \texttt{\{box\}} to be reversed in direction, but the reference point of the \texttt{\{box\}} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\{box\}} having a depth dependent on the height of the original and \textit{vice versa}.

Resizes the \texttt{\{box\}} 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 height only: it does not include any depth. The updated \texttt{\{box\}} is an hbox, irrespective of the nature of the \texttt{\{box\}} before the resizing is applied. A negative \texttt{(y-size)} causes the material in the \texttt{\{box\}} to be reversed in direction, but the reference point of the \texttt{\{box\}} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\{box\}} having a depth dependent on the height of the original and \textit{vice versa}.
Resizes the \texttt{\textbackslash box} 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{\textbackslash box} is an \texttt{hbox}, irrespective of the nature of the \texttt{\textbackslash box} before the resizing is applied. A negative \texttt{(y-size)} causes the material in the \texttt{\textbackslash box} to be reversed in direction, but the reference point of the \texttt{\textbackslash box} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\textbackslash box} having a depth dependent on the height of the original and \textit{vice versa}.

Resizes the \texttt{\textbackslash box} to \texttt{(x-size)} (horizontally), scaling the vertical size by the same amount: \texttt{(x-size)} is a dimension expression. The updated \texttt{\textbackslash box} is an \texttt{hbox}, irrespective of the nature of the \texttt{\textbackslash box} before the resizing is applied. A negative \texttt{(x-size)} causes the material in the \texttt{\textbackslash box} to be reversed in direction, but the reference point of the \texttt{\textbackslash box} is unchanged. Thus a negative \texttt{(x-size)} results in the \texttt{\textbackslash box} having a depth dependent on the height of the original and \textit{vice versa}.

Resizes the \texttt{\textbackslash box} 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{\textbackslash box} is an \texttt{hbox}, irrespective of the nature of the \texttt{\textbackslash box} before the resizing is applied. Negative sizes cause the material in the \texttt{\textbackslash box} to be reversed in direction, but the reference point of the \texttt{\textbackslash box} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\textbackslash box} having a depth dependent on the height of the original and \textit{vice versa}.

Resizes the \texttt{\textbackslash box} 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{\textbackslash box} is an \texttt{hbox}, irrespective of the nature of the \texttt{\textbackslash box} before the resizing is applied. Negative sizes cause the material in the \texttt{\textbackslash box} to be reversed in direction, but the reference point of the \texttt{\textbackslash box} is unchanged. Thus a negative \texttt{(y-size)} results in the \texttt{\textbackslash box} having a depth dependent on the height of the original and \textit{vice versa}.
\texttt{\texttt{\box\rotate:Nn \box\rotate:cn}}
\texttt{\box\grotate:Nn \box\grotate:cn}

Rotates the \texttt{\langle box\rangle} by \texttt{\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 \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\langle box\rangle} before the rotation is applied.

\texttt{\texttt{\box\scale:Nnn \box\scale:cn}}
\texttt{\box\gscale:Nnn \box\gscale:cn}

Scales the \texttt{\langle box\rangle} by factors \texttt{\langle x-scale\rangle} and \texttt{\langle y-scale\rangle} in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\langle box\rangle} before the scaling is applied. Negative scalings 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{\langle y-scale\rangle} results in the \texttt{\langle box\rangle} having a depth dependent on the height of the original and \textit{vice versa}.

\texttt{\texttt{\box\set\clipped:N \box\set\clipped:c}}
\texttt{\box\gset\clipped:N \box\gset\clipped:c}

Clips the \texttt{\langle box\rangle} in the output so that only material inside the bounding box is displayed in the output. The updated \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\langle box\rangle} before the clipping is applied. Additional box levels are also generated by this operation.

\texttt{\texttt{\box\set\trim:Nnnnn \box\set\trim:cnnnn}}
\texttt{\box\gset\trim:Nnnnn \box\gset\trim:cnnnn}

Adjusts the bounding box of the \texttt{\langle box\rangle} \texttt{\langle left\rangle} is removed from the left-hand edge of the bounding box, \texttt{\langle right\rangle} from the right-hand edge and so fourth. All adjustments are \texttt{\langle dim\ exprs\rangle}. Material outside of the bounding box is still displayed in the output unless \texttt{\box\set\clipped:N} is subsequently applied. The updated \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\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.

\texttt{\texttt{\box\set\viewport:Nnnnn \box\set\viewport:cnnnn}}
\texttt{\box\gset\viewport:Nnnnn \box\gset\viewport:cnnnn}

Adjusts the bounding box of the \texttt{\langle box\rangle} such that it has lower-left co-ordinates \texttt{\langle llx\rangle}, \texttt{\langle lly\rangle} and upper-right co-ordinates \texttt{\langle urx\rangle}, \texttt{\langle ury\rangle}. All four co-ordinate positions are \texttt{\langle dim\ exprs\rangle}. Material outside of the bounding box is still displayed in the output unless \texttt{\box\set\clipped:N} is subsequently applied. The updated \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\langle box\rangle} before the viewport operation is applied. Additional box levels are also generated by this operation.

### 34.14 Viewing part of a box

\texttt{\texttt{\box\set\clipped:N \box\set\clipped:c}}

Clips the \texttt{\langle box\rangle} in the output so that only material inside the bounding box is displayed in the output. The updated \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\langle box\rangle} before the clipping is applied. Additional box levels are also generated by this operation.

\texttt{\texttt{\box\set\trim:Nnnnn \box\set\trim:cnnnn}}

Adjusts the bounding box of the \texttt{\langle box\rangle} \texttt{\langle left\rangle} is removed from the left-hand edge of the bounding box, \texttt{\langle right\rangle} from the right-hand edge and so fourth. All adjustments are \texttt{\langle dim\ exprs\rangle}. Material outside of the bounding box is still displayed in the output unless \texttt{\box\set\clipped:N} is subsequently applied. The updated \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\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.

\texttt{\texttt{\box\set\viewport:Nnnnn \box\set\viewport:cnnnn}}

Adjusts the bounding box of the \texttt{\langle box\rangle} such that it has lower-left co-ordinates \texttt{\langle llx\rangle}, \texttt{\langle lly\rangle} and upper-right co-ordinates \texttt{\langle urx\rangle}, \texttt{\langle ury\rangle}. All four co-ordinate positions are \texttt{\langle dim\ exprs\rangle}. Material outside of the bounding box is still displayed in the output unless \texttt{\box\set\clipped:N} is subsequently applied. The updated \texttt{\langle box\rangle} is an \texttt{hbox}, irrespective of the nature of the \texttt{\langle box\rangle} before the viewport operation is applied. Additional box levels are also generated by this operation.
34.15 Primitive box conditionals

\begin{verbatim}
\ifhbox:N \ifhbox:N (box)
  \{true code\}
\else:
  \{false code\}
\fi:
Tests is (box) is a horizontal box.

\TeXhackers note: This is the \TeX primitive \ifhbox.
\end{verbatim}

\begin{verbatim}
\ifvbox:N \ifvbox:N (box)
  \{true code\}
\else:
  \{false code\}
\fi:
Tests is (box) is a vertical box.

\TeXhackers note: This is the \TeX primitive \ifvbox.
\end{verbatim}

\begin{verbatim}
\ifboxempty:N \ifboxempty:N (box)
  \{true code\}
\else:
  \{false code\}
\fi:
Tests is (box) is an empty (void) box.

\TeXhackers note: This is the \TeX primitive \ifvoid.
\end{verbatim}
Chapter 35

The \texttt{l3coffins} package

Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the \texttt{xcoffins} module (in the \texttt{l3experimental} bundle).

35.1 Creating and initialising coffins

\begin{verbatim}
\texttt{\textbackslash coffin\_new\_N} \langle\texttt{coffin}\rangle
\texttt{\textbackslash coffin\_new\_c}
\end{verbatim}

Creates a new \texttt{\langle\texttt{coffin}\rangle} or raises an error if the name is already taken. The declaration is global. The \texttt{\langle\texttt{coffin}\rangle} is initially empty.

\begin{verbatim}
\texttt{\textbackslash coffin\_clear\_N} \langle\texttt{coffin}\rangle
\texttt{\textbackslash coffin\_clear\_c}
\texttt{\textbackslash coffin\_gclear\_N}
\texttt{\textbackslash coffin\_gclear\_c}
\end{verbatim}

Clears the content of the \texttt{\langle\texttt{coffin}\rangle}.

\begin{verbatim}
\texttt{\textbackslash coffin\_set\_eq\_NN} \langle\texttt{coffin}_1\rangle \langle\texttt{coffin}_2\rangle
\texttt{\textbackslash coffin\_set\_eq\:(N|c|cc)}
\texttt{\textbackslash coffin\_gset\_eq\_NN}
\end{verbatim}

Sets both the content and poles of \texttt{\langle\texttt{coffin}_1\rangle} equal to those of \texttt{\langle\texttt{coffin}_2\rangle}.

\begin{verbatim}
\texttt{\textbackslash coffin\_if\_exist\_p\_N} \langle\texttt{coffin}\rangle
\texttt{\textbackslash coffin\_if\_exist\_p\_c} \langle\texttt{coffin}\rangle
\texttt{\textbackslash coffin\_if\_exist\_NTF\_N} \langle\texttt{coffin}\rangle
\texttt{\textbackslash coffin\_if\_exist\_c\_c}
\end{verbatim}

Tests whether the \texttt{\langle\texttt{coffin}\rangle} is currently defined.
35.2 Setting coffin content and poles

\hcoffin_set:Nn \hcoffin_set:cn \hcoffin_gset:Nn \hcoffin_gset:cn

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.

\hcoffin_set:Nw \hcoffin_set:cw \hcoffin_gset:Nw \hcoffin_gset:cw

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.

\vcoffin_set:Nnn \vcoffin_set:cnn \vcoffin_gset:Nnn \vcoffin_gset:cnn

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.

\vcoffin_set:Nnw \vcoffin_set:cnw \vcoffin_gset:Nnw \vcoffin_gset:cnw

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.

\coffin_set_horizontal_pole:Nnn \coffin_set_horizontal_pole:cnn \coffin_gset_horizontal_pole:Nnn \coffin_gset_horizontal_pole:cnn

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 \langle pole \rangle to run vertically through the \langle coffin \rangle. The \langle pole \rangle is placed at the \langle offset \rangle from the left-hand edge of the bounding box of the \langle coffin \rangle. The \langle offset \rangle should be given as a dimension expression.

\textbf{35.3 Coffin affine transformations}

Resizes the \langle coffin \rangle to \langle width \rangle and \langle total-height \rangle, both of which should be given as dimension expressions.

Rotates the \langle coffin \rangle by the given \langle angle \rangle (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 \langle coffin \rangle by a factors \langle x-scale \rangle and \langle y-scale \rangle in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.
35.4 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, \textit{i.e.} \(\text{coffin}_2\) can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating \(\text{handle}_1\), the point of intersection of \(\langle\text{coffin}_1\rangle - \langle\text{pole}_1\rangle\) and \(\langle\text{coffin}_2\rangle - \langle\text{pole}_2\rangle\), and \(\langle\text{handle}_2\rangle\), the point of intersection of \(\langle\text{coffin}_1\rangle - \langle\text{pole}_1\rangle\) and \(\langle\text{coffin}_2\rangle - \langle\text{pole}_2\rangle\). \(\text{coffin}_2\) 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 \(x\text{-offset}\) and \(y\text{-offset}\). The two offsets should be given as dimension expressions.

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 \(\langle\text{coffin}_1\rangle - \langle\text{pole}_1\rangle\) and \(\langle\text{coffin}_2\rangle - \langle\text{pole}_2\rangle\), and \(\langle\text{handle}_2\rangle\), the point of intersection of \(\langle\text{coffin}_1\rangle - \langle\text{pole}_1\rangle\) and \(\langle\text{coffin}_2\rangle - \langle\text{pole}_2\rangle\). \(\text{coffin}_2\) is then attached to \(\langle\text{coffin}_1\rangle\) such that the relationship between \(\langle\text{handle}_1\rangle\) and \(\langle\text{handle}_2\rangle\) is described by the \(x\text{-offset}\) and \(y\text{-offset}\). The two offsets should be given as dimension expressions.

Typsetting 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 \(x\text{-offset}\) and \(y\text{-offset}\). The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

35.5 Measuring coffins

Calculates the depth (below the baseline) of the \(\langle\text{coffin}\rangle\) in a form suitable for use in a \(\langle\text{dim expr}\rangle\).
Calculates the height (above the baseline) of the \texttt{coffin} in a form suitable for use in a \texttt{dim expr}.

Calculates the width of the \texttt{coffin} in a form suitable for use in a \texttt{dim expr}.

### 35.6 Coffin diagnostics

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

This function first calculates the \texttt{handle} for the \texttt{coffin} as defined by the intersection of \texttt{pole}$_1$ and \texttt{pole}$_2$. It then marks the position of the \texttt{handle} on the \texttt{coffin}. The \texttt{handle} are labelled as part of this process: the location of the \texttt{handle} and the label are both printed in the \texttt{color} specified.

This function shows the structural information about the \texttt{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.

This function writes the structural information about the \texttt{coffin} in the log file. See also \texttt{coffin\_show\_structure:N} which displays the result in the terminal.

Shows full details of poles and contents of the \texttt{coffin} in the terminal or log file. See \texttt{coffin\_show\_structure:N} and \texttt{box\_show:N} to show separately the pole structure and the contents.
35.7 Constants and variables

A permanently empty 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.

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 36

The l3color package
Color support

36.1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

\color_group_begin: \color_group_end: ... \color_group_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.

36.2 Color models

A color model is a way to represent sets of colors. Different models are particularly suitable for different output methods, e.g. screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of axes which define a color space. In contrast, various proprietary models are available which define spot colors (more formally separations).

Core models are used to pass color information to output; these are “native” to l3color. Core models use real numbers in the range \([0, 1]\) to represent values. The core models supported here are

- **gray** Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
- **rgb** Red-green-blue color, with three axes, one for each of the components
• **cmyk** Cyan-magenta-yellow-black color, with four axes, one for each of the components.

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based: see below for more detail. The supported interface models are

• **Gray** Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)

• **hsb** Hue-saturation-brightness color, with three axes, all real values in the range 
\[0, 1\] for hue saturation and brightness

• **Hsb** Hue-saturation-brightness color, with three axes, integer in the range \([0, 360]\) for hue, real values in the range \([0, 1]\) for saturation and brightness

• **HSB** Hue-saturation-brightness color, with three axes, integers in the range \([0, 240]\) for hue, saturation and brightness

• **HTML** HTML format representation of RGB color given as a single six-digit hexadecimal number

• **RGB** Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255

• **wave** Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as rgb.

Finally, there are a small number of models which are parsed to allow data transfer from xcolor but which should not be used by end-users. These are

• **cmy** Cyan-magenta-yellow color with three axes, one for each of the components; converted to cmyk

• **tHsb** “Tuned” hue-saturation-brightness color with three axes, integer in the range \([0, 360]\) for hue, real values in the range \([0, 1]\) for saturation and brightness; converted to rgb using the standard tuning map defined by xcolor

• **&spot** Spot color tint with one value; treated as a gray tint as spot color data is not available for extraction

To allow parsing of data from xcolor, any leading model up the first : will be discarded; the approach of selecting an internal form for data is not used in l3color.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 36.9 for more detail of color support for additional models.

When color is selected by model, the \langle values\rangle 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{} 2e xcolor package and in the PostScript Language Reference Manual, published by Addison-Wesley.
36.3 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

\texttt{red!50!green}

is a mixture of 50\% red and 50\% green. A trailing value is interpreted as implicitly followed by \texttt{!white}, and so

\texttt{red!25}

specifies 25\% red mixed with 75\% white.

Where the models for the mixed colors are different, the model of the first color is used. Thus

\texttt{red!50!cyan}

will result in a color specification using the \texttt{rgb} model, made up of 50\% red and 50\% of cyan \textit{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

\texttt{black!50!red}

has the same result as

\texttt{red!50!black}

(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

\texttt{cyan!50!magenta!10!yellow}

the sub-expression

\texttt{cyan!50!magenta}

is first evaluated to give an intermediate color specification, before the second step

\texttt{<intermediate>!10!yellow}

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

\texttt{.!50}

to mean a mixture of 50\% of current color with white.

(Color expressions supported here are a subset of those provided by the \texttt{FPX2e xcolor} package. At present, only such features as are clearly useful have been added here.)
36.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names black, white, red, green, blue, cyan, magenta and yellow have special meaning and should not be redefined. Color names should be made up of letters, numbers and spaces only: other characters are reserved for use in color expressions. In particular, . represents the current color at the start of a color expression.

\color_set:nn \color_set:nn {\langle name\rangle} \{\langle color expression\rangle\}

Evaluates the \langle color expression\rangle and stores the resulting color specification as the \langle name\rangle.

\color_set:nnn \color_set:nnn {\langle name\rangle} \{(\langle model(s)\rangle)\} \{\langle value(s)\rangle\}

Stores the color specification equivalent to the \langle model(s)\rangle and \langle values\rangle as the \langle name\rangle.

\color_set_eq:nn \color_set_eq:nnn {\langle name1\rangle} \{\langle name2\rangle\}

Copies the color specification in \langle name2\rangle to \langle name1\rangle. The special name . may be used to represent the current color, allowing it to be saved to a name.

\color_if_exist_p:n \color_if_exist:nTF {\langle name\rangle} \{\langle true code\rangle\} \{\langle false code\rangle\}

Tests whether \langle name\rangle is currently defined to provide a color specification.

\color_show:n \color_log:n {\langle name\rangle}

Displays the color specification stored in the \langle name\rangle on the terminal or log file.

36.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color (\.). Other more specialised functions such as fill and stroke selectors do not adjust this value.

\color_select:n \color_select:nn {\langle color expression\rangle}

Parses the \langle color expression\rangle and then activates the resulting color specification for typeset material.

\color_select:nn \color_select:nnn {\langle model(s)\rangle} \{\langle value(s)\rangle\}

Activates the color specification equivalent to the \langle model(s)\rangle and \langle value(s)\rangle for typeset material.

\l_color_fixed_model_tl

When this is set to a non-empty value, colors will be converted to the specified model when they are selected. Note that included images and similar are not influenced by this setting.
36.6 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 expression⟩}
\color_stroke:n

Parses the ⟨color expression⟩ and then activates the resulting color specification for filling or stroking.

\color_fill:nn {⟨model(s)⟩} {⟨value(s)⟩}
\color_stroke:nn

Activates the color specification equivalent to the ⟨model(s)⟩ and ⟨value(s)⟩ for filling or stroking.

When using dvips, this PostScript variables hold the stroke color.

36.6.1 Coloring math mode material

Coloring math mode material using \color_select:nn(n) has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating \mathord atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

\color_math:nn {⟨color expression⟩}{⟨content⟩}
\color_math:nnn {⟨model(s)⟩} {⟨value(s)⟩} {⟨content⟩}

Works as for \color_select:n(n) but applies color only to the math mode ⟨content⟩. The function does not generate a group and the ⟨content⟩ therefore retains its math atom states. Sub/superscripts are also properly handled.

This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.

36.7 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 \{ \texttt{cmyk} / \texttt{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.

### 36.8 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:nn\{\langle color expression\rangle\}\{\langle format\rangle\}\{\langle tl\rangle\}

Parses the ⟨color expression⟩ as described earlier, then converts to the ⟨format⟩ specified and assigns the data to the ⟨tl⟩.

\color_export:nnn\{\langle model\rangle\}\{\langle value(s)\rangle\}\{\langle format\rangle\}\{\langle tl\rangle\}

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⟩.
36.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see https://helpx.adobe.com/indesign/using/spot-process-colors.html for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (DeviceCMYK, DeviceRGB, DeviceGray), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that l3color uses the shorter names cmyk, etc.)

\color_model_new:nnn \{\langle model \rangle \} \{\langle family \rangle \} \{\langle params \rangle \}

Creates a new \langle model \rangle which is derived from the color model \langle family \rangle. The latter should be one of

• DeviceN
• ICCBased
• Separation

(The \langle family \rangle may be given in mixed case as in the PDF reference: internally, case of these strings is folded.) Depending on the \langle family \rangle, one or more \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 \langle name \rangle may contain spaces, etc., which are not permitted in the \langle model \rangle.

• alternative-model An alternative device colorspace, one of cmyk, rgb, gray or CIELAB. The three parameter-based models work as described above; see below for details of CIELAB colors.

• alternative-values A comma-separated list of values appropriate to the alternative-model. This information is used by the PDF application if the Separation is not available.

CIELAB color separations are created using the alternative-model = CIELAB setting. These colors must also have an illuminant key, one of a, c, e, d50, d55, d65 or d75. The alternative-values in this case are the three parameters L*, a* and b* of the CIELAB model. Full details of this device-independent color approach are given in the documentation to the colorspace package.

CIELAB colors cannot be converted into other device-dependent color spaces, and as such, mixing can only occur if colors set up using the CIELAB model are also given with an alternative parameter-based model. If that is not the case, l3color will fallback to using black as the colorant in any mixing.

For a DeviceN space, there is one compulsory key.

• names The names of the components of the DeviceN space. Each should be either the \langle name \rangle of a Separation model, a process color name (cyan, etc.) or the special name none.

For a ICCBased space, there is one compulsory key.

• file The name of the file containing the profile.
36.9.1 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:nn {profile} {model}
```

This function applies a `profile` to one of the device `models`. The profile will then apply to all color of the selected `model`. The `profile` should specify an ICC profile file. The `model` has to be one the standard device models: cmyk, gray or rgb.
Chapter 37

The \l3pdf package
Core PDF support

37.1 Objects

\pdf_object_new:n {\langle object \rangle}

Declares \langle object \rangle as a PDF object. The object may be referenced from this point on, and written later using \pdf_object_write:nnn.

\pdf_object_write:nn {\langle object \rangle} {\langle type \rangle} {\langle content \rangle}

Writes the \langle content \rangle as content of the \langle object \rangle. Depending on the \langle type \rangle declared for the object, the format required for the \langle data \rangle will vary

- array A space-separated list of values
- dict Key–value pairs in the form /\langle key \rangle \langle value \rangle
- fstream Two brace groups: \langle file name \rangle and \langle file content \rangle
- stream Two brace groups: \langle attributes (dictionary) \rangle and \langle stream contents \rangle

\pdf_object_ref:n \pdf_object_ref:n {\langle object \rangle}

Inserts the appropriate information to reference the \langle object \rangle in for example page resource allocation
\pdf_object_unnamed_write:nn \pdf_object_unnamed_write:nn \{\textit{type}\} \{\textit{content}\}
\pdf_object_unnamed_write:ne

Revised: 2021-02-10

Writes the \textit{content} as content of an anonymous object. Depending on the \textit{type}, the format required for the \textit{data} will vary.

\textbf{array} A space-separated list of values

\textbf{dict} Key–value pairs in the form /\textit{key} \{\textit{value}\}

\textbf{fstream} Two value pairs: \textit{attributes (dictionary)} and \textit{file name}

\textbf{stream} Two brace groups: \textit{attributes (dictionary)} and \textit{stream contents}

\pdf_object_ref_last:

Revised: 2021-02-10

Inserts the appropriate information to reference the last \textit{object} created. This is particularly useful for anonymous objects.

\pdf_pageobject_ref:n

Revised: 2021-02-10

Inserts the appropriate information to reference the \textit{pageobject}.

\pdf_object_if_exist_p:n \pdf_object_if_exist:nTF

Revised: 2020-05-15

Tests whether an object with name \{\textit{object}\} has been defined.

### 37.2 Version

\pdf_version_compare_p:Nn \pdf_version_compare:NnTF

Revised: 2021-02-10

Compares the version of the PDF being created with the \textit{version} string specified, using the \textit{comparator}. Either the \textit{true code} or \textit{false code} will be left in the output stream.

\pdf_version_gset:n
\pdf_version_min_gset:n

Revised: 2021-02-10

Sets the \textit{version} of the PDF being created. The \textit{min version} will not alter the output version unless it is currently lower than the \textit{version} requested.

This function may only be used up to the point where the PDF file is initialised. With \texttt{dvips} it sets \texttt{\pdf_version_major:} and \texttt{\pdf_version_minor:} and allows to compare the values with \texttt{\pdf_version_compare:Nn}, but the PDF version itself still has to be set with the command line option \texttt{-dCompatibilityLevel} of \texttt{ps2pdf}.

\pdf_version:
\pdf_version_major: \pdf_version_minor:

Revised: 2021-02-10

Expands to the currently-active PDF version.
37.3 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.

37.4 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.

37.5 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. \{\textit{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 \texttt{/FitR}: The backend code defines \texttt{fitr} so that it will with \texttt{pdflatex} and \texttt{LuaLaTeX} 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>\texttt{fit}</td>
<td>/Fit</td>
<td>Fits the page to the window</td>
</tr>
<tr>
<td>\texttt{fith}</td>
<td>/FitH \texttt{top}</td>
<td>Fits the width of the page to the window</td>
</tr>
<tr>
<td>\texttt{fitv}</td>
<td>/FitV \texttt{left}</td>
<td>Fits the height of the page to the window</td>
</tr>
<tr>
<td>\texttt{fitb}</td>
<td>/FitB</td>
<td>Fits the page bounding box to the window</td>
</tr>
<tr>
<td>\texttt{fitbh}</td>
<td>/FitBH \texttt{top}</td>
<td>Fits the width of the page bounding box to the window.</td>
</tr>
<tr>
<td>\texttt{fitbv}</td>
<td>/FitBV \texttt{left}</td>
<td>Fits the height of the page bounding box to the window.</td>
</tr>
<tr>
<td>\texttt{fitr}</td>
<td>/FitR \texttt{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>\texttt{xyz}</td>
<td>/XYZ \texttt{left top null}</td>
<td>Sets a coordinate but doesn’t change the zoom.</td>
</tr>
</tbody>
</table>

\texttt{pdf_destination:nnnn} \{\textit{name}\} \{\textit{width}\} \{\textit{height}\} \{\textit{depth}\}

This creates a destination with \texttt{/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 38

The \texttt{l3candidates} package
Experimental additions to \texttt{l3kernel}

\section{Important notice}

This module provides a space in which functions can be added to \texttt{l3kernel} (\texttt{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:

\begin{itemize}
  \item Be prepared that your public packages might require updating when such functions are being finalized.
  \item 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.
  \item 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).
\end{itemize}

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are not necessarily stable and we may have to adjust them!
38.2 Additions to l3seq

\seq_set_filter:NNn \seq_set_filter:NNn \langle \text{sequence}_1 \rangle \langle \text{sequence}_2 \rangle \{\langle \text{inline boolexpr} \rangle\}

Evaluates the \langle \text{inline boolexpr} \rangle for every \langle \text{item} \rangle stored within the \langle \text{sequence}_2 \rangle. The \langle \text{inline boolexpr} \rangle receives the \langle \text{item} \rangle as #1. The sequence of all \langle \text{items} \rangle for which the \langle \text{inline boolexpr} \rangle evaluated to \text{true} is assigned to \langle \text{sequence}_1 \rangle.

\textbf{TeXhackers note:} Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level \TeX errors.

38.3 Additions to l3tl

\tl_build_get:NN \tl_build_get:NN \langle \text{tl var}_1 \rangle \langle \text{tl var}_2 \rangle

Stores the contents of the \langle \text{tl var}_1 \rangle in the \langle \text{tl var}_2 \rangle. The \langle \text{tl var}_1 \rangle must have been set up with \tl_build_begin:N or \tl_build_gbegin:N. The \langle \text{tl var}_2 \rangle is a “normal” token list variable, assigned locally using \tl_set:Nn.
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>Page</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>271</td>
<td>\bool_case:n</td>
</tr>
<tr>
<td>&amp;h</td>
<td>270</td>
<td>\bool_case:nTF</td>
</tr>
<tr>
<td>*</td>
<td>271</td>
<td>\bool_const:NN</td>
</tr>
<tr>
<td>**</td>
<td>271</td>
<td>\bool_do_until:NN</td>
</tr>
<tr>
<td>+</td>
<td>271</td>
<td>\bool_gset:N</td>
</tr>
<tr>
<td>-</td>
<td>271</td>
<td>\bool_gset:NTF</td>
</tr>
<tr>
<td>/</td>
<td>271</td>
<td>\bool_if:NTF</td>
</tr>
<tr>
<td>:i</td>
<td>43</td>
<td>\bool_if_exist:NTF</td>
</tr>
<tr>
<td>:N</td>
<td>43</td>
<td>\bool_if_exist:p:N</td>
</tr>
<tr>
<td>:v</td>
<td>43</td>
<td>\bool_lazy_all:nn</td>
</tr>
<tr>
<td>:x</td>
<td>43</td>
<td>\bool_lazy_or:nn</td>
</tr>
<tr>
<td>&lt;</td>
<td>271</td>
<td>\bool_log:nn</td>
</tr>
<tr>
<td>=</td>
<td>271</td>
<td>\bool_new:N</td>
</tr>
<tr>
<td>&gt;</td>
<td>271</td>
<td>\bool_not_p:n</td>
</tr>
<tr>
<td>?</td>
<td>271</td>
<td>\bool_set:N</td>
</tr>
<tr>
<td>?:</td>
<td>270</td>
<td>\bool_set_eq:NN</td>
</tr>
<tr>
<td>(\langle\text{scope}\rangle)<em>tmpa(</em>\text{type})</td>
<td>5</td>
<td>\bool_set:NN</td>
</tr>
<tr>
<td>(\langle\text{scope}\rangle)<em>tmpb(</em>\text{type})</td>
<td>5</td>
<td>\bool_set_eq:NN</td>
</tr>
<tr>
<td>^</td>
<td>271</td>
<td>\bool_set_inverse:N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Page</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>271</td>
<td>\bool_new:N</td>
</tr>
<tr>
<td>acos</td>
<td>274</td>
<td>\bool_not_p:n</td>
</tr>
<tr>
<td>acosd</td>
<td>274</td>
<td>\bool_set:N</td>
</tr>
<tr>
<td>acot</td>
<td>275</td>
<td>\bool_set:NN</td>
</tr>
<tr>
<td>acotd</td>
<td>275</td>
<td>\bool_set_eq:NN</td>
</tr>
<tr>
<td>acsc</td>
<td>274</td>
<td>\bool_set_inverse:N</td>
</tr>
<tr>
<td>asec</td>
<td>274</td>
<td>\bool_set_inverse:N</td>
</tr>
</tbody>
</table>
\bool_set_true:N \bool_show:N \bool_to_str:N \box_log:N \box_move_down:n \box_move_left:n \box_move_right:n \box_move_up:n \box_new:N \box_if_horizontal_p:N \box_if_horizontal:NTF \box_if_exist_p:N \box_if_exist:NTF \box_if_empty:NTF \box_ht_plus_dp:N \box_ht:N \box_gset_wd:N \box_gset_trim:Nnnnn \box_gset_to_last:N \box_gset_eq_drop:NN \box_gset_eq:NN \box_gset_dp:Nn \box_gset_clipped:N \box_gscale:Nnn \box_grotate:Nn \box_gresize_to_wd_and_ht_plus_- \box_gresize_to_wd_and_ht:Nnn \box_gresize_to_wd:Nn \box_gresize_to_ht_plus_dp:Nn \box_gresize_to_ht:Nn \box_gresize_to_wd_and_ht_plus_- \box_gresize_to_wd_and_ht:Nnn \cctab_begin:N \cctab_end:Nn \cctab_item:Nn \cctab_if_exist_p:N \cctab_if_exist:NTF \cctab_gset:Nn \cctab_gset_clipped:N \cctab_gset_dp:Nn \cctab_gset_to_last:N \cctab_gset_viewport:Nnnnn \cctab_gset:NN \cctab_gset_eq:NN \cctab_gset_eq_drop:NN \cctab_gset trim:Nnnnn \cctab_gsave current:N \cctab_if_exist:NTF \cctab_if_exist_p:N \cctab_item:Nn \cctab_new:N \cctab_select:N \c_code_cctab \c_document_cctab \cctab_begin:N \cctab_end:Nn \cctab_item:Nn \cctab_new:N \cctab_select:N \c_code_cctab \c_document_cctab

box commands:
\box_autosize_to_wd_and_ht:Nnn . \box_autosize_to_wd_and_ht_plus_- \box_dp:Nnn \box_clear:N \box_new_N \box_move_right:nn \box_move_left:nn \box_move_up:n \box_move_down:n \box_scale:Nnn \box_set_wd:Nn \box_set_viewport:Nnnnn \box_set_dp:Nn \box_set_trim:Nnnnn \box_set_eq:NN \box_set_eq_drop:NN \box_set_to_last:N \box_set_to_last:N \box_set_to_last:N \box_set_to_last:N \box_set_to_last:N \box_set_to_last:N \box_set_to_last:N

C

cc

\box_gclear_new:N \box_gclear:N \box_gautosize_to_wd_and_ht_-

bp

\box_gclear:NN \box_gautosize_to_wd_and_ht:Nnn \box_gautosize_to_wd_and_ht_plus_- \box_gautosize_to_wd_and_ht_plus_- \box_gautosize_to_wd_and_ht:Nnn \box_gautosize_to_wd_and_ht:Nnn \box_gautosize_to_wd_and_ht:Nnn

\box_gscale:NN \box_gset_clipped:N \box_gset_dp:Nn \box_gset_eq:NN \box_gset_dp:Nn \box_gset_clipped:N \box_gset_dp:Nn \box_gset_dp:Nn \box_gset_dp:Nn \box_gset_dp:Nn

\box_gclear:NN \box_gclear:NN \box_gclear:NN \box_gclear:NN \box_gclear:NN

\bool_set_true:N \bool_show:N \bool_to_str:N \box_log:N \box_move_down:n \box_move_left:n \box_move_right:n \box_move_up:n \box_new:N \box_if_horizontal_p:N \box_if_horizontal:NTF \box_if_exist_p:N \box_if_exist:NTF \box_if_empty:NTF \box_ht_plus_dp:N \box_ht:N \box_gset_wd:N \box_gset_trim:Nnnnn \box_gset_to_last:N \box_gset_eq_drop:NN \box_gset_eq:NN \box_gset_dp:Nn \box_gset_clipped:N \box_gscale:Nnn \box_grotate:Nn \box_gresize_to_wd_and_ht_plus_- \box_gresize_to_wd_and_ht:Nnn \box_gresize_to_wd_and_ht:Nnn \box_gresize_to_wd_and_ht:Nnn \box_gresize_to_wd_and_ht:Nnn \box_gresize_to_wd_and_ht:Nnn

\cctab_begin:N \cctab_end:Nn \cctab_item:Nn \cctab_if_exist_p:N \cctab_if_exist:NTF \cctab_gset:NN \cctab_gset_eq:NN \cctab_gset_eq_drop:NN \cctab_gset trim:Nnnnn \cctab_gsave current:N \cctab_if_exist:NTF \cctab_if_exist_p:N \cctab_item:Nn \cctab_new:N \cctab_select:N \c_code_cctab \c_document_cctab

\cctab_begin:N \cctab_end:Nn \cctab_item:Nn \cctab_new:N \cctab_select:N \c_code_cctab \c_document_cctab

\cctab_begin:N \cctab_end:Nn \cctab_item:Nn \cctab_new:N \cctab_select:N \c_code_cctab \c_document_cctab
\c_other_cctab ............. 283 \char_show_value_catcode:n ... 197
\g_tmpa_cctab ............. 283 \char_show_value_lccode:n ... 198
\g_tmpb_cctab ............. 283 \char_show_value_mathcode:n ... 198
ceil ............. 273 \char_show_value_sfcode:n ... 199

char commands:
\l_char_active_seq .......... , 199
\char_foldcase:N .......... 211 \char_str_foldcase:N ........ 199
\char_generate:nn .......... 125, 195 \char_str_lowercase:N ........ 211
\char_get_active_eq:NN .... 195 \char_str_titlecase:N .......... 211
\char_get_active_eq:N .... 195 \char_str_uppercase:N .......... 211
\char_lowercase:N .......... 211 \char_titlecase:N ........... 211
\char_set_active_eq:NN ... 195 \char_value_catcode:n ........ 197
\char_set_active_eq:nN ... 195 \char_value_lccode:n .......... 198
\char_set_active_eq:nN ... 195 \char_value_mathcode:n .......... 198
\char_set_catcode_alignment:N ...... 197 \char_value_sfcode:n .......... 199
\char_set_catcode_active:N ... 197 \char_value_uccode:n .......... 198
\char_set_catcode_active:n ... 197 

choice commands:
.choice: ...................... 239
.choices:nn .................. 239

clist commands:
\clist_concat:NNN .......... 184 \clist_count:n ............ 184
\clist_count:N ............. 184 \clist_gclear:N ........... 184
\clist_gconcat:NNN .......... 184 \clist_gclear_new:N ......... 184
\clist_gclear:nn .......... 185 \clist_gconcat:NN ........... 185
\clist_gclear:nn .......... 185 \clist_gconcat:NN ........... 185
\clist_gclear_all:N ........ 186 \clist_gconcat:NN ........... 185
\clist_gclear_duplicates:N . 186 \clist_gconcat:NN ........... 185
\clist_get_left:N ........ 185 \clist_gconcat:NN ........... 185
\clist_gconcat:NN ........... 185 \clist_gconcat:NN ........... 185
\clist_gconcat:NN ........... 185 \clist_gconcat:NN ........... 185
\clist_gconcat:NN ........... 185 \clist_gconcat:NN ........... 185
\clist_gconcat:NN ........... 185 

330
\color_profile_apply:nn \hspace{145x562} \color_stroke:n \hspace{160x706} \color_set_eq:nn \hspace{160x629} \color_set:nn \hspace{160x640} \color_select:n \hspace{160x640} \cs_end: \hspace{145x540} \cs_generate_variant:Nn \hspace{160x529} \cs_generate_from_arg_count:NNnn \hspace{160x19} \cs_new:Npx \hspace{160x120} \cs_new_nopar:Npe \hspace{160x131} \cs_new_nopar:Nn \hspace{160x176} \cs_new_protected_nopar:Npx \hspace{160x618} \color_set_eq:nn \hspace{160x640} \color_set:nn \hspace{160x684} \color_set_eq:nn \hspace{160x684} \cs_gset_protected_nopar:Npx \hspace{160x319} \cs_gset_protected_nopar:Np \hspace{160x397} \cs_gset_protected:Nn \hspace{160x408} \cs_gset_nopar:Npx \hspace{160x408} \cs_gset_nopar:Npn \hspace{160x430} \cs_gset_nopar:Npe \hspace{160x441} \cs_gset_protected:Np \hspace{160x441} \cs_gset_protected:Nn \hspace{160x452} \cs_gset_nopar:Np \hspace{160x452} \cs_gset_nopar:Nn \hspace{160x463} \cs_gset_nopar:Np \hspace{160x463} \cs_gset_nopar:Nn \hspace{160x474} \cs_gset_nopar:Np \hspace{160x474} \cs_gset_nopar:Nn \hspace{160x485} \cs_gset_nopar:Np \hspace{160x485} \cs_gset_nopar:Nn \hspace{160x496} \cs_gset_nopar:Np \hspace{160x496} \cs_gset_nopar:Nn \hspace{160x507} \cs_gset_nopar:Np \hspace{160x507} \cs_gset_nopar:Nn \hspace{160x518} \cs_gset_nopar:Np \hspace{160x518} \cs_gset_nopar:Nn \hspace{160x529} \cs_gset_nopar:Np \hspace{160x529} \cs_gset_nopar:Nn \hspace{160x540} \cs_gset_nopar:Np \hspace{160x540} \cs_gset_nopar:Nn \hspace{160x551} \cs_gset_nopar:Np \hspace{160x551} \cs_gset_nopar:Nn \hspace{160x562} \cs_gset_nopar:Np \hspace{160x562} \cs_gset_nopar:Nn \hspace{160x573} \cs_gset_nopar:Np \hspace{160x573} \cs_gset_nopar:Nn \hspace{160x584} \cs_gset_nopar:Np \hspace{160x584} \cs_gset_nopar:Nn \hspace{160x595} \cs_gset_nopar:Np \hspace{160x595} \cs_gset_nopar:Nn \hspace{160x607} \cs_gset_nopar:Np \hspace{160x607} \cs_gset_nopar:Nn

\TeX and \LaTeX commands:

\@filelist .......................... 105 \inputpath .......................... 101
\@firstofone .......................... 24 \jobname .......................... 75
\@gobbletwo ......................... 26 \leavevmode .......................... 29
\@gobble ................................ 26 \long ................................ 5, 211
\aftergroup ................................ 14 \luaescapestring ...................... 107
\box ..................................... 302 \makeatletter ......................... 9
\char ..................................... 211 \mathchar .......................... 211
\chardef ................................ 211 \mathcode .......................... 211
\char \@gobbletwo ..................... 202, 203 \mathchardef ......................... 203
\char \@gobbbletwo .................... 28 \mathdchord ........................ 317
\cl ................................. 210 \maxdimen .......................... 228
\csname ................................ 21 \meaning ..................... 20, 200, 210, 211
\day ................................... 75 \message .............................. 34
\def .................................... 210 \month ................................ 75
\detokenize .......................... 116 \newif ............................ 67, 109
\directlua .......................... 106 \newlinechar .......................... 126
\dp ..................................... 296 \newlinecharchar ........................ 126
\edef .................................. 3, 6 \newlinecharchardef .................. 39, 209
\else .................................. 28 \number ................................ 179
\endcsname .......................... 21 \or .................................. 179
\endinput ................................ 85 \outer .................................. 7, 211
\endlinechar ........................ 94, 126, 281–283 \parindent ......................... 29
\endtemplate ........................ 74 \pdfescapecname .................... 143
\escapechar .......................... 116 \pdfescapestring ................ 143
\everypar .......................... 29, 206 \pdfuniformdeviate ................ 276
\expandafter ................................ 39, 41 \pi .................................. 269
\expanded ................................ 3, 6, 26, 34 \protected .......................... 211
\if ..................................... 28, 209 \ProvidesClass ...................... 9
\fmtname ............................ 76 \ProvidesFile ........................... 9
\font .................................. 209 \ProvidesPackage .................... 9
\fontdimen ................................ 62, 254 \read ......................... 94
\halign ................................ 74, 104 \readline ......................... 94
\hakip .................................. 233 \relax .................................. 28, 209
\ht ..................................... 296 \RequirePackage .................... 10
\if ..................................... 29 \romannumeral ..................... 41
\ifcase ................................ 179 \scantokens ....................... 126, 145
\ifcat ................................ 29 \show .................................. 20, 118
\ifcsname ............................ 29 \showgroups ........................... 14
\ifdef .................................. 29 \showtokens .......................... 118
\ifdim .................................. 236 \sin .................................. 269
\ifeof .................................. 100 \string ................................ 200
\iffalse .................................. 28, 67 \tenrm .......................... 209
\ifhbox .............................. 306 \the .................................. 170, 209, 227, 232, 235
\ifhmode ................................ 29 \time .................................. 75
\ifinner ................................ 29 \toks .................................. 44, 157, 179
\ifmmode ................................ 29 \topmark .......................... 210
\ifnum .................................. 179 \unexpanded ................. 40, 117, 118, 122, 123, 154, 160, 161, 186, 190–192, 216
\ifodd .................................. 180 \unexpand ..................... 40, 117, 118, 122, 123, 154, 160, 161, 186, 190–192, 216
\iftrue .................................. 28, 67 \unhbox .......................... 302
\ifx ..................................... 28, 67 \unhcopy .......................... 299
\ifvmode ................................ 29 \uniformdeviate ................ 276
\ifvoid .................................. 306 \unless ................................ 28
\ifx ..................................... 29 \unvbox .......................... 302
\iftty .................................. 269 \unvcopy .......................... 301
\token_if_eq_meaning_p:NN 202 usage commands:
\token_if_expandable:NTF 202 \usage:n 246
\token_if_expandable_p:N 202 \usage:n 246
\token_if_font_selection:NTF 203 \usage:n 21, 181
\token_if_font_selection_p:N 203 \usage:n 24, 26, 111, 209
\token_if_group_begin:NTF 200 \usage:nn 24
\token_if_group_begin_p:N 200 \usage:nn 24
\token_if_group_end:NTF 200 \usage:nnn 24
\token_if_group_end_p:N 200 \usage:i:nn 25
\token_if_int_register:NTF 203 \usage:i:nnn 25
\token_if_int_register_p:N 203 \usage:i:nnnn 25
\token_if_letter:NTF 201 \usage:i:nnnnn 25
\token_if_letter_p:N 201 \usage:i:nnnnnn 25
\token_if_long_macro:NTF 202 \usage:i:nnnnnnn 25
\token_if_long_macro_p:N 202 \usage:i:nnnnnnnnn 25
\token_if_math_subscript:NTF 201 \usage:i:nnnnnnn 25
\token_if_math_subscript_p:N 201 \usage:i:nnnnnnnnn 25
\token_if_math_superscript:NTF 201 \usage:i:nnnnnnnnn 25
\token_if_math_superscript_p:N 201 \usage:i:nnnnnnnnnnn 25
\token_if_math_toggle:NTF 201 \usage:i:nnnnnnnnnnn 25
\token_if_math_toggle_p:N 201 \usage:i:nnnnnnnnnnnnn 25
\token_if_muskip_register:NTF 201 \usage:i:nnnnnnnnnnnnn 25
\token_if_muskip_register_p:N 201 \usage:i:nnnnnnnnnnnnnnn 25
\token_if_protected_macro:NTF 201 \usage:i:nnnnnnnnnnnnnnn 25
\token_if_protected_macro_p:N 201 \usage:i:nnnnnnnnnnnnnnnnn 25
\token_if_protected_long:_macro:NTF 202 \usage:i:nnnnnnnnnnnnnnnnn 25
\token_if_protected_long_macro:_p:N 202 \usage:i:nnnnnnnnnnnnnnnnnnn 25
\token_if_protected_macro:NTF 202 \usage:i:nnnnnnnnnnnnnnnnnnn 25
\token_if_protected_macro_p:N 202 \usage:i:nnnnnnnnnnnnnnnnnnnnn 25
\token_if_skip_register:NTF 203 \usage:iv:mmn 25
\token_if_skip_register_p:N 203 \usage:iv:mmnn 25
\token_if_space:NTF 201 \usage:iv:mmnnn 25
\token_if_space_p:N 201 \usage:iv:mmnnnn 25
\token_if_toks_register:NTF 203 \usage:iv:mmnnnnn 25
\token_if_toks_register_p:N 203 \usage:iv:mmnnnnnn 25
\token_to_catcode:N 200 \usage:ix:mmnnnnnnnn 25
\token_to_meaning:N 200 \usage:none:n 26
\token_to_str:N 7, 22, 99, 129, 200, 210 \usage:none:n 26
\true: 277 \usage:none:n 26
\trunc 273 \usage:none:n 26
U
\undefined: 243 \usage:none:n 26
\use_none:nnnnnnnn \use_none_delimit_by_q_nil:w \use_none_delimit_by_q_recursion_-
\use_none_delimit_by_q_stop:w \use_v:nnnnn \use_v:nnnnnn \use_v:nnnnnnn \use_v:nnnnnnnn \use_v:nnnnnnnnn \use_vi:nnnnnn \use_vi:nnnnnnnn \use_vi:nnnnnnnnn \use_vii:nnnnnnnn \use_vii:nnnnnnnnn \use_vii:nnnnnnnnnn \use_viii:nnnnnnnnnn
\vbox:n \value_commands:
.value_forbidden:n \value_required:n
\vbox_commands:
\vbox:n \vbox_gset:Nn \vbox_gset:Nw \vbox_gset:Ng \vbox_gset_end:
\vbox_gset_split_to_ht:NNn \vbox_gset_to_ht:Nnn \vbox_gset_to_ht:Nnw \vbox_gset_top:Nn
\vbox_set:Nn \vbox_set:Nw \vbox_set:Ng \vbox_set_end:
\vbox_set_split_to_ht:NNn \vbox_set_to_ht:Nnn \vbox_set_to_ht:Nnw \vbox_set_top:Nn \vbox_to_ht:nn \vbox_to_zero:n
\vbox_top:n \vbox_unpack:N \vbox_unpack_drop:N \vcoffin_commands:
\vcoffin_gset:Nnn \vcoffin_gset:Nnw \vcoffin_gset_end:
\vcoffin_set:Nnn \vcoffin_set:Nnw \vcoffin_set_end:

vcoffin

value commands:
.value_forbidden:n \value_required:n \vbox_commands:
\vbox:n