The \LaTeX3 Interfaces

The \LaTeX Project*

Released 2024-05-27

Abstract

This is the reference documentation for the \texttt{expl3} programming environment; see the matching \texttt{source3} PDF for the typeset sources. The \texttt{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 \texttt{\epsilon-\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 \texttt{expl3} modules define an independent low-level \LaTeX3\ programming language.

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

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 8

2 The \l3bootstrap module: Bootstrap code 9
   2.1 Using the \LaTeX3 modules 9

3 The \l3names module: Namespace for primitives 11
   3.1 Setting up the \LaTeX3 programming language 11

III Programming Flow 12

4 The \l3basics module: 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 30
   4.8 Debugging support 30
10 The \texttt{l3sys} module: System/runtime functions

10.1 The name of the job ........................................ 74
10.2 Date and time ........................................... 74
10.3 Engine ................................................... 75
10.4 Output format ........................................... 76
10.5 Platform ................................................ 76
10.6 Random numbers ......................................... 76
10.7 Access to the shell ....................................... 77
10.8 System queries .......................................... 78
10.9 Loading configuration data ......................... 79
  10.9.1 Final settings ..................................... 79

11 The \texttt{l3msg} module: Messages .......................... 80

11.1 Creating new messages .................................. 80
11.2 Customizable information for message modules .... 81
11.3 Contextual information for messages ............... 82
11.4 Issuing messages ....................................... 83
  11.4.1 Messages for showing material ................. 87
  11.4.2 Expandable error messages ................... 87
11.5 Redirecting messages .................................. 88

12 The \texttt{l3file} module: File and I/O operations .... 90

12.1 Input–output stream management .................... 90
  12.1.1 Reading from files ................................ 92
  12.1.2 Reading from the terminal ....................... 96
  12.1.3 Writing to files ................................... 96
  12.1.4 Wrapping lines in output ....................... 98
  12.1.5 Constant input–output streams, and variables 99
  12.1.6 Primitive conditionals ....................... 99
  12.2 File operations ..................................... 99
  12.2.1 Basic file operations ........................... 99
  12.2.2 Information about files and file contents ...... 100
  12.2.3 Accessing file contents ....................... 103

13 The \texttt{l3luatex} module: Lua\TeX{}-specific functions 105

13.1 Breaking out to \LaTeX{} .............................. 105
13.2 Lua interfaces ....................................... 106

14 The \texttt{l3legacy} module: Interfaces to legacy concepts 108

IV Data types .......................... 109
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Subsections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>The <code>l3tl</code> module: Token lists</td>
<td>15.1 Creating and initialising token list variables</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.2 Adding data to token list variables</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.3 Token list conditionals</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.3.1 Testing the first token</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.4 Working with token lists as a whole</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.4.1 Using token lists</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.4.2 Counting and reversing token lists</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.4.3 Viewing token lists</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.5 Manipulating items in token lists</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.5.1 Mapping over token lists</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.5.2 Head and tail of token lists</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.5.3 Items and ranges in token lists</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.5.4 Sorting token lists</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.6 Manipulating tokens in token lists</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.6.1 Replacing tokens</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.6.2 Reassigning category codes</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.7 Constant token lists</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.8 Scratch token lists</td>
<td>126</td>
</tr>
<tr>
<td>16</td>
<td>The <code>l3tl-build</code> module: Piecewise <code>tl</code> constructions</td>
<td>16.1 Constructing <code>&lt;tl var&gt;</code> by accumulation</td>
<td>127</td>
</tr>
<tr>
<td>17</td>
<td>The <code>l3str</code> module: Strings</td>
<td>17.1 Creating and initialising string variables</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.2 Adding data to string variables</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.3 String conditionals</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.4 Mapping over strings</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.5 Working with the content of strings</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.6 Modifying string variables</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.7 String manipulation</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.8 Viewing strings</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.9 Constant strings</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17.10 Scratch strings</td>
<td>141</td>
</tr>
<tr>
<td>18</td>
<td>The <code>l3str-convert</code> module: String encoding conversions</td>
<td>18.1 Encoding and escaping schemes</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.2 Conversion functions</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.3 Conversion by expansion (for PDF contexts)</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.4 Possibilities, and things to do</td>
<td>144</td>
</tr>
<tr>
<td>19</td>
<td>The <code>l3quark</code> module: Quarks and scan marks</td>
<td>19.1 Quarks</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.2 Defining quarks</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.3 Quark tests</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.4 Recursion</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.4.1 An example of recursion with quarks</td>
<td>149</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19.5 Scan marks</td>
<td>150</td>
</tr>
</tbody>
</table>
24 The \texttt{\texttt{l3token}} module: Token manipulation 194
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 ...................................................
24.6 Peeking ahead at the next token ...................................
24.7 Description of all possible tokens ..................................
25 The \texttt{\texttt{l3prop}} module: Property lists 212
25.1 Creating and initialising property lists ............................ 213
25.2 Adding and updating property list entries ......................... 215
25.3 Recovering values from property lists ............................ 216
25.4 Modifying property lists ............................................. 217
25.5 Property list conditionals ...........................................
25.6 Recovering values from property lists with branching ...........
25.7 Mapping over property lists ....................................... 219
25.8 Viewing property lists ............................................. 220
25.9 Scratch property lists ................................................
25.10 Constants ..............................................................
26 The \texttt{\texttt{l3skip}} module: Dimensions and skips 222
26.1 Creating and initialising \texttt{dim} variables ....................... 222
26.2 Setting \texttt{dim} variables .......................................... 223
26.3 Utilities for dimension calculations ................................
26.4 Dimension expression conditionals ................................
26.5 Dimension expression loops ......................................
26.6 Dimension step functions .......................................... 227
26.7 Using \texttt{dim} expressions and variables ....................... 228
26.8 Viewing \texttt{dim} variables .......................................... 230
26.9 Constant dimensions ................................................
26.10 Scratch dimensions ..................................................
26.11 Creating and initialising \texttt{skip} variables ..................... 231
26.12 Setting \texttt{skip} variables ......................................... 232
26.13 Skip expression conditionals ......................................
26.14 Using \texttt{skip} expressions and variables ..................... 233
26.15 Viewing \texttt{skip} variables .........................................
26.16 Constant skips ....................................................... 234
26.17 Scratch skips ...........................................................
26.18 Inserting skips into the output ...................................
26.19 Creating and initialising \texttt{muskip} variables .................. 235
26.20 Setting \texttt{muskip} variables ...................................... 235
26.21 Using \texttt{muskip} expressions and variables ................... 236
26.22 Viewing \texttt{muskip} variables ..................................... 236
26.23 Constant muskips .....................................................
26.24 Scratch muskips ......................................................
26.25 Primitive conditional ..............................................
### 32 The \texttt{t3ctab} module: Category code tables

- **32.1 Creating and initialising category code tables**
- **32.2 Using category code tables**
- **32.3 Category code table conditionals**
- **32.4 Constant and scratch category code tables**

### V Text manipulation

- **33 The \texttt{t3unicode} module: Unicode support functions**

### 34 The \texttt{t3text} module: Text processing

- **34.1 Expanding text**
- **34.2 Case changing**
- **34.3 Removing formatting from text**
- **34.4 Control variables**
- **34.5 Mapping to graphemes**

### VI Typesetting

- **35 The \texttt{t3box} module: Boxes**
- **35.1 Creating and initialising boxes**
- **35.2 Using boxes**
- **35.3 Measuring and setting box dimensions**
- **35.4 Box conditionals**
- **35.5 The last box inserted**
- **35.6 Constant boxes**
- **35.7 Scratch boxes**
- **35.8 Viewing box contents**
- **35.9 Boxes and color**
- **35.10 Horizontal mode boxes**
- **35.11 Vertical mode boxes**
- **35.12 Using boxes efficiently**
- **35.13 Affine transformations**
- **35.14 Viewing part of a box**
- **35.15 Primitive box conditionals**

- **36 The \texttt{t3coffins} module: Coffin code layer**
- **36.1 Creating and initialising coffins**
- **36.2 Setting coffin content and poles**
- **36.3 Coffin affine transformations**
- **36.4 Joining and using coffins**
- **36.5 Measuring coffins**
- **36.6 Coffin diagnostics**
- **36.7 Constants and variables**
37 The \texttt{\textcolor{color}{color}} module: Color support 320
37.1 Color in boxes ........................................ 320
37.2 Color models ........................................ 320
37.3 Color expressions .................................... 322
37.4 Named colors ........................................ 323
37.5 Selecting colors ...................................... 323
37.6 Colors for fills and strokes .......................... 324
  37.6.1 Coloring math mode material .................... 324
37.7 Multiple color models ............................... 324
37.8 Exporting color specifications ....................... 325
37.9 Creating new color models ........................... 326
  37.9.1 Color profiles ................................ 327
38 The \texttt{\textcolor{pdf}{pdf}} module: Core PDF support 328
38.1 Objects .............................................. 328
  38.1.1 Named objects ................................ 328
  38.1.2 Indexed objects ................................ 329
  38.1.3 General functions .............................. 329
38.2 Version ............................................. 330
38.3 Page (media) size ................................... 330
38.4 Compression ....................................... 330
38.5 Destinations ...................................... 331

Index 332
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 LÄTEX programming language is found in expl3.pdf.

1.1 Naming functions and variables

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

- This means \textit{expansion once}. In general, the V and v specifiers are favoured over o for recovering stored information. However, o is useful for correctly processing information with delimited arguments.

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

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

- The f specifier stands for \textit{full expansion}, and in contrast to x stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a \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 \l_mya_tl with the content A\l_myb_tl, as A cannot be expanded and so terminates expansion before \l_myb_tl is considered.

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

- The letter p indicates \LaTeX \texttt{parameters}. Normally this will be used for delimited functions as expl3 provides better methods for creating simple sequential arguments.

- Finally, there is the 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).

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

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

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

- c Constant: global parameters whose value should not be changed.
g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

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

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

**clist** Comma separated list.

**dim** “Rigid” lengths.

**fp** Floating-point values;

**int** Integer-valued count register.

**muskip** “Rubber” lengths for use in mathematics.

**skip** “Rubber” lengths.

**str** String variables: contain character data.

**tl** Token list variables: placeholder for a token list.

Applying \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** Non-negative integer that can be incremented expandably.

**fparray** Fixed-size array of floating point values.

**intarray** Fixed-size array of integers.

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

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

**regex** Regular expression.

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

\(^1\)The module names are not used in case of generic scratch registers defined in the data type modules,
e.g., the \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 \((\text{scope})\_\text{tmp}_a(\text{type})/\(\text{scope}\)_\text{tmp}_b(\text{type})\). These are never used by the core code. The nature of \(\TeX\) grouping means that as with any other scratch variable, these should only be set and used with no intervening third-party code.

1.1.2 Terminological inexactitude

A word of warning. In this document, and others referring to the expl3 programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, \(\TeX\) is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are also macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are almost the same. 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 actual registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in \(\TeX\)’s stomach” (if you are familiar with the \(\TeX\)book parlance).

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

1.2 Documentation conventions

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

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

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

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use _ and : in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed variants of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

\footnote{\TeXnically, 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, \(\text{sequence}\) 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 \texttt{edef} or \texttt{expanded}), as well as within an \texttt{f}-type argument. These fully expandable functions are indicated in the documentation by a star:

\begin{verbatim}
\seq_new:N \seq_new:c
\end{verbatim}

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

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

\begin{verbatim}
\seq_map_function:NN \seq_map_function:NN
\end{verbatim}

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

\begin{verbatim}
\sys_if_engine_xetex:TF \sys_if_engine_xetex:TF \{\text{true code}\} \{\text{false code}\}
\end{verbatim}

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

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

Usually, the illustration will use the \texttt{TF} variant, and so both \(\text{true code}\) and \(\text{false code}\) will be shown. The two variant forms \texttt{T} and \texttt{F} take only \(\text{true code}\) and \(\text{false code}\), 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:

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

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

In some cases, the function is similar to one in \LaTeX\ 2e or plain \TeX. In these cases, the text will include an extra “\TeXhackers note” section:
TEXhackers note: Detail for the experienced \TeX{} or \LaTeX{} programmer. In this case, it would point out that this function is the \TeX{} primitive \texttt{\string}.

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

### 1.3 Formal language conventions which apply generally

As this is a formal reference guide for \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 \texttt{\langle true code\rangle} or the \texttt{\langle false code\rangle} 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 “\texttt{\outer}” macro is *not supported* at all by \LaTeX{}3. As such, the functions provided here may break when used on top of \LaTeX{}2ε if \texttt{\outer} tokens are used in the arguments.
Part II

Bootstrapping
Chapter 2

The l3bootstrap module
Bootstrap code

2.1 Using the \LaTeX3 modules

The modules documented in this file (and source3 for documented sources) are designed to be used on top of \LaTeX2ε and are already pre-loaded since \LaTeX2ε 2020-02-02. To support older formats, the `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions are still available to load them all as one.

As the modules use a coding syntax different from standard \LaTeX2ε it provides a few functions for setting it up.

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

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

\TeXhackers note: Spaces introduced by ~ 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 ~ are equivalent to a single one. However, ~ is not ignored at the end of a line.

These functions act broadly in the same way as the corresponding \LaTeX2ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as \LaTeX2ε provides in turning on `\makeatletter` within package and class code.) The `\langle date \rangle` should be given in the format ⟨year⟩/⟨month⟩/⟨day⟩ or in the ISO date format ⟨year⟩-{⟨month⟩}-{⟨day⟩}. If the `\langle version \rangle` is given then a leading v is optional: if given as a “pure” version string, a v will be prepended.
\GetIdInfo \GetIdInfo $Id: ⟨SVN info field⟩$ ⟨(description)⟩

Updated: 2012-06-04

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 \LaTeXe category codes and the \LaTeX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the \GetIdInfo command you can use the information when loading a package with

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

The l3names module
Namespace for primitives

3.1 Setting up the \LaTeX3 programming language

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

- defines new names for all \TeX primitives;
- emulate required primitives not provided by default in 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: The TeXbook, \TeX by Topic and the manuals for pdflatex, Xe\TeX, Lua\TeX, \pdf\TeX and up\TeX should be consulted for details of the primitives. These are named \texttt{\tex\_⟨name⟩}, typically based on the primitive’s ⟨name⟩ in pdflatex and omitting a leading \texttt{pdf} when the primitive is not related to pdf output.
Part III
Programming Flow
Chapter 4

The \texttt{l3basics} module

Basic definitions

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

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

4.1 No operation functions

\verbatim{\texttt{prg\_do\_nothing}}

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

\verbatim{\texttt{scan\_stop}}

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

4.2 Grouping material

\verbatim{\texttt{group\_begin}} \verbatim{\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{\textbackslash group\_begin} must be matched by a \texttt{\textbackslash 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.

\texttt{\textbackslash \textbackslash texhackers note}: These are the \TeX{} primitives \texttt{\textbackslash begingroup} and \texttt{\textbackslash endgroup}.
\group_insert_after:N \group_insert_after:N \token

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

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

\group_show_list: \group_log_list:

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 \\epsilon-\TeX primitive \showgroups.

4.3 Control sequences and functions

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

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

4.3.1 Defining functions

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

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

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

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

\gset Create a new function with the \gset scope, such as \cs_gset:Npn. The definition is global and does not result in an error if the function is already defined.
Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

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

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

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

- T and F Functionally equivalent to n (you are actually encouraged to use the family of `\prg_new_conditional` functions described in Section 9.1).
- p and w These are special cases.

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

### 4.3.2 Defining new functions using parameter text

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

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

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

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

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

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

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

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

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

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 \texttt{e}-type or \texttt{x}-type argument. The definition is global and an error results if the \texttt{(function)} is already defined.

\texttt{\cs_set_protected_nopar:Nn} \texttt{(cn|Ne|ce)} \texttt{\cs_new_protected_nopar:Nn} \texttt{(code)} \texttt{(function)}

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 \texttt{e}-type or \texttt{x}-type argument. The definition is global and an error results if the \texttt{(function)} is already defined.

\texttt{\cs_set:Nn} \texttt{(cn|Ne|ce)} \texttt{\cs_set:Nn} \texttt{(code)} \texttt{(function)}

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

\texttt{\cs_set_nopar:Nn} \texttt{(cn|Ne|ce)} \texttt{\cs_set_nopar:Nn} \texttt{(code)} \texttt{(function)}

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

\texttt{\cs_set_protected:Nn} \texttt{(cn|Ne|ce)} \texttt{\cs_set_protected:Nn} \texttt{(code)} \texttt{(function)}

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

\texttt{\cs_set_protected_nopar:Nn} \texttt{(cn|Ne|ce)} \texttt{\cs_set_protected_nopar:Nn} \texttt{(code)} \texttt{(function)}

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

\texttt{\textbackslash cs\_gset:Nn \{function\} \{(code)\}}

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

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

\texttt{\textbackslash cs\_gset\_nopar:Nn \{function\} \{(code)\}}

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

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

\texttt{\textbackslash cs\_gset\_protected:Nn \{function\} \{(code)\}}

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

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

\texttt{\textbackslash cs\_gset\_protected\_nopar:Nn \{function\} \{(code)\}}

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

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

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

Updated: 2012-01-14

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

4.3.4 Copying control sequences

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

In the following text “cs” is used as an abbreviation for “control sequence”.
Globally creates \texttt{\texttt{<control sequence>1}} and sets it to have the same meaning as \texttt{\texttt{<control sequence>2}} or \texttt{\texttt{<token>}}. The second control sequence may subsequently be altered without affecting the copy.

Sets \texttt{\texttt{<control sequence>1}} to have the same meaning as \texttt{\texttt{<control sequence>2}} (or \texttt{\texttt{<token>}}). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the \texttt{\texttt{<control sequence>1}} is restricted to the current \TeX{} group level.

Globally sets \texttt{\texttt{<control sequence>1}} to have the same meaning as \texttt{\texttt{<control sequence>2}} (or \texttt{\texttt{<token>}}). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the \texttt{\texttt{<control sequence>1}} is not restricted to the current \TeX{} group level: the assignment is global.

### 4.3.5 Deleting control sequences

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

Sets \texttt{\texttt{<control sequence>}} to be globally undefined.

This function expands to the \texttt{\texttt{meaning}} of the \texttt{\texttt{<control sequence>}} control sequence. For a macro, this includes the \texttt{\texttt{replacement text}}.

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

Displays the definition of the \texttt{\texttt{<control sequence>}} on the terminal.

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

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

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

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

\TeXhackers note: The prefix can be empty, \texttt{\long}, \texttt{\protected} or \texttt{\protected\long} with backslash replaced by the current escape character.
\cs_parameter_spec:N          \cs_parameter_spec:N \langle \text{token} \rangle

If the \langle \text{token} \rangle is a macro, this function leaves the primitive \TeX parameter specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

\begin{verbatim}
\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 \text{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{-\textgreater}, then the function produces incorrect results.

\cs_replacement_spec:N          \cs_replacement_spec:N \langle \text{token} \rangle

\cs_replacement_spec:c          \cs_replacement_spec:c

\texttt{New: 2019-02-27}

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

\begin{verbatim}
\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 \text{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{-\textgreater}, then the function produces incorrect results.

### 4.5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the situation in force when first function absorbs the token).
\use:n \use:n \{ (group_1) \}
\use:nn \use:nn \{ (group_1) \} \{ (group_2) \}
\use:nnn \use:nnn \{ (group_1) \} \{ (group_2) \} \{ (group_3) \}
\use:nnnn \use:nnnn \{ (group_1) \} \{ (group_2) \} \{ (group_3) \} \{ (group_4) \}

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

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

results in the input stream containing

abc \{ def \}

i.e. only the outer braces are removed.

\TeXhackers note: The \use:n function is equivalent to \LaTeX's \@firstofone.
These functions absorb a number ($n$) arguments from the input stream. They then discard all arguments other than that indicated by the roman numeral, which is left in the input stream. For example, \texttt{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

\texttt{abc \{ def \}}

\textit{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 \texttt{n} arguments may be an unbraced single token (\textit{i.e.} an \texttt{N} argument).

\textbf{TEXhackers note:} These are equivalent to \LaTeX2ε’s \texttt{\@gobble, \@gobbletwo, etc.}

\use:e \{ \texttt{\{expandable tokens\}} \}

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

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

\subsection*{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.

\begin{verbatim}
\use_none_delimit_by_q_nil:w  * \use_none_delimit_by_q_nil:w \{balanced text\} \q_nil
\use_none_delimit_by_q_stop:w * \use_none_delimit_by_q_stop:w \{balanced text\} \q_stop
\use_none_delimit_by_q_recursion_stop:w * \use_none_delimit_by_q_recursion_stop:w \{balanced text\}
\q_recursion_stop
\end{verbatim}

Absorb the \texttt{(balanced text)} from the input stream delimited by the marker given in
the function name, leaving nothing in the input stream.
Absorb the \langle balanced text \rangle from the input stream delimited by the marker given in the function name, leaving \langle inserted tokens \rangle in the input stream for further processing.

4.6 Predicates and conditionals

\LaTeX{} has three concepts for conditional flow processing:

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

\[
\texttt{\cs_if_free:cTF \{abc\} \{\langle true code \rangle\} \{\langle false code \rangle\}}
\]

a function that turns the first argument into a control sequence (since it’s marked as \texttt{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 \texttt{TF} function is defined it is usually accompanied by \texttt{T} and \texttt{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 \langle true code \rangle and/or \langle false code \rangle are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

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

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

\[
\texttt{\cs_if_free_p:N}
\]

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

\[
\texttt{\bool_if:nTF \{ \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl \}{\langle true code\rangle} {\langle false code\rangle}}
\]

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

4.6.1 Tests on control sequences

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

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

\cs_if_exist_p:N * \cs_if_exist_p:N \langle control sequence \rangle
\cs_if_exist_p:c * \cs_if_exist_p:c \langle control sequence \rangle
\cs_if_exist:NTF * \cs_if_exist:NTF \langle control sequence \rangle \{\langle true code \rangle\} \{\langle false code \rangle\}
\cs_if_exist:cTF * \cs_if_exist:cTF \langle control sequence \rangle

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

4.6.2 Primitive conditionals

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

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

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

\TeXhackers note: \if_true: and \if_false: are equivalent to their corresponding \TeX primitive conditionals \\texttt{\iftrue} and \texttt{\iffalse}; \\else: and \fi: are the \TeX primitives \texttt{\else} and \texttt{\fi}; \\reverse_if:N is the \\epsilon\TeX primitive \texttt{\unless}.
\if_meaning:w \if_meaning:w \arg_1 \arg_2 \true\ else: \false\ \fi:
\else: \false\ \fi:

\if_meaning:w \executes\ \true\ when \arg_1 \and \arg_2 \are the same, otherwise \executes\ \false. \arg_1 \and \arg_2 \could be functions, variables, tokens; in all cases the unexpanded definitions are compared.

\TeXhackers\ note: This is the \TeX\ primitive \texttt{\ifx}.

\if:w \if:w \token(s) \true\ else: \false\ \fi:
\else: \false\ \fi:
\if_catcode:w \if_catcode:w \token(s) \true\ else: \false\ \fi:
\else: \false\ \fi:
\if_charcode:w \if_charcode:w \token(s) \true\ else: \false\ \fi:
\else: \false\ \fi:
\if_mode_horizontal: \if_mode_horizontal: \true\ else: \false\ \fi:
\else: \false\ \fi:
\if_mode_vertical: \if_mode_vertical: \true\ else: \false\ \fi:
\else: \false\ \fi:
\if_mode_math: \if_mode_math: \true\ else: \false\ \fi:
\else: \false\ \fi:
\if_mode_inner: \if_mode_inner: \true\ else: \false\ \fi:
\else: \false\ \fi:

\if_charcode:w \if_charcode:w is an alternative name for \texttt{\if}. These conditionals expand \texttt{\token(s)} until two unexpandable tokens \texttt{\token_1} and \texttt{\token_2} are found; any further tokens up to the next unbalanced \texttt{\else:} are the true branch, ending with \texttt{\true}. It is executed if the condition is fulfilled, otherwise \texttt{\false} is executed. You can omit \texttt{\else:} when just in front of \texttt{\fi:} and you can nest \texttt{\if...\else:...\fi:} constructs inside the true branch or the \texttt{\false}. With \texttt{\exp_not:N}, you can prevent the expansion of a token.

\if_catcode:w \if_catcode:w tests if \texttt{\token_1} and \texttt{\token_2} have the same category code whereas \texttt{\if:w and \if_charcode:w} test if they have the same character code.

\TeXhackers\ note: \texttt{\if:w and \if_charcode:w} are both the \TeX\ primitive \texttt{\if}. \texttt{\if_catcode:w} is the \TeX\ primitive \texttt{\ifcat}.

\if_cs_exist:N \if_cs_exist:N \cs \true\ else: \false\ \fi:
\else: \false\ \fi:
\if_cs_exist:w \if_cs_exist:w \tokens \cs_end: \true\ else: \false\ \fi:
\else: \false\ \fi:

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

\TeXhackers\ note: These are the \TeX\ primitives \texttt{\ifdefined} and \texttt{\ifcsname}.

\if_mode_horizontal: \if_mode_horizontal: \true\ else: \false\ \fi:
\if_mode_vertical: \if_mode_vertical: \true\ else: \false\ \fi:
\if_mode_math: \if_mode_math: \true\ else: \false\ \fi:
\if_mode_inner: \if_mode_inner: \true\ else: \false\ \fi:

Execute \texttt{\true} if currently in horizontal mode, otherwise execute \texttt{\false}. Similar for the other functions.

\TeXhackers\ note: These are the \TeX\ primitives \texttt{\ifhmode}, \texttt{\ifvmode}, \texttt{\ifmmode}, and \texttt{\ifinner}. 
4.7 Starting a paragraph

\texttt{\textbackslash 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 \texttt{\parindent}, followed by the \texttt{\everypar} token list.

\textbf{\TeX{}hackers note}: This results in the contents of the \texttt{\everypar} token register being inserted, after \texttt{\mode\_leave\_vertical:} is complete. Notice that in contrast to the \LaTeX{}\texttt{2e} \texttt{\leavevmode} approach, no box is used by the method implemented here.

4.8 Debugging support

\texttt{\debug\_on:n} \texttt{\debug\_off:n} \texttt{\debug\_suspend:} \texttt{\debug\_resume:} 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 \texttt{\langle \textit{list} \rangle} are

- \texttt{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;
- \texttt{check-expressions} that checks integer, dimension, skip, and miskip expressions are not terminated prematurely;
- \texttt{deprecation} that makes deprecated commands produce errors;
- \texttt{log-functions} that logs function definitions and variable declarations;
- all that does all of the above.

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

\texttt{\debug\_suspend:} ... \texttt{\debug\_resume:} Suppress (locally) errors and logging from \texttt{debug} commands, except for the \texttt{deprecation} errors. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance \texttt{l3ctab} and \texttt{l3coffins}. 

30
Chapter 5

The \texttt{l3expan} module
Argument expansion

This module provides generic methods for expanding \TeX\ arguments in a systematic manner. The functions in this module all have prefix \texttt{exp}.

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

5.1 Defining new variants

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

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

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

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

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

results in the definition of \texttt{\seq_gpush:No}
Providing variants in this way in style files is safe as the \texttt{\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 \texttt{\cs_generate_variant:Nn}, described next.

\subsection*{5.2 Methods for defining variants}

We recall the set of available argument specifiers.

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

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

- A few odd argument types remain: \texttt{T} and \texttt{F} for conditional processing, otherwise identical to \texttt{n}-type arguments, \texttt{p} for the parameter text in definitions, \texttt{w} for arguments with a specific syntax, and \texttt{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 \LaTeX 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 { NV , cV }
\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 \texttt{check-declarations} is enabled.) If the \langle parent control sequence \rangle is protected or if the \langle variant \rangle involves any \texttt{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 \texttt{n} and \texttt{N} arguments can be changed to other types. The only allowed changes are

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

This means the \langle parent \rangle of a \langle variant \rangle form is always unambiguous, even in cases where both an \texttt{n}-type parent and an \texttt{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 \texttt{n}, \texttt{o}, \texttt{V}, \texttt{v}, \texttt{f}, \texttt{e}, or \texttt{x}-type variants of an \texttt{N}-type argument or \texttt{N} or \texttt{c}-type variants of an \texttt{n}-type argument. Both are deprecated. The first because passing more than one token to an \texttt{N}-type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a \texttt{V}-type or \texttt{v}-type variant instead of \texttt{c}-type. In those cases, using the lower-level \exp_args:No or \exp_args:Nc functions explicitly is preferred to defining confusing variants.
\exp_args_generate:n \exp_args_generate:n \{(\textit{variant argument specifiers})\}

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

5.3 Introducing the variants

The \textit{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 \textit{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 \textit{V} specifier should be used. For those referred to by (cs)name, the \textit{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 \textit{o} specifiers be employed.

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

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

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

Note that \textit{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 \textit{o}-expansion would not expand \texttt{\int_eval:n} fully to its result since that function requires several expansions. Besides the fact that \textit{x}-expansion is protected rather than expandable, another difference between \textit{f}-expansion and \textit{x}-expansion is that \textit{f}-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while \textit{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 \texttt{x}-type. If you use \texttt{f} type expansion in conditional processing then you should stick to using \texttt{TF} type functions only as the expansion does not finish any \texttt{\iff}...\texttt{\fi}: itself!

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

\begin{verbatim}
\exp_after:wN <base function> \exp_after:wN { <argument> }
\end{verbatim}

At the same time, \texttt{f}-type expansion stops at the \textit{first} non-expandable token. This means for example that both

\begin{verbatim}
\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }
\end{verbatim}

and

\begin{verbatim}
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
\end{verbatim}

leave \texttt{\g_tmpb_tl} unchanged: \texttt{\{ 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 \texttt{x}-type arguments (that are fully expanded before being passed to the \texttt{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 \texttt{f} or \texttt{e} expansion.

- In contrast, \texttt{e} expansion (full expansion, almost like \texttt{x} except for the treatment of \texttt{#}) does not prevent variants from being expandable (if the base function is).

- Finally \texttt{f} expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

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

- for fastest processing any \texttt{c}-type arguments should come first followed by all other modified arguments;

- unchanged \texttt{N}-type args that appear before modified ones have a small performance hit;

- unchanged \texttt{n}-type args that appear before modified ones have a relative larger performance hit.

\subsection{5.4 Manipulating the first argument}

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.
\exp_args:Nc \exp_args:Nc \langle \text{function} \rangle \{\langle \text{tokens} \rangle\}

This function absorbs two arguments (the \langle \text{function} \rangle name and the \langle \text{tokens} \rangle). The \langle \text{tokens} \rangle 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 \langle \text{function} \rangle. Thus the \langle \text{function} \rangle may take more than one argument: all others are left unchanged.

The \text{:cc} variant constructs the \langle \text{function} \rangle name in the same manner as described for the \langle \text{tokens} \rangle.

\exp_args:No \exp_args:No \langle \text{function} \rangle \{\langle \text{tokens} \rangle\} ...

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

\exp_args:NV \exp_args:NV \langle \text{function} \rangle \langle \text{variable} \rangle

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

\exp_args:Nv \exp_args:Nv \langle \text{function} \rangle \{\langle \text{tokens} \rangle\}

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

\exp_args:Ne \exp_args:Ne \langle \text{function} \rangle \{\langle \text{tokens} \rangle\}

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

\exp_args:Nf \exp_args:Nf \langle \text{function} \rangle \{\langle \text{tokens} \rangle\}

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

\exp_args:Nx \exp_args:Nx \langle \text{function} \rangle \{\langle \text{tokens} \rangle\}

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

New: 2018-05-15

\exp_args:Nf \begin{align*}
\exp_args:Nf \langle \text{function} \rangle \{\langle \text{tokens} \rangle\}
\end{align*}

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

\exp_args:Nx \exp_args:Nx \langle \text{function} \rangle \{\langle \text{tokens} \rangle\}

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

\exp_args:NNc \langle token_1 \rangle \langle token_2 \rangle \{\langle tokens \rangle\}

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

\exp_args:Nnc \langle token \rangle \{\langle tokens_1 \rangle\} \{\langle tokens_2 \rangle\}

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

\exp_args:Nnx \langle token_1 \rangle \langle token_2 \rangle \{\langle tokens \rangle\}

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

5.6 Manipulating three arguments

\exp_args:NNNo \langle token_1 \rangle \langle token_2 \rangle \langle token_3 \rangle \{\langle tokens \rangle\}

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

\exp_last_unbraced:N no \langle \text{token} \rangle \{ \langle \text{tokens}_1 \rangle \} \{ \langle \text{tokens}_2 \rangle \}

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

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

\exp_last_unbraced:Nx \langle \text{function} \rangle \{ \langle \text{tokens} \rangle \}

This function fully expands the \langle \text{tokens} \rangle and leaves the result in the input stream after reinsertion of the \langle \text{function} \rangle. This function is not expandable.

\exp_last_two_unbraced:Noo \text{*} \exp_last_two_unbraced:Noo \langle \text{token} \rangle \{ \langle \text{tokens}_1 \rangle \} \{ \langle \text{tokens}_2 \rangle \}

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

\exp_after:wN \langle \text{token}_1 \rangle \langle \text{token}_2 \rangle

Carries out a single expansion of \langle \text{token}_2 \rangle (which may consume arguments) prior to the expansion of \langle \text{token}_1 \rangle. If \langle \text{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 \text{token}_1 \rangle may be \textit{any} single token, including group-opening and -closing tokens (\{ or \} assuming normal \TeX\ category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate \texttt{\exp_{-}args:N(variant)} function.

\textbf{\TeXhackers note:} This is the \TeX primitive \texttt{\expandafter}.
5.8 Preventing expansion

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

\exp_not:N \langle \text{token} \rangle

Prevents expansion of the \langle \text{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.

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

\exp_not:c \{ \langle \text{tokens} \rangle \}

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

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

\textbf{\TeXhackers note:} This is the \e\TeX primitive \texttt{\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 \texttt{1}, and \exp_not:n \{\#\} is equivalent to \#, namely it inserts the character \#.

\exp_not:o \{ \langle \text{tokens} \rangle \}

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

\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 \{ \langle \text{tokens} \rangle \}

Expands the \langle \text{tokens} \rangle until only characters remains, and then converts this into a control 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 \{\langle tokens\rangle\}

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

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

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

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

This function terminates an f-type expansion. Thus if a function \foo_bar:f starts an f-type expansion and all of \langle tokens\rangle are expandable \exp_stop_f: terminates the expansion of tokens even if \langle more tokens\rangle are also expandable. The function itself is an implicit space token. Inside an e-type or x-type expansion, it retains its form, but when typeset it produces the underlying space (\hspace{1em}).

### 5.9 Controlled expansion

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

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

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

Expands \langle \text{expandable-tokens} \rangle until reaching \exp:w \exp_end: at which point expansion stops. 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:w \exp_end: will be misinterpreted later on.\(^{3}\)

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

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

where somewhere during the expansion of \@@_case:NnTF the \exp:w \exp_end: gets generated.\(^{3}\)

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

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

Expands \langle \text{expandable-tokens} \rangle until reaching \exp:w \exp_end_continue_f:w \langle \text{further-tokens} \rangle 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 \exp_end: will be misinterpreted later on.\(^{4}\)

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

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

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

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

\exp:w \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_stop_f: \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.

\(^{3}\)Due to the implementation you might get the character in position 0 in the current font (typically \text{"\texttt{\textup{^0}}}\) in the output without any error message!
The difference to \exp_end_continue_f:w is that we first we pick up an argument which is then returned to the input stream. If \further-tokens starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

5.10 Internal functions

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

Internal forms for the base expansion types. These names do not conform to the general \LaTeX{}3 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_args:Ncof { \::c \::o \::f \::: }
\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 \LaTeX{}3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

\footnote{In this particular case you may get a character into the output as well as an error message.}
Chapter 6

The \texttt{l3sort} module

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

\textbf{\TeX{}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{\langle comparison code\rangle} should not call \texttt{\newtoks} or other commands that allocate new \texttt{\toks} registers. On the other hand, altering the value of a previously allocated \texttt{\toks} register is not a problem.
\texttt{\textbackslash sort\_return\_same}: \texttt{\textbackslash seq\_sort:nn (seq var)}
\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.
This module provides functions that are particularly useful in the \l3regex module for mapping through a token list one (token) at a time (including begin-group/end-group tokens). For \tl_analysis_map_inline:Nn or \tl_analysis_map_inline:nn, the token list is given as an argument; the analogous function \peek_analysis_map_inline:n documented in \l3token finds tokens in the input stream instead. In both cases the user provides (inline code) that receives three arguments for each (token):

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

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

\tl_analysis_show:N
\tl_analysis_show:n \{\token list\}
\tl_analysis_log:n \{\token list\}

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

\tl_analysis_map_inline:nn
\tl_analysis_map_inline:nn \{\token list\} \{\inline function\}

\tl_analysis_map_inline:Nn

Updates the (inline function) to each individual (token) in the (token list). The (inline function) receives three arguments as explained above. As all other mappings the mapping is done at the current group level, i.e. any local assignments made by the (inline function) remain in effect after the loop.
Chapter 8

The \texttt{l3regex} module

Regular expressions in \TeX

The \texttt{l3regex} module provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the 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\text{-}cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
\end{verbatim}
the token list variable \texttt{\l_my_tl} holds the text “This 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 { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
\end{verbatim}

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

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using \texttt{\regex_set:Nn}. For example,
\begin{verbatim}
\regex_new:N \l_foo_regex
\regex_set:Nn \l_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
\end{verbatim}
stores in \texttt{\l_foo_regex} a regular expression which matches the starting marker for an environment: \texttt{\begin}, followed by a begin-group token (\texttt{\cB}), then any number of tokens which are neither begin-group nor end-group character tokens (\texttt{\c[^BE].*}), ending with an end-group token (\texttt{\cE}). As explained in the next section, the parentheses “capture” the result of \texttt{\c[^BE].*}, giving us access to the name of the environment when doing replacements.
8.1 Syntax of regular expressions

8.1.1 Regular expression examples

We start with a few examples, and encourage the reader to apply `\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.
- `[^\+\-=\*\+\-]+` matches an explicit integer with at most one sign.
- `[^\+\-=\*\+\-]+\d+` matches an explicit integer with any number of `+` and `−` signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[^\+\-=\*\+\-]+\d+\d+` matches an explicit integer or decimal number; using `[.\]` instead of `\` would allow the comma as a decimal marker.
- `[^\+\-=\*\+\-]+\d+\d+\d+` matches an explicit dimension with any unit that TeX knows, where `(\?i)` means to treat lowercase and uppercase letters identically.
- `[^\+\-=\*\+\-]+\d+\d+\d+\d+` matches an explicit floating point number or the special values `\nan` and `\inf` (with signs and spaces allowed).
- `[^\+\-=\*\+\-]+\d+\cC.+` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\G.*?\K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (\G) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_-extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[^\+\-=\*\+\-]+\d+\d+\d+\d+\d+` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.
8.1.2 Characters in regular expressions
Most characters match exactly themselves, with an arbitrary category code. Some charac-
ters 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.

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].

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

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

\D Any token not matched by \d.

\H Any token not matched by \h.

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

\S Any token not matched by \s.

\V Any token not matched by \v.

\W Any token not matched by \w.

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

Character classes match exactly one token in the subject.

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

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

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

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

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

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

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

8.1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

? 0 or 1, greedy.

?? 0 or 1, lazy.

* 0 or more, greedy.

*? 0 or more, lazy.

+ 1 or more, greedy.
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.

(...:... Non-capturing group.

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

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

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

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

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

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

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

8.1.5 Matching exact tokens

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

• C for control sequences;
• B for begin-group tokens;
• E for end-group tokens;
• M for math shift;
• T for alignment tab tokens;
• P for macro parameter tokens;
• U for superscript tokens (up);
• D for subscript tokens (down);
• S for spaces;
• L for letters;
• O for others; and
• A for active characters.

The \c escape sequence is used as follows.

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

\cX Applies to the next object, which can be a character, escape character sequence such as \x{0A}, character class, or group, and forces this object to only match tokens with category X (any of CBEMTPUDSLOA). For instance, \cL[A-Z\d] matches uppercase letters and digits of category code letter, \cC. matches any control sequence, and \cO(abc) matches abc where each character has category other.\footnote{This last example also captures “abc” as a regex group; to avoid this use a non-capturing group \cO(?!:abc).}

\c[XYZ] Applies to the next object, and forces it to only match tokens with category X, Y, or Z (each being any of 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 $\text{T}_{\text{PX}}$ 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{1_tmpa_regex}D matches the tokens A and
D separated by something that matches the regular expression \l_tmpa_regex. This behaves as if a non-capturing group were surrounding \l_tmpa_regex, and any group contained in \l_tmpa_regex is converted to a non-capturing group. Quantifiers are supported.

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

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

8.1.6 Miscellaneous

Anchors and simple assertions.

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

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

^ or \A Start of the subject token list.

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

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

The option (?i) makes the match case insensitive (treating A–Z and a–z as equivalent, with no support yet for Unicode case changing). This applies until the end of the group in which it appears, and can be reverted using (?-i). For instance, in (?i)(a(?-i)b|c)d, the letters a and d are affected by the i option. Characters within ranges and classes are affected individually: (?i)[?-B] is equivalent to \l?@ABab (and differs from the much larger class \l?-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 (\ldots); similarly for \2, \ldots, \9 and \g{number};
- \␣ inserts a space (spaces are ignored when not escaped);
• \a, \e, \f, \n, \r, \t, \xhh, \x{hhh} correspond to single characters as in regular expressions;
• \c\{cs name\} inserts a control sequence;
• \c\langle category\rangle\langle character\rangle (see below);
• \u\{tl var name\} inserts the contents of the \langle tl \rangle (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:nnN { \[^,]+ } { \u\{l_my_\0_tl\} } \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 \x20 or \x{20} have category code 10 regardless of the prevailing category code regime;
• if the category code would be 0 (escape), 5 (newline), 9 (ignore), 14 (comment) or 15 (invalid), it is replaced by 12 (other) instead.

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

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

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

The escape sequence \u\{\langle var name \rangle\} allows to insert the contents of the variable with name \langle var name \rangle 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,
\begin{verbatim}
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { \[^,]+ } { \u\{l_my_\0_tl\} } \l_my_tl
\end{verbatim}
results in \l_my_tl holding first,\textit{second},first,first.

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

\begin{verbatim}
\tl_clear:N \l_tmpa_tl
\regex_replace_all:nnN { } { \cU\% \cA\~ } \l_tmpa_tl
\end{verbatim}

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

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

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

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

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

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

leaves TRUE then FALSE in the input stream.

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

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

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

results in \l_foo_int taking the value 5.

\regex_match_case:nnTF 
\regex_match_case:nnN 
\regex_match_case:nnN 
\regex_match_case:NnN 
\regex_match_case:NVN

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

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

\regex_extract_once:nnN \regex_extract_once:nnNTF \regex_extract_once:NnN \regex_extract_once:nVN \regex_extract_all:nnN \regex_extract_all:nnNTF \regex_extract_all:NnN \regex_extract_all:nVN

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 \} \{ LaTeX!!! \} \l_foo_seq \{ true \} \{ false \}

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

\regex_extract_all:nnN \regex_extract_all:nnNTF \regex_extract_all:NnN \regex_extract_all:nVN

Finds all matches of the \langle regular expression \rangle in the \langle token list \rangle, and stores all the submatch information in a single sequence (concatenating the results of multiple \regex_extract_once:nnN calls). 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_all:nnNTF \{ \w+ \} \{ Hello,-world! \} \l_foo_seq \{ true \} \{ false \}

Then the regular expression matches twice, the resulting sequence contains the two items \{Hello\} and \{world\}, and the true branch is left in the input stream.
Splits the \langle token list \rangle into a sequence of parts, delimited by matches of the \langle regular expression \rangle. If the \langle regular expression \rangle has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to \langle seq var \rangle is local. If no match is found the resulting \langle seq var \rangle has the \langle token list \rangle as its sole item. If the \langle regular expression \rangle matches the empty token list, then the \langle token list \rangle is split into single tokens. 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 example, after

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

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

### 8.6 Replacement

Searches for the \langle regular expression \rangle in the contents of the \langle tl var \rangle and replaces the first match with the \langle replacement \rangle. In the \langle replacement \rangle, \texttt{\0} represents the full match, \texttt{\1} represent the contents of the first capturing group, \texttt{\2} of the second, etc. The result is assigned locally to \langle tl var \rangle.

Replaces all occurrences of the \langle regular expression \rangle in the contents of the \langle tl var \rangle by the \langle replacement \rangle, where \texttt{\0} represents the full match, \texttt{\1} represent the contents of the first capturing group, \texttt{\2} of the second, etc. Every match is treated independently, and matches cannot overlap. The result is assigned locally to \langle tl var \rangle.
Replaces the earliest match of the regular expression \(?!(\text{regex}_1)|...|(\text{regex}_n)\) 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 \textit{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_case_once:nN}. This is equivalent to checking with \texttt{\regex_match_case:nN} which \textit{regex} matches, then performing the replacement with \texttt{\regex_replace_once:nN}.

Replaces all occurrences of all \textit{regex} in the \textit{token list} by the corresponding \textit{replacement}. Every match is treated independently, and matches cannot overlap. The result is assigned locally to \textit{tl var}, and the \textit{true code} or \textit{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 \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}, and the search resumes at the position that follows this match (and replacement). For instance

\begin{verbatim}
\tl_set:Nn \l_tmpa_tl { Hello,-world! }
\regex_replace_case_all:n
\end{verbatim}

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

59
8.7 Scratch regular expressions

\l_tmpa_regex
\l_tmpb_regex

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

\g_tmpa_regex
\g_tmpb_regex

Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX-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 \_\_regex_balance_tl to build the function \_\_regex_replacement_balance_one_match:n.

• Reduce the number of epsilon-transitions in alternatives.

• Optimize simple strings: use less states (abcade should give two states, for abc and ade). [Does that really make sense?]

• Optimize groups with no alternative.

• Optimize states with a single \_\_regex_action_free:n.

• Optimize the use of \_\_regex_action_success: by inserting it in state 2 directly instead of having an extra transition.

• Optimize the use of \int_step.... functions.

• Groups don’t capture within regexes for csnames; optimize and document.

• Better “show” for anchors, properties, and catcode tests.

• Does \K really need a new state for itself?

• When compiling, use a boolean in_cs and less magic numbers.

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

• General look-ahead/behind assertions.

• Regex matching on external files.

• Conditional subpatterns with look ahead/behind: “if what follows is […] then […]”.

• (*..) and (?..) sequences to set some options.

• UTF-8 mode for pdf\TeX.

• 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 treebboxes.

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 \texttt{PCRE} or Perl will definitely not be implemented.

• Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.

• Recursion: this is a non-regular feature.

• Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.

• Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.

• Backtracking control verbs: intrinsically tied to backtracking.

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

• \texttt{	extbackslash c\textbackslash x}, similar to \TeX\’s own \textbackslash ^\textbackslash x.

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

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

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

The l3prg module

Control structures

Conditional processing in \LaTeX{}3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a state is returned. The states returned are \langle true \rangle and \langle false \rangle.

\LaTeX{}3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean \langle true \rangle or \langle false \rangle. For example, the function \texttt{\cs_if_free_p:N} checks whether the control sequence given as its argument is free and then returns the boolean \langle true \rangle or \langle false \rangle values to be used in testing with \texttt{\if_predicate:w} or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in \texttt{\cs_if_free:NTF} which also takes one argument (the N) and then executes either \texttt{true} or \texttt{false} depending on the result.

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

9.1 Defining a set of conditional functions

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

These functions create a family of conditionals using the same \langle code \rangle to perform the test created. Those conditionals are expandable if \langle code \rangle is. The new versions check for existing definitions and perform assignments globally (\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 \langle conditions \rangle, which should be one or more of p, T, F and TF.
These functions create a family of protected conditionals using the same \(\langle \text{code}\rangle\) to perform the test created. The \(\langle \text{code}\rangle\) does not need to be expandable. The new version check for existing definitions and perform assignments globally (cf. \texttt{\cs_new:Npn}) whereas the set version do not (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 T, F and TF (not 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 true or logical false. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for protected conditionals.
- \(\langle \text{name}\rangle:\langle \text{arg spec}\rangle\)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 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 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 true, while the \(\langle \text{false branch}\rangle\) code in the second argument will be left on the input stream if the test is 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 (cf. \texttt{\cs_new:Nn}, 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:

\begin{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:
\end{verbatim}

64
This defines the function \( \texttt{foo_if_bar_p:NN} \), \( \texttt{foo_if_bar:NNTF} \) and \( \texttt{foo_if_bar:NNT} \) but not \( \texttt{foo_if_bar:NNF} \) (because F is missing from the \( \langle \text{conditions} \rangle \) 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.

The special case where the code of a conditional ends with \( \texttt{prg_return_true:} \), \( \texttt{else:} \) \( \texttt{prg_return_false:} \) \( \texttt{fi:} \) is optimized.

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

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

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

Defines argument-specifier variants of conditionals. This is equivalent to running \( \texttt{cs_generate_variant:Nn} \langle \text{conditional} \rangle \{\langle \text{variant argument specifiers} \rangle \} \langle \text{condition specifiers} \rangle \) on each \( \langle \text{conditional} \rangle \) described by the \( \langle \text{condition specifiers} \rangle \). These base-form \( \langle \text{conditionals} \rangle \) are obtained from the \( \langle \text{name} \rangle \) and \( \langle \text{arg spec} \rangle \) as described for \( \texttt{prg_new_conditional:Npn} \), and they should be defined.

### 9.2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (\( \texttt{e.g., draft/final} \)) and other times you perhaps want to use it as a predicate function in an \( \texttt{if_predicate:w} \) test. The problem of the primitive \( \texttt{if_false:} \) and
\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 bools: \texttt{\c_true_bool} or \texttt{\c_false_bool}. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations \texttt{And}, \texttt{Or}, \texttt{Not}, \textit{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 bools, possibly nested).

\begin{itemize}
\item \texttt{\bool_new:N} \texttt{\bool_new:c} (\textit{boolean}) \quad Creates a new \textit{boolean} or raises an error if the name is already taken. The declaration is global. The \textit{boolean} is initially \texttt{false}.
\item \texttt{\bool_set_true:N} \texttt{\bool_set_true:c} \texttt{\bool_gset_true:N} \texttt{\bool_gset_true:c} (\textit{boolean}) \quad Sets \textit{boolean} logically \texttt{true}.
\item \texttt{\bool_set_inverse:N} \texttt{\bool_set_inverse:c} \texttt{\bool_gset_inverse:N} \texttt{\bool_gset_inverse:c} \quad Toggles the \textit{boolean} from \texttt{true} to \texttt{false} and conversely: sets it to the inverse of its current value.
\end{itemize}

\TeX{}hackers note: The \texttt{\bool_} data type is not implemented using the \texttt{\iffalse}/\texttt{\iftrue} primitives, in contrast to \texttt{\newif}, \textit{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_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 \bool_gset_eq:NN \bool_gset_eq:cc
\bool_set:Nn \bool_set:cn \bool_gset:Nn \bool_gset:cn
\end{verbatim}

\begin{verbatim}
Updated: 2017-07-15
\end{verbatim}

\begin{verbatim}
\bool_set_inverse:N \bool_set_inverse:c \bool_gset_inverse:N \bool_gset_inverse:c
\end{verbatim}

\begin{verbatim}
Updated: 2018-05-10
\end{verbatim}
\bool_if_p:N \bool_if_p:c \bool_if:N \bool_if:cTF
Tests the current truth of \textit{boolean}, and continues expansion based on this result.

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

\bool_show:N \bool_show:c \bool_show:n
Displays the logical truth of the \textit{boolean} or \textit{boolean expression} on the terminal.

\bool_log:N \bool_log:c \bool_log:n
Writes the logical truth of the \textit{boolean} or \textit{boolean expression} in the log file.

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

\c_true_bool \c_false_bool
Constants that represent \texttt{true} and \texttt{false}, respectively. Used to implement predicates.

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

9.2.1 Constant and scratch booleans
A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any \texttt{\LaTeX}3-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 \texttt{true} or \texttt{false} values, it seems only fitting that we also provide a parser for \texttt{boolean expressions}.

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

\begin{verbatim}
\int_compare_p:n { 1 = 1 } \&\&
  \begin{verbatim}
  \int_compare_p:n { 2 = 3 } ||
  \int_compare_p:n { 4 <= 4 } ||
  \str_if_eq_p:nn { abc } { def } \&\&
  \end{verbatim}
\end{verbatim}
! \int_compare_p:n { 2 = 4 }
\end{verbatim}

is a valid boolean expression.

Contrarily to some other programming languages, the operators \&\& and || evaluate both operands in all cases, even when the first operand is enough to determine the result. This “eager” evaluation should be contrasted with the “lazy” evaluation of \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 \texttt{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_and_p:nn
    { \int_compare_p:n { 2 = 3 } }%�行的输出结果为0，因为2 != 3
    { \int_compare_p:n { 4 <= 4 } }%行的输出结果为1，因为4 <= 4是true
      { \int_compare_p:n { 1 = \error } }%第3行被跳过了，因为1 != error
    }%第2行被跳过了，因为第1行已经是false，所以第2行不被计算
  }%第1行被跳过了，因为第0行已经是false，所以第1行不被计算
! \int_compare_p:n { 2 = 4 } ]
\end{verbatim}

68
the line marked with skipped is not expanded because the result of `bool_lazy_any_p`:nn is known once the second boolean expression is found to be logically true. On the other hand, the last line is expanded because its logical value is needed to determine the result of `bool_lazy_and_p`:nn.

```
\bool_if_p:n * \bool_if_p:n \{\{boolean \ expression\}\} {\{true \ code\}} {\{false \ code\}}
```

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

```
\bool_lazy_all_p:n * \bool_lazy_all_p:n \{\{boolexpr\}\} \{\{boolexpr\}\} \cdots \{\{boolexpr\}\}\} {\{true \ code\}} {\{false \ code\}}
```

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

```
\bool_lazy_and_p:nn * \bool_lazy_and_p:nn \{\{boolexpr\}\} \{\{boolexpr\}\} \cdots \{\{boolexpr\}\}\} {\{true \ code\}} {\{false \ code\}}
```

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

```
\bool_lazy_any_p:n * \bool_lazy_any_p:n \{\{boolexpr\}\} \{\{boolexpr\}\} \cdots \{\{boolexpr\}\}\} {\{true \ code\}} {\{false \ code\}}
```

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

```
\bool_lazy_or_p:nn * \bool_lazy_or_p:nn \{\{boolexpr\}\} \{\{boolexpr\}\} \cdots \{\{boolexpr\}\}\} {\{true \ code\}} {\{false \ code\}}
```

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

```
\bool_not_p:n * \bool_not_p:n \{\{boolean \ expression\}\}
```

Function version of !(`\{boolean \ expression\}`) within a boolean expression.
\bool_xor_p:nn \bool_xor_p:nn \{\texttt{boolexpr}_1\} \{\texttt{boolexpr}_2\}

\bool_xor:nnTF \bool_xor:nnTF \{\texttt{boolexpr}_1\} \{\texttt{boolexpr}_2\} \{\texttt{true code}\} \{\texttt{false code}\}

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

### 9.4 Logical loops

Loops using either boolean expressions or stored boolean values.

\bool_do_until:Nn \bool_do_until:Nn \{\texttt{boolean}\} \{\texttt{code}\}

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

\bool_do_while:Nn \bool_do_while:Nn \{\texttt{boolean}\} \{\texttt{code}\}

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

\bool_until_do:Nn \bool_until_do:Nn \{\texttt{boolean}\} \{\texttt{code}\}

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

\bool_while_do:Nn \bool_while_do:Nn \{\texttt{boolean}\} \{\texttt{code}\}

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

\bool_do_until:nn \bool_do_until:nn \{\texttt{boolean expression}\} \{\texttt{code}\}

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

\bool_do_while:nn \bool_do_while:nn \{\texttt{boolean expression}\} \{\texttt{code}\}

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

\bool_until_do:nn \bool_until_do:nn \{\texttt{boolean expression}\} \{\texttt{code}\}

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

This function first checks the logical value of the \textit{boolean expression} (as described for \bool_if:nTF). If it is \texttt{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 \texttt{false}.

\bool_case:n \bool_case:n \{ \{ \textit{boolexpr case}_1 \} \{ \textit{code case}_1 \} \} \{ \{ \textit{boolexpr case}_2 \} \{ \textit{code case}_2 \} \} \ldots \{ \{ \textit{boolexpr case}_n \} \{ \textit{code case}_n \} \} \{ \{ \texttt{true code} \} \{ \texttt{false code} \} \}

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

\bool_case:nF
\{ \dim_compare_p:n \{ \__mypkg_wd_dim <= \num{10} \} \} \{ \texttt{Fits} \} \{ \int_compare_p:n \{ \__mypkg_total_int >= \num{10} \} \} \{ \texttt{Many} \} \{ \__mypkg_special_bool \} \{ \texttt{Special} \} \} \{ \texttt{No idea!} \}

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

### 9.5 Producing multiple copies

\prg_replicate:nn \prg_replicate:nn \{\textit{integer expression}\} \{\textit{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.

### 9.6 Detecting \TeX’s mode

\mode_if_horizontal_p: \mode_if_horizontal_p: \{ \textit{true code} \} \{ \texttt{false code} \}

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

For types with mappings defined in the kernel, \texttt{\type\_map\_break} and \texttt{\type\_map\_break:n} are defined as \texttt{\prg\_map\_break:Nn \type\_map\_break: \{\}} and the same with \{\} omitted.

### 9.8.1 Simple mappings

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

\begin{itemize}
  \item \texttt{\prg\_break:n} \{\textit{code}\} \ldots \texttt{\prg\_break\_point:}
  \end{itemize}

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

### 9.9 Internal programming functions

\begin{itemize}
  \item \texttt{\group\_align\_safe\_begin:}
  \item \texttt{\group\_align\_safe\_end:}
  \end{itemize}

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

The \texttt{l3sys} module

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{\TeXhacksnote{}}: 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

\begin{itemize}
\item \texttt{\textbackslash c\_sys\_minute\_int}
\item \texttt{\textbackslash c\_sys\_hour\_int}
\item \texttt{\textbackslash c\_sys\_day\_int}
\item \texttt{\textbackslash c\_sys\_month\_int}
\item \texttt{\textbackslash c\_sys\_year\_int}
\end{itemize}

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

\textbf{\TeXhacksnote{}}: 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.

\begin{itemize}
\item \texttt{\textbackslash c\_sys\_timestamp\_str}
\end{itemize}

The timestamp for the current job: the format is as described for \texttt{\file\_timestamp\_n}.
10.3 Engine

\sys_if_engine_luatex_p: * \sys_if_engine_pdfTeX_p: * \sys_if_engine_ptex_p: * \sys_if_engine_uptex_p: * \sys_if_engine_xetex_p: * \sys_if_engine_luatex: TF * \sys_if_engine_pdfTeX: TF * \sys_if_engine_ptex: TF * \sys_if_engine_uptex: TF * \sys_if_engine_uptex: TF * \sys_if_engine_xetex: TF *

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 \(\varepsilon\)-p\TeX{} and \(\varepsilon\)-up\TeX{} as expl3 requires the \(\varepsilon\)-\TeX{} extensions. Each conditional is true for \textit{exactly one} supported engine. In particular, \sys_if_engine_ptex_p: is true for \(\varepsilon\)-p\TeX{} but false for \(\varepsilon\)-up\TeX{}.

\c_sys_engine_str

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

\c_sys_engine_exec_str

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

\c_sys_engine_format_str

The name of the preloaded format for the current \TeX{} run given as a lower case string: one of \texttt{lualatex} (or \texttt{dvilualatex}), \texttt{pdflatex} (or \texttt{latex}), \texttt{platex}, \texttt{uplatex} or \texttt{xelatex} for \L\A\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}).

\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

\(\langle\text{major}\rangle.\langle\text{minor}\rangle.\langle\text{revision}\rangle\)

For Xe\TeX{}, the form is

\(\langle\text{major}\rangle.\langle\text{minor}\rangle\)

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 \(\varepsilon\)-\p\TeX{} version.

\(p(\langle\text{major}\rangle.\langle\text{minor}\rangle.\langle\text{revision}\rangle)\text{-}u(\langle\text{major}\rangle.\langle\text{minor}\rangle)-\langle\text{e}\text{p}\text{T}\text{E}\text{X}\rangle\)

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

\sys_timer: *

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

### 10.4 Output format

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

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

### 10.5 Platform

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

The current platform given as a lower case string: one of \texttt{unix}, \texttt{windows} or \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.
\sys_gset_rand_seed:n \sys_gset_rand_seed:n \{(int expr)\}

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

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

### 10.7 Access to the shell

\sys_get_shell:nnN \sys_get_shell:nnN \{(shell command)\} \{\langle setup\rangle\} \{tl var\}

\sys_get_shell:nnNTF \sys_get_shell:nnNTF \{(shell command)\} \{\langle setup\rangle\} \{tl var\} \{\langle true code\rangle\} \{\langle false code\rangle\}

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

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

\c_sys_shell_escape_int

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

1. Shell escape is disabled
2. Unrestricted shell escape is enabled
3. Restricted shell escape is enabled

\sys_if_shell_p: \sys_if_shell:TF \sys_if_shell:TF \{\langle true code\rangle\} \{\langle false code\rangle\}

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

\sys_if_shell_unrestricted_p: \sys_if_shell_unrestricted:TF \sys_if_shell_unrestricted:TF \{\langle true code\rangle\} \{\langle false code\rangle\}

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

\sys_shell_now:n \sys_shell_now:e
\sys_shell_shipout:n \sys_shell_shipout:e

Execute \tokens through shell escape immediately.

Sets the \tl var to the information returned by the \sys_get_query:NN, potentially supplying the \options and \spec to the query call. The valid \cmd names are at present

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

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

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

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

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

10.8 System queries

Some queries can be made about the file system, etc., without needing to use unrestricted shell escape. This is carried out using the script 13sys-query, which is documented separately. The wrappers here use this script, if available, to obtain system information that is not directly available within the \TeX run. Note that if restricted shell escape is disabled, no results can be obtained.
Works as described for \verb|\sys_split_query|:nnnN, but sets the \texttt{seq} to contain one entry for each line returned by \verb|l3sys-query|. This function should therefore be preferred where multi-line return is expected, e.g. for the \verb|ls| command.

## 10.9 Loading configuration data

\verb|\sys_load_backend|:n \verb|\sys_load_backend|:n \{\texttt{backend}\}

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

\verb|\sys_ensure_backend|: \verb|\sys_ensure_backend|:

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

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

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

\verb|\sys_load_debug|: \verb|\sys_load_debug|:

Load the additional configuration file for debugging support.

### 10.9.1 Final settings

\verb|\sys_finalise|: \verb|\sys_finalise|:

Finalises all system-dependent functionality: required before loading a backend.
Chapter 11

The \texttt{l3msg} module

Messages

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

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

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

11.1 Creating new messages

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

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

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

Some authors may find the need to include spaces as `-` characters tedious. This can be avoided by locally 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:nnnn \msg_new:nnee \msg_new:nnn \msg_new:nne

Updated: 2011-08-16

\msg_set:nnnn \msg_set:nnn

\msg_if_exist_p:nn \msg_if_exist:nnTF

Tests whether the \langle message\rangle for the \langle module\rangle is currently defined.

\msg_module_name:n \msg_module_type:n

Expands to the public name of the \langle module\rangle as defined by \g_msg_module_name_prop (or otherwise leaves the \langle module\rangle unchanged).

Expands to the description which applies to the \langle module\rangle, 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 \TeXeX three core messages, an empty entry is set here meaning that they are not described using the standard Package text.

11.2 Customizable information for message modules
11.3 Contextual information for messages

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

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

\msg_fatal_text:n \* \msg_fatal_text:n \{(module)\}
Produces the standard text

Fatal Package \(\langle\text{module}\rangle\) Error
This function can be redefined to alter the language in which the message is given, using \#1 as the name of the \(\langle\text{module}\rangle\) to be included. Any redefinition must produce output containing the \(\langle\text{module}\rangle\) name, and will affect all messages using the expl3 mechanism.

\msg_critical_text:n \* \msg_critical_text:n \{(module)\}
Produces the standard text

Critical Package \(\langle\text{module}\rangle\) Error
This function can be redefined to alter the language in which the message is given, using \#1 as the name of the \(\langle\text{module}\rangle\) to be included. Any redefinition must produce output containing the \(\langle\text{module}\rangle\) name, and will affect all messages using the expl3 mechanism.

\msg_error_text:n \* \msg_error_text:n \{(module)\}
Produces the standard text

Package \(\langle\text{module}\rangle\) Error
This function can be redefined to alter the language in which the message is given, using \#1 as the name of the \(\langle\text{module}\rangle\) to be included. Any redefinition must produce output containing the \(\langle\text{module}\rangle\) name, and will affect all messages using the expl3 mechanism.

\msg_warning_text:n \* \msg_warning_text:n \{(module)\}
Produces the standard text

Package \(\langle\text{module}\rangle\) Warning
This function can be redefined to alter the language in which the message is given, using \#1 as the name of the \(\langle\text{module}\rangle\) to be included. The \(\langle\text{type}\rangle\) of \(\langle\text{module}\rangle\) may be adjusted: Package is the standard outcome: see \msg_module_type:n. Any redefinition must produce output containing the \(\langle\text{module}\rangle\) name, and will affect all messages using the expl3 mechanism.
This function can be redefined to alter the language in which the message is given, using #1 as the name of the \module to be included. The \type of \module may be adjusted: Package is the standard outcome: see \msg_module_type:n. Any redefinition must produce output containing the \module name, and will affect all messages using the expl3 mechanism.

\msg_see_documentation_text:n * \msg_see_documentation_text:n \{\module\}

\msg_see_documentation_text:n \ Updated: 2018-09-30

Produces the standard text

See the \module documentation for further information.

This function can be redefined to alter the language in which the message is given, using #1 as the name of the \module to be included. The name of the \module is produced using \msg_module_name:n.

11.4 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the e-type variants should be used to expand material. Note that this expansion takes place with the standard definitions in effect, which means that shorthands such as \~ or \\ are not available; instead one should use \iow_char:N \~ and \iow_newline:, respectively. The following message classes exist:

- fatal, ending the \TeX run;
- critical, ending the file being input;
- error, interrupting the \TeX run without ending it;
- warning, written to terminal and log file, for important messages that may require corrections by the user;
- note (less common than info) for important information messages written to the terminal and log file;
- info for normal information messages written to the log file only;
- term and log for un-decorated messages written to the terminal and log file, or to the log file only;
- none for suppressed messages.
Issues \texttt{\module error \message}, passing \texttt{\arg one} to \texttt{\arg four} to the text-creating functions. After issuing a fatal error, \TeX run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).

\TeXhackers note: The \TeX \texttt{\endinput} primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

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

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

Issues `{module}` information `{message}`, passing `{arg one}` to `{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 `{module}`, omitting for instance the module name. It is added to the log file by `{module}` while `{module}` 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 \(\langle\text{module}\rangle\) \{\langle\text{message}\rangle\} \{\langle\text{arg one}\rangle\} \{\langle\text{arg two}\rangle\} \{\langle\text{arg three}\rangle\} \{\langle\text{arg four}\rangle\}

New: 2017-12-04

Issues \langle\text{module}\rangle\) information \langle\text{message}\rangle, passing \langle\text{arg one}\rangle to \langle\text{arg four}\rangle to the text-creating functions. The information text is shown on the terminal and the \TeX run is interrupted in a manner similar to \tl_show:n. This is used in conjunction with \msg_show_item:n and similar functions to print complex variable contents completely. If the formatted text does not contain \textgreater{} at the start of a line, an additional line \textgreater{}, 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 *

New: 2017-12-04

Used in the text of messages for \msg_show:nnnnnn to show or log a list of items or key-value pairs. The output of \msg_show_item:n produces a newline, the prefix \textgreater{}, two spaces, then the braced string representation of its argument. The two-argument versions separates the key and value using \textgreater{}=\textgreater{} 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:

\textgreater{}{a}
\textgreater{}{\{b\}}
\textgreater{}{\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_expansible_error:nnnnnn \{module\} \{message\} \{arg one\} \{arg two\} \{arg three\} \{arg four\}
\msg_expansible_error:nnffff
\msg_expansible_error:nnnn
\msg_expansible_error:nnff
\msg_expansible_error:nnf
\msg_expansible_error:nn

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

11.5 Redirecting messages

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

\msg_new:nnnn \{module\} \{my-message\} \{Some-text\} \{Some-more-text\}
\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\}
\msg_redirect_module:nnn \{module\} \{error\} \{warning\}
\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.

\texttt{msg\_redirect\_class:nn} {class one} {class two}

Changes the behaviour of messages of \texttt{class one} so that they are processed using the
code for those of \texttt{class two}. Each \texttt{class} can be one of fatal, critical, error,
warning, note, info, term, log, none.

\texttt{msg\_redirect\_module:nnn} {module} {class one} {class two}

Redirects message of \texttt{class one} for \texttt{module} to act as though they were from \texttt{class
two}. Messages of \texttt{class one} from sources other than \texttt{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 \texttt{module} could be turned off with:

\texttt{msg\_redirect\_module:nnn} { module } { warning } { none }

\texttt{msg\_redirect\_name:nnn} {module} {message} {class}

Redirects a specific \texttt{message} from a specific \texttt{module} to act as a member of \texttt{class}
of messages. No further redirection is performed. This function can be used to make a
selected message “silent” without changing global parameters:

\texttt{msg\_redirect\_name:nnn} { module } { annoying-message } { none }
Chapter 12

The l3file module

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix \file_..., while others are used to work with files on a line by line basis and have prefix \ior_... (reading) or \iow_... (writing).

It is important to remember that when reading external files TeX attempts to locate them using both the operating system path and entries in the TeX file database (most TeX systems use such a database). Thus the “current path” for TeX is somewhat broader than that for other programs.

For functions which expect a ⟨file name⟩ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in \l_char_active_seq) are not expanded, allowing the direct use of these in file names. Quote tokens (") are not permitted in file names as they are reserved for internal use by some TeX primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

12.1 Input–output stream management

As TeX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in l3TeX3. 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{\_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{\_term:...}.

\texttt{\ior_open:Nn} \{\texttt{stream}\} \{\texttt{file name}\}

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 an \texttt{\ior_close:N} instruction is given or the \TeX{} run ends. If the file is not found, an error is raised.

\texttt{\ior_open:NnTF} \{\texttt{stream}\} \{\texttt{true code}\} \{\texttt{false code}\}

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.

\texttt{\ior_open:(NV|cn|cV)}

Opens \texttt{file name} for writing using \texttt{stream} as the control sequence for file access. If the \texttt{stream} was already open it is closed before the new operation begins. The \texttt{stream} is available for access immediately and will remain allocated to \texttt{file name} until an \texttt{\iow_close:N} instruction is given or the \TeX{} run ends. Opening a file for writing clears any existing content in the file (i.e. writing is not additive).

\texttt{\ior_shell_open:Nn} \{\texttt{shell command}\}

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}.

\texttt{\ioc_shell_open:Nn} \{\texttt{shell command}\}

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 an \texttt{\iow_close:N} instruction is given or the \TeX{} run ends. If piped system calls are disabled an error is raised.

For details of handling of the \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.

\subsection{12.1.1 Reading from files}

Reading from files and reading from the terminal are separate processes in expl3. The functions \texttt{iorget:NN} and \texttt{iorstrget:NN}, and their branching equivalents, are designed to work with files.
\ior_get:NN \ior_get:NNTF

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input \langle stream \rangle and stores the result locally in the \langle token list \rangle variable. The material read from the \langle stream \rangle is tokenized by \TeX{} according to the category codes and \texttt{\endlinechar} in force when the function is used. Assuming normal settings, any lines which do not end in a comment character \texttt{\%} 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 \hspace{1pt} b \hspace{1pt} c}. Any blank line is converted to the token \texttt{\par}. Therefore, blank lines can be skipped by using a test such as
\begin{verbatim}
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
\end{verbatim}
...  
Also notice that if multiple lines are read to match braces then the resulting token list can contain \texttt{\par} tokens. In the non-branching version, where the \langle stream \rangle is not open the \langle tl var \rangle is set to \texttt{\q\_no\_value}.

\TeX{}hackers note: This protected macro is a wrapper around the \TeX{} primitive \texttt{\read}. Regardless of settings, \TeX{} replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code \texttt{\endlinechar}, omitted if \texttt{\endlinechar} is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

\ior_str_get:NN \ior_str_get:NNTF

Function that reads one line from the file input \langle stream \rangle and stores the result locally in the \langle token list \rangle variable. The material is read from the \langle stream \rangle as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the \langle token list variable \rangle being empty. Unlike \ior_get:NN, line ends do not receive any special treatment. Thus input
\begin{verbatim}
a b c
\end{verbatim}
results in a token list \texttt{a \hspace{1pt} b \hspace{1pt} c} with the letters \texttt{a}, \texttt{b}, and \texttt{c} having category code 12. In the non-branching version, where the\langle stream \rangle is not open the \langle tl var \rangle is set to \texttt{\q\_no\_value}.

\TeX{}hackers note: This protected macro is a wrapper around the \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 ⟨function⟩ or ⟨code⟩ discussed below remain in effect after the loop.

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

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

\ior_map_variable:NNn
\ior_map_variable:NNn \ior_get:NN
For each set of ⟨lines⟩ obtained by calling \ior_get:NN until reaching the end of the file, stores the ⟨lines⟩ in the ⟨tl var⟩ then applies the ⟨code⟩. The ⟨code⟩ will usually make use of the ⟨variable⟩, but this is not enforced. The assignments to the ⟨variable⟩ are local. Its value after the loop is the last set of ⟨lines⟩, or its original value if the ⟨stream⟩ is empty. \TeX{} ignores any trailing new-line marker from the file it reads. This function is typically faster than \ior_map_inline:Nn.

\ior_str_map_variable:NNn
\ior_str_map_variable:NNn \ior_get:NN
For each ⟨line⟩ in the ⟨stream⟩, stores the ⟨line⟩ in the ⟨variable⟩ then applies the ⟨code⟩. The material is read from the ⟨stream⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨code⟩ will usually make use of the ⟨variable⟩, but this is not enforced. The assignments to the ⟨variable⟩ are local. Its value after the loop is the last ⟨line⟩, or its original value if the ⟨stream⟩ is empty. Note that \TeX{} removes trailing space and tab characters (character codes 32 and 9) from every line upon input. \TeX{} also ignores any trailing new-line marker from the file it reads. This function is typically faster than \ior_str_map_inline:Nn.
\ior_map_break: 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_iør
{\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 ⟨code⟩

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

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

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

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

\ior_if_eof_p:N * \ior_if_eof_p:N ⟨stream⟩
\ior_if_eof:NTF * \ior_if_eof:NTF ⟨stream⟩ \{ ⟨true code⟩ \} \{ ⟨false code⟩ \}

Tests if the end of a file ⟨stream⟩ has been reached during a reading operation. The test also returns a true value if the ⟨stream⟩ is not open.
12.1.2 Reading from the terminal

\ior_get_term:nN \ior_get_term:nN \langle prompt \rangle \langle token list variable \rangle

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 \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_term:n \iow_term:e

\iow_log:n \langle\{tokens\}\rangle

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

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:e \iow_shipout:(Ne|cn|ce)

\iow_shipout:Nn \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 \ior_newline: or using the line-wrapping code \ior_wrap:nnnN are not recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additional unwanted line-breaks.

96
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).

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

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

\iow_now:N \g_my_iow { \iow_char:N \{ text \iow_char:N \} }

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

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

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

\iow_wrap:nnnN {\langle text \rangle} {\langle run-on text \rangle} {\langle set up \rangle} {\langle function \rangle}

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 \texttt{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 \texttt{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:

- \texttt{\\} or \texttt{\iownewline}: may be used to force a new line,
- \texttt{\_} may be used to represent a forced space (for example after a control sequence),
- \texttt{\#, \%, \{, \}, \~} may be used to represent the corresponding character,
- \texttt{\iow_wrap_allow_break}: may be used to allow a line-break without inserting a
  space,
- \texttt{\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
\texttt{\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, \texttt{\iow_wrap:nnnN} carries out an \texttt{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:

In the first argument of \texttt{\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 \{\langle text \rangle\}

In the first argument of \texttt{\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{\\} to force line breaks.

98
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 MiKi\TeX\ systems.

### 12.1.5 Constant input–output streams, and variables

\texttt{\textbackslash g\_tmpa\_ior} \hfill \texttt{\textbackslash g\_tmpb\_ior}\0

Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any \LaTeX\3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\texttt{\textbackslash c\_log\_iow} \hfill \texttt{\textbackslash c\_term\_iow}\0

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

\texttt{\textbackslash g\_tmpa\_iow} \hfill \texttt{\textbackslash g\_tmpb\_iow}\0

Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any \LaTeX\3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

### 12.1.6 Primitive conditionals

\texttt{\textbackslash if\_eof:w \{stream\}}

\texttt{\textbackslash if\_eof:w \{true code\}}

\texttt{\textbackslash else:}

\texttt{\textbackslash false code\}}

\texttt{\textbackslash fi:}

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

\TeX\hackers\ note: This is the \TeX\ primitive \texttt{\textbackslash if\_eof}.

### 12.2 File operations

#### 12.2.1 Basic file operations

\texttt{\textbackslash g\_file\_curr\_dir\_str} \hfill \texttt{\textbackslash g\_file\_curr\_name\_str} \hfill \texttt{\textbackslash g\_file\_curr\_ext\_str}\0

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{\textbackslash c\_sys\_jobname\_str}, which may be different from the real file name (if set using \texttt{\textbackslash --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 \LaTeX{} 2ε, expl3 will automatically append the current \texttt{\input@path} to the set of values from \texttt{\l_file_search_path_seq}.

\section*{12.2.2 Information about files and file contents}

Functions in this section return information about files as expl3 \texttt{str} data, except that the non-expandable functions set their return \texttt{token list} to \texttt{\q_no_value} if the file requested is not found. As such, comparison of file names, hashes, sizes, etc., should use \texttt{\str_{-if_eq:nnTF}} rather than \texttt{\tl_if_eq:nnTF} and so on.

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

Tests if \texttt{(file name)} is found in the path as detailed for \texttt{\file_if_exist:nTF}.

\begin{verbatim}
\file_hex_dump:n \file_hex_dump:V * \file_hex_dump:nnn * \file_hex_dump:Vnn *
\file_hex_dump:n \file_hex_dump:V *
\file_hex_dump:nnn \file_hex_dump:Vnn *
\end{verbatim}

Searches for \texttt{(file name)} 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{{\langle start index\rangle}} and \texttt{{\langle end index\rangle}} values work as described for \texttt{\str_range:nnn}.

\begin{verbatim}
\file_get_hex_dump:nN \file_get_hex_dump:Vn * \file_get_hex_dump:nnnN \file_get_hex_dump:VnnN *
\file_get_hex_dump:nN \file_get_hex_dump:VnN *
\file_get_hex_dump:nnnN \file_get_hex_dump:VnnN *
\end{verbatim}

Sets the \texttt{(tl var)} to the result of applying \texttt{\file_hex_dump:n/\file_hex_dump:nnn} to the \texttt{\langle file\rangle}. If the file is not found, the \texttt{(tl var)} will be set to \texttt{\q_no_value}.
\file_mdfive_hash:n \file_mdfive_hash:V \textit{New: 2019-09-03}
\file_get_mdfive_hash:nN \file_get_mdfive_hash:VN \file_get_mdfive_hash:nN \file_get_mdfive_hash:VN
\textit{New: 2017-07-11}
\textit{Updated: 2019-02-16}
\file_size:n \file_size:V \textit{New: 2019-09-03}
\file_get_size:nN \file_get_size:VN \file_get_size:nN \file_get_size:VN
\textit{New: 2017-07-09}
\textit{Updated: 2019-02-16}
\file_timestamp:n \file_timestamp:V \textit{New: 2019-09-03}
\file_get_timestamp:nN \file_get_timestamp:VN \file_get_timestamp:nN \file_get_timestamp:VN
\textit{New: 2017-07-09}
\textit{Updated: 2019-02-16}

\file_mdfive_hash:n \text{n \{file name\}}
\textit{Searches for \text{\langle} file name \rangle \text{using the current } \text{TeX search path and the additional paths}
\textit{controlled by \textbackslash l_file_search_path_seq}. It then expands to leave the MD5 sum generated
\textit{from the contents of the file in the input stream. The file is read as bytes, which}
\textit{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}
\textit{not found, the result of expansion is empty.}

\file_get_mdfive_hash:nN \text{n \{file name\} \{tl var\}}
\textit{Sets the \text{\langle} tl var \rangle \text{to the result of applying \textbackslash file_mdfive_hash:n to the \{file\}. If the}
\textit{file is not found, the \text{\langle} tl var \rangle \text{will be set to \textbackslash q_no_value.}}

\file_size:n \text{n \{file name\}}
\textit{Searches for \text{\langle} file name \rangle \text{using the current } \text{TeX search path and the additional paths}
\textit{controlled by \textbackslash l_file_search_path_seq}. It then expands to leave the size of the file in
\textit{bytes in the input stream. When the file is not found, the result of expansion is empty.}

\file_get_size:nN \text{n \{file name\} \{tl var\}}
\textit{Sets the \text{\langle} tl var \rangle \text{to the result of applying \textbackslash file_size:n to the \{file\}. If the file is}
\textit{not found, the \text{\langle} tl var \rangle \text{will be set to \textbackslash q_no_value.}}

\file_timestamp:n \text{n \{file name\}}
\textit{Searches for \text{\langle} file name \rangle \text{using the current } \text{TeX search path and the additional paths}
\textit{controlled by \textbackslash l_file_search_path_seq}. It then expands to leave the modification
\textit{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 \textit{plus-minus}\langle hours⟩'⟨minutes⟩'. When the file is not found, the result
\textit{of expansion is empty.}

\file_get_timestamp:nN \text{n \{file name\} \{tl var\}}
\textit{Sets the \text{\langle} tl var \rangle \text{to the result of applying \textbackslash file_timestamp:n to the \{file\}. If the file is}
\textit{not found, the \text{\langle} tl var \rangle \text{will be set to \textbackslash q_no_value.}}
Compares the file stamps on the two files as indicated by the comparator, and inserts either the true code or false code as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different.

```
\file_get_full_name:nN
\file_get_full_name:VN
\file_get_full_name:nNTF
\file_get_full_name:VN
```

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

```
\file_full_name:n
```

Searches for file name in the path as detailed for \file_if_exist:nTF, and if found leaves the fully-qualified name of the file, i.e. the path and file name, in the input stream. This includes an extension .tex when the given file name has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.

```
\file_parse_full_name:NNNN
\file_parse_full_name:VNNN
```

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

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

Before parsing, the (full name) is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (") are invalid in file names and are discarded from the input.
Parses the \textlangle full name \rangle as described for \textbackslash file_parse_full_name:nNNN, and leaves \langle dir \rangle, \langle name \rangle, and \langle ext \rangle in the input stream, each inside a pair of braces.

\textbackslash file_parse_full_name:V
\textbackslash file_parse_full_name_apply:nN \textbackslash file_parse_full_name_apply:VN

Parses the \textlangle full name \rangle as described for \textbackslash file_parse_full_name:nNNN, and passes \langle dir \rangle, \langle name \rangle, and \langle ext \rangle as arguments to \langle function \rangle, as an n-type argument each, in this order.

12.2.3 Accessing file contents

\textbackslash file_get:nnN \textbackslash file_get:VnN \textbackslash file_get:nnNTF \textbackslash file_get:VnNTF

Defines \langle tl \rangle to the contents of \langle file name \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{\textunderscore 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. The file content will be tokenized using the current category code régime.

\textbackslash file_input:n \textbackslash file_input:V

Searches for \langle file name \rangle in the path as detailed for \textbackslash file_if_exist:nTF, and if found reads in the file as additional LATeX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

\textbackslash file_input_raw:n \textbackslash file_input_raw:V

Searches for \langle file name \rangle in the path as detailed for \textbackslash file_if_exist:nTF, and if found reads in the file as additional LATeX source. No data concerning the file is tracked. If the file is not found, no action is taken.

\texttt{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 \textbackslash halign.}
\file_input_stop:  \file_input_stop:  

New: 2017-07-07

Ends the reading of a file started by \file_input:n or similar before the end of the file is reached. Where the file reading is being terminated due to an error, \msg_-critical:nn(nn) should be preferred.

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

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

\file_show_list:  \file_show_list:
\file_log_list:  \file_log_list:

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

The l3luatex module
LuaTEX-specific functions

The LuaTEX engine provides access to the Lua programming language, and with it access to the “internals” of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTeX, dviTeX, uTPeX or XeLaTeX these raise an error: use \sys_if_engine_luatex:T to avoid this. Details on using Lua with the LuaTeX engine are given in the LuaTeX manual.

13.1 Breaking out to Lua

\lua_now:n \{⟨token list⟩\}
\lua_now:e

The (token list) is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting ⟨Lua input⟩ is passed to the Lua interpreter for processing. Each \lua_now:n block is treated by Lua as a separate chunk. The Lua interpreter executes the ⟨Lua input⟩ immediately, and in an expandable manner.

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

\lua_shipout_e:n \lua_shipout:n \{⟨token list⟩\}

The ⟨token list⟩ is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting ⟨Lua input⟩ is passed to the Lua interpreter when the current page is finalised (i.e. at shipout). Each \lua_shipout:n block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨Lua input⟩ during the page-building routine: no TeX expansion of the ⟨Lua input⟩ will occur at this stage.

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

\textbf{TeXhackers note:} At a TeX level, the ⟨Lua input⟩ is stored as a “whatsit”. 
Converts the \textit{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 \textbackslash n and \textbackslash r, respectively.

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

\texttt{\lua_load_module:n} \texttt{\{Lua module name\}}

Loads a Lua module into the Lua interpreter.

\texttt{\lua_now:n} passes its \texttt{\{token list\}} argument to the Lua interpreter as a single line, with characters interpreted under the current catcode regime. These two facts mean that \texttt{\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 \texttt{\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 \texttt{\lua_load_module:n}.

\textbf{\textTeX{hackers note}}: This is a wrapper around the Lua call \texttt{require '\{module\}'}. 

\section{13.2 Lua interfaces}

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

\texttt{\ltx{utils}}

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

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

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

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

Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal \textTeX{} behaviour. If the \texttt{(file)} is not found, nothing is returned with \textit{no error raised}.

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

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

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

where the latter may be \texttt{Z} (UTC) or \texttt{\{plus-minus\}\{hours\}'\{minutes\}'}. If the \texttt{(file)} is not found, nothing is returned with \textit{no error raised}.

106
\texttt{ltx.utils.filesize} \texttt{size} = \texttt{ltx.utils.filesize(\langle file\rangle)}

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

The \texttt{l3}legacy module

Interfaces to legacy concepts

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

\begin{verbatim}
\legacy_if_p:n \legacy_if:nTF \\
\legacy_if_set_true:n \legacy_if_set_false:n \legacy_if_gset_true:n \legacy_if_gset_false:n
\end{verbatim}

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

\begin{verbatim}
\legacy_if_set:nn \legacy_if_gset:nn
\end{verbatim}

New: 2021-05-10

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

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

New: 2021-05-10
Part IV
Data types
Chapter 15

The \texttt{t3tl} module

Token lists

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

\begin{verbatim}
\texttt{\textbackslash foo:n \{ a collection of \texttt{\textbackslash 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{t3tl}. 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{\textbackslash 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, \}, \, 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}

\texttt{\textbackslash tl\_new:C}

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.
\texttt{\tlconst:Nn} \quad \texttt{\tlconst:(Ne|cn|ce)} \quad \texttt{\tlconst:Nn \{tl var\} \{tokens\}}

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 tokens \rangle.

\texttt{\tlclear:N} \quad \texttt{\tlclear:c} \quad \texttt{\tlgclear:N} \quad \texttt{\tlgclear:c}

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

\texttt{\tlclear_new:N} \quad \texttt{\tlclear_new:c} \quad \texttt{\tlgclear_new:N} \quad \texttt{\tlgclear_new:c}

Ensures that the \langle tl var \rangle exists globally by applying \texttt{\tlnew:N} if necessary, then applies \texttt{\tl_(g)clear:N} to leave the \langle tl var \rangle empty.

\texttt{\tlset_eq:NN} \quad \texttt{\tlset_eq:(cN|Nc|cc)} \quad \texttt{\tlgset_eq:NN} \quad \texttt{\tlgset_eq:(cN|Nc|cc)}

Sets the content of \langle tl var \rangle equal to that of \langle tl var2 \rangle.

\texttt{\tlconcat:NNN} \quad \texttt{\tlconcat:ccc} \quad \texttt{\tlgconcat:NNN} \quad \texttt{\tlgconcat:ccc}

Concatenates the content of \langle tl var2 \rangle and \langle tl var3 \rangle together and saves the result in \langle tl var1 \rangle. The \langle tl var2 \rangle is placed at the left side of the new token list.

\texttt{\tlif_exist_p:N} \quad \texttt{\tlif_exist_p:c} \quad \texttt{\tlif_exist:NTF} \quad \texttt{\tlif_exist:cTF}

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.

\begin{itemize}
  \item \texttt{\tlif_exist_p:N} \star \texttt{\tlif_exist_p:c} \star \texttt{\tlif_exist:NTF} \star \texttt{\tlif_exist:cTF}
\end{itemize}

\textbf{New:} 2012-03-03

\section*{15.2 Adding data to token list variables}

\texttt{\tlset:Nn} \quad \texttt{\tlset:(NV|Nv|Ne|Nf|cn|cV|cv|co|ce|cf)} \quad \texttt{\tlgset:Nn} \quad \texttt{\tlgset:(NV|Nv|Ne|Nf|cn|cV|cv|co|ce|cf)}

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

\texttt{\tlput_left:Nn} \quad \texttt{\tlput_left:(NV|Nv|Ne|Nf|cn|cV|cv|ce|co)} \quad \texttt{\tlgput_left:Nn} \quad \texttt{\tlgput_left:(NV|Nv|Ne|Nf|cn|cV|cv|ce|co)}

Appends \langle tokens \rangle to the left side of the current content of \langle tl var \rangle.
15.3 Token list conditionals

\[\begin{align*}
\texttt{\textbackslash tl\_put\textunderscore right:Nn} & \Rightarrow \langle \texttt{tl\ var} \rangle \{\langle \texttt{tokens} \rangle\} \\
\texttt{\textbackslash tl\_put\textunderscore right:(NV\mid Nv\mid No\mid cn\mid cV\mid ce\mid co)} & \Rightarrow \langle \texttt{tokens} \rangle \\
\texttt{\textbackslash tl\_gput\textunderscore right:(NV\mid Nv\mid No\mid cn\mid cV\mid ce\mid co)} & \Rightarrow \langle \texttt{tokens} \rangle
\end{align*}\]

Appends \(\langle \texttt{tokens} \rangle\) to the right side of the current content of \(\langle \texttt{tl\ var} \rangle\).

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

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

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

\[
\begin{align*}
\texttt{\textbackslash tl\_set:Nn} & \ \texttt{\_tmpa\_tl} \ \{ \ abc \} \\
\texttt{\textbackslash tl\_set:Ne} & \ \texttt{\_tmpb\_tl} \ \{ \ \texttt{\_to\_str:n} \ \{ \ abc \} \} \\
\texttt{\textbackslash tl\_if\_eq:NN} & \ \texttt{\_tmpa\_tl} \ \texttt{\_tmpb\_tl} \ \{ \ \texttt{true} \} \ \{ \ \texttt{false} \}
\end{align*}\]

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

Tests if the \(\langle \texttt{tl\ var} \rangle\) and the \(\langle \texttt{token\ list} \rangle\) 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 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 \( \texttt{\textbackslash t l \_i f \_e q: n T F} \) for an expandable version when token lists are stored in variables, or \( \texttt{\textbackslash s t r \_i f \_e q: N N T F} \) if category codes are not important.

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

Tests if the \( \{ \text{token list} \} \) and the special \( \texttt{\textbackslash c \_n o v a l u e \_t l} \) marker contain the same list of tokens, both in respect of character codes and category codes. This means that \( \texttt{\textbackslash e x p \_a r g s: N o} \ \texttt{\textbackslash t l \_i f \_n o v a l u e: N T F} \ \{ \texttt{\textbackslash c \_n o v a l u e \_t l} \} \) is logically \texttt{true} but \( \texttt{\textbackslash t l \_i f \_n o v a l u e: N T F} \ \{ \texttt{\textbackslash c \_n o v a l u e \_t l} \} \) is logically \texttt{false}. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

Tests if the content of the \( \langle \text{item} \rangle \) is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to \( \texttt{\textbackslash t l \_c o u n t: N} \).

Tests if the \( \{ \text{token list} \} \) has exactly one \( \langle \text{item} \rangle \), \textit{i.e.} a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to \( \texttt{\textbackslash t l \_c o u n t: N} \).

Tests if the token list consists of exactly one token, \textit{i.e.} is either a single space character or a single normal token. Token groups \{...\} are not single tokens.
15.3.1 Testing the first token

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

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

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

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

Tests if the first ⟨token⟩ in the ⟨token list⟩ is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields false, as it does not have a normal first token. This function is useful to implement actions on token lists on a token by token basis.
Tests if the first \textit{token} in the \textit{token list} is an explicit space character (explicit token with character code 32 and category code 10). In particular, the test is \textit{false} if the \textit{token list} starts with an implicit token such as \texttt{\textbackslash 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{\textbackslash tl\_to\_str:n \{\textit{token list}\}}

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

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

- an escape character, whose character code is given by the internal parameter \texttt{\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{\textbackslash 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{\textbackslash 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{\textbackslash tl\_use:N \{\textit{tl\_var}\}}

Recover the content of a \texttt{\textbackslash tl\_var} and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a \texttt{\textbackslash tl\_var} directly without an accessor function.
15.4.2 Counting and reversing token lists

\tl_count:n \* \tl_count:n \{(token list)\}

Counts the number of \item{s} in the \tokenlist 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 \tokenlist. See also \tl_count:N. This function requires three expansions, giving an \integer denotation.

\tl_count:N \* \tl_count:N \tl_var
\tl_count:c \* \tl_count:c \tl_var

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

\tl_count_tokens:n \* \tl_count_tokens:n \{(token list)\}

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

\tl_reverse:n \* \tl_reverse:n \{(token list)\}
\tl_reverse:(V|v|e|o) \* \tl_reverse:(V|v|e|o)

Reverses the order of the \item{s} in the \tokenlist, so that \item{1}\item{2}\item{3} \ldots \item{n} becomes \item{n}\item{n-1}\item{n-2} \ldots \item{1}. This process preserves unprotected space within the \tokenlist. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider \tl_reverse_items:n. See also \tl_reverse:N.

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

\tl_reverse:N \* \tl_reverse:N \tl_var
\tl_reverse:c \* \tl_reverse:c
\tl_greverse:N \* \tl_greverse:c

Sets the \tl_var to contain the result of reversing the order of its \item{s}, so that \item{1}\item{2}\item{3} \ldots \item{n} becomes \item{n}\item{n-1}\item{n-2} \ldots \item{1}. This process preserves unprotected spaces within the \tl_var. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. This is equivalent to a combination of an assignment and \tl_reverse:N. See also \tl_reverse:V. For improved performance.

\tl_reverse_items:n \* \tl_reverse_items:n \{(token list)\}

Reverses the order of the \item{s} in the \tokenlist, so that \item{1}\item{2}\item{3} \ldots \item{n} becomes \item{n}\item{n-1}\item{n-2} \ldots \item{1}. This process removes any unprotected space within the \tokenlist. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function \tl_reverse:n.

\TeXhackers note: The result is returned within \unexpanded, which means that the token list does not expand further when appearing in an \e or \x-type argument expansion.
\tl_trim_spaces:n \star \tl_trim_spaces:(v|e|o) \star

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

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

\tl_trim_spaces_apply:nN \star \tl_trim_spaces_apply:oN \star

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

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

Sets the \langle tl var \rangle 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

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

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

\tl_show:n \tl_show:e

Displays the \langle token list \rangle on the terminal.

\TeXhackers 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

Writes the content of the \langle tl var \rangle in the log file. See also \tl_show:N which displays the result in the terminal.

\tl_log:n \tl_log:e

Writes the \langle token list \rangle 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, \textit{i.e.} any local assignments made by the \langle function \rangle or \langle code \rangle discussed below remain in effect after the loop.

\verbatim
\l_map_function:NN \∗
\l_map_function:CN \∗

\begin{verbatim}
\tl_map_function:NN \langle tl var \rangle \langle function \rangle
\end{verbatim}

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

\verbatim
\l_map_function:nN \∗
\l_map_function:cN \∗

\begin{verbatim}
\tl_map_function:nN \{ \langle token list \rangle \} \langle function \rangle
\end{verbatim}

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

\verbatim
\l_map_inline:Nn \tl_map_inline:cn

\begin{verbatim}
\tl_map_inline:Nn \langle tl var \rangle \{ \langle inline function \rangle \}
\end{verbatim}

Applies the \langle inline function \rangle to every \langle item \rangle stored within the \langle tl var \rangle. The \langle inline function \rangle should consist of code which receives the \langle item \rangle as \#1. See also \tl_map_function:nN.

\verbatim
\l_map_inline:nn \tl_map_inline:cn

\begin{verbatim}
\tl_map_inline:nn \{ \langle token list \rangle \} \{ \langle inline function \rangle \}
\end{verbatim}

Applies the \langle inline function \rangle to every \langle item \rangle stored within the \langle token list \rangle. The \langle inline function \rangle should consist of code which receives the \langle item \rangle as \#1. See also \tl_map_function:nN.

\verbatim
\l_map_tokens:Nn \tl_map_tokens:cn \tl_map_tokens:nn

\begin{verbatim}
\tl_map_tokens:Nn \langle tl var \rangle \{ \langle code \rangle \}
\tl_map_tokens:nn \{ \langle token list \rangle \} \{ \langle code \rangle \}
\end{verbatim}

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

\verbatim
\tl_map_tokens:Nn \l_my_tl \{ \prg_replicate:nn \{ 2 \} \}
\end{verbatim}

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

\verbatim
\l_map_variable:NNn \tl_map_variable:cNn

\begin{verbatim}
\tl_map_variable:NNn \langle tl var \rangle \langle variable \rangle \{ \langle code \rangle \}
\tl_map_variable:CNn
\end{verbatim}

Stores each \langle item \rangle of the \langle tl var \rangle in turn in the \langle token list \rangle \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 tl var \rangle, or its original value if the \langle tl var \rangle is blank. See also \tl_map_inline:Nn.
\tl_map_variable:nNn \tl_map_variable:nNn \{\langle\text{token list}\rangle\} \langle\text{variable}\rangle \{\langle\text{code}\rangle\}

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

\tl_map_break: \tl_map_break:

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

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

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

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

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

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

\tl_map_inline:Nn \l_my_tl
{\str_if_eq:nNT { #1 } { bingo } \{ \tl_map_break:n \{ \langle\text{code}\rangle \} \}
% Do something useful
}

Use outside of a \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 \langle\text{code}\rangle is inserted into the input stream. This depends on the design of the mapping function.

### 15.5.2 Head and tail of token lists

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.
Leaves in the input stream the first \langle item \rangle in the \langle token list \rangle, discarding the rest of the \langle token list \rangle. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example
\tl_head:n \{ abc \}

and
\tl_head:n \{ - abc \}

both leave 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 token list \rangle (see \tl_if_blank:nTF) results in \tl_head:n leaving nothing in the input stream.

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

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first \langle item \rangle in the \langle token list \rangle, and leaves the remaining tokens in the input stream. Thus for example
\tl_tail:n \{ a \{ bc \} d \}

and
\tl_tail:n \{ - a \{ bc \} d \}

both leave \(bc\)d in the input stream. A blank \langle token list \rangle (see \tl_if_blank:nTF) results in \tl_tail:n leaving nothing in the input stream.

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

If you wish to handle token lists where the first token may be a space, and this
needs to be treated as the head/tail, this can be accomplished using $\texttt{\tl_if_head_is_space:NTF}$, for example

\begin{verbatim}
\exp_last_unbraced:NNo
  \cs_new:Npn \_\_\_mypkg_gobble_space:w \c_space_tl { }
  \cs_new:Npn \_\_\_mypkg_gobble_space:n #1
  { \\texttt{\tl_if_head_is_space:NTF} \{#1\} }
  \cs_new:Npn \_\_\_mypkg_gobble_space:n #1
  { \\texttt{\exp_not:o { \_\_\_mypkg_gobble_space:w #1 } } }
\end{verbatim}

15.5.3 Items and ranges in token lists

Indexing items in the \texttt{⟨token list⟩} from 1 on the left, this function evaluates the \texttt{⟨integer expression⟩} and leaves the appropriate item from the \texttt{⟨token list⟩} in the input stream. If the \texttt{⟨integer expression⟩} is negative, indexing occurs from the right of the token list, starting at $-1$ for the right-most item. If the index is out of bounds, then the function expands to nothing.

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

\begin{verbatim}
\tl_rand_item:N \tl_rand_item:c \tl_rand_item:n
\end{verbatim}

Selects a pseudo-random item of the \texttt{⟨token list⟩}. If the \texttt{⟨token list⟩} is blank, the result is empty.

\textbf{TeXhackers note:} The result is returned within the $\texttt{\unexpanded}$ primitive, which means that the \texttt{⟨item⟩} does not expand further when appearing in an \texttt{e} or \texttt{x} argument expansion.
Leaves in the input stream the items from the \textit{(start index)} to the \textit{(end index)} inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here \textit{(start index)} and \textit{(end index)} should be \textit{(integer expressions)}. For describing in detail the functions' behavior, let \textit{m} and \textit{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 \textit{l} be the count of the token list.

The \textit{actual start point} is determined as \(M = m\) if \(m > 0\) and as \(M = l + m + 1\) if \(m < 0\). Similarly the \textit{actual end point} is \(N = n\) if \(n > 0\) and \(N = l + n + 1\) if \(n < 0\). If \(M > N\), the result is empty. Otherwise it consists of all items from position \(M\) to position \(N\) inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions \(s\) for \(s \leq 0\) or \(s > l\).

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to \texttt{\tl_trim_spaces:n} to begin with. Thus, with \(l = 7\) as in the examples below, all of the following are equivalent and result in the whole token list

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

are all equivalent and will print \texttt{bcd }\texttt{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 \textit{(item)} does not expand further when appearing in an \texttt{e}-type or \texttt{x}-type argument expansion.

\texttt{\tl_range:nnn} ⋆

\texttt{\tl_range:nnn} ⋆

New: 2017-02-17

Updated: 2017-07-15
15.5.4 Sorting token lists

\tl_sort:Nn \tl_sort:Nn \langle \text{tl var} \rangle \{ \langle \text{comparison code} \rangle \}
\tl_sort:cn \tl_gsort:Nn \tl_gsort:cn
\textit{Rev}: 2017-02-06

\tl_sort:nN * \tl_sort:nN \{ \langle \text{token list} \rangle \} \{ \langle \text{conditional} \rangle \}
\textit{Rev}: 2017-02-06

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

\TeX hackers note: The result is returned within \texttt{\exp_not:n}, which means that the token list does not expand further when appearing in an \texttt{e}-type or \texttt{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:Nnn \langle \text{tl var} \rangle \{ \langle \text{old tokens} \rangle \} \{ \langle \text{new tokens} \rangle \}
\tl_greplace_once:Nnn \tl_greplace_once:Nnn
\textit{Updated}: 2011-08-11

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).
\tl_replace_all:Nnn
\tl_replace_all:(NV|Nn|Nen|Nee|cnV|cnV|cen|cne|cee)
\tl_greplace_all:Nnn
\tl_greplace_all:(NV|Nn|Nen|Nee|cnV|cnV|cen|cne|cee)

Replaces all occurrences of \textit{old tokens} in the \textit{tl var} with \textit{new tokens}. \textit{Old tokens} cannot contain \{, \} or \# (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern \textit{old tokens} may remain after the replacement (see \tl_remove_all:Nn for an example).

\tl_remove_once:Nn
\tl_remove_once:(NV|Ne|cn|cV|ce)
\tl_gremove_once:Nn
\tl_gremove_once:(NV|Ne|cn|cV|ce)

Removes the first (leftmost) occurrence of \textit{tokens} from the \textit{tl var}. The \textit{tokens} cannot contain \{, \} or \# (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tl_remove_all:Nn
\tl_remove_all:(NV|Ne|cn|cV|ce)
\tl_gremove_all:Nn
\tl_gremove_all:(NV|Ne|cn|cV|ce)

Removes all occurrences of \textit{tokens} from the \textit{tl var}. The \textit{tokens} cannot contain \{, \} or \# (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern \textit{tokens} may remain after the removal, for instance,

\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl \tl_remove_all:Nn \l_tmpa_tl \tl_set:Nn \l_tmpa_tl {bc}

results in \l_tmpa_tl containing abcd.

15.6.2 Reassigning category codes

These functions allow the rescanning of tokens: re-apply \TeX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using \char_generate:nn).
Sets \( \langle \text{tl var} \rangle \) to contain \( \langle \text{tokens} \rangle \), applying the category code régime specified in the \( \langle \text{setup} \rangle \) before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the \( \langle \text{setup} \rangle \) are those in force at the point of use of \( \text{\tl_set_rescan:}\text{Nnn} \).) This allows the \( \langle \text{tl var} \rangle \) to contain material with category codes other than those that apply when \( \langle \text{tokens} \rangle \) are absorbed. The \( \langle \text{setup} \rangle \) is run within a group and may contain any valid input, although only changes in category codes, such as uses of \text{\vctab_select:N}, are relevant. See also \( \text{\tl_rescan:nn} \).

\textbf{T\kern-0.16em X\kern-0.16em hack\kern-0.16em er\kern-0.16em s note:} The \( \langle \text{tokens} \rangle \) are first turned into a string (using \text{\tl_to_str:n}). If the string contains one or more characters with character code \text{\newlinechar} (set equal to \text{\endlinechar} unless that is equal to 32, before the user \( \langle \text{setup} \rangle \)), 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 \text{\scantokens \epsilon-\TeX} primitive, \text{\tl_rescan:nn} tokenizes the whole string in the same category code régime rather than one token at a time, so that directives such as \text{\verb} that rely on changing category codes will not function properly.

### 15.7 Constant token lists

\texttt{\textbackslash \c_empty_tl} Constant that is always empty.
A marker for the absence of an argument. This constant \texttt{tl} can safely be typeset (\textit{cf.} \texttt{q-nil}), with the result being \texttt{-NoValue-}. It is important to note that \texttt{c_novalue_tl} is constructed such that it will \textit{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.

\subsection{Scratch 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 \LaTeX-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 \LaTeX-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 module
Piecewise \texttt{tl} constructions

16.1 Constructing \texttt{⟨tl var⟩} by accumulation

When creating a \texttt{⟨tl var⟩} by accumulation of many tokens, the performance available using a combination of \texttt{\tl_set:Nn} and \texttt{\tl_put_right:Nn} or similar begins to become an issue. To address this, a set of functions are available to “build” a \texttt{⟨tl var⟩}. The performance of this approach is much more efficient than the standard \texttt{\tl_put_right:Nn}, but the constructed token list cannot be accessed during construction other than by methods provided in this section.

Whilst the exact performance difference is dependent on the size of each added block of tokens and the total number of blocks, in general, the \texttt{\tl_build\_\textit{(g)put...}} functions will out-perform the basic \texttt{\textit{(g)put...}} equivalent if more than 100 non-empty addition operations occur. See https://github.com/latex3/latex3/issues/1393#issuecomment-1880164756 for a more detailed analysis.

\begin{verbatim}
\tl_build_begin:N \tl_build_gbegin:N
\tl_build_put_left:Nn \tl_build_put_left:Ne \tl_build_gput_left:Nn \tl_build_gput_left:Ne
\tl_build_put_right:Nn \tl_build_put_right:Ne \tl_build_gput_right:Nn \tl_build_gput_right:Ne
\end{verbatim}

\begin{verbatim}
\tl_build_begin:N \tl_build_gbegin:N \tl_build_put_left:N \tl_build_gput_left:N \tl_build_put_right:N \tl_build_gput_right:N
\end{verbatim}

127
\texttt{\tl_build_end:N} \texttt{\tl_build_end:N \{tl var\}}

Gets the contents of \texttt{tl var} and stores that into the \texttt{tl var} using \texttt{\tl_set:Nn} or \texttt{\tl_gset:Nn}. The \texttt{tl var} must have been set up with \texttt{\tl_build_begin:N} or \texttt{\tl_build_gbegin:N}. The \texttt{end} and \texttt{gend} functions must be used for local and global \texttt{tl var} respectively. These functions completely remove the setup code that enabled \texttt{tl var} to be used for other \texttt{\tl_build_...} functions. After the action of \texttt{end/gend}, the \texttt{tl var} may be manipulated using standard \texttt{tl} functions.

\texttt{\tl_build_get_intermediate:NN \tl_build_get_intermediate:NN \{tl var\}_1 \{tl var\}_2}

Stores the contents of the \texttt{tl var}_1 in the \texttt{tl var}_2. The \texttt{tl var}_1 must have been set up with \texttt{\tl_build_begin:N} or \texttt{\tl_build_gbegin:N}. The \texttt{tl var}_2 is a “normal” token list variable, assigned locally using \texttt{\tl_set:Nn}. 
Chapter 17

The \texttt{l3str} module

Strings

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

A \LaTeX{} string (and thus an \texttt{expl3} string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 ("space"). Thus at a technical level, a \LaTeX{} string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix \texttt{...str}. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using \texttt{\tl_to_str:n} for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions \texttt{\cs_to_str:N, \tl_to_str:n, \tl_to_str:N} and \texttt{\token_to_str:N} (and variants) generate strings from the appropriate input: these are documented in \texttt{l3basics}, \texttt{l3tl} and \texttt{l3token}, respectively.

Most expandable functions in this module come in three flavours:

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

\str_new:N \str_new:c
\str_new:NN \str_new:NNN

Creates a new \textit{str var} or raises an error if the name is already taken. The declaration is global. The \textit{str var} is initially empty.

\str_const:Nn \str_const:N \str_const:cc
\str_const:(NV|Nc|cV|ce)

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

\str_clear:N \str_clear:c \str_gclear:N \str_gclear:c
\str_clear_new:N \str_clear_new:c \str_gclear_new:N \str_gclear_new:c

Clears the content of the \textit{str var}.

\str_set_eq:NN \str_set_eq:NN \str_set_eq:cc
\str_gset_eq:NN \str_gset_eq:cc

Sets the content of \textit{str var} equal to that of \textit{str var}.

\str_concat:NNN \str_concat:ccc \str_gconcat:NNN \str_gconcat:ccc

Concatenates the content of \textit{str var} and \textit{str var} together and saves the result in \textit{str var}. The \textit{str var} is placed at the left side of the new string variable. The \textit{str var} and \textit{str var} must indeed be strings, as this function does not convert their contents to a string.

\str_if_exist_p:N \str_if_exist_p:cc \str_if_exist:NTF \str_if_exist:cTF

Tests whether the \textit{str var} is currently defined. This does not check that the \textit{str var} really is a string.
17.2 Adding data to string variables

\texttt{\textbackslash str\_set:Nn}  \\
\texttt{\textbackslash str\_set:\langle\text{NV}\mid\text{Ne}\mid\text{cn}\mid\text{cV}\mid\text{ce}\rangle}  \\
\texttt{\textbackslash str\_gset:Nn}  \\
\texttt{\textbackslash str\_gset:\langle\text{NV}\mid\text{Ne}\mid\text{cn}\mid\text{cV}\mid\text{ce}\rangle}

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

\texttt{\textbackslash str\_put\_left:Nn}  \\
\texttt{\textbackslash str\_put\_left:\langle\text{NV}\mid\text{Ne}\mid\text{cn}\mid\text{cV}\mid\text{ce}\rangle}  \\
\texttt{\textbackslash str\_gput\_left:Nn}  \\
\texttt{\textbackslash str\_gput\_left:\langle\text{NV}\mid\text{Ne}\mid\text{cn}\mid\text{cV}\mid\text{ce}\rangle}

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

\texttt{\textbackslash str\_put\_right:Nn}  \\
\texttt{\textbackslash str\_put\_right:\langle\text{NV}\mid\text{Ne}\mid\text{cn}\mid\text{cV}\mid\text{ce}\rangle}  \\
\texttt{\textbackslash str\_gput\_right:Nn}  \\
\texttt{\textbackslash str\_gput\_right:\langle\text{NV}\mid\text{Ne}\mid\text{cn}\mid\text{cV}\mid\text{ce}\rangle}

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

17.3 String conditionals

\texttt{\textbackslash str\_if\_empty\_p:NN}  \\
\texttt{\textbackslash str\_if\_empty\_p:n}  \\
\texttt{\textbackslash str\_if\_empty\_p:NTF}  \\
\texttt{\textbackslash str\_if\_empty\_p:NN}  \\
\texttt{\textbackslash str\_if\_empty\_p:n}  \\
\texttt{\textbackslash str\_if\_empty\_p:NTF}

Tests if the \langle\textit{string variable}\rangle is entirely empty \textit{(i.e.} contains no characters at all).
\str_if_eq_p:nn * \str_if_eq_p:nn \{\tl_1\} \{\tl_2\}
\str_if_eq_p:nn\{\{Vn\}|on|no|\{nV\}|\{Vn\}|ee\} * \str_if_eq:nnTF \{\tl_1\} \{\tl_2\} \{\{true code\}\} \{\{false code\}\}
\str_if_eq:nnTF *
\str_if_eq:nn\{\{Vn\}|on|no|\{nV\}|\{Vn\}|ee\}TF *

Updated: 2018-06-18

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

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

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

\str_if_in:NnTF \str_if_in:nnTF \{\tl_1\} \{\tl_2\} \{\{true code\}\} \{\{false code\}\}

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

\str_case:nn * \str_case:nnTF \{\test string\}
\str_case:nnTF * \{\{string case_1\}\} \{\{code case_1\}\}
\str_case:nnTF * \{\{string case_2\}\} \{\{code case_2\}\}
\str_case:nn * \{\{\ldots\}\}
\str_case:nnTF * \{\{string case_{n}\}\} \{\{code case_{n}\}\}

New: 2013-07-24
Updated: 2022-03-21

Compares the \{test string\} in turn with each of the \{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 \{code\} is left in the input stream and other cases are discarded. If any of the cases are matched, the \{true code\} is also inserted into the input stream (after the code for the appropriate case), while if none match then the \{false code\} is inserted. The function \str_case:nn, which does nothing if there is no match, is also available.

This set of functions performs no expansion on each \{string case\} argument, so any variable in there will be compared as a string. If expansion is needed in the \{string case\}_s, then \str_case_e:nn(TF) should be used instead.
\str_case_e:nn  \str_case_e:nnTF \{\langle test string\rangle\}  
\str_case_e:en  \{  
  \{\langle string case_1\rangle\} \{\langle code case_1\rangle\}  
  \{\langle string case_2\rangle\} \{\langle code case_2\rangle\}  
  \ldots  
  \{\langle string case_n\rangle\} \{\langle code case_n\rangle\}  
\} \{\langle true code\rangle\} \{\langle false code\rangle\}

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.

\str_compare_p:nNn  \str_compare_p:eNe  \str_compare:nNnTF  \str_compare:eNe  
\str_compare_p:nNn \{\langle tl_1\rangle\} \{\langle relation\rangle\} \{\langle tl_2\rangle\}  
\str_compare:nNnTF \{\langle tl_1\rangle\} \{\langle relation\rangle\} \{\langle tl_2\rangle\}  \{\langle true code\rangle\} \{\langle false code\rangle\}

Compares the two \textit{(token lists)} on a character by character basis (namely after converting them to strings) in a lexicographic order according to the character codes of the characters. The \textit{(relation)} can be \textless, \texttt{=} or \textgreater and the test is \texttt{true} under the following conditions:

- for \textless, if the first string is earlier than the second in lexicographic order;
- for \texttt{=}, if the two strings have exactly the same characters;
- for \textgreater, if the first string is later than the second in lexicographic order.

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

\texttt{\str_compare_p:nNn \{ ab \} < \{ abc \}}

\TeXhackers note: 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.

### 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.

\str_map_function:nN  \str_map_function:NN  \str_map_function:cN  
\str_map_function:nN \{\langle token list\rangle\} \{\langle function\rangle\}  
\str_map_function:NN \{\langle str var\rangle\} \{\langle function\rangle\}  
\str_map_function:cN \{\langle character\rangle\}

Converts the \textit{(token list)} to a \textit{(string)} then applies \textit{(function)} to every \textit{(character)} in the \textit{(string)} including spaces.

New: 2018-06-19

New: 2021-05-17

New: 2017-11-14

133
\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}\}
\str_map_inline:cn
Converts the \{\textit{token list}\} to a \{\textit{string}\} then applies the \{\textit{inline function}\} to every \textit{character} in the \{\textit{str var}\} including spaces. The \{\textit{inline function}\} should consist of code which receives the \textit{character} 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}\}
\str_map_tokens:cn ☆
Converts the \{\textit{token list}\} to a \{\textit{string}\} then applies \{\textit{code}\} to every \textit{character} in the \{\textit{string}\} including spaces. The \{\textit{code}\} receives each character as a trailing brace group. This is equivalent to \str_map_function:nN if the \{\textit{code}\} 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
Converts the \{\textit{token list}\} to a \{\textit{string}\} then stores each \textit{character} in the \{\textit{string}\} (including spaces) in turn in the \{\textit{string} or 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{character} in the \{\textit{string}\}, or its original value if the \{\textit{string}\} is empty. See also \str_map_inline:nn.

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

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

See also \str_map_break:n. Use outside of a \str_map... scenario leads to low level \TeX errors.

\textbf{\LaTeX hackers note}: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.
\str_map_break:n \str_map_break:n \{\langle code\rangle\}

Used to terminate a \str_map_inline:Nn \l_my_str
\{\langle code\rangle\}
\{ \str_map_break:n \{ \langle code\rangle \} \}
\%
\Do someth\ing useful  

Use outside of a \str_map_inline:Nn 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.

\section{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_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:cnn *
\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,

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

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

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

both yield empty strings.

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

\begin{verbatim}
\iow_term:e \{ \str_range:nnn { abcdefg } { 2 } { 5 } \}
\iow_term:e \{ \str_range:nnn { abcdefg } { 2 } { -3 } \}
\iow_term:e \{ \str_range:nnn { abcdefg } { -6 } { 5 } \}
\iow_term:e \{ \str_range:nnn { abcdefg } { -6 } { -3 } \}
\iow_term:e \{ \str_range:nnn { ab-e-fg } { 2 } { 5 } \}
\iow_term:e \{ \str_range:nnn { ab-e-fg } { 2 } { -3 } \}
\iow_term:e \{ \str_range:nnn { ab-e-fg } { -6 } { 5 } \}
\iow_term:e \{ \str_range:nnn { ab-e-fg } { -6 } { -3 } \}
\iow_term:e \{ \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } \}
\iow_term:e \{ \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } \}
\iow_term:e \{ \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } \}
\iow_term:e \{ \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } \}
\end{verbatim}
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}

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

\texttt{\textbackslash str\_replace\_all:Nnn}  \\texttt{\textbackslash str\_replace\_all:cn}  \\texttt{\textbackslash str\_greplace\_all:Nnn}  \\texttt{\textbackslash str\_greplace\_all:cn}

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

\texttt{\textbackslash str\_remove\_once:Nn}  \\texttt{\textbackslash str\_remove\_once:cn}  \\texttt{\textbackslash str\_gremove\_once:Nn}  \\texttt{\textbackslash str\_gremove\_once:cn}

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

\texttt{\textbackslash str\_remove\_all:Nn}  \\texttt{\textbackslash str\_remove\_all:cn}  \\texttt{\textbackslash str\_gremove\_all:Nn}  \\texttt{\textbackslash str\_gremove\_all:cn}

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

\texttt{\textbackslash str\_set:Nn \l_tmpa\_str \{abbc\textbackslash c\textbackslash d\} \textbackslash str\_remove\_all:Nn \l_tmpa\_str \{bc\}}

results in \texttt{\l_tmpa\_str} containing abcd.
17.7 String manipulation

\str_lowercase:n \str_lowercase:f \str_uppercase:n \str_uppercase:f

Converses 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

\begin{verbatim}
define N (\myfunc:n) #1#2 {
    \set_protected:n {
        user
        \str_uppercase:f { \tl_head:n {#1} }
        \str_lowercase:f { \tl_tail:n {#1} }
    }
    #2
}
\end{verbatim}

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

These functions should not be used for

- Caseless comparisons: use \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.
\str_casefold:n \{\langle\text{tokens}\rangle}\}
Converts the input $\langle\text{tokens}\rangle$ to their string representation, as described for $\tl_to_str:n$, and then folds the case of the resulting $\langle\text{string}\rangle$ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by $\str_casefold:n$ 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. I always folds to i and not to î).

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

### 17.8 Viewing strings

\str_show:N \{\text{str var}\}
Displays the content of the $\langle\text{str var}\rangle$ on the terminal.

\str_log:N \{\text{str var}\}
Writes the content of the $\langle\text{str var}\rangle$ in the log file.
17.9 Constant strings

\c_\texttt{ampersand\_str}  Constant strings, containing a single character token, with category code 12.
\c_\texttt{atsign\_str}
\c_\texttt{backslash\_str}
\c_\texttt{left\_brace\_str}
\c_\texttt{right\_brace\_str}
\c_\texttt{circumflex\_str}
\c_\texttt{colon\_str}
\c_\texttt{dollar\_str}
\c_\texttt{hash\_str}
\c_\texttt{percent\_str}
\c_\texttt{tilde\_str}
\c_\texttt{underscore\_str}
\c_\texttt{zero\_str}

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

\c_\texttt{empty\_str}  Constant that is always empty.

New: 2023-12-07

17.10 Scratch strings

\l_\texttt{tmpa\_str}  Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
\l_\texttt{tmpb\_str}

\g_\texttt{tmpa\_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_\texttt{tmpb\_str}
Chapter 18

The \texttt{l3str-convert} module
String encoding conversions

18.1 Encoding and escaping schemes

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

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be \texttt{utf-16}, \texttt{iso 8859-1}, \textit{etc.} See Table 1 for a list of supported encodings.\textsuperscript{6}

- Bytes are translated to \TeX tokens through a given “escaping”. Those are defined for the most part by the PDF file format. See Table 2 for a list of escaping methods supported.\textsuperscript{6}

\textsuperscript{6}Encodings and escapings will be added as they are requested.
Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

<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, hexagonal</td>
<td>byte = two hexadecimal digits</td>
</tr>
<tr>
<td>name</td>
<td>see \pdffontname</td>
</tr>
<tr>
<td>string</td>
<td>see \pdfescapestring</td>
</tr>
<tr>
<td>url</td>
<td>encoding used in URLs</td>
</tr>
</tbody>
</table>
18.2 Conversion functions

\texttt{\str_set_convert:Nnnn} \str_set_convert:Nnnn (\texttt{str var}) \{\langle string\rangle\} \{\langle name 1\rangle\} \{\langle name 2\rangle\}

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

For example,

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

results in the variable \l_foo_str holding the string FEFF00480065006C0066006F0021. This is obtained by converting each character in the (native) string Hello! to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark “FEFF, which can be avoided by specifying the encoding utf16be/hex.

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

\texttt{\str_set_convert:NnnnTF} \str_set_convert:NnnnTF (\texttt{str var}) \{\langle string\rangle\} \{\langle name 1\rangle\} \{\langle name 2\rangle\} \{\langle true code\rangle\} \{\langle false code\rangle\}

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

18.3 Conversion by expansion (for PDF contexts)

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

\texttt{\str_convert_pdfname:n} \str_convert_pdfname:n \langle \texttt{string} \rangle

As \texttt{\str_convert:Nnnn}, converts the \langle \texttt{string} \rangle on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

18.4 Possibilities, and things to do

Encoding/escaping-related tasks.
• In 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{\textbackslash 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{url} escaping. Perhaps the characters !'()*-./0123456789_ are safe, and all other characters should be escaped?

• Automate generation of 8-bit mapping files.

• Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.

• More encodings (see Heiko’s \texttt{stringenc}). CESU?

• More escapings: \texttt{ascii85}, shell escapes, lua escapes, \textit{etc}?
Chapter 19

The \texttt{l3quark} module
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{\textbackslash q}, and scan marks start with \texttt{\textbackslash s}.

19.1 Quarks

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore \textit{never} be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (\textit{i.e.} \texttt{\textbackslash 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{\textbackslash 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{\textbackslash 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.

146
19.2 Defining quarks

\quark_new:N \quark_new:N \quark \langle quark \rangle

Creates a new \textit{(quark)} which expands only to \textit{(quark)}. The \textit{(quark)} is defined globally, and an error message is raised if the name was already taken.

\quark_stop

Used as a marker for delimited arguments, such as

\texttt{\textbackslash cs\_set:Npn \textbackslash tmp:w \#1\#2 \quark_stop \{\#1\}}

\quark_mark

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

\quark_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 \quark_stop, which is only ever used as a delimiter).

\quark_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 \texttt{\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 \quark_if_nil_p:noV \quark_if_nil_p:nVF
\quark_if_nil:NTF \quark_if_nil:nTF \quark_if_nil:nTFoV \quark_if_nil:nTFoVF

Tests if the \texttt{(token)} is equal to \texttt{\quark\_nil}.

\quark_if_no_value_p:N \quark_if_no_value_p:n \quark_if_no_value_p:noV \quark_if_no_value_p:nVF
\quark_if_no_value:NTF \quark_if_no_value:nTF \quark_if_no_value:nTFoV \quark_if_no_value:nTFoVF

Tests if the \texttt{(token list)} contains only \texttt{\quark\_nil} (distinct from \texttt{(token list) being empty or containing \texttt{\quark\_nil} 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 \star \quark_if_recursion_tail_stop:N \langle \text{token} \rangle

Tests if \langle \text{token} \rangle 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 \star \quark_if_recursion_tail_stop:o \star

Updated: 2011-09-06

Tests if the \langle \text{token list} \rangle contains only \q_recursion_tail, and if so uses \use_none_delimit_by_q_recursion_stop:w to terminate the recursion that this belongs to. The recursion input must include the marker tokens \q_recursion_tail and \q_recursion_stop as the last two items.

\quark_if_recursion_tail_stop_do:Nn \star \quark_if_recursion_tail_stop_do:Nn \langle \text{token} \rangle \langle \text{insertion} \rangle

Tests if \langle \text{token} \rangle 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 \langle \text{insertion} \rangle code is then added to the input stream after the recursion has ended.

\quark_if_recursion_tail_stop_do:nn \star \quark_if_recursion_tail_stop_do:nn \langle \text{token list} \rangle \langle \text{insertion} \rangle

Updated: 2011-09-06

Tests if the \langle \text{token list} \rangle 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 \langle \text{insertion} \rangle code is then added to the input stream after the recursion has ended.
Tests if \texttt{token list} contains only \texttt{q_recursion_tail}, and if so terminates the recursion using \texttt{(type)}\textunderscore map\textunderscore break:. The recursion end should be marked by \texttt{prg\_break\_point}:Nn \texttt{(type)}\textunderscore map\textunderscore break:.

### 19.4.1 An example of recursion with quarks

Quarks are mainly used internally in the \texttt{expl3} code to define recursion functions such as \texttt{tl\_map\_inline:nn} and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called \texttt{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 “[-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{q\_recursion\_tail}, with delimiters according to the type of recursion (here a pair of \texttt{q\_recursion\_tail}), concluding with \texttt{q\_recursion\_stop}. These quarks are used to mark the end of the token list being operated upon.

\begin{verbatim}
\cs_new:Npn \my_map_dbl:nn #1#2 
{ 
\cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
\__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail \q_recursion_stop
}
\end{verbatim}

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

\begin{verbatim}
\cs_new:Nn \__my_map_dbl:nn 
{ 
\quark_if_recursion_tail_stop:n {#1}
\quark_if_recursion_tail_stop:n {#2}
\__my_map_dbl_fn:nn #1 #2
}
\end{verbatim}

Finally, recurse:

\begin{verbatim}
\__my_map_dbl:nn
\end{verbatim}

Note that contrary 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}.

149
19.5 Scan marks

Scan marks are control sequences set equal to \texttt{\char46 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 \texttt{l3regex}).

\texttt{\char46 scan_new:N} \hspace{1em} \texttt{\char46 scan_new:N \langle \text{scan mark} \rangle}

\texttt{New: 2018-04-01}

Creates a new \texttt{\langle \text{scan mark} \rangle} which is set equal to \texttt{\char46 scan_stop:}. The \texttt{\langle \text{scan mark} \rangle} is defined globally, and an error message is raised if the name was already taken by another scan mark.

\texttt{\char46 s_stop} \hspace{1em} \texttt{\char46 s_stop}

\texttt{New: 2018-04-01}

Used at the end of a set of instructions, as a marker that can be jumped to using \texttt{\use_\texttt{-\char46 none_delimit_by_s_stop:w}}.

\texttt{\use\_none\_delimit\_by\_s\_stop:w} \hspace{1em} \texttt{\use\_none\_delimit\_by\_s\_stop:w \langle \text{tokens} \rangle \char46 s\_stop}

\texttt{New: 2018-04-01}

Removes the \texttt{\langle \text{tokens} \rangle} and \texttt{\char46 s\_stop} from the input stream. This leads to a low-level \TeX{} error if \texttt{\char46 s\_stop} is absent.
Chapter 20

The \texttt{l3seq} module
Sequences and stacks

\LaTeX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any \textit{balanced text}. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in \LaTeX3. This is achieved using a number of dedicated stack functions.

20.1 Creating and initialising sequences

\begin{verbatim}
\seq_new:N \seq_new:cx
\seq_clear:N \seq_clear:cx \seq_gclear:N \seq_gclear:cx
\seq_set_eq:NN \seq_set_eq:N \seq_gset_eq:NN \seq_gset_eq:N
\seq_gset_eq:ccx
\end{verbatim}

\texttt{\seq_new:N \seq_new:cx} \hspace{1cm} \texttt{\seq_new:N \seq_new:cx} (seq var)

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.

\texttt{\seq_clear:N \seq_clear:cx \seq_gclear:N \seq_gclear:cx} \hspace{1cm} \texttt{\seq_clear:N \seq_clear:cx \seq_gclear:N \seq_gclear:cx} (seq var)

Clears all items from the \texttt{seq var}.

\texttt{\seq_set_eq:NN \seq_set_eq:N \seq_gset_eq:NN \seq_gset_eq:N \seq_gset_eq:ccx} \hspace{1cm} \texttt{\seq_set_eq:NN \seq_set_eq:N \seq_gset_eq:NN \seq_gset_eq:N \seq_gset_eq:ccx} (seq var1) (seq var2)

Sets the content of \texttt{seq var1} equal to that of \texttt{seq var2}.
Converts the data in the \textit{comma list} into a \textit{seq var}: the original \textit{comma list} is unchanged.

Splits the \textit{token list} into \textit{items} separated by \texttt{delimiter}, and assigns the result to the \textit{seq var}. Spaces on both sides of each \textit{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 \textit{items} are preserved by \texttt{seq_split:Nnn}, and can be removed afterwards using \texttt{seq_remove_all:Nn \{seq var\} \{\}}. The \texttt{delimiter} may not contain \{,\} or \# (assuming \TeX’s normal category code régime). If the \texttt{delimiter} is empty, the \textit{token list} is split into \textit{items} as a \textit{token list}. See also \texttt{seq_split_keep_spaces:Nnn}, which omits space stripping.

Splits the \textit{token list} into \textit{items} separated by \texttt{delimiter}, and assigns the result to the \textit{seq var}. One set of outer braces is removed (if any) but any surrounding spaces are retained: any braces \textit{inside} one or more spaces are therefore kept. Empty \textit{items} are preserved by \texttt{seq_split_keep_spaces:Nnn}, and can be removed afterwards using \texttt{seq_remove_all:Nn \{seq var\} \{\}}. The \texttt{delimiter} may not contain \{,\} or \# (assuming \TeX’s normal category code régime). If the \texttt{delimiter} is empty, the \textit{token list} is split into \textit{items} as a \textit{token list}. See also \texttt{seq_split:Nnn}, which removes spaces around the delimiters.
Evaluates the \(\text{inline booleanexpr}\) for every \(\langle\text{item}\rangle\) stored within the \(\langle\text{seq var}\rangle_2\). The \(\text{inline booleanexpr}\) receives the \(\langle\text{item}\rangle\) as \#1. The sequence of all \(\langle\text{items}\rangle\) for which the \(\text{inline booleanexpr}\) evaluated to true is assigned to \(\langle\text{seq var}\rangle_1\).

**TeXhackers note:** Contrarily to other mapping functions, \seqmapbreak: cannot be used in this function, and would lead to low-level TeX errors.

Concatenates the content of \(\langle\text{seq var}\rangle_2\) and \(\langle\text{seq var}\rangle_3\) together and saves the result in \(\langle\text{seq var}\rangle_1\). The items in \(\langle\text{seq var}\rangle_2\) are placed at the left side of the new sequence.

Tests whether the \(\langle\text{seq var}\rangle\) is currently defined. This does not check that the \(\langle\text{seq var}\rangle\) really is a sequence variable.

### 20.2 Appending data to sequences

Appends the \(\langle\text{item}\rangle\) to the left of the \(\langle\text{seq var}\rangle\).

Appends the \(\langle\text{item}\rangle\) to the right of the \(\langle\text{seq var}\rangle\).

### 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 \textsf{token list variable} used with \tlset:Nn and never \tlgset:Nn.

Stores the left-most item from a \(\langle\text{seq var}\rangle\) in the \textsf{token list variable} without removing it from the \(\langle\text{seq var}\rangle\). The \textsf{token list variable} is assigned locally. If \(\langle\text{seq var}\rangle\) is empty the \textsf{token list variable} is set to the special marker \q_custom{\text{n}}.
Stores the right-most item from a \langle \text{seq var} \rangle in the \langle \text{token list variable} \rangle without removing it from the \langle \text{seq var} \rangle. The \langle \text{token list variable} \rangle is assigned locally. If \langle \text{seq var} \rangle is empty the \langle \text{token list variable} \rangle is set to the special marker \texttt{\q_no_-value}.

Pops the left-most item from a \langle \text{seq var} \rangle into the \langle \text{token list variable} \rangle, i.e. removes the item from the sequence and stores it in the \langle \text{token list variable} \rangle. Both of the variables are assigned locally. If \langle \text{seq var} \rangle is empty the \langle \text{token list variable} \rangle is set to the special marker \texttt{\q_no_-value}.

Pops the left-most item from a \langle \text{seq var} \rangle into the \langle \text{token list variable} \rangle, i.e. removes the item from the sequence and stores it in the \langle \text{token list variable} \rangle. The \langle \text{seq var} \rangle is modified globally, while the assignment of the \langle \text{token list variable} \rangle is local. If \langle \text{seq var} \rangle is empty the \langle \text{token list variable} \rangle is set to the special marker \texttt{\q_no_-value}.

Pops the right-most item from a \langle \text{seq var} \rangle into the \langle \text{token list variable} \rangle, i.e. removes the item from the sequence and stores it in the \langle \text{token list variable} \rangle. The \langle \text{seq var} \rangle is modified globally, while the assignment of the \langle \text{token list variable} \rangle is local. If \langle \text{seq var} \rangle is empty the \langle \text{token list variable} \rangle is set to the special marker \texttt{\q_no_-value}.

Indexing items in the \langle \text{seq var} \rangle from 1 at the top (left), this function evaluates the \langle \text{integer expression} \rangle and leaves the appropriate item from the sequence in the input stream. If the \langle \text{integer expression} \rangle is negative, indexing occurs from the bottom (right) of the sequence. If the \langle \text{integer expression} \rangle is larger than the number of items in the \langle \text{seq var} \rangle (as calculated by \seq_count:N) then the function expands to nothing.

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

Selects a pseudo-random item of the \seq. If the \seq is empty the result is empty.

\textbf{\TeX}hackers 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 \texttt{e}-type or \texttt{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:NN \seq_get_right:NNTF \seq_get_right:NN \seq_pop_left:NNTF \seq_pop_left:NN \seq_gpop_left:NNTF \seq_gpop_left:NN

If the \seq is empty, leaves the \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 \seq is non-empty, stores the left-most item from the \seq in the \texttt{token list variable} without removing it from the \seq, then leaves the \true code in the input stream. The \texttt{token list variable} is assigned locally.

If the \seq is empty, leaves the \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 \seq is non-empty, stores the right-most item from the \seq in the \texttt{token list variable} without removing it from the \seq, then leaves the \true code in the input stream. The \texttt{token list variable} is assigned locally.

If the \seq is empty, leaves the \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 \seq is non-empty, pops the left-most item from the \seq in the \texttt{token list variable}, i.e. removes the item from the \seq, then leaves the \true code in the input stream. Both the \seq and the \texttt{token list variable} are assigned locally.

If the \seq is empty, leaves the \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 \seq is non-empty, pops the left-most item from the \seq in the \texttt{token list variable}, i.e. removes the item from the \seq, then leaves the \true code in the input stream. The \seq is modified globally, while the \texttt{token list variable} is assigned locally.
20.5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

\seq_pop_right\NTF\seq_var\token_list_variable \{\true_code\} \{\false_code\}

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.

\seq_pop_right\NTF\seq_var\token_list_variable \{\true_code\} \{\false_code\}

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.

\seq_remove_duplicates:N\seq_var

Removes duplicate items from the \seq_var, leaving the left most copy of each item in the \seq_var. The \item comparison takes place on a token basis, as for \tl_if_eq:nTF.

\TeXhackersnote: This function iterates through every item in the \seq_var and does a comparison with the items already checked. It is therefore relatively slow with large sequences.

\seq_remove_all:Nn\seq_var\item

Removes every occurrence of \item from the \seq_var. The \item comparison takes place on a token basis, as for \tl_if_eq:nTF.

\TeXhackersnote: In cases where the assignment was successful, \true_code is run afterwards.
\seq_reverse:N \seq_reverse:c \seq_greverse:N \seq_greverse:c
Reverses the order of the items stored in the \seq var.

\seq_sort:Nn \seq_sort:cn \seq_gsort:Nn \seq_gsort:cn
Sorts the items in the \seq var according to the \comparison code, and assigns the result to \seq var. The details of sorting comparison are described in Section 6.1.

\seq_shuffle:N \seq_shuffle:c \seq_gshuffle:N \seq_gshuffle:c
Sets the \seq var to the result of placing the items of the \seq var in a random order. Each item is (roughly) as likely to end up in any given position.

\seq_if_empty_p:N \seq_if_empty:NTF \seq_if_empty:N \seq_if_empty:TF
\seq_if_in:NnTF \seq_if_in:Nn \seq_if_in:NTF \seq_if_in:NTF \seq_if_in:NTF \seq_if_in:NTF
Tests if the \seq var is empty (containing no items).
Tests if the \item is present in the \seq var.

\seq_map_function:NN \seq_map_function:cn \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. To pass further arguments to the \function, see \seq_map_tokens:Nn. The function \seq_map_inline:NN is faster than \seq_map_function:NN for sequences with more than about 10 items.

**20.6 Sequence conditionals**

**20.7 Mapping over sequences**

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

\seq_sort:Nn \seq_sort:cn \seq_gsort:Nn \seq_gsort:cn
Sorts the items in the \seq var according to the \comparison code, and assigns the result to \seq var. The details of sorting comparison are described in Section 6.1.

\seq_shuffle:N \seq_shuffle:c \seq_gshuffle:N \seq_gshuffle:c
Sets the \seq var to the result of placing the items of the \seq var in a random order. Each item is (roughly) as likely to end up in any given position.

\seq_if_empty_p:N \seq_if_empty:NTF \seq_if_empty:N \seq_if_empty:TF
\seq_if_in:NnTF \seq_if_in:Nn \seq_if_in:NTF \seq_if_in:NTF \seq_if_in:NTF \seq_if_in:NTF
Tests if the \seq var is empty (containing no items).
Tests if the \item is present in the \seq var.

\seq_map_function:NN \seq_map_function:cn \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. To pass further arguments to the \function, see \seq_map_tokens:Nn. The function \seq_map_inline:NN is faster than \seq_map_function:NN for sequences with more than about 10 items.

**20.6 Sequence conditionals**

**20.7 Mapping over sequences**

All mappings are done at the current group level, \textit{i.e.} any local assignments made by the \function or \code discussed below remain in effect after the loop.
\seq_map_inline:Nn \seq_map_inline:cn
Applies \textit{inline function} to every \textit{item} stored within the \textit{seq var}. The \textit{inline function} should consist of code which will receive the \textit{item} as \#1. The \textit{items} are returned from left to right.

\seq_map_tokens:Nn \seq_map_tokens:cn
Analogue of \seq_map_function:NN which maps several tokens instead of a single function. The \textit{code} receives each item in the \textit{seq var} as a trailing brace group. For instance,

\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }
expands to twice each item in the \textit{seq var}: for each item in \texttt{\l_my_seq} the function \texttt{\prg_replicate:nn} receives \texttt{2} and \textit{item} as its two arguments. The function \texttt{\seq_map_inline:Nn} is typically faster but it is not expandable.

\seq_map_variable:NNn \seq_map_variable:NNm \seq_map_variable:NN (Ncn|cNn|ccNn)
Updated: 2012-06-29
Stores each \textit{item} of the \textit{seq var} in turn in the (token list) \textit{variable} and applies the \textit{code}. The \textit{code} will usually make use of the \textit{variable}, but this is not enforced. The assignments to the \textit{variable} are local. Its value after the loop is the last \textit{item} in the \textit{seq var}, or its original value if the \textit{seq var} is empty. The \textit{items} are returned from left to right.

\seq_map_indexed_function:NN \seq_map_indexed_function:NN (function)
Updated: 2018-05-03
Applies \textit{function} to every entry in the \textit{seq var}. The \textit{function} should have signature :nn. It receives two arguments for each iteration: the \textit{index} (namely 1 for the first entry, then 2 and so on) and the \textit{item}.

\seq_map_indexed_inline:Nn \seq_map_indexed_inline:NN (inline function)
Updated: 2018-05-03
Applies \textit{inline function} to every entry in the \textit{seq var}. The \textit{inline function} should consist of code which receives the \textit{index} (namely 1 for the first entry, then 2 and so on) as \#1 and the \textit{item} as \#2.

\seq_map_pairwise_function:NNn \seq_map_pairwise_function:NNm \seq_map_pairwise_function:NN (function)
Updated: 2023-05-10
Applies \textit{function} to every pair of items \textit{seq}_1-item–\textit{seq}_2-item from the two sequences, returning items from both sequences from left to right. The \textit{function} receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (\textit{i.e.} whichever sequence has fewer items determines how many iterations occur).
\seq_map_break: \seq_map_break:

Used to terminate a \seq_map... function before all entries in the \seqvar have been processed. This normally takes place within a conditional statement, for example

\seq_map_inline:Nn \l_my_seq
{\str_if_eq:nnTF { #1 } { bingo }\seq_map_break: }%
% Do something useful
}

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

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

\seq_map_break:n \seq_map_break:n \langle\text{code}\rangle

Used to terminate a \seq_map... function before all entries in the \seqvar have been processed, inserting the \langle\text{code}\rangle 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 \langle\text{code}\rangle}%
% 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\text{code}\rangle is inserted into the input stream. This depends on the design of the mapping function.

\seq_set_map:NNn \seq_gset_map:NNn \langle\seqvar_1\rangle \langle\seqvar_2\rangle \langle\text{inline function}\rangle

Applies \langle\text{inline function}\rangle to every \langle\item\rangle stored within the \seqvar. The \langle\text{inline function}\rangle should consist of code which will receive the \langle\item\rangle as \#1. The sequence resulting applying \langle\text{inline function}\rangle to each \langle\item\rangle is assigned to \langle\seqvar_1\rangle.

**\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

Applies \textit{inline function} to every \textit{item} stored within the \textit{seq var}_2. The \textit{inline function} should consist of code which will receive the \textit{item} as \#1. The sequence resulting from e-expanding \textit{inline function} applied to each \textit{item} is assigned to \textit{seq var}_1. As such, the code in \textit{inline function} should be expandable.

\textbf{\TeXhackers note:} Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level \TeXX errors.

\seq_count:N \seq_count:c

Leaves the number of items in the \textit{seq var} in the input stream as an \textit{integer denotation}. The total number of items in a \textit{seq var} includes those which are empty and duplicates, \ie every item in a \textit{seq var} is unique.

\textbf{20.8 Using the content of sequences directly}

\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | \{de\} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }

inserts “\textit{a, b, c, de, and f}” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

\textbf{\TeXhackers note:} The result is returned within the \texttt{\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.
Places the contents of the \langle seq var \rangle 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,
\begin{verbatim}
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | \{de\} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
\end{verbatim}

inserts “a and b and c and de and f” in the input stream.

\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.

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.

\begin{verbatim}
\seq_get:NN \seq_get:cN
\seq_pop:NN \seq_pop:cN
\seq_gpop:NN \seq_gpop:cN
\seq_get:NTF \seq_get:cNTF
\end{verbatim}

Updated: 2012-05-19

If the \langle seq var \rangle 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 \langle seq var \rangle is non-empty, stores the top item from a \langle seq var \rangle in the \langle token list variable \rangle without removing it from the \langle seq var \rangle. 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 \langle seq var \rangle only has distinct items, use \seq_remove_duplicates:N \langle seq var \rangle. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set \langle seq var \rangle are straightforward. For instance, \seq_count:N \langle seq var \rangle expands to the number of items, while \seq_if_in:NnTF \langle seq var \rangle \{ \langle item \rangle \} tests if the \langle item \rangle is in the set.

Adding an \langle item \rangle to a set \langle seq var \rangle can be done by appending it to the \langle seq var \rangle if it is not already in the \langle seq var \rangle:

\seq_if_in:NnF \langle seq var \rangle \{ \langle item \rangle \} \{ \seq_put_right:Nn \langle seq var \rangle \{ \langle item \rangle \} \}

Removing an \langle item \rangle from a set \langle seq var \rangle can be done using \seq_remove_all:Nn,

\seq_remove_all:Nn \langle seq var \rangle \{ \langle item \rangle \}

The intersection of two sets \langle seq var_1 \rangle and \langle seq var_2 \rangle can be stored into \langle seq var_3 \rangle by collecting items of \langle seq var_1 \rangle which are in \langle seq var_2 \rangle.
\seq_clear:N \seq_var_3
\seq_map_inline:Nn \seq_var_1
{ \seq_if_in:NnT \seq_var_2 \#1
  \seq_put_right:Nn \seq_var_3 \#1
}

The code as written here only works if \seq_var_3 is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence \l__⟨pkg⟩_internal_seq, then \seq_var_3 should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets \seq_var_1 and \seq_var_2 can be stored into \seq_var_3 through
\seq_concat:NNN \seq_var_3 \seq_var_1 \seq_var_2
\seq_remove_duplicates:N \seq_var_3
or by adding items to (a copy of) \seq_var_3 one by one
\seq_set_eq:NN \seq_var_3 \seq_var_1
\seq_map_inline:Nn \seq_var_2
{ \seq_if_in:NnF \seq_var_3 \#1
  \seq_put_right:Nn \seq_var_3 \#1
}

The second approach is faster than the first when the \seq_var_2 is short compared to \seq_var_1.

The difference of two sets \seq_var_1 and \seq_var_2 can be stored into \seq_var_3 by removing items of the \seq_var_2 from (a copy of) the \seq_var_1 one by one.
\seq_set_eq:NN \seq_var_3 \seq_var_1
\seq_map_inline:Nn \seq_var_2
{ \seq_remove_all:Nn \seq_var_3 \#1
}

The symmetric difference of two sets \seq_var_1 and \seq_var_2 can be stored into \seq_var_3 by computing the difference between \seq_var_1 and \seq_var_2 and storing the result as \l__⟨pkg⟩_internal_seq, then the difference between \seq_var_2 and \seq_var_1, and finally concatenating the two differences to get the symmetric differences.
\seq_set_eq:NN \l__⟨pkg⟩_internal_seq \seq_var_1
\seq_map_inline:Nn \seq_var_2
{ \seq_remove_all:Nn \l__⟨pkg⟩_internal_seq \#1
}\seq_set_eq:NN \seq_var_3 \seq_var_2
\seq_map_inline:Nn \seq_var_1
{ \seq_remove_all:Nn \seq_var_3 \#1
}\seq_concat:NNN \seq_var_3 \seq_var_3 \l__⟨pkg⟩_internal_seq

20.11 Constant and scratch sequences

\c_empty_seq Constant that is always empty.

Revised: 2012-07-02
\l_tmpa_seq  Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\l_tmpb_seq

\l_tmpa_seq  \l_tmpb_seq  
Rev: 2012-04-26

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

\g_tmpb_seq

\g_tmpa_seq  \g_tmpb_seq  
Rev: 2012-04-26

\section{20.12 Viewing sequences}

\seq_log:N  \seq_log:N \langle \text{seq var} \rangle
\seq_log:c

\seq_log:N  \seq_log:N \langle \text{seq var} \rangle
\seq_log:c

Updated: 2021-04-29

\seq_show:N  \seq_show:N \langle \text{seq var} \rangle
\seq_show:c

\seq_show:N  \seq_show:N \langle \text{seq var} \rangle
\seq_show:c

Updated: 2021-04-29
Chapter 21

The \texttt{l3int} module

Integers

Calculation and comparison of integer values can be carried out using literal numbers, \texttt{int}
registers, constants and integers stored in token list variables. The standard operators $+$,
$-$, $/$ and $*$ and parentheses can be used within such expressions to carry arithmetic
operations. This module carries out these functions on integer expressions ("\langle \text{int expr} \rangle").

21.1 Integer expressions

Throughout this module, (almost) all \texttt{n}-type argument allow for an \langle \text{int expr} \rangle argument
with the following syntax. The \texttt{integer expression} should consist, after expansion,
of $+$, $-$, $*$, $/$, (, ) and of course integer operands. The result is calculated by applying
standard mathematical rules with the following peculiarities:

- $/$ denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute
  value of any intermediate result exceeds $2^{31} - 1$, except in the case of scaling oper-
  ations $a*b/c$, for which $a*b$ may be arbitrarily large (but the operands $a$, $b$, $c$ are
  still constrained to an absolute value at most $2^{31} - 1$);
- parentheses may not appear after unary $+$ or $-$, namely placing $+(\text{ or } -\text{ at the
  start of an expression or after } +, -, *, / \text{ or } ( \text{ leads to an error.}$

Each integer operand can be either an integer variable (with no need for \texttt{\int_use:N}) or
an integer denotation. For example both

\begin{verbatim}
\int_show:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
\end{verbatim}

and

\begin{verbatim}
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { 5 }
\int_new:N \l_my_int
\int_set:Nn \l_my_int { 4 }
\int_show:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
\end{verbatim}
show the same result −6 because \_my\_tl expands to the integer denotation 5 while the integer variable \_my\_int takes the value 4. As the ⟨integer expression⟩ is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and \exp\_not:n and its variants have no effect while \exp\_not:N may incorrectly interrupt the expression.

\TeXhackers note: Exactly two expansions are needed to evaluate \int_eval:n. The result is not an ⟨internal integer⟩, and therefore should be terminated by a space if used in \int_value:w or in a \TeX-style integer assignment.

As all \TeX integers, integer operands can also be: \value{⟨LATEX 2ε counter⟩}; dimension or skip variables, converted to integers in \sp; the character code of some character given as ‘⟨char⟩’ or ‘\⟨char⟩’; octal numbers given as ‘\’ followed by digits from 0 to 7; or hexadecimal numbers given as ‘\’ followed by digits and upper case letters from A to F.

\int_eval:n \{⟨int expr⟩\}
Evaluates the ⟨int expr⟩ and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with 0, for negative results − followed by such a sequence, and 0 for zero.

\TeXhackers note: Exactly two expansions are needed to evaluate \int_eval:n. The result is not an ⟨internal integer⟩, and therefore requires suitable termination if used in a \TeX-style integer assignment.

As all \TeX integers, integer operands can also be dimension or skip variables, converted to integers in \sp, or octal numbers given as ‘\’ followed by digits other than 8 and 9, or hexadecimal numbers given as ‘\’ followed by digits or upper case letters from A to F, or the character code of some character or one-character control sequence, given as ‘\langlechar⟩’.

\int_eval:w \{⟨int expr⟩\}
Evaluates the ⟨int expr⟩ as described for \int_eval:n. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is \scan_stop: it is removed, otherwise not. Spaces do not terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, \int_eval:w 1\_9\_1\_0 (with explicit space tokens inserted using ~ in a code setting) expands to 29 since the digit 9 is not part of the expression. Expansion details, etc., are as given for \int_eval:n.

\int_sign:n \{⟨int expr⟩\}
Evaluates the ⟨int expr⟩ then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

\int_abs:n \{⟨int expr⟩\}
Evaluates the ⟨int expr⟩ as described for \int_eval:n and leaves the absolute value of the result in the input stream as an ⟨integer denotation⟩ after two expansions.
\int_div_round:nn \{\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:c \text{integer}

Creates a new \{\text{integer}\} or raises an error if the name is already taken. The declaration is global. The \{\text{integer}\} is initially equal to 0.

\int_const:Nn \text{integer} \{\text{int expr}\}
\int_const:cn \text{integer} \{\text{int expr}\}

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 \text{integer}
\int_zero:c \text{integer}
\int_gzero:N \text{integer}
\int_gzero:c \text{integer}

Sets \{\text{integer}\} to 0.

\int_zero_new:N \text{integer}
\int_zero_new:c \text{integer}
\int_gzero_new:N \text{integer}
\int_gzero_new:c \text{integer}

Ensures that the \{\text{integer}\} exists globally by applying \int_new:N if necessary, then applies \int_(g)zero:N to leave the \{\text{integer}\} set to zero.
\int_set_eq:NN \langle \text{integer}_1 \rangle \langle \text{integer}_2 \rangle

Sets the content of \langle \text{integer}_1 \rangle equal to that of \langle \text{integer}_2 \rangle.

\int_set_eq:(cN|N|cc)
\int_gset_eq:(cN|N|cc)
\int_if_exist_p:N \langle \text{int} \rangle
\int_if_exist:NTF \langle \text{int} \rangle \{\langle \text{true code} \rangle \} \{\langle \text{false code} \rangle \}

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 \langle \text{integer} \rangle \{\langle \text{int expr} \rangle \}

Adds the result of the \langle \text{int expr} \rangle to the current content of the \langle \text{integer} \rangle.

\int_decr:N \langle \text{integer} \rangle
\int_decr:c
\int_gdecr:N
\int_gdecr:c

Decreases the value stored in \langle \text{integer} \rangle by 1.

\int_incr:N \langle \text{integer} \rangle
\int_incr:c
\int_gincr:N
\int_gincr:c

Increases the value stored in \langle \text{integer} \rangle by 1.

\int_set:Nn \langle \text{integer} \rangle \{\langle \text{int expr} \rangle \}

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 \langle \text{integer} \rangle \{\langle \text{int expr} \rangle \}

Subtracts the result of the \langle \text{int expr} \rangle from the current content of the \langle \text{integer} \rangle.
21.4 Using integers

\int_use:N \int_use:c

Recovers the content of an \texttt{integer} and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an \texttt{integer} is required (such as in the first and third arguments of \int_compare:nNnTF).

\textbf{TeXhackers note:} \int_use:N is the \TeX{} primitive \texttt{the}: this is one of several \LaTeX{}3 names for this primitive.

21.5 Integer expression conditionals

\int_compare_p:nNn \int_compare:nNnTF

\texttt{\{int expr\}_1} \texttt{\{relation\}} \texttt{\{int expr\}_2}

\texttt{\{true code\}} \texttt{\{false code\}}

This function first evaluates each of the \texttt{int expr} as described for \int_eval:n. The two results are then compared using the \texttt{relation}:

\begin{align*}
\text{Equal} & = \\
\text{Greater than} & > \\
\text{Less than} & < 
\end{align*}

This function is less flexible than \int_compare:nTF but around 5 times faster.
This function evaluates the \{int expr\}s as described for \inteval:n and compares consecutive result using the corresponding \{relation\}, namely it compares \{int expr\} and \{int expr\} using the \{relation\}, then \{int expr\} and \{int expr\} using the \{relation\}, until finally comparing \{int expr\} and \{int expr\} using the \{relation\}. The test yields true if all comparisons are true. Each \{int expr\} is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is false, then no other \{integer expression\} is evaluated and no other comparison is performed. The \{relations\} can be any of the following:

\begin{align*}
& \text{Equal} \quad = \text{ or } == \\
& \text{Greater than or equal to} \quad >= \\
& \text{Greater than} \quad > \\
& \text{Less than or equal to} \quad <= \\
& \text{Less than} \quad < \\
& \text{Not equal} \quad !=
\end{align*}

This function is more flexible than \intcompare:nNnTF but around 5 times slower.
This function evaluates the \(\text{test int expr}\) and compares this in turn to each of the \(\text{int expr cases}\). If the two are equal then the associated \(\text{code}\) is left in the input stream and other cases are discarded. If any of the cases are matched, the \(\text{true code}\) is also inserted into the input stream (after the code for the appropriate case), while if none match then the \(\text{false code}\) is inserted. The function \(\text{int_case:nn}\), which does nothing if there is no match, is also available. For example

\[
\text{\texttt{int_case:nnF}} \{ 2 \times 5 \}
\{
\{ 5 \} \text { Small }
\{ 4 \times 6 \} \text { Medium }
\{ 2 \times 10 \} \text { Negative }
\}
\{ \text { No idea! } \}
\]

leaves “Medium” in the input stream.

This function first evaluates the \(\text{int expr}\) as described for \(\text{int_eval:n}\). It then evaluates if this is odd or even, as appropriate.

This function first evaluates the \(\text{int expr}\) as described for \(\text{int_eval:n}\). It then evaluates if this is zero or not.

\section*{21.6 Integer expression loops}

Places the \(\text{code}\) in the input stream for \(\text{\LaTeX}\) to process, and then evaluates the relationship between the two \(\text{int expr}\)s as described for \(\text{int_compare:nNnTF}\). If the test is \texttt{false} then the \(\text{code}\) is inserted into the input stream again and a loop occurs until the \(\text{relation}\) is \texttt{true}.

171
\texttt{\texttt{\texttt{\textbackslash int\_do\textunderscore while:nNnn}}} \texttt{\{int expr\} \{relation\} \{(int expr_2)\} \{(code)\}}

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \texttt{(int expr)s} as described for \texttt{\texttt{\textbackslash int\_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}.

\texttt{\texttt{\texttt{\textbackslash int\_until\_do:nNnn}}} \texttt{\{int expr\} \{relation\} \{(int expr_2)\} \{(code)\}}

Evaluates the relationship between the two \texttt{(int expr)s} as described for \texttt{\texttt{\textbackslash int\_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{false}.

\texttt{\texttt{\texttt{\textbackslash int\_while\_do:nNnn}}} \texttt{\{int expr\} \{relation\} \{(int expr_2)\} \{(code)\}}

Evaluates the relationship between the two \texttt{(int expr)s} as described for \texttt{\texttt{\textbackslash int\_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{\texttt{\texttt{\textbackslash int\_until\_do:nn}}} \texttt{\{integer relation\} \{(code)\}}

Updated: 2013-01-13

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the \texttt{(integer relation)} as described for \texttt{\texttt{\textbackslash int\_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}.

\texttt{\texttt{\texttt{\textbackslash int\_do\textunderscore while:nn}}} \texttt{\{integer relation\} \{(code)\}}

Updated: 2013-01-13

Places the \texttt{(code)} in the input stream for \TeX{} to process, and then evaluates the \texttt{(integer relation)} as described for \texttt{\texttt{\textbackslash int\_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}.

\texttt{\texttt{\texttt{\textbackslash int\_until\_do:nn}}} \texttt{\{integer relation\} \{(code)\}}

Updated: 2013-01-13

Evaluates the \texttt{(integer relation)} as described for \texttt{\texttt{\textbackslash int\_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}.

\texttt{\texttt{\texttt{\textbackslash int\_while\_do:nn}}} \texttt{\{integer relation\} \{(code)\}}

Updated: 2013-01-13

Evaluates the \texttt{(integer relation)} as described for \texttt{\texttt{\textbackslash int\_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}.
21.7 Integer step functions

\texttt{$\int_{\text{step function}}:nN$} \hspace{1em} \texttt{$\int_{\text{step function}}:nnN$} \hspace{1em} \texttt{$\int_{\text{step function}}:nnnN$}

This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, all of which should be integer expressions. 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

\begin{verbatim}
cs_set:Npn \my_func:n #1 { [I saw #1] \quad }
\int_step_function:nnN \{ 1 \} \{ 5 \} \my_func:n
\end{verbatim}

would print

[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]

The functions $\int_{\text{step function}}:nN$ and $\int_{\text{step function}}:nnN$ both use a fixed \langle step \rangle of 1, and in the case of $\int_{\text{step function}}:nN$ the \langle initial value \rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\texttt{$\int_{\text{step inline}}:nn$} \hspace{1em} \texttt{$\int_{\text{step inline}}:nnn$}

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 $\int_{\text{step inline}}:nn$ and $\int_{\text{step inline}}:nnn$ both use a fixed \langle step \rangle of 1, and in the case of $\int_{\text{step inline}}:nn$ the \langle initial value \rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\texttt{$\int_{\text{step variable}}:nNn$} \hspace{1em} \texttt{$\int_{\text{step variable}}:nnNn$} \hspace{1em} \texttt{$\int_{\text{step variable}}:nnnNn$}

This function first evaluates the \langle initial value \rangle, \langle step \rangle and \langle final value \rangle, all of which should be integer expressions. Then for each \langle value \rangle from the \langle initial value \rangle to the \langle final value \rangle in turn (using \langle step \rangle between each \langle value \rangle), the \langle code \rangle is inserted into the input stream, with the \langle tl var \rangle defined as the current \langle value \rangle. Thus the \langle code \rangle should make use of the \langle tl var \rangle.

The functions $\int_{\text{step variable}}:nNn$ and $\int_{\text{step variable}}:nnNn$ both use a fixed \langle step \rangle of 1, and in the case of $\int_{\text{step variable}}:nNn$ the \langle initial value \rangle is also fixed as 1. These functions are provided as simple short-cuts for code clarity.
21.8 Formatting integers

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

\int_to_arabic:n \{ ⟨int expr⟩ \}
\int_to_arabic:v

Places the value of the \{ ⟨int expr⟩ \} in the input stream as digits, with category code 12 (other).

\int_to_alph:n \{ ⟨int expr⟩ \}
\int_to_Alph:n

Evaluates the \{ ⟨int expr⟩ \} and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

\int_to_alph:n \{ 1 \}
places a in the input stream,

\int_to_alph:n \{ 26 \}
is represented as z and

\int_to_alph:n \{ 27 \}
is converted to aa. For conversions using other alphabets, use \int_to_symbols:nnn to define an alphabet-specific function. The basic \int_to_alph:n and \int_to_Alph:n functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_symbols:nnn
\int_to_symbols:nnn \{(int expr)\} \{(total symbols)\}
\{(value to symbol mapping)\}

This is the low-level function for conversion of an \{ ⟨int expr⟩ \} into a symbolic form (often letters). The \{ total symbols \} available should be given as an integer expression. Values are actually converted to symbols according to the \{ value to symbol mapping \}. This should be given as \{ total symbols \} pairs of entries, a number and the appropriate symbol. Thus the \int_to_alph:n function is defined as

\cs_new:Npn \int_to_alph:n #1
{\int_to_symbols:nnn\{#1\}\{26\} }
\texttt{\textbackslash int\_to\_bin:n \{(int expr)\}} \par \texttt{Nov: 2014-02-11}

Calculates the value of the \texttt{(int expr)} and places the binary representation of the result in the input stream.

\texttt{\textbackslash int\_to\_hex:n \{(int expr)\}} \par \texttt{\textbackslash int\_To\_Hex:n \{(int expr)\}} \par \texttt{Nov: 2014-02-11}

Calculates the value of the \texttt{(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 \texttt{\textbackslash int\_to\_hex:n} and upper case ones for \texttt{\textbackslash int\_to\_Hex:n}. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\texttt{\textbackslash int\_to\_oct:n \{(int expr)\}} \par \texttt{Nov: 2014-02-11}

Calculates the value of the \texttt{(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).

\texttt{\textbackslash int\_to\_base:nn \{(int expr)\} \{\langle base\rangle\}} \par \texttt{\textbackslash int\_To\_Base:nn \{(int expr)\} \{\langle base\rangle\}} \par \texttt{Updated: 2014-02-11}

Calculates the value of the \texttt{(int expr)} and converts it into the appropriate representation in the \texttt{\langle base\rangle}; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for \texttt{\textbackslash int\_to\_base:n} and upper case ones for \texttt{\textbackslash int\_to\_Base:n}. The maximum \texttt{\langle base\rangle} value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\texttt{\textbackslash int\_to\_roman:n} \par \texttt{\textbackslash int\_to\_Roman:n} \par \texttt{Updated: 2011-10-22}

Places the value of the \texttt{(int expr)} in the input stream as Roman numerals, either lower case \texttt{\textbackslash int\_to\_roman:n} or upper case \texttt{\textbackslash 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 \emph{mdclxvi}, repeated as needed: the notation with bars (such as \textbackslash bar{v} for 5000) is not used. For instance \texttt{\textbackslash int\_to\_roman:n} \{8249\} expands to \texttt{mmmmmmmmccxlix}.

\texttt{\textbackslash int\_from\_alph:n} \par \texttt{\textbackslash int\_from\_Alph:n} \par \texttt{Updated: 2014-08-28}

Converts the \texttt{(letters)} into the integer (base 10) representation and leaves this in the input stream. The \texttt{(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 \texttt{+} and \texttt{-}. This is the inverse function of \texttt{\textbackslash int\_to\_alph:n} and \texttt{\textbackslash int\_to\_Alph:n}.

\texttt{\textbackslash int\_from\_bin:n} \par \texttt{\textbackslash int\_from\_Bin:n} \par \texttt{Updated: 2014-08-28}

Converts the \texttt{(binary number)} into the integer (base 10) representation and leaves this in the input stream. The \texttt{(binary number)} is first converted to a string, with no expansion. The function accepts a leading sign, made of \texttt{+} and \texttt{-}, followed by binary digits. This is the inverse function of \texttt{\textbackslash int\_to\_bin:n}.

\textbf{2.19 Converting from other formats to integers}
\int_from_hex:n \{\langle hexadecimal number \rangle\}
Converts the \langle hexadecimal number \rangle into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the \langle hexadecimal number \rangle by upper or lower case letters. The \langle hexadecimal number \rangle is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and -. This is the inverse function of \int_to_hex:n and \int_to_Hex:n.

\int_from_oct:n \{\langle octal number \rangle\}
Converts the \langle octal number \rangle into the integer (base 10) representation and leaves this in the input stream. The \langle octal number \rangle is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by octal digits. This is the inverse function of \int_to_oct:n.

\int_from_roman:n \{\langle roman numeral \rangle\}
Converts the \langle roman numeral \rangle into the integer (base 10) representation and leaves this in the input stream. The \langle roman numeral \rangle is first converted to a string, with no expansion. The \langle roman numeral \rangle may be in upper or lower case; if the numeral contains characters besides mdclxvi or MDCLXVI then the resulting value is -1. This is the inverse function of \int_to_roman:n and \int_to_Roman:n.

\int_from_base:nn \{\langle number \rangle\} \{\langle base \rangle\}
Converts the \langle number \rangle expressed in \langle base \rangle into the appropriate value in base 10. The \langle number \rangle is first converted to a string, with no expansion. The \langle number \rangle should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum \langle base \rangle 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 \{\langle int expr1 \rangle\} \{\langle int expr2 \rangle\}
Evaluates the two \langle int expr \rangle s and produces a pseudo-random number between the two (with bounds included).

\int_rand:n \{\langle int expr \rangle\}
Evaluates the \langle int expr \rangle then produces a pseudo-random number between 1 and the \langle int expr \rangle (included).

21.11 Viewing integers

\int_show:N \langle integer \rangle
\int_show:c
Displays the value of the \langle integer \rangle on the terminal.
\int_show:n \{\text{int expr}\}

Displays the result of evaluating the \{(int expr)\} on the terminal.

\int_log:N \text{integer}
\int_log:c

Writes the value of the \{(integer)\} in the log file.

\int_log:n \{\text{int expr}\}

Writes the result of evaluating the \{(int expr)\} in the log file.

### 21.12 Constant integers

\c_zero_int \c_one_int

Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.

\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 \texttt{\LaTeX3}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_int \g_tmpb_int

Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any \texttt{\LaTeX3}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
21.14 Direct number expansion

\int_value:w \ ⟨integer⟩ \ ⟨integer denotation⟩ ⟨optional space⟩

Expands the following tokens until an ⟨integer⟩ is formed, and leaves a normalized form
(no leading sign except for negative numbers, no leading digit 0 except for zero) in the
input stream as category code 12 (other) characters. The ⟨integer⟩ can consist of any
number of signs (with intervening spaces) followed by

- an integer variable (in fact, any \TeX register except \toks) or
- explicit digits (or by ‘⟨octal digits⟩’ or ‘⟨hexadecimal digits⟩’ or ‘⟨character⟩’).

In this last case expansion stops once a non-digit is found; if that is a space it is removed
as in f-expansion, and so \exp_stop_f: may be employed as an end marker. Note that
protected functions are expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable
for use in cases where a number is required “directly”. In general, \int_eval:n is the
preferred approach to generating numbers.

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

21.15 Primitive conditionals

\if_int_compare:w \ ⟨integer1⟩ \ ⟨relation⟩ \ ⟨integer2⟩
\ ⟨true code⟩
\else:
\ ⟨false code⟩
\fi:

Compare two integers using ⟨relation⟩, which must be one of =, < or > with category
code 12. The \else: branch is optional.

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

\if_case:w \ ⟨integer⟩ \ ⟨case0⟩
\or: \ ⟨case1⟩
\or: ... 
\else: \ ⟨default⟩
\fi:

Selects a case to execute based on the value of the ⟨integer⟩. The first case (⟨case0⟩) is
executed if ⟨integer⟩ is 0, the second (⟨case1⟩) if the ⟨integer⟩ is 1, etc. The ⟨integer⟩
may be a literal, a constant or an integer expression (e.g. using \int_eval:n).

\TeXhackers note: These are the \TeX primitives \ifcase and \or.
\if_int_odd:w \if_int_odd:w (tokens) (optional space)  
(true code)  
\else:  
(true code)  
\fi:  

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

\TeXhackers{} note: This is the \TeX{} primitive \texttt{\textbackslash ifodd}.
Chapter 22

The l3flag module

Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any (small) non-negative value, which we call its \texttt{height}. In expansion-only contexts, a flag can only be “raised”: this increases the \texttt{height} by 1. The \texttt{height} can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by l3str-convert, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height and that the memory cannot be reclaimed even if the flag is cleared. Flags should not be used unless it is unavoidable.

In earlier versions, flags were referenced by an \texttt{n-type \langle flag name \rangle} such as \texttt{fp-overflow}, used as part of \texttt{\use:c} constructions. All of the commands described below have \texttt{n-type} analogues that can still appear in old code, but the \texttt{N-type} commands are to be preferred moving forward. The \texttt{n-type \langle flag name \rangle} is simply mapped to \texttt{\l\langle flag name \rangle\_flag}, which makes it easier for packages using public flags (such as l3fp) to retain backwards compatibility.

22.1 Setting up flags

\begin{verbatim}
\flag_new:N \flag_new:N \langle flag var \rangle
\flag_new:c \langle flag var \rangle
\end{verbatim}

Creates a new \texttt{\langle flag var \rangle}, or raises an error if the name is already taken. The declaration is global, but flags are always local variables. The \texttt{\langle flag var \rangle} initially has zero height.
\flag_clear:N \flag_clear:N (flag var)  \flag_clear:c
Sets the height of the (flag var) to zero. The assignment is local.

\flag_clear_new:N \flag_clear_new:N (flag var)  \flag_clear_new:c
Ensures that the (flag var) exists globally by applying \flag_new:N if necessary, then
applies \flag_clear:N, setting the height to zero locally.

\flag_show:N \flag_show:N (flag var)  \flag_show:c
Displays the height of the (flag var) in the terminal.

\flag_log:N \flag_log:N (flag var)  \flag_log:c
Writes the height of the (flag var) in the log file.

22.2 Expandable flag commands

\flag_if_exist_p:N \flag_if_exist_p:N (flag var)  \flag_if_exist_p:c
\flag_if_exist:NTF \flag_if_exist:NTF (flag var) {true code} {false code}
\flag_if_exist:cTF \flag_if_exist:cTF
This function returns true if the (flag var) is currently defined, and false otherwise.
This does not check that the (flag var) really is a flag variable.

\flag_if_raised_p:N \flag_if_raised_p:N (flag var)  \flag_if_raised_p:c
\flag_if_raised:NTF \flag_if_raised:NTF (flag var) {true code} {false code}
\flag_if_raised:cTF \flag_if_raised:cTF
This function returns true if the (flag var) has non-zero height, and false if the
(flag var) has zero height.

\flag_height:N \flag_height:N (flag var)  \flag_height:c
Expands to the height of the (flag var) as an integer denotation.

\flag_raise:N \flag_raise:N (flag var)  \flag_raise:c
The height of (flag var) is increased by 1 locally.

\flag_ensure_raised:N \flag_ensure_raised:N (flag var)  \flag_ensure_raised:c
Ensures the (flag var) is raised by making its height at least 1, locally.
Scratch flag for local assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
Chapter 23

The \texttt{l3clist} module

Comma separated lists

Comma lists (in short, \texttt{clist}) contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in \texttt{l3seq}) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with \LaTeX or other code that expects or provides items separated by commas.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an \texttt{\ExplSyntaxOn} ... \texttt{\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, b, c~\d, 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 \lshow functions such as \tl_show:N can be applied to them. (These functions often have \clist analogues, which should be preferred.)

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using \seq_set_fromclist:Nn (see \lseq) may be advisable if speed is important. The exception is that \clist_if_in:NnTF and \clist_remove_duplicates:N may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains \{, \} or \# (assuming the usual \TeX{} category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

23.1 Creating and initialising comma lists

\clist_new:N \clist_new:N \langle\clist var\rangle

Creates a new \langle\clist var\rangle or raises an error if the name is already taken. The declaration is global. The \langle\clist var\rangle initially contains no items.

\clist_init:N \clist_init:(Ne|cn|ce) \newline \textit{New: 2014-07-05}

\clist_init:N \clist_init:{\langle\clist var\rangle} \{\langlecomma list\rangle\}

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

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

Clears all items from the \langle\clist var\rangle.

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

Ensures that the \langle\clist 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\clist var_1\rangle \langle\clist var_2\rangle

Sets the content of \langle\clist var_1\rangle equal to that of \langle\clist var_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) \newline \textit{New: 2014-07-17}

Converts the data in the \langle\seq var\rangle into a \langle\clist var\rangle: the original \langle\seq var\rangle is unchanged. Items which contain either spaces or commas are surrounded by braces.
\clist_concat:NNN \clist_concat:ccc \clist_gconcat:NNN \clist_gconcat:ccc
Concatenates the content of \clist var \clist2 and \clist var \clist3 together and saves the result in \clist var \clist1. The items in \clist var \clist2 are placed at the left side of the new comma list.

\clist_concat:NNN \clist_concat:ccc \clist_gconcat:NNN \clist_gconcat:ccc
\clist_if_exist_p:N \clist_if_exist:NTF \clist_if_exist:N \clist_if_exist:c
Tests whether the \clist var is currently defined. This does not check that the \clist var really is a comma list.

New: 2012-03-03

23.2 Adding data to comma lists

\clist_set:Nn \clist_set:N \clist_gset:Nn \clist_gset:N
Sets \clist var to contain the \items, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some \tokens as a single \item even if the \tokens contain commas or spaces, add a set of braces: \clist_set:Nn \clist var \{ \{ \tokens \} \}.

\clist_set:Nn \clist_set:N \clist_gset:Nn \clist_gset:N
\clist_put_left:Nn \clist_put_left:N \clist_gput_left:Nn \clist_gput_left:N
Appends the \items to the left of the \clist var. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some \tokens as a single \item even if the \tokens contain commas or spaces, add a set of braces: \clist_put_left:Nn \clist var \{ \{ \tokens \} \}.

\clist_put_right:Nn \clist_put_right:N \clist_gput_right:Nn \clist_gput_right:N
Appends the \items to the right of the \clist var. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some \tokens as a single \item even if the \tokens contain commas or spaces, add a set of braces: \clist_put_right:Nn \clist var \{ \{ \tokens \} \}.

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_var
\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_var \item
\clist_remove_all:(cn|NV|cV)
\clist_gremove_all:Nn
\clist_gremove_all:(cn|NV|cV)
```

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_var
\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.
23.4 Comma list conditionals

\clist_sort:Nn \clist_sort:cn \clist_gsort:Nn \clist_gsort:cn
\clist_sort:Nn (clist var) \{\langle comparison code \rangle\}
Sorts the items in the \langle clist var \rangle according to the \langle comparison code \rangle, and assigns the result to \langle clist var \rangle. The details of sorting comparison are described in Section 6.1.

\clist_sort:cn \clist_gsort:cn

23.4 Comma list conditionals

\clist_if_empty_p:N * \clist_if_empty_p:c \clist_if_empty:NTF \clist_if_empty:nTF
\clist_if_empty_p:N (clist var)
\clist_if_empty_p:c (clist var) \{\langle true code \rangle\} \{\langle false code \rangle\}
Tests if the \langle clist var \rangle is empty (containing no items).

\clist_if_empty_p:n \clist_if_empty:nTF
\clist_if_empty:p:n \clist_if_empty:p:NTF \clist_if_empty:nTF (comma list) \{\langle true code \rangle\} \{\langle false code \rangle\}
Tests if the \langle clist var \rangle is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list \{~,~,~,\} (without outer braces) is empty, while \{~,{},\} (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

\clist_if_in:NnTF \clist_if_in:NN \clist_if_in:nTF \clist_if_in:nFT
\clist_if_in:NnTF (clist var) \{\langle item \rangle\} \{\langle true code \rangle\} \{\langle false code \rangle\}
\clist_if_in:NN (NV|No|cn|cV|co)T \clist_if_in:nnTF (nV|no)T
Tests if the \langle item \rangle is present in the \langle clist var \rangle. In the case of an n-type \langle comma list \rangle, the usual rules of space trimming and brace stripping apply. Hence,

\clist_if_in:nFT \{ a , \{b\}~ , \{b\} , c \} \{ b \} \{true\} \{false\}
yields true.

\TeX\textbf{hackers note}: The function may fail if the \langle item \rangle contains \{, \}, or # (assuming the usual \TeX\ category codes apply).

23.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, \emph{i.e.} any local assignments made by the \langle function \rangle or \langle code \rangle discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is \{a,\{b\},\{c\}\}, then the arguments passed to the mapped function are ‘a’, ‘\{b\}’, an empty argument, and ‘c’.
When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

\clist_map_function:NN \clist_map_function:cN \clist_map_function:nN \clist_map_function:eN
\clist_map_inline:Nn \clist_map_inline:cn \clist_map_inline:nn
\clist_map_variable:NNn \clist_map_variable:cNn \clist_map_variable:nNn
\clist_map_tokens:Nn \clist_map_tokens:cn \clist_map_tokens:nn
\clist_map_break:

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

\TeX{}hackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.
\clist_map_break:n \clist_map_break:n \clist_map_inline:Nn \l_my_clist
\str_if_eq:nnTF { #1 } { bingo }
{ \clist_map_break:n \clist_map_break:n \langle \text{code} \rangle }
{ % Do something useful }
}

Use outside of a \clist_map_inline:Nn scenario leads to low level \TeX{} errors.

\textbf{\TeX{}hackers note}: When the mapping is broken, additional tokens may be inserted before the \langle \text{code} \rangle is inserted into the input stream. This depends on the design of the mapping function.

\clist_count:N \clist_count:N \clist_count:n \clist_count:e

Leaves the number of items in the \langle \text{clist var} \rangle in the input stream as an \langle \text{integer denotation} \rangle. The total number of items in a \langle \text{clist var} \rangle includes those which are duplicates, \textit{i.e.} every item in a \langle \text{clist var} \rangle is counted.
23.6 Using the content of comma lists directly

\clist_use:Nnnn \clist_use:cnnn

Places the contents of the \clist var in the input stream, with the appropriate \separator between the items. Namely, if the comma list has more than two items, the \separator between more than two is placed between each pair of items except the last, for which the \separator between final two is used. If the comma list has exactly two items, then they are placed in the input stream separated by the \separator between two. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

\TeXhackers note: The result is returned within the \exp_not:n primitive, which means that the \items do not expand further when appearing in an e-type or x-type argument expansion.

\clist_use:Nn \clist_use:cn

Places the contents of the \clist var in the input stream, with the \separator between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { -and- }

inserts “a and b and c and de and f” in the input stream.

\TeXhackers note: The result is returned within the \exp_not:n primitive, which means that the \items do not expand further when appearing in an e-type or x-type argument expansion.
\clist_use:nnnn \clist_use:nn \clist_use:nn \clist_use:nn
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{\TeX{}hackers note:} The result is returned within the \texttt{\unexpanded}\ primitive (\exp_not:n), which means that the \textit{items} do not expand further when appearing in an e-type or 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
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 \texttt{\q_no_value}.

\clist_pop:NN \clist_pop:cN Updated: 2011-09-06
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.

\clist_gpop:NN \clist_gpop:cN
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.

\clist_pop:NNTF \clist_pop:CTF Updated: 2012-05-14
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.
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_gpop:NNTF \clist_gpop:cNTF
\clist_gpop:NNTF \clist_gpop:cNTF
\clist_gpush:Nn\clist_gpush:cN\clist_gpush:NN\clist_gpush:CN
\clist_push:Nn \clist_push:(NV)|No|cn|cV|co
\clist_push:Nn\clist_push:NV\clist_push:NO\clist_push:cn\clist_push:cV
\clist_item:Nn\clist_item:en\clist_item:nn\clist_item:cn\clist_item:co
\clist_rand_item:N\clist_rand_item:n\clist_rand_item:N\clist_rand_item:c
\clist_rand_item:N\clist_rand_item:NN\clist_rand_item:CN\clist_rand_item:NC
\clist_show:N\clist_show:c\clist_show:N\clist_show:CN

23.8 Using a single item

\clist_item:Nn \clist_item:en\clist_item:nn \clist_item:cn \clist_item:co
\clist_item:Nn \clist_item:en \clist_item:nn \clist_item:cn \clist_item:co
\clist_rand_item:N \clist_rand_item:n \clist_rand_item:N \clist_rand_item:c
\clist_rand_item:N \clist_rand_item:NN \clist_rand_item:CN \clist_rand_item:NC
\clist_show:N \clist_show:c \clist_show:N \clist_show:CN

23.9 Viewing comma lists
\clist_show:n \clist_show:n \{\tokens\} \newline Displays the entries in the comma list in the terminal.

\clist_log:N \clist_log:N \clist_log:c \newline \clist_log:N \clist_var \newline Writes the entries in the \clist_var in the log file. See also \clist_show:N which displays the result in the terminal.

\clist_log:n \clist_log:n \clist_log:n \{\tokens\} \newline \clist_log:n \newline 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 \newline Constant that is always empty.

\l_tmpa_clist \l_tmpb_clist \newline 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 \newline Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any \LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
Chapter 24

The \texttt{l3token} module

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let’s try with a better description: When programming in \TeX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: \texttt{\textbackslash \text{token\_}} for anything that deals with tokens and \texttt{\textbackslash \text{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{\textbackslash if:w}, \texttt{\textbackslash if\_charcode:w}, and \texttt{\textbackslash 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{\textbackslash show\_until\_if:w} defined below takes everything until \texttt{\textbackslash if:w} as an argument, despite the presence of other copies of \texttt{\textbackslash 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 24.7.
24.1 Creating character tokens

\char_set_active_eq:NN \char_set_active_eq:Nc \char_gset_active_eq:NN \char_gset_active_eq:Nc

Updated: 2015-11-12

Sets the behaviour of the \langle char \rangle in situations where it is active (category code 13) to be equivalent to that of the definition of the \langle function \rangle at the time \char_set_active_eq:NN is used. The category code of the \langle char \rangle is unchanged by this process. The \langle function \rangle may itself be an active character.

\char_set_active_eq:nN \char_set_active_eq:nc \char_gset_active_eq:nN \char_gset_active_eq:nc

New: 2015-11-12

Sets the behaviour of the \langle char \rangle which has character code as given by the \langle integer expression \rangle in situations where it is active (category code 13) to be equivalent to that of the \langle function \rangle at the time \char_set_active_eq:nN is used. The category code of the \langle char \rangle is unchanged by this process. The \langle function \rangle may itself be an active character.

\char_generate:nn \char_generate:nc

Generates a character token of the given \langle charcode \rangle and \langle catcode \rangle (both of which may be integer expressions). The \langle catcode \rangle 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 \langle charcode \rangle may be any one valid for the engine in use, except that for \langle catcode \rangle 10, \langle charcode \rangle 0 is not allowed. Active characters cannot be generated in older versions of Xe\TeX. Another way to build token lists with unusual category codes is \regex_replace:nnN {.*} {\langle replacement \rangle} {\tl var}.

\TeXhackers note: Exactly two expansions are needed to produce the character.

\c_catcode_active_space_tl

Rev: 2017-08-07

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

\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_letter:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

Sets the category code of the \texttt{character} to that indicated in the function name. Depending on the current category code of the \texttt{token} the escape token may also be needed:

\char_set_catcode_other:N \%

The assignment is local.
Sets the category code of the \texttt{character} which has character code as given by the \texttt{integer expression}. This version can be used to set up characters which cannot otherwise be given (\textit{cf.} the \texttt{N}-type variants). The assignment is local.

These functions set the category code of the \texttt{character} which has character code as given by the \texttt{integer expression}. The first \texttt{integer expression} is the character code and the second is the category code to apply. The setting applies within the current \TeX{} group. In general, the symbolic functions \texttt{char_set_catcode\_\langle type\rangle} should be preferred, but there are cases where these lower-level functions may be useful.

Expands to the current category code of the \texttt{character} with character code given by the \texttt{integer expression}.

Displays the current category code of the \texttt{character} with character code given by the \texttt{integer expression} on the terminal.

Sets up the behaviour of the \texttt{character} when found inside \texttt{text\_lowercase:n}, such that \texttt{character1} will be converted into \texttt{character2}. The two \texttt{characters} may be specified using an \texttt{integer expression} for the character code concerned. This may include the \TeX{} \texttt{\langle character\rangle} method for converting a single character into its character code:

\begin{verbatim}
\char_set_lccode:n { '\A' } { '\a' } \% Standard behaviour
\char_set_lccode:n { '\A' } { '\A + 32' }
\char_set_lccode:n { 50 } { 60 }
\end{verbatim}

The setting applies within the current \TeX{} group.
\texttt{\char_value_lccode:n} \{(\texttt{integer expression})\}

Expands to the current lower case code of the \texttt{\{character\}} with character code given by the \texttt{\{integer expression\}}.

\texttt{\char_show_value_lccode:n} \{(\texttt{integer expression})\}

Displays the current lower case code of the \texttt{\{character\}} with character code given by the \texttt{\{integer expression\}} on the terminal.

\texttt{\char_set_uccode:nn} \{(\texttt{int expr}_1)\} \{(\texttt{int expr}_2)\}

Sets up the behaviour of the \texttt{\{character\}} when found inside \texttt{\text_uppercase:n}, such that \texttt{\{character\}_1} will be converted into \texttt{\{character\}_2}. The two \texttt{\{characters\}} may be specified using an \texttt{\{integer expression\}} for the character code concerned. This may include the \TeX{} \texttt{'\{character\}'} method for converting a single character into its character code:

\begin{verbatim}
\char_set_uccode:nn { \char \a } { \char \A } % Standard behaviour
\char_set_uccode:nn { \char \A } { \char \A - 32 }
\char_set_uccode:nn { 60 } { 50 }
\end{verbatim}

The setting applies within the current \TeX{} group.

\texttt{\char_value_uccode:n} \{(\texttt{integer expression})\}

Expands to the current upper case code of the \texttt{\{character\}} with character code given by the \texttt{\{integer expression\}}.

\texttt{\char_show_value_uccode:n} \{(\texttt{integer expression})\}

Displays the current upper case code of the \texttt{\{character\}} with character code given by the \texttt{\{integer expression\}} on the terminal.

\texttt{\char_set_mathcode:nn} \{(\texttt{int expr}_1)\} \{(\texttt{int expr}_2)\}

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

\texttt{\char_value_mathcode:n} \{(\texttt{integer expression})\}

Expands to the current math code of the \texttt{\{character\}} with character code given by the \texttt{\{integer expression\}}.

\texttt{\char_show_value_mathcode:n} \{(\texttt{integer expression})\}

Displays the current math code of the \texttt{\{character\}} with character code given by the \texttt{\{integer expression\}} on the terminal.

\texttt{\char_set_sfcode:nn} \{(\texttt{int expr}_1)\} \{(\texttt{int expr}_2)\}

This function sets up the space factor for the \texttt{\{character\}}. The \texttt{\{character\}} is specified as an \texttt{\{integer expression\}} which will be used as the character code of the relevant character. The setting applies within the current \TeX{} group.

198
\char_value sfcode:n \char_value sfcode:n \{(integer expression)\}

Expands to the current space factor for the \texttt{character} with character code given by the \texttt{integer expression}.

\char show_value sfcode:n \char show_value sfcode:n \{(integer expression)\}

Displays the current space factor for the \texttt{character} with character code given by the \texttt{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 \texttt{active} (catcode 13). Each entry in the sequence consists of a single escaped token, for example \texttt{\textasciitilde}. 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 \texttt{letter} (catcode 11) or \texttt{other} (catcode 12). Each entry in the sequence consists of a single escaped token, for example \texttt{\textbackslash} for the backslash or \texttt{\{ 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.

\textit{\LaTeX}hackers note: The tokens \texttt{\c group begin token}, \texttt{\c group end token}, and \texttt{\c space token} are expl3 counterparts of \B{\LaTeX}2e’s \texttt{\bgroup}, \texttt{\egroup}, and \texttt{\@sptoken}.

\c catcode letter token \c catcode other token

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

\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

Inserts the current meaning of the \token into the input stream as a series of characters of category code 12 (other). This is the primitive \TeX description of the \token, thus for example both functions defined by \cs_set_nopar:Npn and token list variables defined using \tl_new:N are described as macros.

\TeXhackers note: This is the \TeX primitive \meaning. The \token can thus be an explicit space token or an explicit begin-group or end-group character token ({ or } when normal \TeX category codes apply) even though these are not valid N-type arguments.

\token_to_str:N * \token_to_str:N \token

\token_to_str:c * \token_to_str:c \token

Converts the given \token into a series of characters with category code 12 (other). If the \token is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the \token). This function requires only a single expansion.

\TeXhackers note: \token_to_str:N is the \TeX primitive \string. The \token can thus be an explicit space tokens or an explicit begin-group or end-group character token ({ or } when normal \TeX category codes apply) even though these are not valid N-type arguments.

\token_to_catcode:N * \token_to_catcode:N \token

Converts the given \token into a number describing its category code. If \token is a control sequence this expands to 16. This can’t detect the categories 0 (escape character), 5 (end of line), 9 (ignored character), 14 (comment character), or 15 (invalid character). Control sequences or active characters let to a token of one of the detectable category codes will yield that category.

24.5 Token conditionals

\token_if_group_begin_p:N * \token_if_group_begin_p:N \token

\token_if_group_begin:NTF * \token_if_group_begin:NTF \token \{\true\} \{\false\}

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

\token_if_group_end_p:N * \token_if_group_end_p:N \token

\token_if_group_end:NTF * \token_if_group_end:NTF \token \{\true\} \{\false\}

Tests if \token has the category code of an end group token (} when normal \TeX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.
\token_if_math_toggle_p:N \token_if_math_toggle_p:N {token} \token_if_math_toggle:NTF \token_if_math_toggle:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of a math shift token ($ when normal TeX category codes are in force).

\token_if_alignment_p:N \token_if_alignment_p:N {token} \token_if_alignment:NTF \token_if_alignment:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of an alignment token (& when normal TeX category codes are in force).

\token_if_parameter_p:N \token_if_parameter_p:N {token} \token_if_parameter:NTF \token_if_parameter:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of a macro parameter token (# when normal TeX category codes are in force).

\token_if_math_superscript_p:N \token_if_math_superscript_p:N {token} \token_if_math_superscript:NTF \token_if_math_superscript:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of a superscript token (^ when normal TeX category codes are in force).

\token_if_math_subscript_p:N \token_if_math_subscript_p:N {token} \token_if_math_subscript:NTF \token_if_math_subscript:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of a subscript token (_ when normal TeX category codes are in force).

\token_if_space_p:N \token_if_space_p:N {token} \token_if_space:NTF \token_if_space:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

\token_if_letter_p:N \token_if_letter_p:N {token} \token_if_letter:NTF \token_if_letter:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of a letter token.

\token_if_other_p:N \token_if_other_p:N {token} \token_if_other:NTF \token_if_other:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of an “other” token.

\token_if_active_p:N \token_if_active_p:N {token} \token_if_active:NTF \token_if_active:NTF {token} \{true code\} \{false code\}
Tests if \langle token \rangle has the category code of an active character.

\token_if_eq_catcode_p:NN \token_if_eq_catcode_p:NN {token_1} \{token_2\} \token_if_eq_catcode:NTF \token_if_eq_catcode:NTF {token_1} \{token_2\} \{true code\} \{false code\}
Tests if the two \langle tokens \rangle have the same category code.
\token_if_eq_charcode_p:NN \ token_if_eq_charcode_p:NN \ (token_1) \ (token_2)
\token_if_eq_charcode:NNTF \ token_if_eq_charcode:NNTF \ (token_1) \ (token_2) \ {true code} \ {false code}

Tests if the two \textit{tokens} have the same character code.

\texttt{\token_if_eq_meaning_p:NN} \ \texttt{\token_if_eq_meaning_p:NN}
\token_if_eq_meaning:NNTF \ (token_1) \ (token_2) \ {true code} \ {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:NTF \ (token) \ {true code} \ {false code}

Tests if the \textit{token} is a \LaTeX{} macro.

\token_if_cs_p:N \ \token_if_cs_p:N
\token_if_cs:NTF \ (token) \ {true code} \ {false code}

Tests if the \textit{token} is a control sequence.

\token_if_expandable_p:N \ \token_if_expandable_p:N
\token_if_expandable:NTF \ (token) \ {true code} \ {false code}

Tests if the \textit{token} is expandable. This test returns \texttt{false} for an undefined token.

\token_if_long_macro_p:N \ \token_if_long_macro_p:N
\token_if_long_macro:NTF \ (token) \ {true code} \ {false code}

Tests if the \textit{token} is a long macro.

\token_if_protected_macro_p:N \ \token_if_protected_macro_p:N
\token_if_protected_macro:NTF \ (token) \ {true code} \ {false code}

Tests if the \textit{token} is a protected macro: for a macro which is both protected and long this returns \texttt{false}.

\token_if_protected_long_macro_p:N \ \token_if_protected_long_macro_p:N
\token_if_protected_long_macro:NTF \ (token) \ {true code} \ {false code}

Tests if the \textit{token} is a protected long macro.

\token_if_chardef_p:N \ \token_if_chardef_p:N
\token_if_chardef:NTF \ (token) \ {true code} \ {false code}

Tests if the \textit{token} is defined to be a chardef.

\textbf{\LaTeX{} hackers note:} Booleans, boxes and small integer constants are implemented as \texttt{chardefs}.
\token_if_mathchardef_p:N \token_if_mathchardef_p:N \token_if_font_selection_p:N \token_if_font_selection_p:N \token_if_dim_register_p:N \token_if_dim_register_p:N \token_if_int_register_p:N \token_if_int_register_p:N \token_if_muskip_register_p:N \token_if_muskip_register_p:N \token_if_skip_register_p:N \token_if_skip_register_p:N

Tests if the \textit{token} is defined to be a mathchardef.

Updated: 2012-01-20

\token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF

Tests if the \textit{token} is defined to be a font selection command.

New: 2020-10-27

\token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF

Tests if the \textit{token} is defined to be a dimension register.

Updated: 2012-01-20

\token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF

Tests if the \textit{token} is defined to be an integer register.

Updated: 2012-01-20

\token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF

Tests if the \textit{token} is defined to be a muskip register.

New: 2012-02-15

\token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF

Tests if the \textit{token} is defined to be a skip register.

Updated: 2012-01-20

\token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF

Tests if the \textit{token} is defined to be a toks register (not used by \LaTeX{}3).

Updated: 2012-01-20

\token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF

Tests if the \textit{token} is defined to be a primitive. In \LuaTeX{} this includes primitive-like commands defined using \token.set_lua.

\texhacksnote{Constant integers may be implemented as integer registers, \texttt{\chardefs{}}, or \texttt{\mathchardefs{}s} depending on their value.}

\texttt{\token_if_mathchardef_p:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF \token_if_mathchardef_p:N \token_if_mathchardef:N \token_if_mathchardef:NTF}

Updated: 2012-01-20
This function compares the \langle \text{test token} \rangle in turn with each of the \langle \text{token cases} \rangle. If the two are equal (as described for \text{token_if_eq_catcode:NNTF}, \text{token_if_eq_charcode:NNTF} and \text{token_if_eq_meaning:NNTF}, respectively) then the associated \langle \text{code} \rangle is left in the input stream and other cases are discarded. If any of the cases are matched, the \langle \text{true code} \rangle is also inserted into the input stream (after the code for the appropriate case), while if none match then the \langle \text{false code} \rangle is inserted. The functions \text{token_case_catcode:Nn}, \text{token_case_charcode:Nn}, and \text{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 \text{peek_after:Nw} is provided along with a family of predefined tests for common cases. Peeking ahead does not skip spaces: rather, \text{peek_remove_spaces:n} should be used. In addition, using \text{peek_analysis_map_inline:n}, one can map through the following tokens in the input stream and repeatedly perform some tests.

\text{\textbackslash peek\string} \text{\string after:Nw} \langle \text{function} \rangle \langle \text{token} \rangle

Locally sets the test variable \text{l_peek_token} equal to \langle \text{token} \rangle (as an implicit token, not as a token list), and then expands the \langle \text{function} \rangle. The \langle \text{token} \rangle remains in the input stream as the next item after the \langle \text{function} \rangle. The \langle \text{token} \rangle here may be \textunderscore, \{ or \} (assuming normal \TeX{} category codes), i.e. it is not necessarily the next argument which would be grabbed by a normal function.

\text{\textbackslash peek\string} \text{\string after:Nw} \langle \text{function} \rangle \langle \text{token} \rangle

Globally sets the test variable \text{g_peek_token} equal to \langle \text{token} \rangle (as an implicit token, not as a token list), and then expands the \langle \text{function} \rangle. The \langle \text{token} \rangle remains in the input stream as the next item after the \langle \text{function} \rangle. The \langle \text{token} \rangle here may be \textunderscore, \{ or \} (assuming normal \TeX{} category codes), i.e. it is not necessarily the next argument which would be grabbed by a normal function.

\text{l_peek_token} Token set by \text{peek_after:Nw} and available for testing as described above.

\text{g_peek_token} Token set by \text{peek_after:Nw} and available for testing as described above.
\peek_catcode:NTF \peek_catcode:NTF \token (test token) \{(true code)\} \{(false code)\}

Tests if the next \token in the input stream has the same category code as the \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:NTF \peek_catcode_remove:NTF \token (test token) \{(true code)\} \{(false code)\}

Tests if the next \token in the input stream has the same category code as the \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:NTF \peek_charcode:NTF \token (test token) \{(true code)\} \{(false code)\}

Tests if the next \token in the input stream has the same character code as the \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:NTF \peek_charcode_remove:NTF \token (test token) \{(true code)\} \{(false code)\}

Tests if the next \token in the input stream has the same character code as the \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:NTF \peek_meaning:NTF \token (test token) \{(true code)\} \{(false code)\}

Tests if the next \token in the input stream has the same meaning as the \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:NTF \peek_meaning_remove:NTF \token (test token) \{(true code)\} \{(false code)\}

Tests if the next \token in the input stream has the same meaning as the \token (as defined by the test \token_if_eq_meaning:NNTF). Spaces are respected by the test and the \token is removed from the input stream if the test is true. The function then places either the \true code or \false code in the input stream (as appropriate to the result of the test).

\peek_remove_spaces:n \peek_remove_spaces:n \code

Peeks ahead and detect if the following token is a space (category code 10 and character code 32). If so, removes the token and checks the next token. Once a non-space token is found, the \code will be inserted into the input stream. Typically this will contain a peek operation, but this is not required.
\peek_remove_filler:n \peek_remove_filler:n \{\langle code\rangle\}

New: 2022-01-10

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to \scan_stop:. If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using 1-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 code\rangle will be inserted into the input stream. Typically this will contain a peek operation, but this is not required.

\TeXhackers note: This is essentially a macro-based implementation of how \TeX handles the search for a left brace after for example \everypar, except that any non-expandable token cleanly ends the \langle filler\rangle (i.e. it does not lead to a \TeX error).

In contrast to \TeX’s filler removal, a construct \exp_not:N \foo will be treated in the same way as \foo.

\peek_N_type:TF \peek_N_type:TF \{\langle true code\rangle\} \{\langle false code\rangle\}

Updated: 2012-12-20

Tests if the next \langle token\rangle in the input stream can be safely grabbed as an \n-type argument. The test is \langle false\rangle if the next \langle 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 \LaTeXx) and \langle true\rangle in all other cases. Note that a \langle true\rangle result ensures that the next \langle token\rangle is a valid \n-type argument. However, if the next \langle token\rangle is for instance \c_space_token, the test takes the \langle false\rangle branch, even though the next \langle token\rangle is in fact a valid \n-type argument. The \langle token\rangle is left in the input stream after the \langle true code\rangle or \langle false code\rangle (as appropriate to the result of the test).
Repeatedly removes one \textit{token} from the input stream and applies the \textit{inline function} to it, until \texttt{\peek_analysis_map_break} is called. The \textit{inline function} receives three arguments for each \textit{token} in the input stream:

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

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

- \textit{catcode}, a capital hexadecimal digit which denotes the category code of the \textit{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{\char\textit{catcode}}.

These arguments are the same as for \texttt{\tl_analysis_map_inline:nn} defined in \texttt{l3tl-analysis}. The \textit{char code} and \textit{catcode} do not take the meaning of a control sequence or active character into account: for instance, upon encountering the \textit{token} \texttt{\c_group-begin_token} in the input stream, \texttt{\peek_analysis_map_inline:n} calls the \textit{inline function} with \texttt{#1} being \texttt{\exp_not:n \{ \c_group_begin_token \}} (with the current implementation), \texttt{#2} being \texttt{-1}, and \texttt{#3} being \texttt{0}, as for any other control sequence. In contrast, upon encountering an explicit begin-group token \texttt{\{}, the \textit{inline function} is called with arguments \texttt{\exp_after:wN \{ \if_false: \fi: \}} 123 and 1.

The mapping is done at the current group level, \textit{i.e.} any local assignments made by the \textit{inline function} remain in effect after the loop. Within the code, \texttt{\l_peek_token} is set equal (as a token, not a token list) to the token under consideration.

Peek functions cannot be used within this mapping function (nor other mapping functions) since the input stream contains trailing material necessary for the functioning of the loop.

\textbf{TeXhacker note:} In case the input stream has not yet been tokenized (converted from characters to tokens), characters are tokenized one by one as needed by \texttt{\peek_analysis_map_inline:n} using the current category code regime.

\texttt{\peek_analysis_map_break:n \peek_analysis_map_break:n \{ \ldots \} \peek_analysis_map_break:n \{\textit{code}\}}

Stops the \texttt{\peek_analysis_map_inline:n} loop from seeking more tokens, and inserts \texttt{\textit{code}} in the input stream (empty for \texttt{\peek_analysis_map_break:n}).
\peek_regex:nTF \peek_regex:NTF \peek_regex:NTF
\peek_regex:nTF \peek_regex:NTF \peek_regex:NTF
New: 2020-12-03

Tests if the \langle tokens \rangle that follow in the input stream match the \langle regular expression \rangle. Any \langle tokens \rangle that have been read are left in the input stream after the \langle true code \rangle or \langle false code \rangle (as appropriate to the result of the test). See \texttt{l3regex} for documentation of the syntax of regular expressions. The \langle regular expression \rangle is implicitly anchored at the start, so for instance \texttt{\peek_regex:nTF \{ a \} } is essentially equivalent to \texttt{\peek_\-charcode:NTF a}.

\textbf{\texttt{\textsc{\texttt{\textup{TeX}}}}hackers note:} Implicit character tokens are correctly considered by \texttt{\peek_regex:nTF} as control sequences, while functions that inspect individual tokens (for instance \texttt{\peek_\-charcode:NTF}) only take into account their meaning.

The \texttt{\peek_regex:nTF} function only inspects as few tokens as necessary to determine whether the regular expression matches. For instance \texttt{\peek_regex:nTF \{ abc | [a-z] \} \{ \} \{ \} abc} will only inspect the first token \texttt{a} even though the first branch \texttt{abc} of the alternative is preferred in functions such as \texttt{\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.

\texttt{\peek_regex_remove_once:nTF} \peek_regex_remove_once:nTF \texttt{\{regex\} \{\true code\} \{false code\}}
\peek_regex_remove_once:nTF \peek_regex_remove_once:nTF \texttt{\{\true code\} \{false code\}}
New: 2020-12-03

Tests if the \langle tokens \rangle that follow in the input stream match the \langle regex \rangle. If the test is true, the \langle tokens \rangle are removed from the input stream and the \langle true code \rangle is inserted, while if the test is false, the \langle false code \rangle is inserted followed by the \langle tokens \rangle that were originally in the input stream. See \texttt{l3regex} for documentation of the syntax of regular expressions. The \langle regular expression \rangle is implicitly anchored at the start, so for instance \texttt{\peek_regex_remove_once:nTF \{ a \} } is essentially equivalent to \texttt{\peek_\-charcode_remove:NTF a}.

\textbf{\texttt{\textsc{\texttt{\textup{TeX}}}}hackers note:} Implicit character tokens are correctly considered by \texttt{\peek_regex_remove_once:nTF} as control sequences, while functions that inspect individual tokens (for instance \texttt{\peek_charcode:NTF}) only take into account their meaning.

208
If the ⟨tokens⟩ that follow in the input stream match the ⟨regex⟩, replaces them according to the ⟨replacement⟩ as for \regex_replace_once:nn, and leaves the result in the input stream, after the ⟨true code⟩. Otherwise, leaves ⟨false code⟩ followed by the ⟨tokens⟩ that were originally in the input stream, with no modifications. See \l3regex for documentation of the syntax of regular expressions and of the ⟨replacement⟩: for instance \0 in the ⟨replacement⟩ is replaced by the tokens that were matched in the input stream. The ⟨regular expression⟩ is implicitly anchored at the start. In contrast to \regex_replace_once:nn, no error arises if the ⟨replacement⟩ leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

\TeXhacker{note}: Implicit character tokens are correctly considered by \peek_regex_replace_once:nn 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 \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 ⟨token⟩ is expandable) results in an internal token, displayed (temporarily) as \notexpanded: ⟨token⟩, whose shape coincides with the ⟨token⟩ and whose meaning differs from \relax.
• An `\texttt{outer endtemplate}`: can be encountered when peeking ahead at the next token; this expands to another internal token, `\texttt{end of alignment template}`.

• Tricky programming might access a frozen `\texttt{endwrite}`.

• Some frozen tokens can only be accessed in interactive sessions: `\texttt{cr, right, endgroup, fi, inaccessible}`.

• In LuaTeX, there is also the strange case of “bytes” 1100xy where \(x, y\) are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from 110000 = 1114112 to 1100ff = 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 (not characters) whose `\texttt{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{token_to_str:N or token_to_meaning:N}` to these tokens is unsafe. Unfortunately, they don’t seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens explicit character tokens. Category codes that a character token can have are listed below by giving a sample output of the \TeX primitive `\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 `&`),
5. macro parameter character (`\texttt{parameter}`, often `#`),
6. superscript character (`\texttt{math_superscript}`, often `^`),
7. subscript character (`\texttt{math_subscript}`, often `_`),
8. blank space (`\texttt{space}`, often character code 32),
9. the letter (`\texttt{letter}`, such as `A`),
10. 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 implicit character tokens. The meaning is otherwise in the following list:

• a macro, used in \LATEX for most functions and some variables (\texttt{tl}, \texttt{fp}, \texttt{seq}, ...),
• a primitive such as `\texttt{def or topmark}`, used in \LATEX for some functions,
• a register such as `\texttt{count123}`, used in \LATEX for the implementation of some variables (\texttt{int, dim, ...}).
- a constant integer such as \texttt{\char"56} or \texttt{\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

\texttt{\langle prefix\rangle macro:\langle argument\rangle ->\langle replacement\rangle}

where \texttt{(prefix)} is among \texttt{\protected\long\outer}, \texttt{(argument)} describes parameters that the macro expects, such as \texttt{#1#2#3}, and \texttt{(replacement)} 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 \texttt{10} (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code \texttt{32} and category code \texttt{10}.

When a macro takes an undelimited argument, explicit space characters (with character code \texttt{32} and category code \texttt{10}) are ignored. If the following token is an explicit character token with category code \texttt{1} (begin-group) and an arbitrary character code, then \TeX{} scans ahead to obtain an equal number of explicit character tokens with category code \texttt{1} (begin-group) and \texttt{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 :\texttt{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 \textit{not} ignored in this case (and they prevent brace-stripping).
Chapter 25

The \texttt{l3prop} module

Property lists

\texttt{expl3} implements a “property list” data type, which contain an unordered list of entries each of which consists of a \langle \texttt{key} \rangle \ (string) and an associated \langle \texttt{value} \rangle \ (token list). The \langle \texttt{key} \rangle \ and \langle \texttt{value} \rangle \ may both be given as any balanced text, and the \langle \texttt{key} \rangle \ is processed using \texttt{\tl_to_str:n}, meaning that category codes are ignored. Entries can be manipulated individually, as well as collectively by applying a function to every key–value pair within the list.

Each entry in a property list must have a unique \langle \texttt{key} \rangle; if an entry is added to a property list which already contains the \langle \texttt{key} \rangle then the new entry overwrites the existing one. The \langle \texttt{keys} \rangle \ 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. They can be converted from and to key–value lists, which are a form of \texttt{input} parsed by the \texttt{l3keys} module. If a key–value list contains a \langle \texttt{key} \rangle \ multiple times, only the last \langle \texttt{value} \rangle \ associated to it will be kept in the conversion to a property list.

Internally, property lists can use two distinct implementations with different data storage, which are decided when declaring the property list variable using \texttt{\prop_new:N} \ (“flat” storage) or \texttt{\prop_new_linked:N} \ (“linked” storage). After a property list is declared with \texttt{\prop_new:N} \ or \texttt{\prop_new_linked:N}, the type of internal data storage can be changed by \texttt{\prop_make_flat:N} \ or \texttt{\prop_make_linked:N}, but only at the outermost group level. All other \texttt{l3prop} functions transparently manipulate either storage method and convert as needed.

- The (default) “flat” storage method is suited for a relatively small number of entries, or when the property list is likely to be manipulated (copied, mapped) as a whole rather than entry-wise. It is significantly faster for \texttt{\prop_set_eq:NN}, and only slightly faster for \texttt{\prop_clear:N}, \texttt{\prop_concat:NNN}, and mapping functions \texttt{\prop_map_...}.

- The “linked” storage method is meant for property lists with a large numbers of entries. It takes up more of \TeX{}’s memory during a run, but is significantly faster (for long lists) when accessing or modifying individual entries using functions such as \texttt{\prop_if_in:Nn}, \texttt{\prop_item:Nn}, \texttt{\prop_put:Nnn}, \texttt{\prop_get:NnN}, \texttt{\prop_pop:NnN}, \texttt{\prop_remove:Nn}, as it takes a constant time for these operations (rather
than the number of items for a “flat” property list). A technical drawback is that memory is permanently used\(^7\) by (keys) stored in a “linked” property list, even after they are removed and the property list is deleted.

### 25.1 Creating and initialising property lists

\begin{verbatim}
\prop_new:N \prop_new:N \prop_new:c
\prop_new_linked:N \prop_new_linked:N \prop_new_linked:c
\prop_clear:N \prop_clear:N \prop_clear:c \prop_gclear:N \prop_gclear:c
\prop_set_eq:NN \prop_set_eq:NN \prop_gset_eq:NN \prop_gset_eq:NN
\prop_clear_new:N \prop_clear_new:N \prop_gclear_new:N \prop_gclear_new:N \prop_clear_new_linked:N \prop_clear_new_linked:c \prop_gclear_new_linked:N \prop_gclear_new_linked:c
\end{verbatim}

\begin{itemize}
\item \texttt{\prop_new:N} \texttt{\prop_new:N} \texttt{\prop_new:c}
\begin{itemize}
\item Creates a new “flat” \texttt{(property list)} or raises an error if the name is already taken.
\item The declaration is global. The \texttt{(property list)} initially contains no entries. See also \texttt{\prop_new_linked:N}.
\end{itemize}
\item \texttt{\prop_new_linked:N} \texttt{\prop_new_linked:N} \texttt{\prop_new_linked:c}
\begin{itemize}
\item Creates a new “linked” \texttt{(property list)} or raises an error if the name is already taken.
\item The declaration is global. The \texttt{(property list)} initially contains no entries. The internal data storage differs from that produced by \texttt{\prop_new:N} and it is optimized for property lists with a large number of entries.
\end{itemize}
\item \texttt{\prop_clear:N} \texttt{\prop_clear:N} \texttt{\prop_clear:c} \texttt{\prop_gclear:N} \texttt{\prop_gclear:c}
\begin{itemize}
\item Clears all entries from the \texttt{(property list)}.
\end{itemize}
\item \texttt{\prop_clear_new:N} \texttt{\prop_clear_new:N} \texttt{\prop_gclear_new:N} \texttt{\prop_gclear_new:c}
\begin{itemize}
\item Ensures that the \texttt{(property list)} exists globally by applying \texttt{\prop_new:N} if necessary, then applies \texttt{\prop_(g)clear:N} to leave the list empty.
\item \texttt{\prop_clear_new_linked:N} \texttt{\prop_clear_new_linked:c} \texttt{\prop_gclear_new_linked:N} \texttt{\prop_gclear_new_linked:c}
\begin{itemize}
\item Ensures that the \texttt{(property list)} exists globally by applying \texttt{\prop_new_linked:N} if necessary, then applies \texttt{\prop_(g)clear:N} to leave the list empty.
\end{itemize}
\end{itemize}

**\texttt{\prop_set_eq:NN}** \texttt{\prop_set_eq:NN} \texttt{\prop_gset_eq:NN} \texttt{\prop_gset_eq:(cN|Nc|cc)}
\begin{itemize}
\item Sets the content of \texttt{(property list\textsubscript{1})} equal to that of \texttt{(property list\textsubscript{2})}. This converts as needed between the two storage types.
\end{itemize}

\(^7\)Until the end of the run, that is.

213
Sets \textit{(property list)} to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept. In contrast to most keyval lists (e.g. those in \texttt{l3keys}), each key here must be followed with an = sign even to specify an empty \textit{(value)}.

Spaces are trimmed around every \textit{(key)} and every \textit{(value)}, and if the result of trimming spaces consists of a single brace group then a set of outer braces is removed. The \textit{(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.

Creates a new constant “flat” \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.

Creates a new constant “linked” \textit{(prop var)} or raises an error if the name is already taken. The \textit{(prop var)} 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.

Changes the internal storage type of the \textit{(property list)} to be the same “flat” storage as \texttt{\prop_new:N}. The key–value pairs of the \textit{(property list)} are preserved by the change. If the property list was already flat then nothing is done. This function can only be used at the outermost group level.

Changes the internal storage type of the \textit{(property list)} to be the same “linked” storage as \texttt{\prop_new_linked:N}. The key–value pairs of the \textit{(property list)} are preserved by the change. If the property list was already linked then nothing is done. This function can only be used at the outermost group level.
25.2 Adding and updating property list entries

\prop_put:Nnn \prop_put: \prop_gput:Nnn

Add a new 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_put_if_not_in:Nnn \prop_put_if_not_in: \prop_gput_if_not_in:Nnn

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}.

\prop_concat:NNN \prop_concat:ccc \prop_gconcat:NNN \prop_gconcat:ccc

Combines the key–value pairs of \texttt{property list} and \texttt{property list3}, and saves the result in \texttt{property list1}. If a key appears in both \texttt{property list2} and \texttt{property list3} then the last value, namely the value in \texttt{property list3} is kept. This converts as needed between the two storage types.
Updates the \emph{(property list)} by adding entries for each key–value pair given in the second argument. The addition is done through \texttt{\prop_put:Nnn}, hence if the \emph{(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 \texttt{\prop_set_from_keyval:Nn}, then combining \emph{(property list)} with the temporary variable using \texttt{\prop_concat:NNN}. In particular, the \emph{(keys)} and \emph{(values)} are space-trimmed and unbraced as described in \texttt{\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

Recover the \emph{(value)} stored with \emph{(key)} from the \emph{(property list)}, and places this in the \emph{(token list variable)}. If the \emph{(key)} is not found in the \emph{(property list)} then the \emph{(token list variable)} is set to the special marker \texttt{\q_no_value}. The \emph{(token list variable)} is set within the current \TeX{} group. See also \texttt{\prop_get:NnNTF}. 

Recover the \emph{(value)} stored with \emph{(key)} from the \emph{(property list)}, and places this in the \emph{(token list variable)}. If the \emph{(key)} is not found in the \emph{(property list)} then the \emph{(token list variable)} is set to the special marker \texttt{\q_no_value}. The \emph{(key)} and \emph{(value)} are then deleted from the property list. Both assignments are local. See also \texttt{\prop_pop:NnNTF}. 

Recover the \emph{(value)} stored with \emph{(key)} from the \emph{(property list)}, and places this in the \emph{(token list variable)}. If the \emph{(key)} is not found in the \emph{(property list)} then the \emph{(token list variable)} is set to the special marker \texttt{\q_no_value}. The \emph{(key)} and \emph{(value)} are then deleted from the property list. The \emph{(property list)} is modified globally, while the assignment of the \emph{(token list variable)} is local. See also \texttt{\prop_gpop:NnNTF}. 

216
Expands to the ⟨value⟩ corresponding to the ⟨key⟩ in the ⟨property list⟩. If the ⟨key⟩ is missing, this has an empty expansion.

**TeXhackers note:** For “flat” property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on \prop_get:NnN. (For “linked” property lists both functions are fast.)

The result is returned within the \unexpanded primitive ⟨\exp_not:n⟩, which means that the ⟨value⟩ does not expand further when appearing in an e-type or x-type argument expansion.

Leaves the number of key–value pairs in the ⟨property list⟩ in the input stream as an ⟨integer denotation⟩.

Expands to the ⟨property list⟩ in a key–value notation. Keep in mind that a ⟨property list⟩ is unordered, while key–value interfaces are not necessarily, so this cannot be used for arbitrary interfaces.

**TeXhackers note:** The result is returned within the \unexpanded primitive ⟨\exp_not:n⟩, which means that the key–value list does not expand further when appearing in an e-type or x-type argument expansion. It also needs exactly two steps of expansion.

Removes the entry listed under ⟨key⟩ from the ⟨property list⟩. If the ⟨key⟩ is not found in the ⟨property list⟩ no change occurs, i.e there is no need to test for the existence of a key before deleting it.

Tests whether the ⟨property list⟩ is currently defined. This does not check that the ⟨property list⟩ really is a property list variable.
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 code follows depending on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

If the ⟨key⟩ is not present in the ⟨property list⟩, leaves the ⟨false code⟩ in the input stream. The value of the ⟨token list variable⟩ is not defined in this case and should not be relied upon. If the ⟨key⟩ is present in the ⟨property list⟩, stores the corresponding ⟨value⟩ in the ⟨token list variable⟩ without removing it from the ⟨property list⟩, then leaves the ⟨true code⟩ in the input stream. The ⟨token list variable⟩ is assigned locally.
If the \langle key \rangle is not present in the \langle property list \rangle, leaves the \langle false code \rangle in the input stream. The value of the \langle token list variable \rangle is not defined in this case and should not be relied upon. If the \langle key \rangle is present in the \langle property list \rangle, pops the corresponding \langle value \rangle in the \langle token list variable \rangle, i.e. removes the item from the \langle property list \rangle. The \langle property list \rangle is modified globally, while the \langle token list variable \rangle is assigned locally.

25.7 Mapping over property lists

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

\prop_map_function:NN \prop_map_function:cn
\prop_map_inline:Nn \prop_map_inline:cn
\prop_map_tokens:Nn \prop_map_tokens:cn

expands to the value corresponding to mykey: for each pair in \l_my_prop the function \\str_if_eq:nnT receives mykey, the \langle key \rangle and the \langle value \rangle as its three arguments. For that specific task, \prop_item:Nn is faster.
\prop_map_break: Used to terminate a \prop_map_... function before all entries in the \textit{property list} have been processed. This normally takes place within a conditional statement, for example

\begin{verbatim}
\prop_map_inline:Nn \l_my_prop
{\str_if_eq:nnTF { #1 } { bingo }
{ \prop_map_break: }
{ % Do something useful
}
}
\end{verbatim}

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

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

\prop_map_break:n \begin{verbatim}
\prop_map_break:n \{ \textit{code} \}
\end{verbatim}

Used to terminate a \prop_map_... function before all entries in the \textit{property list} have been processed, inserting the \textit{code} after the mapping has ended. This normally takes place within a conditional statement, for example

\begin{verbatim}
\prop_map_inline:Nn \l_my_prop
{\str_if_eq:nnTF { #1 } { bingo }
{ \prop_map_break:n \{ \textit{code} \} }
{ % Do something useful
}
}
\end{verbatim}

Use outside of a \prop_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{code} 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 \textit{property list} in the terminal, and specifies its storage type.
25.9 Scratch property lists

There is no need to include both flat and linked property lists as scratch variables. We arbitrarily pick the older implementation.

\l_tmpa_prop \l_tmpb_prop

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

\g_tmpa_prop \g_tmpb_prop

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

25.10 Constants

\c_empty_prop

A permanently-empty property list used for internal comparisons.
Chapter 26

The \texttt{l3skip} module

Dimensions and skips

\LaTeXe{}X3 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:C
\dim_new:N \dim_new:C
\end{verbatim}

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.

\begin{verbatim}
\dim_const:Nn \dim_const:cn
\dim_const:Nn \dim_const:cn\ New: 2012-03-05
\end{verbatim}

Creates a new constant \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 the \langle \texttt{dim expr} \rangle.

\begin{verbatim}
\dim_zero:N \dim_zero:c \dim_zero:N \dim_zero:c
\end{verbatim}

Sets \langle \texttt{dimension} \rangle to 0 pt.

\begin{verbatim}
\dim_zero_new:N \dim_zero_new:C \dim_zero_new:N \dim_zero_new:C\ New: 2012-01-07
\end{verbatim}

Ensures that the \langle \texttt{dimension} \rangle exists globally by applying \texttt{dim_new:N} if necessary, then applies \texttt{dim\_}(g)\texttt{zero:N} to leave the \langle \texttt{dimension} \rangle set to zero.
26.2 Setting dim variables

\dim_add:Nn  \dim_add:N (\dim expr)
\dim_gadd:Nn \dim_gadd:cn

Adds the result of the \dim expr to the current content of the \dim. Updated: 2011-10-22

\dim_set:Nn  \dim_set:N (\dim expr)
\dim_gset:Nn \dim_gset:cn

Sets \dim to the value of \dim expr, which must evaluate to a length with units. Updated: 2011-10-22

\dim_set_eq:NN  \dim_set_eq:NN \dim_1 \dim_2
\dim_set_eq:cc  \dim_set_eq:cc (cN|Nc|cc)
\dim_gset_eq:NN \dim_gset_eq:cc (cN|Nc|cc)

Sets the content of \dim_1 equal to that of \dim_2.

\dim_sub:Nn \dim_sub:N (\dim expr)
\dim_gsub:Nn \dim_gsub:cn

Subtracts the result of the \dim expr from the current content of the \dim. Updated: 2011-10-22

26.3 Utilities for dimension calculations

\dim_abs:n  \dim_abs:n (\dim expr)

Converts the \dim expr to its absolute value, leaving the result in the input stream as a \dim denotation. Updated: 2012-09-26

\dim_max:nn  \dim_max:nn (\dim expr_1) (\dim expr_2)
\dim_min:nn  \dim_min:nn (\dim expr_1) (\dim expr_2)

Evaluates the two \dim exprs and leaves either the maximum or minimum value in the input stream as appropriate, as a \dim denotation. Updated: 2012-09-26
\dim_ratio:nn \((\text{dim expr}_1)\) \((\text{dim expr}_2)\)

Parses the two \((\text{dim exprs})\) and converts the ratio of the two to a form suitable for use inside a \((\text{dim expr})\). This ratio is then left in the input stream, allowing syntax such as

\[
\dim_set:Nn \l_my_dim
{ 10 \text{ pt} \ast \dim_ratio:nn{5 \text{ pt}}{10 \text{ pt}}}
\]

The output of \dim_ratio:nn on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

\[
\tl_set:Ne \l_my_tl \{ \dim_ratio:nn{5 \text{ pt}}{10 \text{ pt}} \}
\tl_show:N \l_my_tl
\]

displays \(327680/655360\) on the terminal.

### 26.4 Dimension expression conditionals

\dim_compare_p:nNn \((\text{dim expr}_1)\) \((\text{relation})\) \((\text{dim expr}_2)\)

\dim_compare:nNnTF

\((\text{dim expr}_1)\) \((\text{relation})\) \((\text{dim expr}_2)\)

\((\text{true code})\) \((\text{false code})\)

This function first evaluates each of the \((\text{dim exprs})\) as described for \dim_eval:n. The two results are then compared using the \((\text{relation})\):

- Equal  \(=\)
- Greater than  \(>\)
- Less than  \(<\)

This function is less flexible than \dim_compare:nTF but around 5 times faster.
\dim_compare_p:n \dim_compare:nTF

\dim_compare_p:n
{ (dim expr1) (relation1) 
... 
(dim exprN) (relationN) 
(dim expr\text{\text{N}}\text{\text{+}}\text{\text{1}}) 
}
\dim_compare:nTF
{ (dim expr1) (relation1) 
... 
(dim exprN) (relationN) 
(dim expr\text{\text{N}}\text{\text{+}}\text{\text{1}}) 
}
{\\text{(true code)}} {\\text{(false code)}}

This function evaluates the \langle \text{dim exprs} \rangle as described for \dim_eval:n and compares consecutive result using the corresponding \langle \text{relation} \rangle, namely it compares \langle \text{dim \ expr}_1 \rangle and \langle \text{dim \ expr}_2 \rangle using the \langle \text{relation}_1 \rangle, then \langle \text{dim \ expr}_2 \rangle and \langle \text{dim \ expr}_3 \rangle using the \langle \text{relation}_2 \rangle, until finally comparing \langle \text{dim \ expr}_N \rangle and \langle \text{dim \ expr}_{N\text{\text{+}}\text{\text{1}}} \rangle using the \langle \text{relation}_N \rangle. The test yields \text{true} if all comparisons are \text{true}. Each \langle \text{dim \ expr} \rangle is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is \text{false}, then no other \langle \text{dim \ expr} \rangle is evaluated and no other comparison is performed. The \langle \text{relations} \rangle can be any of the following:

- Equal \quad = \text{ or } ==
- 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 \dim_compare:nNnTF but around 5 times slower.
This function evaluates the \texttt{test dim expr} and compares this in turn to each of the \texttt{dim expr cases}. If the two are equal then the associated \texttt{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 \texttt{dim_case:nn}, which does nothing if there is no match, is also available. For example

\begin{verbatim}
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
  { 2 \l_tmpa_dim }
  { 5 pt } { Small }
  { 4 pt + 6 pt } { Medium }
  { - 10 pt } { Negative }
  { No idea! }
\end{verbatim}

leaves “Medium” in the input stream.

### 26.5 Dimension expression loops

\begin{verbatim}
\dim_do_until:nNnn
\dim_do_until:nNnn
\dim_do_while:nNnn
\dim_until_do:nNnn
\end{verbatim}
26.6 Dimension step functions

\dim_step_function:nnnN
\dim_step_function:nnnN \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{\text{function}\}

This function first evaluates the \text{initial value}, \text{step} and \text{final value}, all of which should be dimension expressions. The \text{function} is then placed in front of each \text{value} from the \text{initial value} to the \text{final value} in turn (using \text{step} between each \text{value}). The \text{step} must be non-zero. If the \text{step} is positive, the loop stops when the \text{value} becomes larger than the \text{final value}. If the \text{step} is negative, the loop stops when the \text{value} becomes smaller than the \text{final value}. The \text{function} should absorb one argument.

\dim_step_inline:nnn
\dim_step_inline:nnn \{\text{initial value}\} \{\text{step}\} \{\text{final value}\} \{\text{code}\}

This function first evaluates the \text{initial value}, \text{step} and \text{final value}, all of which should be dimension expressions. Then for each \text{value} from the \text{initial value} to the \text{final value} in turn (using \text{step} between each \text{value}), the \text{code} is inserted into the input stream with \texttt{#1} replaced by the current \text{value}. Thus the \text{code} should define a function of one argument (\texttt{#1}).
This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, all of which should be dimension expressions. Then for each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩), the ⟨code⟩ is inserted into the input stream, with the ⟨tl var⟩ defined as the current ⟨value⟩. Thus the ⟨code⟩ should make use of the ⟨tl var⟩.

26.7 Using \texttt{dim} expressions and variables

\eval: Evaluates the ⟨dim expr⟩, expanding any dimensions and token list variables within the ⟨expression⟩ to their content (without requiring \use:N/\tl_use:N) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨dimension denotation⟩ after two expansions. This is expressed in points (pt), and requires suitable termination if used in a \TeX\-style assignment as it is not an ⟨internal dimension⟩.

\sign: Evaluates the ⟨dim expr⟩ then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

\use: Recovers the content of a ⟨dimension⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨dimension⟩ is required (such as in the argument of \eval:).

\texttt{\TeX}hackers note: \use:N is the \TeX\ primitive \the: this is one of several \TeX\ names for this primitive.

\decimal: Evaluates the ⟨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

\decimal { 1bp }

leaves 1.00374 in the input stream, \textit{i.e.} the magnitude of one “big point” when converted to (\TeX) points.
\%dim_to_decimal_in_bp:n \{\langle dim expr\rangle\}

Evaluates the \langle dim expr\rangle, and leaves the result, expressed in big points (bp) in the input
stream, with no units. The result is rounded by \TeX to at most five decimal places. If
the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example
\dim_to_decimal_in_bp:n { 1pt }

leaves 0.99628 in the input stream, i.e. the magnitude of one (\TeX) point when converted
to big points.

\TeXhackers note: The implementation of this function is re-entrant: the result of
\dim_compare:nNnTF
{ <x>bp } =
{ \dim_to_decimal_in_bp:n { <x>bp } bp }

will be logically true. The decimal representations may differ provided they produce the same
\TeX dimension.

---

\%dim_to_decimal_in_cc:n
\%dim_to_decimal_in_cm:n
\%dim_to_decimal_in_dd:n
\%dim_to_decimal_in_in:n
\%dim_to_decimal_in_mm:n
\%dim_to_decimal_in_pc:n

Evaluates the \langle dim expr\rangle, and leaves the result, expressed with the appropriate scaling
in the input stream, with no units. If the decimal part of the result is zero, it is omitted,
together with the decimal marker. The 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 1\TeX 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 that these will result
in \TeX errors; the behavior if additional decimal places are given depends on the \TeX internals and thus larger values are \textit{not} supported by expl3.

\TeXhackers note: The implementation of these functions is re-entrant: the result of
\dim_compare:nNnTF
{ <x><unit> } =
{ \dim_to_decimal_in_<unit>:n { <x><unit> } <unit> }

will be logically true. The decimal representations may differ provided they produce the same
\TeX dimension.
\dim_to_decimal_in_sp:n \{\dim_expr\}

Evaluates the \dim_expr, and leaves the result, expressed in scaled points (sp) in the input stream, with no units. The result is necessarily an integer.

\dim_to_decimal_in_unit:nn \{\dim_expr_1\} \{\dim_expr_2\}

Evaluates the \dim_exprs, and leaves the value of \dim_expr_1, expressed in a unit given by \dim_expr_2, in the input stream. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

For example
\dim_to_decimal_in_unit:nn \{ 1bp \} \{ 1mm \}

leaves 0.35278 in the input stream, i.e. the magnitude of one big point when expressed in millimetres. The conversions do not guarantee that LaTeX would yield identical results for the direct input in an equality test, thus for instance
\dim_compare:nNnTF
\{ 1bp \} =
\{ \dim_to_decimal_in_unit:nn \{ 1bp \} \{ 1mm \} mm \}

will take the false branch.

\dim_to_fp:n \{\dim_expr\}

Expands to an internal floating point number equal to the value of the \dim_expr in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, \dim_to_fp:n can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

### 26.8 Viewing dim variables

\dim_show:N \dim_show:N \{\dim\}
\dim_show:C

Displays the value of the \dim on the terminal.

\dim_show:n \{\dim_expr\}

Displays the result of evaluating the \dim_expr on the terminal.

\dim_log:N \dim_log:N \{\dim\}
\dim_log:C

Writes the value of the \dim in the log file.
\dim_log:n \dim_log:n \{\dim expr\}

Wrote the result of evaluating the \{\dim expr\} in the log file.

New: 2014-08-22
Updated: 2015-06-07

26.9 Constant dimensions

\c_max_dim The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

\c_zero_dim A zero length as a dimension. This can also be used as a component of a skip.

26.10 Scratch dimensions

\l_tmpa_dim \l_tmpb_dim Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any \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.

26.11 Creating and initialising \skip variables

\skip_new:N \skip_new:N \{\skip\}
\skip_new:c

Creates a new \{\skip\} or raises an error if the name is already taken. The declaration is global. The \{\skip\} is initially equal to 0 pt.

\skip_const:Nn \skip_const:Nn \{\skip\} \{\skip expr\}
\skip_const:cn

New: 2012-03-05

Creates a new constant \{\skip\} or raises an error if the name is already taken. The value of the \{\skip\} is set globally to the \{\skip expr\}.

\skip_zero:N \skip_zero:N \{\skip\}
\skip_zero:c
\skip_gzero:N \skip_gzero:N
\skip_gzero:c

Sets \{\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  \langle \texttt{true code} \rangle \{ \langle \texttt{false code} \rangle \}
Tests whether the \texttt{skip} is currently defined. This does not check that the \texttt{skip} really is a skip variable.

26.12 Setting skip variables

\skip_add:Nn  \skip_add:cn  \skip_gadd:Nn  \skip_gadd:cn
Adds the result of the \texttt{skip expr} to the current content of the \texttt{skip}.

\skip_set:Nn  \skip_set:cn  \skip_gset:Nn  \skip_gset:cn
Updated: 2011-10-22
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).

\skip_set_eq:NN  \skip_set_eq:NN  \skip_set_eq:NN  \skip_set_eq:NN
\langle \texttt{skip1} \rangle \langle \texttt{skip2} \rangle
Sets the content of \texttt{skip1} equal to that of \texttt{skip2}.

\skip_sub:Nn  \skip_sub:cn  \skip_gsub:Nn  \skip_gsub:cn
Updated: 2011-10-22
Subtracts the result of the \texttt{skip expr} from the current content of the \texttt{skip}.
26.13 Skip expression conditionals

\skip_if_eq_p:nn \{⟨skip expr₁⟩\} \{⟨skip expr₂⟩\}
\skip_if_eq:nnTF \{⟨skip expr₁⟩\} \{⟨skip expr₂⟩\} \{⟨true code⟩\} \{⟨false code⟩\}

This function first evaluates each of the ⟨skip exprs⟩ as described for \skip_eval:n. The two results are then compared for exact equality, i.e. both the fixed and rubber components must be the same for the test to be true.

\skip_if_finite_p:n \{⟨skip expr⟩\}
\skip_if_finite:nTF \{⟨skip expr⟩\} \{⟨true code⟩\} \{⟨false code⟩\}

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

26.14 Using skip expressions and variables

\skip_eval:n \{⟨skip expr⟩\}

Updated: 2011-10-22

Evaluates the ⟨skip expr⟩, expanding any skips and token list variables within the ⟨expression⟩ to their content (without requiring \skip_use:N/\tl_use:N) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨glue denotation⟩ after two expansions. This is expressed in points (pt), and requires suitable termination if used in a \TeX-style assignment as it is not an ⟨internal glue⟩.

\skip_use:N \{⟨skip⟩\}
\skip_use:c

Recovers the content of a ⟨skip⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨dimension⟩ or ⟨skip⟩ is required (such as in the argument of \skip_eval:n).

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

26.15 Viewing skip variables

\skip_show:N \{⟨skip⟩\}
\skip_show:c

Displays the value of the ⟨skip⟩ on the terminal.

Updated: 2015-08-03

\skip_show:n \{⟨skip expr⟩\}

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

Displays the result of evaluating the ⟨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 \texttt{\langle \textit{skip} \rangle}
\skip_horizontal:N \texttt{\langle \textit{dim} \rangle}
\skip_horizontal:n \texttt{\langle \textit{skip expr} \rangle}
\skip_horizontal:n

Inserts a horizontal \langle \textit{skip} \rangle into the current list. The argument can also be a \langle \textit{dim} \rangle.

\TeXhackers note: \texttt{\skip_horizontal:N} is the \TeX primitive \texttt{\textbackslash hskip}. 

234
\skip_vertical:N \skip_vertical:C \skip_vertical:n

Inserts a vertical \langle \text{skip} \rangle into the current list. The argument can also be a \langle \text{dim} \rangle.

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

26.19 Creating and initialising \texttt{muskip} variables

\muskip_new:N \muskip_new:C

Creates a new \langle \texttt{muskip} \rangle or raises an error if the name is already taken. The declaration is global. The \langle \texttt{muskip} \rangle is initially equal to 0 mu.

\muskip_const:N \muskip_const:C

\texttt{New: 2012-03-05}

Creates a new constant \langle \texttt{muskip} \rangle or raises an error if the name is already taken. The value of the \langle \texttt{muskip} \rangle is set globally to the \langle \texttt{muskip expr} \rangle.

\muskip_zero:N \muskip_zero:C \muskip_gzero:N \muskip_gzero:C

Sets \langle \texttt{muskip} \rangle to 0 mu.

\texttt{New: 2012-01-07}

Ensures that the \langle \texttt{muskip} \rangle exists globally by applying \muskip_new:N if necessary, then applies \muskip_(g)zero:N to leave the \langle \texttt{muskip} \rangle set to zero.

\texttt{New: 2012-03-03}

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

26.20 Setting \texttt{muskip} variables

\muskip_add:Nn \muskip_add:cn \muskip_gadd:Nn \muskip_gadd:cn

\texttt{Updated: 2011-10-22}

Adds the result of the \langle \texttt{muskip expr} \rangle to the current content of the \langle \texttt{muskip} \rangle.
Sets \( \text{muskip} \) to the value of \( \text{muskip expr} \), which must evaluate to a math length with units and may include a rubber component (for example \( 1 \text{ mu plus } 0.5 \text{ mu} \)).

Sets the content of \( \text{muskip}_1 \) equal to that of \( \text{muskip}_2 \).

Subtracts the result of the \( \text{muskip expr} \) from the current content of the \( \text{muskip} \).

\( \text{muskip} \) expressions and variables

Evaluates the \( \text{muskip expr} \), expanding any skips and token list variables within the \( \text{expression} \) to their content (without requiring \( \text{muskip use:N/tl use:N} \)) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a \( \text{muglue denotation} \) after two expansions. This is expressed in \( \text{mu} \), and requires suitable termination if used in a \( \text{T\LaTeX} \)-style assignment as it is not an \( \text{internal muglue} \).

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 \( \text{dimension} \) is required (such as in the argument of \( \text{muskip eval:n} \)).

\( \text{T\LaTeX} \)hackers note: \( \text{muskip use:N} \) is the \( \text{T\LaTeX} \) primitive \( \text{\the} \): this is one of several \( \text{ET\LaTeX}3 \) names for this primitive.

Displays the value of the \( \text{muskip} \) on the terminal.
\muskip_show:n \muskip_show:n \{\muskip \text{expr}\}

Displays the result of evaluating the \textit{\muskip \text{expr}} on the terminal.

\muskip_log:N \muskip_log:c
\muskip_log:N \{\muskip\}

\text{New: 2014-08-22}
\text{Updated: 2015-08-03}

\muskip_log:n \muskip_log:n \{\muskip \text{expr}\}

\text{New: 2014-08-22}
\text{Updated: 2015-08-07}

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

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

26.25 Primitive conditional

\if_dim:w \if_dim:w \{\text{dimen}_1\} \{\text{relation}\} \{\text{dimen}_2\}
\{\text{true code}\}
\else:
\{\text{false}\}
\fi:

Compare two dimensions. The \textit{\text{relation}} is one of \textless, \text{=} or \textgreater{} with category code 12.

\TeXhackers note: This is the \TeX primitive \texttt{ifdim}.
Chapter 27

The \texttt{l3keys} module

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

\MyModuleSetup{
  key-one = value one,
  key-two = value two
}

or

\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more \texttt{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 \texttt{2e} package. See Section 27.2 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 \texttt{/} is reserved for sub-division of keys into different subsets. Functions and variables are not expanded when creating key names, and so

\begin{verbatim}
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
    \l_mymodule_tmp_tl .code:n = code
}
\end{verbatim}

creates a key called \texttt{\l_mymodule_tmp_tl}, and not one called \texttt{key}.

\section{Creating keys}

\texttt{\keys_define:nn} \texttt{\keys_define:ne} Updated: 2017-11-14

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 definitions are local.
Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as `.code:n`, `.tl_set:N`, etc., override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

\keys_define:nn { mymodule }
\begin{verbatim}
{  
  keyname .code:n = Some-code-using-#1,  
  keyname .value_required:n = true
}
\end{verbatim}
\keys_define:nn { mymodule }
\begin{verbatim}
{  
  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 `.undefine:` property undefines the key within the current \TeX{} group.

\begin{verbatim}
(key) .bool_set:N = (boolean variable)
\end{verbatim}

Defines `(key)` to set `(boolean variable)` to `(value)` (which must be either “true” or “false”). If the variable does not exist, it will be created globally at the point that the key is set up.

\begin{verbatim}
(key) .bool_set_inverse:N = (boolean variable)
\end{verbatim}

Defines `(key)` to set `(boolean variable)` to the logical inverse of `(value)` (which must be either “true” or “false”). If the `(boolean variable)` does not exist, it will be created globally at the point that the key is set up.

\begin{verbatim}
(key) .choice:
\end{verbatim}

Sets `(key)` to act as a choice key. Each valid choice for `(key)` must then be created, as discussed in section 27.3.

\begin{verbatim}
(key) .choices:nn = {(choices)} {(code)}
\end{verbatim}

Sets `(key)` to act as a choice key, and defines a series `(choices)` which are implemented using the `(code)`. Inside `(code)`, \l_keys_choice_tl will be the name of the choice made, and \l_keys_choice_int will be the position of the choice in the list of `(choices)` (indexed from 1). Choices are discussed in detail in section 27.3.

\begin{verbatim}
(key) .clist_set:N = (comma list variable)
\end{verbatim}

Defines `(key)` to set `(comma list variable)` to `(value)`. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.
\code:n\par Stores the \code\ for execution when \key\ is used. The \code\ can include one parameter (#1), which will be the \value\ given for the \key\.

\cs_set:Np\par Defines \key\ to set \controlsequence\ \argspec\ \repltext\.

\default:n\par Creates a \default\ value for \key, which is used if no value is given. This will be used if only the key name is given, but not if a blank \value\ is given:

\keys_define:nn { mymodule } \par \{
  key .code:n = Hello-#1, 
  key .default:n = World 
\}

\keys_set:nn { mymodule } \par \{
  key = Fred, % Prints ‘Hello Fred’
  key, % Prints ‘Hello World’
  key = , % Prints ‘Hello’
\}

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

When no value is given for a key as part of \keys_set:nn, the \default:n value provides the value before key properties are considered. The only exception is when the \value_required:n property is active: a required value cannot be supplied by the default, and must be explicitly given as part of \keys_set:nn.

\dim_set:N\par Defines \key\ to set \dimension\ to \valuex\ (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

\fp_set:N\par Defines \key\ to set \floatingpoint\ to \valuex\ (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.
_{groups:n} ⟨key⟩.groups:n = {⟨groups⟩}

New: 2013-07-14
Defines ⟨key⟩ as belonging to the ⟨groups⟩ (a comma-separated list). Groups provide a “secondary axis” for selectively setting keys, and are described in Section 27.7.

TeXhackers note: The ⟨groups⟩ argument is turned into a string then interpreted as a comma-separated list, so group names cannot contain commas nor start or end with a space character.

.inherit:n ⟨key⟩.inherit:n = {⟨parents⟩}

New: 2016-11-22
Specifies that the ⟨key⟩ path should inherit the keys listed as any of the ⟨parents⟩ (a comma list), which can be a module or a sub-division thereof. For example, after setting
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
setting
\keys_set:nn { bar } { test = a }
will be equivalent to
\keys_set:nn { foo } { test = a }

Inheritance applies at point of use, not at definition, thus keys may be added to the ⟨parent⟩ after the use of .inherit:n and will be active. If more than one ⟨parent⟩ is specified, the presence of the ⟨key⟩ will be tested for each in turn, with the first successful hit taking priority.

.initial:n ⟨key⟩.initial:n = {⟨value⟩}

Updated: 2013-07-09
Initialises the ⟨key⟩ with the ⟨value⟩, equivalent to
\keys_set:nn {⟨module⟩} { ⟨key⟩ = ⟨value⟩ }

.int_set:N ⟨key⟩.int_set:N = ⟨integer⟩

.int_set:c
.int_gset:N
.int_gset:c
Updated: 2020-01-17
Defines ⟨key⟩ to set ⟨integer⟩ to ⟨value⟩ (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.legacy_if_set:n ⟨key⟩.legacy_if_set:n = ⟨switch⟩

.legacy_if_gset:n
.legacy_if_set_inverse:n
.legacy_if_gset_inverse:n
Updated: 2022-01-15
Defines ⟨key⟩ to set legacy \if⟨switch⟩ to ⟨value⟩ (which must be either “true” or “false”). The ⟨switch⟩ is the name of the switch without the leading if. The inverse versions will set the ⟨switch⟩ to the logical opposite of the ⟨value⟩.

.meta:n ⟨key⟩.meta:n = {⟨keyval list⟩}

Updated: 2013-07-10
Makes ⟨key⟩ a meta-key, which will set ⟨keyval list⟩ in one go. The ⟨keyval list⟩ can refer as #1 to the value given at the time the ⟨key⟩ is used (or, if no value is given, the ⟨key⟩’s default value).
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 ⟨choices⟩ which are implemented using the ⟨code⟩. Inside ⟨code⟩, \l_keys_choice_tl will be the name of the choice made, and \l_keys_choice_int will be the position of the choice in the list of ⟨choices⟩ (indexed from 1). Choices are discussed in detail in section 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 put the ⟨value⟩ onto the ⟨property list⟩ stored under the ⟨key⟩. If the variable does not exist, it is created globally at the point that the key is set up.

 Defines ⟨key⟩ to set ⟨skip⟩ to ⟨value⟩ (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

 Defines ⟨key⟩ to set ⟨string variable⟩ to ⟨value⟩. If the variable does not exist, it is created globally at the point that the key is set up.

 Defines ⟨key⟩ to set ⟨string variable⟩ to ⟨value⟩, which will be subjected to an e-type expansion (i.e. using \str_set:N). If the variable does not exist, it is created globally at the point that the key is set up.
### Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several subsets for a given module. This can be achieved either by adding a sub-division to the module name:

\[ \texttt{\keys_define:nn} \{ \texttt{mymodule} / \texttt{subset} \} \{ \texttt{key} .\texttt{code:n} = \texttt{code} \} \]

or to the key name:

\[ \texttt{\keys_define:nn} \{ \texttt{mymodule} \} \{ \texttt{subset} / \texttt{key} .\texttt{code:n} = \texttt{code} \} \]

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `mymodule/subset/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.
27.3 Choice and multiple choice keys

The \l3{keys} system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the .choice: property:

\keys_define:nn { mymodule } { key .choice: }

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependent only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the .choices:nn property.

\keys_define:nn { mymodule } { key .choices:nn = { choice-a, choice-b, choice-c } { You-gave-choice-\tl_use:N \l_keys_choice_tl, which-is-in-position-\int_use:N \l_keys_choice_int \c_space_tl in-the-list. } }

The index \l_keys_choice_int in the list of choices starts at 1.

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

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

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

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

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special unknown choice. The general behavior of the unknown key is described in Section 27.6. A typical example in the case of a choice would be to issue a custom error message:

\keys_define:nn { mymodule }
{  
  key .choice:,  
  key / choice-a .code:n = code-a,  
  key / choice-b .code:n = code-b,  
  key / choice-c .code:n = code-c,  
  key / unknown .code:n =  
  \msg_error:nneee { mymodule } { unknown-choice }
  { key } % Name of choice key  
  { choice-a , choice-b , choice-c } % Valid choices  
  { \exp_not:n {#1} } % Invalid choice given
%
%
}

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties \mult choice: and \mult choices:nn. As with mutually exclusive choices, multiple choices are defined as sub-keys. Thus both

\keys_define:nn { mymodule }
{  
  key .multichoice:nn =  
  { choice-a, choice-b, choice-c }
  {  
    You-gave-choice-‘\tl_use:N \l_keys_choice_tl’,-  
    which-is-in-position-  
    \int_use:N \l_keys_choice_int \c_space_tl  
    in-the-list.
  }
}

and

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

are valid.

When a multiple choice key is set
\keys_set:nn { mymodule }
{
    key = { a , b , c } \% 'key' defined as a multiple choice
}

Each choice is applied in turn, equivalent to a clist mapping or to applying each value individually:
\keys_set:nn { mymodule }
{
    key = a ,
    key = b ,
    key = c ,
}

Thus each separate choice will have passed to it the \l_keys_choice_tl and \l_keys_choice_int in exactly the same way as described for .choices:nn.

### 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 \{ (module) \} \{ (keyval list) \}
Parses the (keyval list), and sets those keys which are defined for (module). The behaviour on finding an unknown key can be set by defining a special unknown key: this is illustrated later.
For each key processed, information of the full path of the key, the name of the key and the value of the key is available within two string and one token list variables. These may be used within the code of the key.

The path of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

\keys_set:nn { mymodule } { key-a = some-value }

has path mymodule/key-a while

\keys_set:nn { mymodule } { subset / key-a = some-value }

has path mymodule/subset/key-a. This information is stored in \l\_keys\_path\_str.

The name of the key is the part of the path after the last /, and thus is not unique. In the preceding examples, both keys have name key-a despite having different paths. This information is stored in \l\_keys\_key\_str.

The value is everything after the =, which may be empty if no value was given. This is stored in \l\_keys\_value\_tl, and is not processed in any way by \keys_set:nn.

### 27.6 Handling of unknown keys

If a key has not previously been defined (is unknown), \keys_set:nn looks for a special unknown key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts. The unknown key also supports the .default:n property.

\keys_define:nn { mymodule }
\{  
unknown .code:n =  
\ You-tried-to-set-key-’\l\_keys\_key\_str’-to-’#1’. ,  
unknown .default:V = \c\_no\_value\_tl  
\}  

### 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_known:nn {\langle module\rangle} {\langle keyval list\rangle}
\keys_set_known:nnN {\langle module\rangle} {\langle keyval list\rangle} \{tl var\}
\keys_set_known:nnnN {\langle module\rangle} {\langle keyval list\rangle} \{\langle root\rangle\} \{tl var\}
\keys_set_known:nn {\langle module\rangle} {\langle keyval list\rangle}
\keys_set_known:nnN {\langle module\rangle} {\langle keyval list\rangle} \{tl var\}
\keys_set_known:nnnN {\langle module\rangle} {\langle keyval list\rangle} \{\langle root\rangle\} \{tl var\}

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:nnN and \keys_set_known:nnnN store the key-value pairs for unknown keys in the \langle tl var\rangle in comma-separated form (i.e. an edited version of the \langle keyval list\rangle). When a \langle root\rangle is given \langle \keys_set_known:nnnN \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.
These functions activate key selection in an “opt-in” sense: only keys assigned to one or more of the \( \langle \text{groups} \rangle \) specified are set. The \( \langle \text{groups} \rangle \) are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

In addition, \( \keys_set_groups:nnnN \) and \( \keys_set_groups:nnnnN \) store the key–value pairs for skipped keys in the \( \langle \text{tl var} \rangle \) in comma-separated form (i.e. an edited version of the \( \langle \text{keyval list} \rangle \)). When a \( \langle \text{root} \rangle \) is given \( \langle \keys_set_groups:nnnnN \rangle \), 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.

These functions activate key selection in an “opt-out” sense: keys assigned to one or more of the \( \langle \text{groups} \rangle \) specified are not set. The \( \langle \text{groups} \rangle \) are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set.

In addition, \( \keys_set_exclude_groups:nnnN \) and \( \keys_set_exclude_groups:nnnnN \) store the key–value pairs for skipped keys in the \( \langle \text{tl var} \rangle \) in comma-separated form (i.e. an edited version of the \( \langle \text{keyval list} \rangle \)). When a \( \langle \text{root} \rangle \) is given \( \langle \keys_set_exclude_groups:nnnnN \rangle \), 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.

27.8 Digesting keys

\[ \keys_precompile:nnN \] \( \langle \text{module} \rangle \) \( \langle \text{keyval list} \rangle \) \( \langle \text{tl var} \rangle \)

 Parses the \( \langle \text{keyval list} \rangle \) as for \( \keys_set:nn \), placing the resulting code for those which set variables or functions into the \( \langle \text{tl var} \rangle \). Thus this function “precompiles” the keyval list into a set of results which can be applied rapidly.
27.9 Utility functions for keys

\keys_if_exist_p:nn = \keys_if_exist_p:nn \{\text{module}\} \{\text{key}\}
\keys_if_exist_p:ne = \keys_if_exist_p:nnTF \{\text{module}\} \{\text{key}\} \{\text{true code}\} \{\text{false code}\}
\keys_if_choice_exist_p:nn = \keys_if_choice_exist_p:nn \{\text{module}\} \{\text{key}\} \{\text{choice}\}
\keys_if_choice_exist_p:nnTF = \keys_if_choice_exist_p:nnTF \{\text{module}\} \{\text{key}\} \{\text{choice}\} \{\text{true code}\} \{\text{false code}\}

Tests if the \texttt{\{key\}} exists for \texttt{\{module\}}, i.e. if any code has been defined for \texttt{\{key\}}.

\keys_show:nn = \keys_show:nn \{\text{module}\} \{\text{key}\}

Displays in the terminal the information associated to the \texttt{\{key\}} for a \texttt{\{module\}}, including the function which is used to actually implement it.

\keys_log:nn = \keys_log:nn \{\text{module}\} \{\text{key}\}

Writes in the log file the information associated to the \texttt{\{key\}} for a \texttt{\{module\}}. See also \keys_show:nn which displays the result in the terminal.

27.10 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

\begin{verbatim}
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
\end{verbatim}

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 \texttt{\{key–value list\}} into \texttt{\{keys\}} and associated \texttt{\{values\}}. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double # tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category “other” (12) so that the
parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

\[\text{key} = \{\text{value here}\},\]

and

\[\text{key} = \text{value here},\]

are treated identically.

\keyval_parse:nnn \(\langle\text{code} 1\rangle\) \(\langle\text{code} 2\rangle\) \(\langle\text{key-value list}\rangle\)

Parses the \(\langle\text{key-value list}\rangle\) into a series of \(\langle\text{keys}\rangle\) and associated \(\langle\text{values}\rangle\), or keys alone (if no \(\langle\text{value}\rangle\) was given). \(\langle\text{code} 1\rangle\) receives each \(\langle\text{key}\rangle\) (with no \(\langle\text{value}\rangle\)) as a trailing brace group, whereas \(\langle\text{code} 2\rangle\) is appended by two brace groups, the \(\langle\text{key}\rangle\) and \(\langle\text{value}\rangle\). The order of the \(\langle\text{keys}\rangle\) in the \(\langle\text{key-value list}\rangle\) is preserved. Thus

\keyval_parse:nnn
{ \use_none:nn \{ code 1 \} }  
{ \use_none:nnn \{ code 2 \} }  
{ key1 = value1 , key2 = value2, key3 = , key4 }  

is converted into an input stream

\use_none:nn \{ code 2 \} \{ key1 \} \{ value1 \}  
\use_none:nn \{ code 2 \} \{ key2 \} \{ value2 \}  
\use_none:nn \{ code 2 \} \{ key3 \} \}  
\use_none:nn \{ code 1 \} \{ key4 \}  

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the \(\langle\text{key}\rangle\) and \(\langle\text{value}\rangle\), then one \emph{outer} set of braces is removed from the \(\langle\text{key}\rangle\) and \(\langle\text{value}\rangle\) 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.

\TeXhackers\ \textbf{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) 

Parses the \langle key–value list \rangle into a series of \langle keys \rangle and associated \langle values \rangle, or keys alone (if no \langle value \rangle was given). \langle function_1 \rangle should take one argument, while \langle function_2 \rangle should absorb two arguments. After \keyval_parse:NNn has parsed the \langle key–value list \rangle, \langle function_1 \rangle is used to process keys given with no value and \langle function_2 \rangle is used to process keys given with a value. The order of the \langle keys \rangle in the \langle key–value list \rangle is preserved. Thus

\keyval_parse:NNn \function:n \function:nn
\{ key1 = value1 , key2 = value2, key3 = , key4 \}

is converted into an input stream

\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the \langle key \rangle and \langle value \rangle, then one outer set of braces is removed from the \langle key \rangle and \langle value \rangle as part of the processing.

This shares the implementation of \keyval_parse:nnn, the difference is only semantically.

\TeXhackers\note: The result is returned within \exp_not:n, which means that the converted input stream does not expand further when appearing in an e-type or x-type argument expansion.
Chapter 28

The \texttt{l3intarray} module

Fast global integer arrays

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast \texttt{l3seq}, where access time is linear). These arrays have several important features

\begin{itemize}
  \item The size of the array is fixed and must be given at point of initialisation
  \item The absolute value of each entry has maximum $2^{30} - 1$ (\textit{i.e.} one power lower than the usual \texttt{\_c\_max\_int} ceiling of $2^{31} - 1$)
\end{itemize}

The use of \texttt{intarray} data is therefore recommended for cases where the need for fast access is of paramount importance.

28.1 Creating and initialising integer array variables

\begin{verbatim}
\intarray_new:Nn \intarray_new:cn
\end{verbatim}

Evaluates the integer expression \texttt{(size)} and allocates an \texttt{(integer array variable)} with that number of (zero) entries. The variable name should start with \texttt{\_g\_} because assignments are always global.

\begin{verbatim}
\intarray_const_from_clist:Nn \intarray_const_from_clist:cn
\end{verbatim}

Creates a new constant \texttt{(integer array variable)} or raises an error if the name is already taken. The \texttt{(integer array variable)} is set (globally) to contain as its items the results of evaluating each \texttt{(integer expression)} in the \texttt{(comma list)}.

\begin{verbatim}
\intarray_gzero:N \intarray_gzero:cn
\end{verbatim}

Sets all entries of the \texttt{(integer array variable)} to zero. Assignments are always global.
28.2 Adding data to integer arrays

\intarray_gset:Nnn \intarray_gset:nnn \intarray_gset:cn

Stores the result of evaluating the integer expression \langle value \rangle into the \langle integer array variable \rangle at the (integer expression) \langle position \rangle. If the \langle position \rangle is not between 1 and the \intarray_count:N, or the \langle value \rangle's absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

28.3 Counting entries in integer arrays

\intarray_count:N \intarray_count:n \intarray_count:cn

Expands to the number of entries in the \langle integer array variable \rangle. Contrarily to \seq_count:N this is performed in constant time.

28.4 Using a single entry

\intarray_item:Nn \intarray_item:nn \intarray_item:cn

Expands to the integer entry stored at the (integer expression) \langle position \rangle in the \langle integer array variable \rangle. If the \langle position \rangle is not between 1 and the \intarray_count:N, an error occurs.

\intarray_rand_item:N \intarray_rand_item:nn \intarray_rand_item:cn

Selects a pseudo-random item of the \langle integer array \rangle. If the \langle integer array \rangle is empty, produce an error.

28.5 Integer array conditional

\intarray_if_exist_p:N \intarray_if_exist:np \intarray_if_exist:pc \intarray_if_exist:N \intarray_if_exist:ntf \intarray_if_exist:ctf

Tests whether the \langle intarray var \rangle is currently defined. This does not check that the \langle intarray var \rangle really is an integer array variable.

28.6 Viewing integer arrays

\intarray_show:N \intarray_show:n \intarray_show:cn \intarray_log:N \intarray_log:n \intarray_log:cn

Displays the items in the \langle integer array variable \rangle in the terminal or writes them in the log file.
28.7 Implementation notes

It is a wrapper around the \fontdimen primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to \l3seq sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive \fontdimen stores dimensions but the \l3intarray module transparently converts these from/to integers. Assignments are always global.

While LuaTeX’s memory is extensible, other engines can “only” deal with a bit less than $4 \times 10^9$ entries in all \fontdimen arrays combined (with default \TeX Live settings).
Chapter 29
The l3fp module
Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions ("\langle fp expr \rangle") support the following operations with their usual precedence.

- Basic arithmetic: addition \( x + y \), subtraction \( x - y \), multiplication \( x * y \), 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 \( !x \), conjunction \( x && y \), disjunction \( x || y \), ternary operator \( x ? y : z \).
- Exponentials: \( \exp x \), \( \ln x \), \( x^y \), \( \log_b x \).
- Integer factorial: \( \text{fact} x \).
- Trigonometry: \( \sin x \), \( \cos x \), \( \tan x \), \( \cot x \), \( \sec x \), \( \csc x \) expecting their arguments in radians, and \( \sin^d x \), \( \cos^d x \), \( \tan^d x \), \( \cot^d x \), \( \sec^d x \), \( \csc^d x \) expecting their arguments in degrees.
- Inverse trigonometric functions: \( \arcsin x \), \( \arccos x \), \( \arctan x \), \( \arccot x \), \( \arcsec x \), \( \arccsc x \) giving a result in radians, and \( \arcsin^d x \), \( \arccos^d x \), \( \arctan^d x \), \( \arccot^d x \), \( \arcsec^d x \), \( \arccsc^d x \) giving a result in degrees.

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

- Extrema: \( \max(x_1, x_2, \ldots) \), \( \min(x_1, x_2, \ldots) \), \( \text{abs}(x) \).
- Rounding functions, controlled by two optional values, \( n \) (number of places, 0 by default) and \( t \) (behavior on a tie, \( \text{nan} \) by default):
  - \( \text{trunc}(x, n) \) rounds towards zero,
  - \( \text{floor}(x, n) \) rounds towards \(-\infty\),

257
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: rand(), randint(m, n).
- Constants: pi, deg (one degree in radians).
- Dimensions, automatically expressed in points, e.g., pc is 12.
- Automatic conversion (no need for \langle\texttt{type\_use:N}\rangle) of integer, dimension, and skip
variables to floating point numbers, expressing dimensions in points and ignoring
the stretch and shrink components of skips.
- Tuples: (x₁,...,xₙ) 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 sec-
tion 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.

\documentclass{article}
\usepackage{siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnum } { m }
{ \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnum { 2 \pi \ast \sin ( 2.3 ^ 5 ) }
\end{document}

See the documentation of siunitx for various options of \num.
29.1 Creating and initialising floating point variables

\fp_new:N \fp_new:N (fp var)
Creates a new (fp var) or raises an error if the name is already taken. The declaration is global. The (fp var) is initially +0.

\fp_new:N \fp_new:c Updated: 2012-05-08
\fp_new:N (fp var) \fp_new:c
\fp_new:N
\fp_new:c

\fp_const:Nn \fp_const:Nn (fp var) \{\fp expr\}
Creates a new constant (fp var) or raises an error if the name is already taken. The (fp var) is set globally equal to the result of evaluating the (fp expr).

\fp_const:Nn \fp_const:cn Updated: 2012-05-08
\fp_const:Nn \fp_const:cn
\fp_const:Nn \fp_const:cn
\fp_const:Nn
\fp_const:cn

\fp_zero:N \fp_zero:N (fp var)
Sets the (fp var) to +0.

\fp_zero:N \fp_zero:N (fp var) \fp_zero:N
\fp_zero:N \fp_zero:N (fp var) \fp_zero:N
\fp_zero:N
\fp_zero:N

\fp_zero_new:N \fp_zero_new:N (fp var)
Ensures that the (fp var) exists globally by applying \fp_new:N if necessary, then applies \fp_(g)zero:N to leave the (fp var) set to +0.

\fp_zero_new:N \fp_zero_new:N
\fp_zero_new:N \fp_zero_new:N
\fp_zero_new:N
\fp_zero_new:N

29.2 Setting floating point variables

\fp_set:Nn \fp_set:Nn (fp var) \{\fp expr\}
Sets (fp var) equal to the result of computing the (fp expr).

\fp_set:Nn \fp_set:cn \fp_set:Nn Updated: 2012-05-08
\fp_set:cn \fp_set:cn
\fp_set:cn
\fp_set:cn
\fp_set:cn

\fp_set_eq:NN \fp_set_eq:NN (fp var1) (fp var2)
Sets the floating point variable (fp var1) equal to the current value of (fp var2).

\fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN Updated: 2012-05-08
\fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN
\fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN
\fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN
\fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN
\fp_set_eq:NN \fp_set_eq:NN \fp_set_eq:NN
\fp_set_eq:NN \fp_set_eq:NN

\fp_add:Nn \fp_add:Nn (fp var) \{\fp expr\}
Adds the result of computing the (fp expr) to the (fp var). This also applies if (fp var) and (floating point expression) evaluate to tuples of the same size.

\fp_add:Nn \fp_add:Nn \fp_add:cn Updated: 2012-05-08
\fp_add:cn \fp_add:cn
\fp_add:cn \fp_add:cn
\fp_add:cn
\fp_add:cn
\fp_add:cn
Subtracting the result of computing the (floating point expression) from the (fp var). This also applies if (fp var) and (floating point expression) evaluate to tuples of the same size.

29.3 Using floating points

Evaluates the (fp expr) and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values ±∞ and nan trigger an “invalid operation” exception. For a tuple, each item is converted using \fp_eval:n and they are combined as ((fp_1), (fp_2), \ldots (fp_n)) if n > 1 and ((fp_1),) or () for fewer items. This function is identical to \fp_to_decimal:n.

Evaluates the (fp expr) and leaves its sign in the input stream using \fp_eval:n \{sign(result\})\}: +1 for positive numbers and for +∞, −1 for negative numbers and for −∞, ±0 for ±0. If the operand is a tuple or is nan, then “invalid operation” occurs and the result is 0.

Evaluates the (fp expr) and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values ±∞ and nan trigger an “invalid operation” exception. For a tuple, each item is converted using \fp_to_decimal:n and they are combined as ((fp_1), (fp_2), \ldots (fp_n)) if n > 1 and ((fp_1),) or () for fewer items.

Evaluates the (fp expr) and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to \fp_to_decimal:n, with an additional trailing pt (both letter tokens). In particular, the result may be outside the range \([-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]\) of valid \TeX\ dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values ±∞ and nan, trigger an “invalid operation” exception.

Evaluates the (fp expr), and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range \([-2^{31} + 1, 2^{31} - 1]\) of valid \TeX\ integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values ±∞ and nan, trigger an “invalid operation” exception.
\fp_to_scientific:N \fp_to_scientific:c \fp_to_scientific:n

Evaluates the \PROP{fp expr} and expresses the result in scientific notation:

\begin{verbatim}
(optional \-)\langle digit \rangle.(15 digits)e\langle optional sign \rangle\langle exponent \rangle
\end{verbatim}

The leading \langle digit \rangle is non-zero except in the case of ±0. The values ±∞ and nan trigger an “invalid operation” exception. Normal category codes apply: thus the e is category code 11 (a letter). For a tuple, each item is converted using \fp_to_scientific:n and they are combined as \langle (fp_1), (fp_2), … (fp_n) \rangle if n > 1 and ((fp_1),) or () for fewer items.

\fp_to_tl:N \fp_to_tl:c \fp_to_tl:n

Evaluates the \PROP{fp expr} and expresses the result in (almost) the shortest possible form. Numbers in the ranges (0, 10^{-3}) and [10^{16}, ∞) are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from \fp_to_scientific:n). Numbers in the range [10^{-3}, 10^{16}) are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see \fp_to_decimal:n). Negative numbers start with -. The special values ±0, ±∞ and nan are rendered as 0, -0, inf, -inf, and nan respectively. Normal category codes apply and thus inf or nan, if produced, are made up of letters. For a tuple, each item is converted using \fp_to_tl:n and they are combined as \langle (fp_1), (fp_2), … (fp_n) \rangle if n > 1 and ((fp_1),) or () for fewer items. This function is identical to \fp_to_decimal:N.

\fp_use:N \fp_use:c

Inserts the value of the \PROP{fp var} into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values ±∞ and nan trigger an “invalid operation” exception. For a tuple, each item is converted using \fp_to_decimal:n and they are combined as \langle (fp_1), (fp_2), … (fp_n) \rangle if n > 1 and ((fp_1),) or () for fewer items. This function is identical to \fp_to_decimal:n.

\section{Floating point conditionals}

\fp_if_exist_p:N \fp_if_exist_p:c \fp_if_exist:N \fp_if_exist:NTF \fp_if_exist:cTF

Tests whether the \PROP{fp var} is currently defined. This does not check that the \PROP{fp var} really is a floating point variable.
\fp_compare_p:nNn \fp_compare:nNn {\textit{fp expr}_1} \{\textit{relation}\} \{\textit{fp expr}_2\}
\fp_compare:nNnTF \fp_compare:nNn {\textit{fp expr}_1} \{\textit{relation}\} \{\textit{fp expr}_2\} \{\textit{true code}\}\{\textit{false code}\}

Compares 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 \textit{nan} or is a tuple, unless they are equal tuples. Note that a \textit{nan} is distinct from any value, even another \textit{nan}, hence \(x = x\) is not true for a \textit{nan}. To test if a value is \textit{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 \textit{nan}). At present any other comparison with tuples yields \(\textit{?}\) (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}. 

Updated: 2012-05-08

262
\fp_compare_p:n \fp_compare:nTF
\fp_if_nan_p:n \fp_if_nan:nTF
\fp_do_until:nNnn
\fp_if_nan_p:n
\fp_if_nan:nTF
\fp_do_until:nNnn

29.5 Floating point expression loops
\fp_do_while:nNnn \{\text{\textit{code}}\} \{\text{\textit{code}}\}
\textbf{New: 2012-08-16}
Places the \textit{\langle code\rangle} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}. If the test is \textit{true} then the \textit{\langle code\rangle} is inserted into the input stream again and a loop occurs until the \textit{(relation)} is \textit{false}.

\fp_until_do:nNnn \{\text{\textit{code}}\} \{\text{\textit{code}}\}
\textbf{New: 2012-08-16}
Evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \textit{\langle code\rangle} in the input stream if the \textit{(relation)} is \textit{false}. After the \textit{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \textit{true}.

\fp_do_until:nn \{\text{\textit{code}}\} \{\text{\textit{code}}\}
\textbf{New: 2012-08-16}
Places the \textit{\langle code\rangle} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}. If the test is \textit{false} then the \textit{\langle code\rangle} is inserted into the input stream again and a loop occurs until the \textit{(relation)} is \textit{true}.

\fp_until_do:nn \{\text{\textit{code}}\} \{\text{\textit{code}}\}
\textbf{New: 2012-08-16}
\textbf{Updated: 2013-12-14}
Evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \textit{\langle code\rangle} in the input stream if the \textit{(relation)} is \textit{true}. After the \textit{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \textit{false}.

\fp_do_until:nn \{\text{\textit{code}}\} \{\text{\textit{code}}\}
\textbf{New: 2012-08-16}
\textbf{Updated: 2013-12-14}
Places the \textit{\langle code\rangle} in the input stream for \TeX{} to process, and then evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}. If the test is \textit{true} then the \textit{\langle code\rangle} is inserted into the input stream again and a loop occurs until the \textit{(relation)} is \textit{false}.

\fp_until_do:nn \{\text{\textit{code}}\} \{\text{\textit{code}}\}
\textbf{New: 2012-08-16}
\textbf{Updated: 2013-12-14}
Evaluates the relationship between the two \textit{(floating point expressions)} as described for \texttt{\textbackslash fp\_compare:nNnTF}, and then places the \textit{\langle code\rangle} in the input stream if the \textit{(relation)} is \textit{true}. After the \textit{(code)} has been processed by \TeX{} the test is repeated, and a loop occurs until the test is \textit{false}.
This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The ⟨function⟩ is then placed in front of each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩). The ⟨step⟩ must be non-zero. If the ⟨step⟩ is positive, the loop stops when the ⟨value⟩ becomes larger than the ⟨final value⟩. If the ⟨step⟩ is negative, the loop stops when the ⟨value⟩ becomes smaller than the ⟨final value⟩. The ⟨function⟩ should absorb one numerical argument. For example

\cs_set:Npn \my_func:n \#1 \{ I saw \#1 \quad \}
\fp_step_function:nnnN \{ 1.0 \} \{ 0.1 \} \{ 1.5 \} \my_func:n

would print

[I saw 1.0] [I saw 1.1] [I saw 1.2] [I saw 1.3] [I saw 1.4] [I saw 1.5]

\TeX\ Hackers note: Due to rounding, it may happen that adding the ⟨step⟩ to the ⟨value⟩ does not change the ⟨value⟩; such cases give an error, as they would otherwise lead to an infinite loop.

This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩), the ⟨code⟩ is inserted into the input stream with \#1 replaced by the current ⟨value⟩. Thus the ⟨code⟩ should define a function of one argument (\#1).

This function first evaluates the ⟨initial value⟩, ⟨step⟩ and ⟨final value⟩, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each ⟨value⟩ from the ⟨initial value⟩ to the ⟨final value⟩ in turn (using ⟨step⟩ between each ⟨value⟩), the ⟨code⟩ is inserted into the input stream, with the ⟨tl var⟩ defined as the current ⟨value⟩. Thus the ⟨code⟩ should make use of the ⟨tl var⟩.

### 29.6 Symbolic expressions

Floating point expressions support variables: these can only be set locally, so act like standard \l... variables.

\fp_new_variable:n \{ A \}
\fp_set:Nn \l_tmpb_fp \{ 1 * \sin(A) + 3**2 \}
\fp_show:n \{ \l_tmpb_fp \}
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn \{ A \} \{ \pi/2 \}

265
defines $A$ to be a variable, then defines $\l_{tmpb\_fp}$ to stand for $1\times \sin(A)+9$ (note that $3^2$ is evaluated, but the $1\times$ product is not simplified away). Until $\l_{tmpb\_fp}$ is changed, $\fp\_show:N\ l_{tmpb\_fp}$ will show $((1\times \sin(A))+9)$ 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 $1\times \sin(A)+9$, 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 \{ \l_{tmpb\_fp} \}
\fp\_show:n \l_{tmpb\_fp}
\fp\_set\_variable:nn \{ A \} \{ 0 \}
\fp\_show:n \l_{tmpb\_fp}
\fp\_show:N \l_{tmpb\_fp}
\end{verbatim}

\fp\_new\_variable:n \{ (identifier) \}
\begin{verbatim}
New: 2023-10-19
\fp\_new\_variable:n \{ (identifier) \}
\end{verbatim}
Declares the \texttt{(identifier)} 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 \texttt{(identifier)} must consist entirely of Latin letters among \texttt{[a-zA-Z]}.

\fp\_set\_variable:nn \{ (identifier) \} \{ (fp expr) \}
\begin{verbatim}
New: 2023-10-19
\fp\_set\_variable:nn \{ (identifier) \} \{ (fp expr) \}
\end{verbatim}
Defines the \texttt{(identifier)} to stand in any further expression for the result of evaluating the \texttt{(floating point expression)} 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 \times 1 \}
\fp\_show:n \{ C + 4 \}
\fp\_set\_variable:nn \{ A \} \{ 4 \}
\fp\_show:n \{ C + 4 \}
\end{verbatim}
shows $((9+(B\times1))+4)$ twice: changing the value of $A$ to $4$ does not alter $C$ because $A$ was replaced by its value $3$ when evaluating $A^2+B\times1$.
\texttt{\textbackslash fp\_clear\_variable:n}\{(\textit{identifier})\}

Removes any value given by \texttt{\textbackslash fp\_set\_variable:nn} to the variable with this \textit{identifier}. For instance,

\begin{verbatim}
\fp_new_variable:n { A }
\fp_set_variable:nn { A } { 3 }
\fp_show:n { A \cdot 2 }
\fp_clear_variable:n { A }
\fp_show:n { A \cdot 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}\{(\textit{identifier})\}

Declares the \textit{identifier} as a function, which allows it to be used in floating point expressions. For instance,

\begin{verbatim}
\fp_new_function:n { foo }
\fp_show:n { foo ( 1 + 2 , foo(3), A ) \cdot 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 \textit{identifier} must consist entirely of Latin letters \texttt{[a-zA-Z]}.

\texttt{\textbackslash fp\_set\_function:nnn}\{(\textit{identifier})\}\{(\textit{vars})\}\{(\textit{fp expr})\}

Defines the \textit{identifier} to stand in any further expression for the result of evaluating the \textit{floating point expression}, with the \textit{identifier} accepting the \textit{vars} (a non-empty comma-separated list). The result may contain other functions, which are then replaced by their results if they have any. For instance,

\begin{verbatim}
\fp_new_function:n { foo }
\fp_set_function:nnn { npow } { a,b } { a**b }
\fp_show:n { npow(16,0.25) }
\end{verbatim}

shows 2. The names of the \textit{vars} must consist entirely of Latin letters \texttt{[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}\{(\textit{identifier})\}

Removes any definition given by \texttt{\textbackslash fp\_set\_function:nnn} to the function with this \textit{identifier}. 

267
29.8 Some useful constants, and scratch variables

\( \texttt{c\_zero\_fp} \) Zero, with either sign.
\( \texttt{c\_minus\_zero\_fp} \)
New: 2012-05-08

\( \texttt{c\_one\_fp} \) One as an fp: useful for comparisons in some places.
New: 2012-05-08

\( \texttt{c\_inf\_fp} \) Infinity, with either sign. These can be input directly in a floating point expression as \texttt{inf} and \texttt{-inf}.
\( \texttt{c\_minus\_inf\_fp} \)
New: 2012-05-08

\( \texttt{c\_nan\_fp} \) Not a number. This can be input directly in a floating point expression as \texttt{nan}.
New: 2012-05-08

\( \texttt{c\_e\_fp} \) The value of the base of the natural logarithm, \( e = \exp(1) \).
Updated: 2012-05-08

\( \texttt{c\_pi\_fp} \) The value of \( \pi \). This can be input directly in a floating point expression as \texttt{pi}.
Updated: 2013-11-17

\( \texttt{c\_one\_degree\_fp} \) 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 \texttt{deg}.
New: 2012-05-08
Updated: 2013-11-17

29.9 Scratch variables

\( \texttt{l\_tmpa\_fp} \) Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any \texttt{FM}X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
\( \texttt{l\_tmpb\_fp} \)

\( \texttt{g\_tmpa\_fp} \) Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any \texttt{FM}X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
\( \texttt{g\_tmpb\_fp} \)
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 ±\(\infty\).

- **Underflow** occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ±\(0\).

- **Invalid operation** occurs for operations with no defined outcome, for instance 0/0 or \(\sin(\infty)\), and results in a \texttt{nan}. It also occurs for conversion functions whose target type does not have the appropriate infinite or \texttt{nan} value (e.g., \texttt{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 ±\(\infty\).

(Not yet) **Inexact** occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in \LaTeX3.

To each exception we associate a “flag”: \texttt{l_fp_overflow_flag}, \texttt{l_fp_underflow_flag}, \texttt{l_fp_invalid_operation_flag} and \texttt{l_fp_division_by_zero_flag}. The state of these flags can be tested and modified with commands from \texttt{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 \texttt{fp_trap:nn}) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

\texttt{\textbackslash fp\_trap:nn \{exception\} \{trap type\}}

All occurrences of the \texttt{\textbackslash fp\_trap:nn} within the current group are treated as \texttt{\{trap type\}}, which can be

- **none**: the \texttt{\{exception\}} will be entirely ignored, and leave no trace;
- **flag**: the \texttt{\{exception\}} will turn the corresponding flag on when it occurs;
- **error**: additionally, the \texttt{\{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.

\texttt{\textbackslash l\_fp\_overflow\_flag}
\texttt{\textbackslash l\_fp\_underflow\_flag}
\texttt{\textbackslash l\_fp\_invalid\_operation\_flag}
\texttt{\textbackslash l\_fp\_division\_by\_zero\_flag}

Flags denoting the occurrence of various floating-point exceptions.
29.11 Viewing floating points

\fp_show:N \fp_show:N \(<\text{fp var}>\)
\fp_show:n \{\fp_show:n \(<\text{fp expr}>\)\}

Evaluates the \(<\text{fp expr}>\) and displays the result in the terminal.

\fp_log:N \fp_log:N \(<\text{fp var}>\)
\fp_log:n \{\fp_log:n \(<\text{fp expr}>\)\}

Evaluates the \(<\text{fp expr}>\) and writes the result in the log file.

29.12 Floating point expressions

29.12.1 Input of floating point numbers

We support four types of floating point numbers:

- \(\pm m \cdot 10^n\), a floating point number, with integer \(1 \leq m \leq 10^{16}\), and \(-10000 \leq n \leq 10000\);
- \(\pm 0\), zero, with a given sign;
- \(\pm \infty\), infinity, with a given sign;
- \(\text{nan}\), is “not a number”, and can be either quiet or signalling (\textit{not yet}: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures. On input, a normal floating point number consists of:

- \(\langle \text{sign}\rangle\): a possibly empty string of \(+\) and \(-\) characters;
- \(\langle \text{significand}\rangle\): a non-empty string of digits together with zero or one dot;
- \(\langle \text{exponent}\rangle\) optionally: the character \(\text{e}\) or \(\text{E}\), followed by a possibly empty string of \(+\) and \(-\) tokens, and a non-empty string of digits.

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

A special case arises if the resulting \(\langle \text{exponent}\rangle\) is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by \(\pm \infty\)), or an underflow (resulting in \(\pm 0\)).
The result is thus ±0 if and only if \( \langle \text{significand} \rangle \) contains no non-zero digit (i.e., consists only in characters 0, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

The \( \langle \text{significand} \rangle \) must be non-empty, so e1 and e-1 are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as \( e \) and should be input as \( \exp(1) \) or \( \text{\c_e_fp} \) (which is faster).

Special numbers are input as follows:

- \( \text{inf} \) represents +\( \infty \), and can be preceded by any \( \langle \text{sign} \rangle \), yielding ±\( \infty \) as appropriate.
- \( \text{nan} \) 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 \( \text{nan} \).
- Note that commands such as \( \text{\infty} \), \( \text{\pi} \), or \( \text{\sin} \) do not work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

29.12.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (\( \sin, \ln, \text{etc} \)).
- Binary ** and ^ (right associative).
- Unary +, -, !.
- Implicit multiplication by juxtaposition (2\( \pi \)) when neither factor is in parentheses.
- Binary * and /, implicit multiplication by juxtaposition with parentheses (for instance 3(4+5)).
- Binary + and -.
- Comparisons \( >, \!, =, <, \text{etc} \).
- 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

\[
\frac{1}{2\pi} = \frac{1}{(2\pi),}
\]

\[
\frac{1}{2}\pi(\pi + \pi) = (2\pi)^{-1}(\pi + \pi) \simeq 1,
\]

\[
\sin 2\pi = \sin(2\pi) \neq 0,
\]

\[
2^{-2}\max(3, 5) = 2^2 \max(3, 5) = 20,\]

\[
1\text{in}/1\text{cm} = (1\text{in})/(1\text{cm}) = 2.54.
\]

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.

```latex
\fp_eval:n { ⟨operand⟩ ? ⟨operand⟩ : ⟨operand⟩ }
```

The ternary operator `?:` results in `⟨operand⟩` if `⟨operand⟩` is true (not $\pm 0$), and `⟨operand⟩` if `⟨operand⟩` is false ($\pm 0$). All three `⟨operands⟩` are evaluated in all cases; they may be tuples. The operator is right associative, hence

\[
\fp_eval:n
\{
1 + 3 > 4 ? 1 :
2 + 4 > 5 ? 2 :
3 + 5 > 6 ? 3 : 4
\}
\]

first tests whether $1 + 3 > 4$; since this isn’t true, the branch following `:` is taken, and $2 + 4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

```latex
\fp_eval:n { ⟨operand⟩ || ⟨operand⟩ }
```

If `⟨operand⟩` is true (not $\pm 0$), use that value, otherwise the value of `⟨operand⟩`. Both `⟨operands⟩` are evaluated in all cases; they may be tuples. In `⟨operand⟩ || ⟨operand⟩ || ... || ⟨operands⟩`, the first true (nonzero) `⟨operand⟩` is used and if all are zero the last one ($\pm 0$) is used.

```latex
\fp_eval:n { ⟨operand⟩ && ⟨operand⟩ }
```

If `⟨operand⟩` is false (equal to $\pm 0$), use that value, otherwise the value of `⟨operand⟩`. Both `⟨operands⟩` are evaluated in all cases; they may be tuples. In `⟨operand⟩ && ⟨operand⟩ && ... && ⟨operands⟩`, the first false ($\pm 0$) `⟨operand⟩` is used and if none is zero the last one is used.
Each \(\text{(relation)}\) consists of a non-empty string of \(<\), \(=\), \(>\), optionally preceded by \(!\), and may not start with \(?\). This evaluates to +1 if all comparisons \(\langle\text{operand}\rangle\) \(\langle\text{relation}\rangle\) \(\langle\text{operand}+1\rangle\) are true, and +0 otherwise. All \(\langle\text{operands}\rangle\) are evaluated (once) in all cases. See \\textbf{\faux{fp_compare:nTF}} for details.

- \(\fp_eval:n\) \{ \langle\text{operand}\rangle + \langle\text{operand}2\rangle \}
- \(\fp_eval:n\) \{ \langle\text{operand}\rangle - \langle\text{operand}2\rangle \}
- Computes the sum or the difference of its two \(\langle\text{operands}\rangle\). 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 \text{nan}.

- \(\fp_eval:n\) \{ \langle\text{operand}\rangle \times \langle\text{operand}2\rangle \}
- \(\fp_eval:n\) \{ \langle\text{operand}\rangle / \langle\text{operand}2\rangle \}
- Computes the product or the ratio of its two \(\langle\text{operands}\rangle\). The “invalid operation” exception occurs for \(\infty / \infty\), 0/0, or 0 * \infty. “Division by zero” occurs when dividing a finite non-zero number by ±0. “Underflow” and “overflow” occur when appropriate. When \(\langle\text{operand}\rangle\) is a tuple and \(\langle\text{operand}2\rangle\) is a floating point number, each item of \(\langle\text{operand}\rangle\) is multiplied or divided by \(\langle\text{operand}2\rangle\). Multiplication also supports the case where \(\langle\text{operand}\rangle\) is a floating point number and \(\langle\text{operand}2\rangle\) a tuple. Other combinations yield an “invalid operation” exception and a \text{nan} result.

- \(\fp_eval:n\) \{ + \langle\text{operand}\rangle \}
- \(\fp_eval:n\) \{ - \langle\text{operand}\rangle \}
- \(\fp_eval:n\) \{ ! \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 ±0) and 0 otherwise (this is the \textbf{not} boolean function). Those operations never raise exceptions.

- ** \(\fp_eval:n\) \{ \langle\text{operand}\rangle \times \langle\text{operand}2\rangle \}
- ** \(\fp_eval:n\) \{ \langle\text{operand}\rangle ^\langle\text{operand}2\rangle \}
- Raises \(\langle\text{operand}\rangle\) to the power \(\langle\text{operand}2\rangle\). This operation is right associative, hence 2 ** 2 ** 3 equals \(2^2^3 = 256\). If \(\langle\text{operand}\rangle\) is negative or -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}\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 ±0 to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.

\textbf{abs} \(\fp_eval:n\) \{ 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 \\textbf{\faux{fp_abs:n}}.
\texttt{exp \fp_eval:n \{ exp( \fp expr ) \}}

Computes the exponential of the \texttt{fp expr}. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

\texttt{fact \fp_eval:n \{ fact( \fp expr ) \}}

Computes the factorial of the \texttt{fp expr}. If the \texttt{fp expr} is an integer between \(-0\) and 3248 included, the result is finite and correctly rounded. Larger positive integers give \(+\infty\) with “overflow”, while \texttt{fact}(+\infty) = +\infty and \texttt{fact(nan)} = \texttt{nan} with no exception. All other inputs give \texttt{nan} with the “invalid operation” exception.

\texttt{ln \fp_eval:n \{ ln( \fp expr ) \}}

Computes the natural logarithm of the \texttt{fp expr}. 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.

\texttt{logb \fp_eval:n \{ logb( \fp expr ) \}}

Determines the exponent of the \texttt{fp expr}, 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}.

\texttt{max \fp_eval:n \{ max( \fp expr1 , \fp expr2 , ... ) \}}
\texttt{min \fp_eval:n \{ min( \fp expr1 , \fp expr2 , ... ) \}}

Evaluates each \texttt{fp expr} and computes the largest (smallest) of those. If any of the \texttt{fp expr} 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 \fp_eval:n { round ( \langle \text{fp expr} \rangle ) } \\
trunc \fp_eval:n { round ( \langle \text{fp expr}_1 \rangle , \langle \text{fp expr}_2 \rangle ) } \\
ceil \fp_eval:n { round ( \langle \text{fp expr}_1 \rangle , \langle \text{fp expr}_2 \rangle , \langle \text{fp expr}_3 \rangle ) } \\
floor \\

Only \text{round} accepts a third argument. Evaluates $\langle \text{fp expr}_1 \rangle = x$ and $\langle \text{fp expr}_2 \rangle = n$ and $\langle \text{fp expr}_3 \rangle = t$ then rounds $x$ to $n$ places. If $n$ is an integer, this rounds $x$ to a multiple of $10^{-n}$; if $n = +\infty$, this always yields $x$; if $n = -\infty$, this yields one of $\pm 0$, $\pm \infty$, or $\text{nan}$; if $n = \text{nan}$, this yields $\text{nan}$; if $n$ is neither $\pm \infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle \text{fp expr}_2 \rangle$ is omitted, $n = 0$, i.e., $\langle \text{fp expr}_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.

- \text{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 $\text{nan}$ (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”).

- \text{floor} yields the largest multiple of $10^{-n}$ smaller or equal to $x$ (“round towards negative infinity”);

- \text{ceil} yields the smallest multiple of $10^{-n}$ greater or equal to $x$ (“round towards positive infinity”);

- \text{trunc} yields a multiple of $10^{-n}$ with the same sign as $x$ and with the largest absolute value less than that of $x$ (“round towards zero”).

“Overflow” occurs if $x$ is finite and the result is infinite (this can only happen if $\langle \text{fp expr}_2 \rangle < -9984$). If any operand is a tuple, “invalid operation” occurs.

sign \fp_eval:n { sign( \langle \text{fp expr} \rangle ) } \\

Evaluates the $\langle \text{fp expr} \rangle$ and determines its sign: +1 for positive numbers and for $+\infty$, $-1$ for negative numbers and for $-\infty$, $\pm 0$ for $\pm 0$, and $\text{nan}$ for $\text{nan}$. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

sin \fp_eval:n { \sin( \langle \text{fp expr} \rangle ) } \\
cos \fp_eval:n { \cos( \langle \text{fp expr} \rangle ) } \\
tan \fp_eval:n { \tan( \langle \text{fp expr} \rangle ) } \\
cot \fp_eval:n { \cot( \langle \text{fp expr} \rangle ) } \\
csc \fp_eval:n { \csc( \langle \text{fp expr} \rangle ) } \\
sec \fp_eval:n { \sec( \langle \text{fp expr} \rangle ) } \\

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle \text{fp expr} \rangle$ given in radians. For arguments given in degrees, see \text{sind}, \text{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.
\texttt{sind} \quad \texttt{fp\_eval:n \{ sind\( (fp\ expr) \) \}}
\texttt{cosd} \quad \texttt{fp\_eval:n \{ cosd\( (fp\ expr) \) \}}
\texttt{tand} \quad \texttt{fp\_eval:n \{ tand\( (fp\ expr) \) \}}
\texttt{cotd} \quad \texttt{fp\_eval:n \{ cotd\( (fp\ expr) \) \}}
\texttt{cscd} \quad \texttt{fp\_eval:n \{ cscd\( (fp\ expr) \) \}}
\texttt{secd} \quad \texttt{fp\_eval:n \{ secd\( (fp\ expr) \) \}}

\texttt{Nov: 2013-11-02} Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the \(\langle fp\ expr \rangle\) 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.

\texttt{asind} \quad \texttt{fp\_eval:n \{ asind\( (fp\ expr) \) \}}
\texttt{acosd} \quad \texttt{fp\_eval:n \{ acosd\( (fp\ expr) \) \}}
\texttt{acscd} \quad \texttt{fp\_eval:n \{ acscd\( (fp\ expr) \) \}}
\texttt{asecd} \quad \texttt{fp\_eval:n \{ asecd\( (fp\ expr) \) \}}

\texttt{Nov: 2013-11-02} Computes the arcsine, arccosine, arccosecant, or arcsecant of the \(\langle fp\ expr \rangle\) and returns the result in radians, in the range \([-\pi/2, \pi/2]\) for \texttt{asind} and \texttt{acscd} and \([0, \pi]\) for \texttt{acosd} and \texttt{asecd}. For a result in degrees, use \texttt{asin}, \texttt{etc}. If the argument of \texttt{asind} or \texttt{acosd} lies outside the range \([-1, 1]\), or the argument of \texttt{acscd} or \texttt{asecd} inside the range \((-1, 1)\), an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

\texttt{Nov: 2013-11-02} Computes the arcsine, arccosine, arccosecant, or arcsecant of the \(\langle fp\ expr \rangle\) and returns the result in degrees, in the range \([-90, 90]\) for \texttt{asind} and \texttt{acscd} and \([0, 180]\) for \texttt{acosd} and \texttt{asecd}. For a result in radians, use \texttt{asin}, \texttt{etc}. If the argument of \texttt{asind} or \texttt{acosd} lies outside the range \([-1, 1]\), or the argument of \texttt{acscd} or \texttt{asecd} inside the range \((-1, 1)\), an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
\texttt{atan} \quad \texttt{fp_eval:n \{ atan( \langle fp \ expr \rangle ) \}}
\texttt{acot} \quad \texttt{fp_eval:n \{ acot( \langle fp \ expr \rangle ) \}}
\texttt{New: 2013-11-02}

Those functions yield an angle in radians: \texttt{atan} and \texttt{acot} 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 arctangent computes the angle in polar coordinates of the point with Cartesian coordinates \((\langle fp \ expr_{2} \rangle, \langle fp \ expr_{1} \rangle)\): this is the arctangent of \(\langle fp \ expr_{1} \rangle/\langle fp \ expr_{2} \rangle\), possibly shifted by \(\pi\) depending on the signs of \(\langle fp \ expr_{1} \rangle\) and \(\langle fp \ expr_{2} \rangle\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\langle fp \ expr_{1} \rangle, \langle fp \ expr_{2} \rangle)\), equal to the arccotangent of \(\langle fp \ expr_{1} \rangle/\langle fp \ expr_{2} \rangle\), possibly shifted by \(\pi\). Both two-argument functions take values in the wider range \([-\pi, \pi]\]. The ratio \(\langle fp \ expr_{1} \rangle/\langle fp \ expr_{2} \rangle\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm0\), or when both yield \(\pm\infty\), the resulting angle is one of \(\{\pm\pi/4, \pm3\pi/4\}\) depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

\texttt{atan} \quad \texttt{fp_eval:n \{ atan( \langle fp \ expr \rangle , \langle fp \ expr_{2} \rangle ) \}}
\texttt{acot} \quad \texttt{fp_eval:n \{ acot( \langle fp \ expr \rangle , \langle fp \ expr_{2} \rangle ) \}}
\texttt{New: 2013-11-02}

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 \((\langle fp \ expr_{2} \rangle, \langle fp \ expr_{1} \rangle)\): this is the arctangent of \(\langle fp \ expr_{1} \rangle/\langle fp \ expr_{2} \rangle\), possibly shifted by \(180\) depending on the signs of \(\langle fp \ expr_{1} \rangle\) and \(\langle fp \ expr_{2} \rangle\). The two-argument arccotangent computes the angle in polar coordinates of the point \((\langle fp \ expr_{1} \rangle, \langle fp \ expr_{2} \rangle)\), equal to the arccotangent of \(\langle fp \ expr_{1} \rangle/\langle fp \ expr_{2} \rangle\), possibly shifted by \(180\). Both two-argument functions take values in the wider range \([-180, 180]\]. The ratio \(\langle fp \ expr_{1} \rangle/\langle fp \ expr_{2} \rangle\) need not be defined for the two-argument arctangent: when both expressions yield \(\pm0\), or when both yield \(\pm\infty\), the resulting angle is one of \(\{\pm45, \pm135\}\) depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

\texttt{sqrt} \quad \texttt{fp_eval:n \{ sqrt( \langle fp \ expr \rangle ) \}}
\texttt{New: 2013-12-14}

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 \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 \TeX. The random seed can be queried using \sys_rand_seed: and set using \sys_gset_rand_seed:n.

\TeXhackers\note: This is based on pseudo-random numbers provided by the engine’s primitive \pdfuniformdeviate in pdf\TeX, \pdftex, \up\TeX and \uniformdeviate in Lua\TeX\ and X\TeX. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of “The Art of Computer Programming, Volume 2”.

While we are more careful than \uniformdeviate to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

randint \fp_eval:n \{\randint{\langle\fp\ expr\rangle}\}

\fp_eval:n \{\randint{\langle\fp\ expr\rangle}, \langle\fp\ expr\rangle}\}

New: 2016-12-05

Produces a pseudo-random integer between 1 and \langle\fp\ expr\rangle or between \langle\fp\ expr\rangle and \langle\fp\ expr\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 rand for important comments on how these pseudo-random numbers are generated.

inf The special values \(+\infty\), \(−\infty\), and \(\text{nan}\) are represented as \(\text{inf}\), \(-\text{inf}\) and \(\text{nan}\) (see \c_-\inf_fp, \c_minus_inf_fp and \c_nan_fp).

\Pi The value of \(\pi\) (see \c_p\i\_fp).

\deg The value of 1° in radians (see \c_one_degree_fp).
Those units of measurement are equal to their values in pt, namely

\[
\begin{align*}
1 \text{ in} & = 72.27 \text{ pt} \\
1 \text{ pt} & = 1 \text{ pt} \\
1 \text{ pc} & = 12 \text{ pt} \\
1 \text{ cm} & = \frac{1}{2.54} \text{ in} = 28.45275590551181 \text{ pt} \\
1 \text{ mm} & = \frac{1}{25.4} \text{ in} = 2.845275590551181 \text{ pt} \\
1 \text{ dd} & = 0.376065 \text{ mm} = 1.07000856496063 \text{ pt} \\
1 \text{ cc} & = 12 \text{ dd} = 12.84010277952756 \text{ pt} \\
1 \text{ nd} & = 0.375 \text{ mm} = 1.066978346456693 \text{ pt} \\
1 \text{ sp} & = 2^{-16} \text{ pt} = 1.52587890625 \times 10^{-5} \text{ pt}.
\end{align*}
\]

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

---

true Other names for 1 and +0.
false

\[
\begin{align*}
\text{\texttt{fp\_abs:n \{\langle fp \ expr\rangle\}}} & \quad \text{Evaluates the} \ \langle \text{fp expr}\rangle \text{ as described for} \ \text{\texttt{fp\_eval:n}} \text{ and leaves the absolute value of the result in the input stream. If the argument is} \pm \infty, \text{nan} \text{ or a tuple, “invalid operation” occurs. Within floating point expressions, abs() can be used; it accepts} \pm \infty \text{ and nan as arguments.}
\end{align*}
\]

\[
\begin{align*}
\text{\texttt{fp\_max:nn \{\langle fp \ expr1\rangle\} \{\langle fp \ expr2\rangle\}}} & \quad \text{Evaluates the} \ \langle \text{fp exprs}\rangle \text{ as described for} \ \text{\texttt{fp\_eval:n}} \text{ and leaves the resulting larger (max) or smaller (min) value in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, max() and min() can be used.}
\end{align*}
\]

### 29.13 Disclaimer and roadmap

This module may break 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.
• Support signalling \texttt{nan}.

• Modulo and remainder, and rounding function \texttt{quantize} (and its friends analogous to \texttt{trunc}, \texttt{ceil}, \texttt{floor}).

• \texttt{\fp\_format:nn \{\texttt{fp expr}\} \{\texttt{format}\}}, but what should \texttt{(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 \(x\) in base \(b\).

• \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 \(-\infty\), \texttt{\dim\_to\_fp:n \{0pt\}} should return \(-0\), not \(+0\).

• The result of \((\pm0) + (\pm0)\), of \(x + (\neg x)\), and of \((\neg x) + x\) should depend on the rounding mode.

• \texttt{0e9999999999} gives a \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 29.12.1, write a grammar.
• It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.

• Some functions should get an `_o` ending to indicate that they expand after their result.

• More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.

• The code for the `ternary` set of functions is ugly.

• There are many `-` missing in the doc to avoid bad line-breaks.

• The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking \( c = \frac{2000}{\lceil 200x \rceil + 1} \in [10, 95] \) 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:NNNW` and `\__fp_basics_pack_weird_high:NNNNNNNNWW` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.

• Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.

• Add bibliography. Some of Kahan’s articles, some previous `TEX` fp packages, the international standards,…

• Also take into account the “inexact” exception?

• Support multi-character prefix operators (e.g., `@/` or whatever)?
Chapter 30

The \l3fparray module
Fast global floating point arrays

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast \l3seq, where access time is linear). The interface is very close to that of \l3intarray. The size of the array is fixed and must be given at point of initialisation

30.1 Creating and initialising floating point array variables

\fparray_new:Nn \fparray_new:cn
New: 2018-05-05
\fparray_gzero:N \fparray_gzero:c
New: 2018-05-05

30.2 Adding data to floating point arrays

\fparray_gset:Nnn \fparray_gset:cmn
New: 2018-05-05
Stores the result of evaluating the floating point expression (value) into the (floating point array variable) at the (integer expression) (position). If the (position) is not between 1 and the \fparray_count:N, an error occurs. Assignments are always global.
30.3 Counting entries in floating point arrays

\fparray_count:N \fparray_count:c

Expands to the number of entries in the (floating point array variable). This is performed in constant time.

30.4 Using a single entry

\fparray_item:Nn \fparray_item:cn \fparray_item_to_tl:Nn \fparray_item_to_tl:cn

Applies \fp_use:N or \fp_to_tl:N (respectively) to the floating point entry stored at the (integer expression) (position) in the (floating point array variable). If the (position) is not between 1 and the \fparray_count:N (fparray var), an error occurs.

30.5 Floating point array conditional

\fparray_if_exist_p:N \fparray_if_exist_p:c \fparray_if_exist:NTF \fparray_if_exist:CTF

Tests whether the (fparray var) is currently defined. This does not check that the (fparray var) really is a floating point array variable.
Chapter 31

The \texttt{l3bitset} module

Bitsets

This module defines and implements the data type \texttt{bitset}, a vector of bits. The size of the vector may grow dynamically. Individual bits can be set and unset by names pointing to an index position. The names 1, 2, 3, ... are predeclared and point to the index positions 1, 2, 3, ... More names can be added and existing names can be changed. The index is like all other indices in expl3 modules 1-based. A \texttt{bitset} can be output as binary number or—as needed e.g. in a PDF dictionary—as decimal (arabic) number. Currently only a small subset of the functions provided by the \texttt{bitset} package are implemented here, mainly the functions needed to use bitsets in PDF dictionaries.

The bitset is stored as a string (but one shouldn’t rely on the internal representation) and so the vector size is theoretically unlimited, only restricted by \TeX-memory. But the functions to set and clear bits use integer functions for the index so bitsets can’t be longer than $2^{31} - 1$. The export function \texttt{\textbackslash bitset\_to\_arabic:N} can use functions from the \texttt{int} module only if the largest index used for this bitset is smaller than 32, for longer bitsets \texttt{fp} is used and this is slower.
31.1 Creating bitsets

\bitset_new:N \bitset_new:N \bitset_new:N
\bitset_new:n \bitset_new:Nn
\bitset_new:cn
\bitset_new:cn

New: 2023-11-15

\bitset_addto_named_index:Nn \bitset_addto_named_index:Nn \bitset_addto_named_index:Nn
\bitset_addto_named_index:n \bitset_addto_named_index:Nn

New: 2023-11-15

Creates a new \texttt{bitset var} or raises an error if the name is already taken. The declaration is global. The \texttt{bitset var} is initially 0.

Bitsets are implemented as string variables consisting of 1’s and 0’s. The rightmost number is the index position 1, so the string variable can be viewed directly as the binary number. But one shouldn’t rely on the internal representation, but use the dedicated \texttt{bitset_to_bin:N} instead to get the binary number.

The name–index pairs given in the second argument of \texttt{bitset_new:Nn} declares names for some indices, which can be used to set and unset bits. The names 1, 2, 3, ... are predeclared and point to the index positions 1, 2, 3, ... .

\texttt{(index...) } should be a positive number or an \texttt{integer expression} which evaluates to a positive number. The expression is evaluated when the index is used, not at declaration time. The names \texttt{(name...)} should be unique. Using a number as name, e.g. \texttt{10=1}, is allowed, it then overwrites the predeclared name \texttt{10}, but the index position \texttt{10} can then only be reached if some other name for it exists, e.g. \texttt{ten=10}. It is not necessary to give every index a name, and an index can have more than one name. The named index can be extended or changed with the next function.

For example after these settings

\texttt{\bitset_new:Nn \l_pdfannot_F_bitset }
\{ 
  \texttt{Invisible = 1,}
  \texttt{Hidden = 2,}
  \texttt{Print = 3,}
  \texttt{NoZoom = 4,}
  \texttt{NoRotate = 5,}
  \texttt{NoView = 6,}
  \texttt{ReadOnly = 7,}
  \texttt{Locked = 8,}
  \texttt{ToggleNoView = 9,}
  \texttt{LockedContents = 10}
\}

\texttt{\bitset_addto_named_index:Nn \l_pdfannot_F_bitset }
\{ 
\}

285
it is possible to set 3 by using any of these alternatives:

\bitset_set_true:Nn \l_pdfannot_F_bitset {Print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {3}

\bitset_if_exist_p:N ⟨bitset var⟩
\bitset_if_exist:NTF ⟨bitset var⟩ {(true code)} {(false code)}
Tests whether the ⟨bitset var⟩ exist.

31.2 Setting and unsetting bits

\bitset_set_true:Nn ⟨bitset var⟩ {(name)}
This sets the bit of the index position represented by ⟨(name)⟩ to 1. ⟨name⟩ should be either one of the predeclared names 1, 2, 3, ..., or one of the names added manually. Index position are 1-based. If needed the length of the bit vector is enlarged.

\bitset_set_false:Nn ⟨bitset var⟩ {(name)}
This unsets the bit of the index position represented by ⟨(name)⟩ (sets it to 0). ⟨name⟩ should be either one of the predeclared names 1, 2, 3, ..., or one of the names added manually. The index is 1-based. If the index position is larger than the current length of the bit vector nothing happens. If the leading (left most) bit is unset, zeros are not trimmed but stay in the bit vector and are still shown by \bitset_show:N.

\bitset_clear:N ⟨bitset var⟩
This resets the bitset to the initial state. The declared names are not changed.

31.3 Using bitsets

\bitset_item:Nn ⟨bitset var⟩ ⟨(name)⟩
\bitset_item:cn
\bitset_gitem:Nn
\bitset_gitem:cn
\bitset_item:Nn outputs 1 if the bit with the index number represented by ⟨(name)⟩ is set and 0 otherwise. ⟨name⟩ is either one of the predeclared names 1, 2, 3, ..., or one of the names added manually.
\bitset_to_bin:N \bitset_to_bin:c * \bitset_to_bin:N \bitset_to_bin:c * New: 2023-11-15
This leaves the current value of the bitset expressed as a binary (string) number in the input stream. If no bit has been set yet, the output is zero.

\bitset_to_arabic:N \bitset_to_arabic:c * \bitset_to_arabic:N \bitset_to_arabic:c * New: 2023-11-15
This leaves the current value of the bitset expressed as a decimal number in the input stream. If no bit has been set yet, the output is zero. The function uses \int_from_bin:n if the largest index that have been set or unset is smaller than 32, and a slower implementation based on \fp_eval:n otherwise.

\bitset_show:N \bitset_show:c \bitset_show:N \bitset_show:c New: 2023-11-15
Displays the binary and decimal values of the (bitset var) on the terminal.

\bitset_log:N \bitset_log:c \bitset_log:N \bitset_log:c New: 2023-11-15
Writes the binary and decimal values of the (bitset var) in the log file.

\bitset_show_named_index:N \bitset_show_named_index:c \bitset_show_named_index:N \bitset_show_named_index:c New: 2023-11-15
Displays declared name–index pairs of the (bitset var) on the terminal.

\bitset_log_named_index:N \bitset_log_named_index:c \bitset_log_named_index:N \bitset_log_named_index:c New: 2023-12-11
Writes declared name–index pairs of the (bitset var) in the log file.
Chapter 32

The \texttt{l3ctab} module

Category code tables

A category code table enables rapid switching of all category codes in one operation. For \LaTeX{}, 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 \texttt{expl3} also saves and restores the \TeX{} \texttt{endlinechar} primitive value, meaning they could be used for example to implement \texttt{\ExplSyntaxOn}.

32.1 Creating and initialising category code tables

\begin{verbatim}
\cctab_new:N \cctab_new:Nn \cctab_new:c
\cctab_const:Nn \cctab_const:cn \cctab_const:N \cctab_const:cn
\cctab_gset:Nn \cctab_gset:cn \cctab_gset:N \cctab_gset:cn
\cctab_gsave_current:N \cctab_gsave_current:c
\end{verbatim}

\begin{itemize}
\item \cctab_new:N \cctab_new:Nn \cctab_new:c
\item \cctab_const:Nn \cctab_const:cn
\item \cctab_gset:Nn \cctab_gset:cn
\item \cctab_gsave_current:N \cctab_gsave_current:c
\end{itemize}

\begin{itemize}
\item Creates a new \texttt{category code table} \variable{\texttt{\cctab_new:N \cctab_new:Nn \cctab_new:c}}\texttt{\cctab_new:Nn \cctab_new:c} variable or raises an error if the name is already taken. The declaration is global. The \texttt{\cctab_new:Nn \cctab_new:c} is initialised with the codes as used by \texttt{iniTeX}.
\item \cctab_const:Nn \cctab_const:cn
\item \cctab_gset:Nn \cctab_gset:cn
\item \cctab_gsave_current:N \cctab_gsave_current:c
\end{itemize}

Updated: 2020-07-02

Updated: 2020-07-07

Updated: 2023-05-26

\begin{itemize}
\item \cctab_gset:cn
\item \cctab_gsave_current:c
\end{itemize}

\begin{verbatim}
\end{verbatim}

\begin{itemize}
\item Saves the current prevailing category codes in the \texttt{\cctab_gsave_current:N \cctab_gsave_current:c}.\texttt{\cctab_gsave_current:c}
\end{itemize}
32.2 Using category code tables

\cctab_begin:N \cctab_begin:c
Switches locally the category codes in force to those stored in the \textit{category code table}. The prevailing codes before the function is called are added to a stack, for use with \cctab_end:. This function does not start a \TeX{} group.

\cctab_end:
Ends the scope of a \textit{category code table} started using \cctab_begin:N, returning the codes to those in force before the matching \cctab_begin:N was used. This must be used within the same \TeX{} group (and at the same \TeX{} group level) as the matching \cctab_begin:N.

\cctab_select:N \cctab_select:c
Selects the \textit{category code table} for the scope of the current group. This is in particular useful in the \textit{setup} arguments of \tl_set_rescan:Nnn, \tl_rescan:nn, \cctab_const:Nn, and \cctab_gset:Nn.

\cctab_item:Nn \cctab_item:cn
Determines the \textit{character} with character code given by the \textit{int expr} and expands to its category code specified by the \textit{category code table}.

32.3 Category code table conditionals

\cctab_if_exist_p:N \cctab_if_exist_p:c \cctab_if_exist:N \cctab_if_exist:c
Tests whether the \textit{category code table} is currently defined. This does not check that the \textit{category code table} really is a category code table.

32.4 Constant and scratch category code tables

\c_code_cctab
Category code table for the expl3 code environment; this does not include "\text{" which is retained as an “other” character. Sets the \texttt{\endlinechar} value to 32 (a space).

\c_document_cctab
Category code table for a standard \LaTeX{} document, as set by the \LaTeX{} kernel. In particular, the upper-half of the 8-bit range will be set to “active” with pdf\LaTeX{} only. No babel shorthands will be activated. Sets the \texttt{\endlinechar} value to 13 (normal line ending).
\c_initex_cctab
Updated: 2020-07-02

Category code table as set up by \texttt{initex}.

\c_other_cctab
Updated: 2020-07-02

Category code table where all characters have category code 12 (other). Sets the \texttt{endlinechar} value to $-1$.

\c_str_cctab
Updated: 2020-07-02

Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space). Sets the \texttt{endlinechar} value to $-1$.

\g_tmpa_cctab
\g_tmpb_cctab

Scratch category code tables.

New: 2023-05-26
Part V
Text manipulation
Chapter 33

The \texttt{l3unicode} module
Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. Most of the code here is internal, but there are a small set of public functions. These work with Unicode (\texttt{codepoints}) and are designed to give usable results with both Unicode-aware and 8-bit engines.
\codepoint_generate:nn \codepoint_generate:nn \langle\textit{codepoint}\rangle \{\langle\textit{catcode}\rangle\}

Generates one or more character tokens representing the \langle\textit{codepoint}\rangle. With Unicode engines, exactly one character token will be generated, and this will have the \langle\textit{catcode}\rangle specified as the second argument:

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the \langle\textit{codepoint}\rangle. For all codepoints outside of the classical ASCII range, the generated character tokens will be active (category code 13); for codepoints in the ASCII range, the given \langle\textit{catcode}\rangle will be used. To allow the result of this function to be used inside an expansion context, the result is protected by \exp_not:n.

\textbf{\TeX hacklers 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 \texttt{\textbackslash kcatcode} of the \langle\textit{codepoint}\rangle, any value outside the ASCII range will result in a series of active bytes being generated.

\codepoint_str_generate:n \codepoint_str_generate:n \langle\textit{codepoint}\rangle

Generates one or more character tokens representing the \langle\textit{codepoint}\rangle. With Unicode engines, exactly one character token will be generated. For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the \langle\textit{codepoint}\rangle. All of the generated character tokens will be of category code 12, except any spaces (codepoint 32), which will be category code 10.
\codepoint_to_category:n \codepoint_to_category:n \{(codepoint)\}

Expands to the Unicode general category identifier of the \codepoint. The general
category identifier is a string made up of two letter characters, the first uppercase and
the second lowercase. The uppercase letters divide codepoints into broader groups, which
are then refined by the lowercase letter. For example, codepoints representing letters all
have identifiers starting L, for example Lu (uppercase letter), Lt (titlecase letter), etc.
Full details are available in the documentation provided by the Unicode Consortium: see
https://www.unicode.org/reports/tr44/#General_Category_Values

\codepoint_to_nfd:n \codepoint_to_nfd:n \{(codepoint)\}

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 34

The \text module
Text processing

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the \text are normalized and become \{ and \}, respectively.

34.1 Expanding text

\textexpand:n \textexpand:n \{\text\}

Takes user input \text and expands the content. Protected commands (typically formatting) are left in place, and no processing takes place of math mode material (as delimited by pairs given in \ltext_math_delims_tl or as the argument to commands listed in \ltext_math_arg_tl). Commands which are neither engine- nor \LaTeX protected are expanded exhaustively. Any commands listed in \ltextexpand_exclude_tl are excluded from expansion, as are those in \ltext_case_exclude_arg_tl and \ltext_math_arg_tl.

\textdeclareexpandequivalent:Nn \textdeclareexpandequivalent:Nn \textdeclareexpandequivalent:cn

Declares that the \replacement tokens should be used whenever the \cmd (a single token) is encountered. The \replacement tokens should be expandable. A token can be “replaced” by itself if the defined replacement wraps it in \expnot:n, for example

\textdeclareexpandequivalent:Nn \textdeclareexpandequivalent:Nn \textdeclareexpandequivalent:cn
34.2 Case changing

\text_lowercase:n  *
\text_uppercase:n  {{(tokens)}}
\text_titlecase_all:n  *
\text_titlecase_first:n  *
\text_lowercase:nn  *
\text_uppercase:nn  *
\text_titlecase_all:nn  *
\text_titlecase_first:nn  *

* Takes user input ⟨text⟩ first applies \text_expand:n, then transforms the case of character tokens as specified by the function name. The category code of letters are not changed by this process when Unicode engines are used; in 8-bit engines, case changed charters in the ASCII range will have the current prevailing category code, while those outside of it will be represented by active characters.

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first character of the ⟨tokens⟩ to uppercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example 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 \lstr_lowercase:n, \lstr_uppercase:n and \lstr_casefold:n.

Case changing does not take place within math mode material so for example

\text_uppercase:n { Some-text-$y = mx + c$-with-{Braces} }

becomes

\text_uppercase:n \{ Some-text-$y = mx + c$ WITH {BRACES} \}

The first mandatory argument of commands listed in \l_text_case_exclude_arg-tl is excluded from case changing; the latter are entirely non-textual content (such as labels).

The standard mappings here follow those defined by the Unicode Consortium in UnicodeData.txt and SpecialCasing.txt. For \pTeX, only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Locale-sensitive conversions are enabled using the ⟨BCP-47⟩ argument, and follow Unicode Consortium guidelines. Currently, the locale strings recognized for special handling are as follows.

- Armenian (hy and hy-x-yiwn) The setting hy maps the codepoint U+0587, the ligature of letters ech and yiw, to the codepoints for capital ech and yew when upercasing: this follows the spelling reform which is used in Armenia. The alternative hy-x-yiwn maps U+0587 to capital ech and yiw on upercasing (also the output if Armenian is not selected at all).

- Azeri and Turkish (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 ypoge-grammaeni (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-mediaval). 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 \text{\textit{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 \text{\textit{text-titlecase-check-letter-bool}} is false, the first codepoint is uppercased, irrespective of the general code of the character.

\begin{verbatim}
\text{\textit{text-declare-case-equivalent:Nn}} \{\langle cmd \rangle \{\langle replacement \rangle\}}
\end{verbatim}

Declares that the \langle replacement \rangle tokens should be used whenever the \langle cmd \rangle (a single token) is encountered during case changing.

\begin{verbatim}
\text{\textit{text-declare-lowercase-mapping:nn}} \{\langle codepoint \rangle \{\langle replacement \rangle\}}
\text{\textit{text-declare-lowercase-mapping:nnn}} \{\langle BCP-47 \rangle \{\langle codepoint \rangle\}}
\text{\textit{text-declare-titlecase-mapping:nn}} \{\langle replacement \rangle\}
\text{\textit{text-declare-titlecase-mapping:nnn}}
\text{\textit{text-declare-uppercase-mapping:nn}}
\text{\textit{text-declare-uppercase-mapping:nnn}}
\end{verbatim}

Declares that the \langle replacement \rangle tokens should be used when case mapping the \langle codepoint \rangle, 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.

\begin{verbatim}
\text{\textit{text-case-switch:nnn \{normal\} \{upper\} \{lower\} \{title\}}
\end{verbatim}

Context-sensitive function \langle normal \rangle, \langle upper \rangle, \langle lower \rangle or \langle title \rangle tokens depending on the current case changing operation. Outside of case changing, the \langle normal \rangle tokens are produced. Within case changing, the appropriate mapping tokens are inserted.
34.3 Removing formatting from text

\text_purify:n \text_purify:n \{(\text)\}

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:Ne

Declares that the \text{} tokens should be used whenever the \text{cmd} (a single token) is encountered. The \text{} tokens should be expandable.

34.4 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.
34.5 Mapping to graphemes

Grapheme splitting is implemented using the algorithm described in Unicode Standard Annex #29. This includes support for extended grapheme clusters. Text starting with a line feed or carriage return character will drop this due to standard TEX processing. At present extended pictograms are not supported: these may be added in a future release.

\text_map_function:nN \star \text_map_function:nN \{\text{function}\}  
\text{New: 2022-08-04}  
Takes user input \text{text} and expands as described for \text\text_expand:n, then maps over the graphemes within the result, passing each grapheme to the \text{function}. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The \text{function} should accept one argument as \text{balanced text}: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also \text_map_inline:nn.

\text_map_inline:nn \text_map_inline:nn \{\text{inline function}\}  
\text{New: 2022-08-04}  
Takes user input \text{text} and expands as described for \text\text_expand:n, then maps over the graphemes within the result, passing each grapheme to the \text{inline function}. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The \text{inline function} should consist of code which receives the grapheme as \text{balanced text}: 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:  \star \text_map_break:  \star \text_map_break:n \{\text{code}\}  
\text{New: 2022-08-04}  
Used to terminate a \text_map... function before all entries in the \text{text} have been processed. This normally takes place within a conditional statement.
Part VI
Typesetting
Chapter 35

The l3box module

Boxes

Box variables contain typeset material that can be inserted on the page or in other boxes. Their contents cannot be converted back to lists of tokens. There are three kinds of box operations: horizontal mode denoted with prefix \hbox_, vertical mode with prefix \vbox_, and the generic operations working in both modes with prefix \box_. For instance, a new box variable containing the words “Hello, world!” (in a horizontal box) can be obtained by the following code.

\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, ~ world! }

The argument is typeset inside a TEX group so that any variables assigned during the construction of this box restores its value afterwards.

Box variables from l3box are compatible with those of LaTeX2ε and plain TEX and can be used interchangeably. The l3box commands to construct boxes, such as \hbox:n or \hbox_set:Nn, are “color-safe”, meaning that

\hbox:n { \color_select:n { blue } Hello, } ~ world!

will result in “Hello,” taking the color blue, but “world!” remaining with the prevailing color outside the box.

35.1 Creating and initialising boxes

\box_new:N \box_new:c
\box_clear:N \box_clear:c \box_gclear:N \box_gclear:c

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}. 
\box_clear_new:N
\box_clear_new:c
\box_gclear_new:N
\box_gclear_new:c

\box_set_eq:NN
\box_gset_eq:NN

\box_if_exist_p:N
\box_gif_exist:NTF
\box_if_exist:N
\box_gif_exist:c

\copy

\box_use:N \box_use:c

\box_move_right:nn \box_move_left:nn

\box_move_up:nn \box_move_down:nn

35.2 Using boxes

\box_use:N \box_use:c

\box_move_right:nn \box_move_left:nn

\box_move_up:nn \box_move_down:nn

Ensures that the \texttt{box} exists globally by applying \texttt{\box_new:N} if necessary, then applies \texttt{\box_\texttt{(g)clear:N}} to leave the \texttt{box} empty.

Sets the content of \texttt{box\textsubscript{1}} equal to that of \texttt{box\textsubscript{2}}.

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

\TeX{}hackers note: This is the \TeX{} primitive \texttt{\copy}.

This function operates in vertical mode, and inserts the material specified by the \texttt{box function} such that its reference point is displaced horizontally by the given \texttt{dim expr} from the reference point for typesetting, to the right or left as appropriate. The \texttt{box function} should be a box operation such as \texttt{\box_use:N \texttt{\box}} or a “raw” box specification such as \texttt{\vbox:n \{ xyz \}}.

This function operates in horizontal mode, and inserts the material specified by the \texttt{box function} such that its reference point is displaced vertically by the given \texttt{dim expr} from the reference point for typesetting, up or down as appropriate. The \texttt{box function} should be a box operation such as \texttt{\box_use:N \texttt{\box}} or a “raw” box specification such as \texttt{\vbox:n \{ xyz \}}.
35.3 Measuring and setting box dimensions

\box_dp:N \box_dp:N \langle box \rangle
Calculates the depth (below the baseline) of the \langle box \rangle in a form suitable for use in a \langle dim expr \rangle.

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

\box_ht:N \box_ht:N \langle box \rangle
Calculates the height (above the baseline) of the \langle box \rangle in a form suitable for use in a \langle dim expr \rangle.

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

\box_wd:N \box_wd:N \langle box \rangle
Calculates the width of the \langle box \rangle in a form suitable for use in a \langle dim expr \rangle.

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

\box_ht_plus_dp:N \box_ht_plus_dp:N \langle box \rangle
Calculates the total vertical size (height plus depth) of the \langle box \rangle in a form suitable for use in a \langle dim expr \rangle.

\box_set_dp:Nn \box_set_dp:cn \langle box \rangle \{ \langle \texttt{dim expr} \rangle \}
Set the depth (below the baseline) of the \langle box \rangle to the value of the \{ \langle \texttt{dim expr} \rangle \}.

\texttt{Updated: 2019-01-22}

\box_set_ht:Nn \box_set_ht:cn \langle box \rangle \{ \langle \texttt{dim expr} \rangle \}
Set the height (above the baseline) of the \langle box \rangle to the value of the \{ \langle \texttt{dim expr} \rangle \}.

\texttt{Updated: 2019-01-22}

\box_set_wd:Nn \box_set_wd:cn \langle box \rangle \{ \langle \texttt{dim expr} \rangle \}
Set the width of the \langle box \rangle to the value of the \{ \langle \texttt{dim expr} \rangle \}.

\texttt{Updated: 2019-01-22}
### 35.4 Box conditionals

\( \texttt{\textbackslash box\_if\_empty\_p:N} \)  
\( \texttt{\textbackslash box\_if\_empty\_p:c} \)  
\( \texttt{\textbackslash box\_if\_empty:NTF} \)  
\( \texttt{\textbackslash box\_if\_empty:cTF} \)  
Tests if \( \langle \text{box} \rangle \) is an empty (equal to \texttt{\textbackslash c\_empty\_box}).

\( \texttt{\textbackslash box\_if\_horizontal\_p:N} \)  
\( \texttt{\textbackslash box\_if\_horizontal\_p:c} \)  
\( \texttt{\textbackslash box\_if\_horizontal:NTF} \)  
\( \texttt{\textbackslash box\_if\_horizontal:cTF} \)  
Tests if \( \langle \text{box} \rangle \) is a horizontal box.

\( \texttt{\textbackslash box\_if\_vertical\_p:N} \)  
\( \texttt{\textbackslash box\_if\_vertical\_p:c} \)  
\( \texttt{\textbackslash box\_if\_vertical:NTF} \)  
\( \texttt{\textbackslash box\_if\_vertical:cTF} \)  
Tests if \( \langle \text{box} \rangle \) is a vertical box.

### 35.5 The last box inserted

\( \texttt{\textbackslash box\_set\_to\_last:N} \)  
\( \texttt{\textbackslash box\_set\_to\_last:c} \)  
\( \texttt{\textbackslash box\_gset\_to\_last:N} \)  
\( \texttt{\textbackslash box\_gset\_to\_last:c} \)  
Sets the \( \langle \text{box} \rangle \) equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the \( \langle \text{box} \rangle \) is always void as it is not possible to recover the last added item.

### 35.6 Constant boxes

\( \texttt{\textbackslash c\_empty\_box} \)  
This is a permanently empty box, which is neither set as horizontal nor vertical.

\textbf{\texttt{\textbackslash \textnormal{T\enquote{E}X}hackers note:}} At the \texttt{\textnormal{T\enquote{E}X}} level this is a void box.

### 35.7 Scratch boxes

\( \texttt{\textbackslash l\_tmpa\_box} \)  
\( \texttt{\textbackslash l\_tmpb\_box} \)  
Updated: 2012-11-04

Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any \texttt{\textnormal{E\enquote{T}P\enquote{X}3}}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\( \texttt{\textbackslash g\_tmpa\_box} \)  
\( \texttt{\textbackslash g\_tmpb\_box} \)  
Updated: 2012-11-04

Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any \texttt{\textnormal{E\enquote{T}P\enquote{X}3}}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
35.8 Viewing box contents

\box_show:N  \box_show:N  ⟨box⟩
Shows full details of the content of the ⟨box⟩ in the terminal.

\box_show:N  \box_show:c
Updated: 2012-05-11

\box_show:N  \box_show:Nnn  ⟨box⟩  {⟨int expr⟩}  {⟨int expr⟩}
Display the contents of ⟨box⟩ in the terminal, showing the first ⟨int expr⟩ items of the box, and descending into ⟨int expr⟩ group levels.

\box_log:N  \box_log:N  ⟨box⟩
Updated: 2012-05-11

\box_log:N  \box_log:Nnn  ⟨box⟩  {⟨int expr⟩}  {⟨int expr⟩}
Writes the contents of ⟨box⟩ to the log, showing the first ⟨int expr⟩ items of the box, and descending into ⟨int expr⟩ group levels.

35.9 Boxes and color

All \emph{LaTeX}3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

35.10 Horizontal mode boxes

\ hbox:n  \ hbox:n  ⟨contents⟩
Typesets the ⟨contents⟩ into a horizontal box of natural width and then includes this box in the current list for typesetting.

\ hbox_to_wd:nn  \ hbox_to_wd:nn  ⟨dim expr⟩  ⟨contents⟩
Typesets the ⟨contents⟩ into a horizontal box of width ⟨dim expr⟩ and then includes this box in the current list for typesetting.

\ hbox_to_zero:n  \ hbox_to_zero:n  ⟨contents⟩
Typesets the ⟨contents⟩ into a horizontal box of zero width and then includes this box in the current list for typesetting.

\ hbox_set:Nn  \ hbox_set:cn  \ hbox_gset:Nn  \ hbox_gset:cn
Typesets the ⟨contents⟩ at natural width and then stores the result inside the ⟨box⟩.
\hbox_set_to_wd:Nnn \hbox_set_to_wd:cnn \hbox_gset_to_wd:Nnn \hbox_gset_to_wd:cnn

Updated: 2017-04-05

\hbox_overlap_center:n \hbox_overlap_center:n \hbox_overlap_right:n \hbox_overlap_left:n

Updated: 2017-04-05

\hbox_set:Nw \hbox_set:cw \hbox_set_end:
\hbox_set:Nw \hbox_set:cw \hbox_set_end:
\hbox_set:Nw \hbox_set:cw \hbox_set_end:
\hbox_set:Nw \hbox_set:cw \hbox_set_end:

Updated: 2017-06-08

\hbox_set_to_wd:Nnw \hbox_set_to_wd:cnw \hbox_gset_to_wd:Nnw \hbox_gset_to_wd:cnw

New: 2017-06-08

\hbox_unpack:N \hbox_unpack:c

Unpacks the content of the horizontal \texttt{box}, retaining any stretching or shrinking applied when the \texttt{box} was set.

\TeXhackers note: This is the \TeX{} primitive \texttt{\unhcopy}.

\section*{35.11 Vertical mode boxes}

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are
top boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

\vbox:n \{ contents \}
Updated: 2017-04-05
Typesets the \( \langle \text{contents} \rangle \) into a vertical box of natural height and includes this box in the current list for typesetting.

\vbox_top:n \{ contents \}
Updated: 2017-04-05
Typesets the \( \langle \text{contents} \rangle \) into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the first item added to the box.

\vbox_to_ht:nn \{ dim expr \} \{ contents \}
Updated: 2017-04-05
Typesets the \( \langle \text{contents} \rangle \) into a vertical box of height \( \langle \text{dim expr} \rangle \) and then includes this box in the current list for typesetting.

\vbox_to_zero:n \{ contents \}
Updated: 2017-04-05
Typesets the \( \langle \text{contents} \rangle \) into a vertical box of zero height and then includes this box in the current list for typesetting.

\vbox_set:Nn \langle box \rangle \{ contents \}
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn
Updated: 2017-04-05
Typesets the \( \langle \text{contents} \rangle \) at natural height and then stores the result inside the \( \langle \text{box} \rangle \).

\vbox_set_top:Nn \langle box \rangle \{ contents \}
\vbox_set_top:cn
\vbox_gset_top:Nn
\vbox_gset_top:cn
Updated: 2017-04-05
Typesets the \( \langle \text{contents} \rangle \) at natural height and then stores the result inside the \( \langle \text{box} \rangle \). The baseline of the box is equal to that of the first item added to the box.

\vbox_set_to_ht:Nnn \langle box \rangle \{ dim expr \} \{ contents \}
\vbox_set_to_ht:cn
\vbox_gset_to_ht:Nn
\vbox_gset_to_ht:cn
Updated: 2017-04-05
Typesets the \( \langle \text{contents} \rangle \) to the height given by the \( \langle \text{dim expr} \rangle \) and then stores the result inside the \( \langle \text{box} \rangle \).

307
\vbox_set_to_ht:Nnw \vbox_set_to_ht:cnw \vbox_gset_to_ht:Nnw \vbox_gset_to_ht:cnw

New: 2017-06-08

\vbox_set_split_to_ht:NNn \vbox_set_split_to_ht:NNn \vbox_set_split_to_ht:NNn \vbox_gset_split_to_ht:NNn \vbox_gset_split_to_ht:NNn

Updated: 2018-12-29

Sets \(\text{box}_1\) to contain material to the height given by the \(\dim\) expr by removing content from the top of \(\text{box}_2\) (which must be a vertical box).

\vbox_unpack:N \vbox_unpack:c

Unpacks the content of the vertical \(\text{box}\), retaining any stretching or shrinking applied when the \(\text{box}\) was set.

\TeX{}hackers note: This is the \TeX{} primitive \texttt{unvcopy}.

### 35.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other 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 \textit{where the box was originally set} rather than necessarily \textit{at the current group level}. For example, with

\begin{verbatim}
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
  \hbox_set:Nn \l_tmpa_box { B }
  \group_begin:
    \box_use_drop:N \l_tmpa_box
  \group_end:
  \box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
\end{verbatim}

the first use of \texttt{\box\_show:N} will show an entirely cleared (void) box, and the second will show the letter \texttt{A} in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of drop functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.
\texttt{\box_use_drop:N} \texttt{\box_use_drop:c}

Inserts the current content of the \texttt{\box} onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

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

\texttt{\box_set_eq_drop:NN} \texttt{\box_set_eq_drop:(cN|Nc|cc)}

Sets the content of \texttt{\{box\}_1} equal to that of \texttt{\{box\}_2}, then drops \texttt{\{box\}_2}.

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

\texttt{\hbox_unpack_drop:N} \texttt{\hbox_unpack_drop:c}

Unpacks the content of the horizontal \texttt{\box}, retaining any stretching or shrinking applied when the \texttt{\box} was set. The original \texttt{\box} is then dropped.

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

\texttt{\vbox_unpack_drop:N} \texttt{\vbox_unpack_drop:c}

Unpacks the content of the vertical \texttt{\box}, retaining any stretching or shrinking applied when the \texttt{\box} was set. The original \texttt{\box} is then dropped.

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

\section*{35.13 Affine transformations}

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in \TeX\ by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.
Resizes the ⟨box⟩ to fit within the given ⟨x-size⟩ (horizontally) and ⟨y-size⟩ (vertically); both of the sizes are dimension expressions. The ⟨y-size⟩ is the height only: it does not include any depth. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. The final size of the ⟨box⟩ is the smaller of \{⟨x-size⟩\} and \{⟨y-size⟩\}, i.e. the result fits within the dimensions specified. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and vice versa.

Resizes the ⟨box⟩ to fit within the given ⟨x-size⟩ (horizontally) and ⟨y-size⟩ (vertically); both of the sizes are dimension expressions. The ⟨y-size⟩ is the total vertical size (height plus depth). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. The final size of the ⟨box⟩ is the smaller of \{⟨x-size⟩\} and \{⟨y-size⟩\}, i.e. the result fits within the dimensions specified. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and vice versa.

Resizes the ⟨box⟩ to ⟨y-size⟩ (vertically), scaling the horizontal size by the same amount; ⟨y-size⟩ is a dimension expression. The ⟨y-size⟩ is the height only: it does not include any depth. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative ⟨y-size⟩ causes the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and vice versa.
Resizes the ⟨box⟩ to (y-size) (vertically), scaling the horizontal size by the same amount: (y-size) is a dimension expression. The (y-size) is the total vertical size (height plus depth). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative (y-size) causes the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative (y-size) results in the ⟨box⟩ having a depth dependent on the height of the original and vice versa.

Resizes the ⟨box⟩ to (x-size) (horizontally), scaling the vertical size by the same amount: (x-size) is a dimension expression. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative (x-size) causes the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative (x-size) results in the ⟨box⟩ having a depth dependent on the height of the original and vice versa.

Resizes the ⟨box⟩ to (x-size) (horizontally) and (y-size) (vertically): both of the sizes are dimension expressions. The (y-size) is the height only and does not include any depth. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative (y-size) results in the ⟨box⟩ having a depth dependent on the height of the original and vice versa.

Resizes the ⟨box⟩ to (x-size) (horizontally) and (y-size) (vertically): both of the sizes are dimension expressions. The (y-size) is the total vertical size (height plus depth). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative (y-size) results in the ⟨box⟩ having a depth dependent on the height of the original and vice versa.
\box_rotate:NNn \box_rotate:cn \box_grotate:NNn \box_grotate:cn

Rotates the \texttt{⟨box⟩} by \langle angle⟩ (a \texttt{fp expr} in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated \texttt{⟨box⟩} is an \texttt{hbox}, irrespective of the nature of the \texttt{⟨box⟩} before the rotation is applied.

\box_scale:NNn \box_scale:cn \box_gscale:NNn \box_gscale:cn

Scales the \texttt{⟨box⟩} by factors \langle x-scale⟩ and \langle y-scale⟩ in the horizontal and vertical directions, respectively (both scales are \texttt{fp expr}). The updated \texttt{⟨box⟩} is an \texttt{hbox}, irrespective of the nature of the \texttt{⟨box⟩} before the scaling is applied. Negative scalings 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 \langle y-scale⟩ results in the \texttt{⟨box⟩} having a depth dependent on the height of the original and \textit{vice versa}.

\box_set_clipped:N \box_set_clipped:c \box_gset_clipped:N \box_gset_clipped:c

Clips the \texttt{⟨box⟩} in the output so that only material inside the bounding box is displayed in the output. The updated \texttt{⟨box⟩} is an \texttt{hbox}, irrespective of the nature of the \texttt{⟨box⟩} before the clipping is applied. Additional box levels are also generated by this operation.

\TeXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

\box_set_trim:NNnn \box_set_trim:cnnn \box_gset_trim:NNnn \box_gset_trim:cnnn

Adjusts the bounding box of the \texttt{⟨box⟩} \langle left⟩ is removed from the left-hand edge of the bounding box, \langle right⟩ from the right-hand edge and so forth. All adjustments are \texttt{dim exprs}. Material outside of the bounding box is still displayed in the output unless \texttt{\box_set_clipped:N} is subsequently applied. The updated \texttt{⟨box⟩} is an \texttt{hbox}, irrespective of the nature of the \texttt{⟨box⟩} before the trim operation is applied. Additional box levels are also generated by this operation. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

\box_set_viewport:NNnn \box_set_viewport:cnnn \box_gset_viewport:NNnn \box_gset_viewport:cnnn

Adjusts the bounding box of the \texttt{⟨box⟩} such that it has lower-left coordinates \langle llx⟩, \langle lly⟩ and upper-right coordinates \langle urx⟩, \langle ury⟩). All four coordinate positions are \texttt{dim exprs}. Material outside of the bounding box is still displayed in the output unless \texttt{\box_set_clipped:N} is subsequently applied. The updated \texttt{⟨box⟩} is an \texttt{hbox}, irrespective of the nature of the \texttt{⟨box⟩} before the viewport operation is applied. Additional box levels are also generated by this operation.

35.14 Viewing part of a box

\textbf{\TeXhackers note:}
35.15 Primitive box conditionals

\if_hbox:N ∗ \if_hbox:N \langle box⟩
  \{true code\}
\else:
  \{false code\}
\fi:
Tests is \langle box⟩ is a horizontal box.

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

\if_vbox:N ∗ \if_vbox:N \langle box⟩
  \{true code\}
\else:
  \{false code\}
\fi:
Tests is \langle box⟩ is a vertical box.

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

\if_box_empty:N ∗ \if_box_empty:N \langle box⟩
  \{true code\}
\else:
  \{false code\}
\fi:
Tests is \langle box⟩ is an empty (void) box.

\TeXhackers note: This is the \TeX primitive \ifvoid.
Chapter 36

The l3coffins module
Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

36.1 Creating and initialising coffins

\coffin_new:N \coffin_new:c
New: 2011-08-17

\coffin_new:N \coffin_new:c
Creates a new \texttt{coffin} or raises an error if the name is already taken. The declaration is global. The \texttt{coffin} is initially empty.

\coffin_clear:N \coffin_clear:c \coffin_gclear:N \coffin_gclear:c
New: 2011-08-17
Updated: 2019-01-21

\coffin_clear:N \coffin_clear:c \coffin_gclear:N \coffin_gclear:c
Clears the content of the \texttt{coffin}.

\coffin_set_eq:NN \coffin_set_eq:\texttt{(Nc|cN|cc)} \coffin_gset_eq:NN \coffin_gset_eq:\texttt{(Nc|cN|cc)}
New: 2011-08-17
Updated: 2019-01-21

\coffin_set_eq:NN \coffin_set_eq:\texttt{(Nc|cN|cc)} \coffin_gset_eq:NN \coffin_gset_eq:\texttt{(Nc|cN|cc)}
Sets both the content and poles of \texttt{coffin}_1 equal to those of \texttt{coffin}_2.

\coffin_if_exist_p:N \coffin_if_exist_p:c \coffin_if_exist:N \coffin_if_exist:NTF \coffin_if_exist:TF
New: 2012-06-20

\coffin_if_exist_p:N \coffin_if_exist_p:c \coffin_if_exist:N \coffin_if_exist:NTF \coffin_if_exist:TF
Tests whether the \texttt{coffin} is currently defined.
### 36.2 Setting coffin content and poles

\hcoffin_set:Nn \hcoffin_set:cn \hcoffin_gset:Nn \hcoffin_gset:cn

New: 2011-08-17
Updated: 2019-01-21

Typesets the (material) in horizontal mode, storing the result in the (coffin). The standard poles for the (coffin) are then set up based on the size of the typeset material.

\hcoffin_set:Nw \hcoffin_set:cw \hcoffin_gset:Nw \hcoffin_gset:cw

New: 2011-09-10
Updated: 2019-01-21

Typesets the (material) in horizontal mode, storing the result in the (coffin). The standard poles for the (coffin) are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\vcoffin_set:Nnn \vcoffin_set:cnn \vcoffin_gset:Nnn \vcoffin_gset:cnn

New: 2011-08-17
Updated: 2023-02-03

Typesets the (material) in vertical mode constrained to the given (width) and stores the result in the (coffin). The standard poles for the (coffin) are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\vcoffin_set:Nnw \vcoffin_set:cnw \vcoffin_gset:Nnw \vcoffin_gset:cnw

New: 2011-09-10
Updated: 2023-02-03

Typesets the (material) in vertical mode constrained to the given (width) and stores the result in the (coffin). The standard poles for the (coffin) are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\coffin_set_horizontal_pole:Nnn \coffin_set_horizontal_pole:cnn \coffin_gset_horizontal_pole:Nnn \coffin_gset_horizontal_pole:cnn

New: 2012-07-20
Updated: 2019-01-21

Sets the (pole) to run horizontally through the (coffin). The (pole) is placed at the (offset) from the baseline of the (coffin). The (offset) should be given as a dimension expression.
Sets the ⟨pole⟩ to run vertically through the ⟨coffin⟩. The ⟨pole⟩ is placed at the ⟨offset⟩ from the left-hand edge of the bounding box of the ⟨coffin⟩. The ⟨offset⟩ should be given as a dimension expression.

Resets the poles of the ⟨coffin⟩ to the standard set, removing any custom or inherited poles. The poles will therefore be equal to those that would be obtained from \hcoffin_set:Nn or similar; the bounding box of the coffin is not reset, so any material outside of the formal bounding box will not influence the poles.

36.3 Coffin affine transformations

Resized the ⟨coffin⟩ to ⟨width⟩ and ⟨total-height⟩, both of which should be given as dimension expressions.

Rotates the ⟨coffin⟩ by the given ⟨angle⟩ (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.

Scales the ⟨coffin⟩ by factors ⟨x-scale⟩ and ⟨y-scale⟩ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.
36.4 Joining and using coffins

This function attaches \(\text{coffin}_2\) to \(\text{coffin}_1\) such that the bounding box of \(\text{coffin}_1\) is not altered, \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 \(\text{coffin}_1\)-pole and \(\text{coffin}_2\)-pole, and \(\text{handle}_2\), the point of intersection of \(\text{coffin}_2\)-pole and \(\text{coffin}_1\)-pole. \(\text{coffin}_2\) is then attached to \(\text{coffin}_1\) such that the relationship between \(\text{handle}_1\) and \(\text{handle}_2\) is described by the \(x\)-offset and \(y\)-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 \(\text{handle}_1\), the point of intersection of \(\text{coffin}_2\)-pole and \(\text{coffin}_1\)-pole, and \(\text{handle}_2\), the point of intersection of \(\text{coffin}_1\)-pole and \(\text{coffin}_2\)-pole. \(\text{coffin}_1\) is then attached to \(\text{coffin}_2\) such that the relationship between \(\text{handle}_1\) and \(\text{handle}_2\) is described by the \(x\)-offset and \(y\)-offset. The two offsets should be given as dimension expressions.

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

36.5 Measuring coffins

Calculates the depth (below the baseline) of the \(\text{coffin}\) 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}.

\texttt{coffin_ht:N} \texttt{coffin} \texttt{coffin_ht:c}

Calculates the width of the \texttt{coffin} in a form suitable for use in a \texttt{dim expr}.

\texttt{coffin_wd:N} \texttt{coffin} \texttt{coffin_wd:c}

\section*{36.6 Coffin diagnostics}

\texttt{coffin_display_handles:Nn} \texttt{coffin} \texttt{coffin_display_handles:cn}

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.

\texttt{coffin_mark_handle:Nnnn} \texttt{coffin} \texttt{coffin_mark_handle:cn}

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.

\texttt{coffin_show_structure:N} \texttt{coffin} \texttt{coffin_show_structure:c}

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 \textit{x} and \textit{y} coordinates of a point that the pole passes through and the \textit{x} and \textit{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.

\texttt{coffin_log_structure:N} \texttt{coffin} \texttt{coffin_log_structure:c}

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.

\texttt{coffin_show:N} \texttt{coffin} \texttt{coffin_show:c} \texttt{coffin_log:N} \texttt{coffin} \texttt{coffin_log:c}

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.
\coffin_show:Nnn \coffin_show:cnn \coffin_log:Nnn \coffin_log:cnn

Shows poles and contents of the \texttt{coffin} in the terminal or log file, showing the first \texttt{int expr} items in the coffin, and descending into \texttt{int expr} group levels. See \texttt{coffin-structure:N} and \texttt{box_show:Nnn} to show separately the pole structure and the contents.

### 36.7 Constants and variables

\verb|\c_empty_coffin|

A permanently empty coffin.

\verb|\l_tmpa_coffin| \verb|\l_tmpb_coffin|

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

\verb|\g_tmpa_coffin| \verb|\g_tmpb_coffin|

Scratch coffins for global assignment. These are never used by the kernel code, and so are safe for use with any \texttt{\LaTeX3}-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
Chapter 37

The \texttt{l3color} module

Color support

37.1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

\begin{description}
\item[$\texttt{color\_group\_begin}$:] \texttt{color\_group\_begin:} \texttt{...}
\item[$\texttt{color\_group\_end}$:] \texttt{color\_group\_end:}
\end{description}

New: 2011-09-03

Creates a color group: one used to “trap” color settings. This grouping is built in to for example \texttt{ hbox\_set:\texttt{Nn}.}

\begin{description}
\item[$\texttt{color\_ensure\_current}$:] \texttt{color\_ensure\_current:}
\end{description}

New: 2011-09-03

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 \texttt{color\_group\_begin: ... \color\_group\_end:} group.

37.2 Color models

A color \textit{model} is a way to represent sets of colors. Different models are particularly suitable for different output methods, \textit{e.g.} screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of \textit{axes} which define a color space. In contrast, various proprietary models are available which define \textit{spot} colors (more formally separations).

Core models are used to pass color information to output; these are “native” to \texttt{l3color}. Core models use real numbers in the range $[0,1]$ to represent values. The core models supported here are

\begin{itemize}
\item \texttt{gray} Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
\item \texttt{rgb} Red-green-blue color, with three axes, one for each of the components
\end{itemize}
• **cmyk** Cyan-magenta-yellow-black color, with four axes, one for each of the components.

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based: see below for more detail. The supported interface models are:

- **Gray** Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)
- **hsb** Hue-saturation-brightness color, with three axes, all real values in the range \([0, 1]\) for hue saturation and brightness
- **Hsb** Hue-saturation-brightness color, with three axes, integer in the range \([0, 360]\) for hue, real values in the range \([0, 1]\) for saturation and brightness
- **HSB** Hue-saturation-brightness color, with three axes, integers in the range \([0, 240]\) for hue, saturation and brightness
- **HTML** HTML format representation of RGB color given as a single six-digit hexadecimal number
- **RGB** Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255
- **wave** Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as **rgb**.

Finally, there are a small number of models which are parsed to allow data transfer from **xcolor** but which should not be used by end-users. These are:

- **cmy** Cyan-magenta-yellow color with three axes, one for each of the components; converted to **cmyk**
- **tHsb** “Tuned” hue-saturation-brightness color with three axes, integer in the range \([0, 360]\) for hue, real values in the range \([0, 1]\) for saturation and brightness; converted to **rgb** using the standard tuning map defined by **xcolor**
- **&spot** Spot color tint with one value; treated as a gray tint as spot color data is not available for extraction

To allow parsing of data from **xcolor**, any leading model up the first : will be discarded; the approach of selecting an internal form for data is *not* used in **3color**.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 37.9 for more detail of color support for additional models.

When color is selected by model, the \((values)\) given are specified as a comma-separated list. The length of the list will therefore be determined by the detail of the model involved.

Color models (and interconversion) are complex, and more details are given in the manual to the **E\TeX\ 2e xcolor** package and in the **PostScript Language Reference Manual**, published by Addison–Wesley.
37.3  Color expressions

In addition to allowing specification of color by model and values, \texttt{13color} also supports
color expressions. These are created by combining one or more color names, with the
amount of each specified as a value in the range 0–100. The value should be given between
! symbols in the expression. Thus for example

\begin{verbatim}
red!50!green
\end{verbatim}

is a mixture of 50\% red and 50\% green. A trailing value is interpreted as implicitly
followed by \texttt{!white}, and so

\begin{verbatim}
red!25
\end{verbatim}

specifies 25\% red mixed with 75\% white.

Where the models for the mixed colors are different, the model of the first color is
used. Thus

\begin{verbatim}
red!50!cyan
\end{verbatim}

will result in a color specification using the \texttt{rgb} model, made up of 50\% red and 50\%
of cyan \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

\begin{verbatim}
black!50!red
\end{verbatim}

has the same result as

\begin{verbatim}
red!50!black
\end{verbatim}

(the predefined colors \texttt{black} and \texttt{white} use the \texttt{gray} model).

Where more than two colors are mixed in an expression, evaluation takes place in a
stepwise fashion. Thus in

\begin{verbatim}
cyan!50!magenta!10!yellow
\end{verbatim}

the sub-expression

\begin{verbatim}
cyan!50!magenta
\end{verbatim}

is first evaluated to give an intermediate color specification, before the second step

\begin{verbatim}
<intermediate>!10!yellow
\end{verbatim}

where \texttt{<intermediate>} represents this transitory calculated value.

Within a color expression, \texttt{.} may be used to represent the color active for typesetting
(the current color). This allows for example

\begin{verbatim}
.\!50
\end{verbatim}

to mean a mixture of 50\% of current color with white.

(Color expressions supported here are a subset of those provided by the \LaTeX xcolor package. At present, only such features as are clearly useful have been added here.)
37.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names black, white, red, green, blue, cyan, magenta and yellow have special meaning and should not be redefined. Color names should be made up of letters, numbers and spaces only: other characters are reserved for use in color expressions. In particular, . represents the current color at the start of a color expression.

\color_set:nn \color_set:nn \{(name)\} \{(color expression)\}

Evaluates the \{color expression\} and stores the resulting color specification as the \{name\}.

\color_set:nnn \color_set:nn \{(name)\} \{(model(s))\} \{(value(s))\}

Stores the color specification equivalent to the \{model(s)\} and \{values\} as the \{name\}.

\color_set_eq:nn \color_set_eq:nn \{(name1)\} \{(name2)\}

Copies the color specification in \{name2\} to \{name1\}. 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 \{(name)\}

Tests whether \{name\} is currently defined to provide a color specification.

\color_show:n \color_show:n \{(name)\}
\color_log:n \color_log:n \{(name)\}

Displays the color specification stored in the \{name\} on the terminal or log file.

37.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color (.): other more specialised functions such as fill and stroke selectors do not adjust this value.

\color_select:n \color_select:nn \{(color expression)\}

 Parses the \{color expression\} and then activates the resulting color specification for typeset material.

\color_select:nn \color_select:nn \{(model(s))\} \{(value(s))\}

Activates the color specification equivalent to the \{model(s)\} and \{value(s)\} 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.
37.6 Colors for fills and strokes

Colors for drawing operations and so forth are split into strokes and fills (the latter may also be referred to as non-stroke color). The fill color is used for text under normal circumstances. Depending on the backend, stroke color may use a stack, in which case it exhibits the same page breaking behavior as general color. However, dvips/dvisvgm do not support this, and so color will need to be contained within a scope, such as \draw_begin:/\draw_end:.

\color_fill:n \color_stroke:n

Parses the \langle color expression \rangle and then activates the resulting color specification for filling or stroking.

\color_fill:nn \color_stroke:nn

Activates the color specification equivalent to the \langle model(s) \rangle and \langle value(s) \rangle for filling or stroking.

\color_sc

When using dvips, this PostScript variables hold the stroke color.

37.6.1 Coloring math mode material

Coloring math mode material using \color_select:nn(n) has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating \mathord atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

\color_math:nn \color_math:nnn

Works as for \color_select:n(n) but applies color only to the math mode \langle content \rangle. The function does not generate a group and the \langle content \rangle therefore retains its math atom states. Sub/superscripts are also properly handled.

\l_color_math_active_tl

This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.

37.7 Multiple color models

When selecting or setting a color with an explicit model, it is possible to give values for more than one model at one time. This is particularly useful where automated conversion between models does not give the desired outcome. To do this, the list of models and list of values are both subdivided using / characters (as for the similar function in xcolor).

For example, to save a color with explicit cmyk and rgb values, one could use

\color_set:nnn { foo } { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
The manually-specified conversion will be used in preference to automated calculation whenever the model(s) listed are used: both in expressions and when a fixed model is active.

Similarly, the same syntax can be applied to directly selecting a color.

```
\color_select:nn { cmyk / rgb }
  { 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

Again, this list is used when a fixed model is active: the first entry is used unless there is a fixed model matching one of the other entries.

### 37.8 Exporting color specifications

The major use of color expressions is in setting typesetting output, but there are other places in which some form of color information is required. These may need data in a different format or using a different model to the internal representation. Thus a set of functions are available to export colors in different formats.

Valid export targets are

- **backend** Two brace groups: the first containing the model, the second containing space-separated values appropriate for the model; this is the format required by backend functions of expl3
- **comma-sep-cmyk** Comma-separated cyan-magenta-yellow-black values
- **comma-sep-rgb** Comma-separated red-green-blue values suitable for use as a PDF annotation color
- **HTML** Uppercase two-digit hexadecimal values, expressing a red-green-blue color; the digits are *not* separated
- **space-sep-cmyk** Space-separated cyan-magenta-yellow-black values
- **space-sep-rgb** Space-separated red-green-blue values suitable for use as a PDF annotation color

---

```
\color_export:nnN {⟨color expression⟩} {⟨format⟩} {⟨tl⟩}
```

Parses the ⟨color expression⟩ as described earlier, then converts to the ⟨format⟩ specified and assigns the data to the ⟨tl⟩.

```
\color_export:nnnN {⟨model⟩} {⟨value(s)⟩} {⟨format⟩} {⟨tl⟩}
```

Expresses the combination of ⟨model⟩ and ⟨value(s)⟩ in an internal representation, then converts to the ⟨format⟩ specified and assigns the data to the ⟨tl⟩.
37.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see https://helpx.adobe.com/indesign/using/spot-process-colors.html for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (DeviceCMYK, DeviceRGB, DeviceGray), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that l3color uses the shorter names cmyk, etc.)

\color_model_new:nnn \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 \(L^\ast\), \(a^\ast\) and \(b^\ast\) 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 an ICCBased space, there is one compulsory key.

- file The name of the file containing the profile.
37.9.1 Color profiles

Color profiles are used to ensure color accuracy by linking to collaboration. Applying a profile can be used to standardize color which is otherwise device-dependence.

\texttt{\color_profile_apply:nn \{\textit{profile}\} \{\textit{model}\}}

This function applies a \textit{profile} to one of the device \textit{models}. The profile will then apply to all color of the selected \textit{model}. The \textit{profile} should specify an ICC profile file. The \textit{model} has to be one the standard device models: \texttt{cmyk}, \texttt{gray} or \texttt{rgb}.
Chapter 38

The l3pdf module
Core PDF support

38.1 Objects

38.1.1 Named objects

An ⟨object⟩ name should fully expand to tokens suitable for use in a label-like context.

\pdf_object_new:n \pdf_object_new:n \{⟨object⟩\}

Declares ⟨object⟩ as a PDF object. The object may be referenced from this point on, and written later using \pdf_object_write:nnn.

\pdf_object_write:nnn \pdf_object_write:nnn \{⟨object⟩\} \{⟨type⟩\} \{⟨content⟩\}

Writes the ⟨content⟩ as content of the ⟨object⟩. Depending on the ⟨type⟩ declared for the object, the format required for the ⟨data⟩ will vary:

- array A space-separated list of values
- dict Key–value pairs in the form /⟨key⟩ ⟨value⟩
- fstream Two brace groups: ⟨file name⟩ and ⟨file content⟩
- stream Two brace groups: ⟨attributes (dictionary)⟩ and ⟨stream contents⟩

\pdf_object_ref:n \pdf_object_ref:n \{⟨object⟩\}

Inserts the appropriate information to reference the ⟨object⟩ in for example page resource allocation. If the ⟨object⟩ does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

\pdf_object_if_exist_p:n \pdf_object_if_exist_p:n \{⟨object⟩\}

\pdf_object_if_exist:nTF \pdf_object_if_exist:nTF \{⟨object⟩\} \{⟨true code⟩\} \{⟨false code⟩\}

Tests whether an object with name ⟨⟨object⟩⟩ has been defined.

328
38.1.2 Indexed objects

Objects can also be created using a pair of \langle class \rangle and \langle index \rangle; the \langle class \rangle argument should expand to character tokens, whilst the \langle index \rangle is an \langle int expr \rangle and starts at 1. For large families of objects, this approach is more efficient than using individual names.

```
pdf_object_new_indexed:nn \{\langle class \rangle\} \{\langle index \rangle\}
```

Declares a PDF object of \langle class \rangle and \langle index \rangle. The object may be referenced from this point on, and written later using `pdf_object_write_indexed:nnnn`.

```
pdf_object_write_indexed:nnnn \{\langle class \rangle\} \{\langle index \rangle\} \{\langle type \rangle\} \{\langle content \rangle\}
```

Writes the \langle content \rangle as content of the object of \langle class \rangle and \langle index \rangle. Depending on the \langle type \rangle declared for the object, the format required for the \langle content \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_indexed:nn * \{\langle class \rangle\} \{\langle index \rangle\}
```

Inserts the appropriate information to reference the object of \langle class \rangle and \langle index \rangle in for example page resource allocation. If the \langle class \rangle/\langle index \rangle combination does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

38.1.3 General functions

```
pdf_object_unnamed_write:nn \{\langle type \rangle\} \{\langle content \rangle\}
```

```
pdf_object_unnamed_write:ne
```

New: 2021-02-10

Writes the \langle content \rangle as content of an anonymous object. Depending on the \langle type \rangle, 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 attributes (dictionary) \rangle and \langle file name \rangle
- **stream** Two brace groups: \langle attributes (dictionary) \rangle and \langle stream contents \rangle
Inserts the appropriate information to reference the last \textit{object} created. This is particularly useful for anonymous objects.

Inserts the appropriate information to reference the \textit{abspage}; the latter is expanded fully before further processing.

### 38.2 Version

\texttt{\pdf_version_compare_p:Nn} \texttt{\pdf_version_compare_NnTF}

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.

\texttt{\pdf_version_gset:n} \texttt{\pdf_version_min_gset:n}

Sets the \textit{version} of the PDF being created. The \texttt{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:} \texttt{\pdf_version_compare:}, but the PDF version itself still has to be set with the command line option \texttt{-dCompatibilityLevel} of \texttt{ps2pdf}.

\texttt{\pdf_version:} \texttt{\pdf_version_major:} \texttt{\pdf_version_minor:}

Expands to the currently-active PDF version.

### 38.3 Page (media) size

\texttt{\pdf_pagesize_gset:nn}

Sets the page size (mediabox) of the PDF being created to the \textit{width} and \textit{height}, both of which are \texttt{dimexpr}. The page size can only be set at the start of the output with \texttt{dvips}; with other backends, this can be adjusted on a per-page basis.

### 38.4 Compression

\texttt{\pdf_uncompress:}

Disables any compression of the PDF, where possible.

This function may only be used up to the point where the PDF file is initialised.
38.5 Destinations

Destinations are the places a link jumped to. 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.

\pdf_destination:nn \pdf_destination:nnn {\langle name\rangle} {\langle type or integer\rangle}

This creates a destination. {\langle type or integer\rangle} can be one of fit, fith, fitv, fitb, fitbh, fitbv, fitr, xyz or an integer representing a scale factor in percent. fitr here gives only a lightweight version of /FitR: The backend code defines fitr so that it will with pdfTex and LuaTeX use the coordinates of the surrounding box, with dvips and dvipdfmx it falls back to fit. For full control use \pdf_destination:nnnn.

The keywords match to the PDF names as described in the following tabular.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>PDF</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit</td>
<td>/Fit</td>
<td>Fits the page to the window</td>
</tr>
<tr>
<td>fith</td>
<td>/FitH top</td>
<td>Fits the width of the page to the window</td>
</tr>
<tr>
<td>fitv</td>
<td>/FitV left</td>
<td>Fits the height of the page to the window</td>
</tr>
<tr>
<td>fitb</td>
<td>/FitB</td>
<td>Fits the page bounding box to the window</td>
</tr>
<tr>
<td>fitbh</td>
<td>/FitBH top</td>
<td>Fits the width of the page bounding box to the window</td>
</tr>
<tr>
<td>fitbv</td>
<td>/FitBV left</td>
<td>Fits the height of the page bounding box to the window</td>
</tr>
<tr>
<td>fitr</td>
<td>/FitR left bottom right top</td>
<td>Fits the rectangle specified by the four coordinates to the window (see above for the restrictions)</td>
</tr>
<tr>
<td>xyz</td>
<td>/XYZ left top null</td>
<td>Sets a coordinate but doesn’t change the zoom.</td>
</tr>
<tr>
<td>{\langle integer\rangle}</td>
<td>/XYZ left top zoom</td>
<td>Sets a coordinate and a zoom meaning {\langle integer\rangle}%.</td>
</tr>
</tbody>
</table>

\pdf_destination:nnnn \pdf_destination:nnnn {\langle name\rangle} {\langle width\rangle} {\langle height\rangle} {\langle depth\rangle}

This creates a destination with /FitR type with the given dimensions relative to the current location. The destination is in a box of size zero, but it doesn’t switch to horizontal mode.
## Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

### Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>273</td>
</tr>
<tr>
<td>&amp;k</td>
<td>272</td>
</tr>
<tr>
<td>*</td>
<td>273</td>
</tr>
<tr>
<td>**</td>
<td>273</td>
</tr>
<tr>
<td>+</td>
<td>273</td>
</tr>
<tr>
<td>-</td>
<td>273</td>
</tr>
<tr>
<td>/</td>
<td>273</td>
</tr>
<tr>
<td>:i</td>
<td>45</td>
</tr>
<tr>
<td>:i</td>
<td>45</td>
</tr>
<tr>
<td>:n</td>
<td>45</td>
</tr>
<tr>
<td>:c</td>
<td>45</td>
</tr>
<tr>
<td>:e</td>
<td>45</td>
</tr>
<tr>
<td>:e</td>
<td>45</td>
</tr>
<tr>
<td>:f</td>
<td>45</td>
</tr>
<tr>
<td>:f</td>
<td>45</td>
</tr>
<tr>
<td>:n</td>
<td>45</td>
</tr>
<tr>
<td>:o</td>
<td>45</td>
</tr>
<tr>
<td>:o</td>
<td>45</td>
</tr>
<tr>
<td>:v</td>
<td>45</td>
</tr>
<tr>
<td>:v</td>
<td>45</td>
</tr>
<tr>
<td>:x</td>
<td>45</td>
</tr>
<tr>
<td>:x</td>
<td>45</td>
</tr>
<tr>
<td>&lt;</td>
<td>273</td>
</tr>
<tr>
<td>=</td>
<td>273</td>
</tr>
<tr>
<td>&gt;</td>
<td>273</td>
</tr>
<tr>
<td>?</td>
<td>273</td>
</tr>
<tr>
<td>?:</td>
<td>272</td>
</tr>
<tr>
<td>??</td>
<td>88</td>
</tr>
<tr>
<td>^</td>
<td>273</td>
</tr>
</tbody>
</table>

### B

bitset commands:

- `\bitset_addto_named_index:Nn` | 285
- `\bitset_clear:N` | 286
- `\bitset_gclear:N` | 286
- `\bitset_gset_false:Nn` | 286
- `\bitset_true:Nn` | 287
- `\bitset_gset_exist:NTF` | 286
- `\bitset_if_exist:p:N` | 286
- `\bitset_if_exist_p:N` | 286
- `\bitset_item:N` | 286
- `\bitset_log:N` | 287
- `\bitset_log_named_index:N` | 287
- `\bitset_new:N` | 285
- `\bitset_new:Nn` | 285
- `\bitset_set_false:Nn` | 286
- `\bitset_set_true:Nn` | 286
- `\bitset_show:N` | 286, 287
- `\bitset_show_named_index:N` | 287
- `\bitset_to_arabic:N` | 284, 287
- `\bitset_to_bin:N` | 285, 287

### bool commands:

- `\bool_case:n` | 71
- `\bool_case:NTF` | 71
- `\bool_const:NN` | 66
- `\bool_do_until:Nn` | 70
- `\bool_do_until:nn` | 70
- `\bool_do_while:nn` | 70
- `\bool_do_while:Nn` | 70
- `\bool_if:NTF` | 66
- `\bool_if_inverse:NN` | 66
- `\bool_if_eq:NN` | 66
- `\bool_if_same:NN` | 66
- `\bool_if_same:NTF` | 66
- `\bool_if_gset_inverse:NN` | 66
- `\bool_if_gset_eq:NN` | 66
- `\bool_if_gset_exist:NTF` | 66
- `\bool_if_gset_exist:p:N` | 66

### A

- `abs` | 273
- `acos` | 276
- `acosd` | 276
- `acot` | 277
- `acotd` | 277
- `acsc` | 276
- `acscd` | 276
- `asec` | 276
- `asecd` | 276
- `asin` | 276
- `asind` | 276

332
C

cc ........................................... 279

cctab commands:
\cctab_begin:N ......................... 289
\cctab_const:N ......................... 288, 289
\cctab_end: .......................... 289
\cctab_gsave_current:N ............. 288
\cctab_gset:N ......................... 288, 289
\cctab_if_exist:NTF ................. 289
\cctab_if_exist_p:N ................. 289
\cctab_item:N ......................... 289
\cctab_new:N .......................... 288
\cctab_select:N ..................... 125, 288, 289
\c_code_cctab ......................... 289
\c_document_cctab .................... 289
\c_inix_cctab ......................... 290
\c_other_cctab ......................... 290
\g_tampa_cctab ........................ 290
\g_tmpb_cctab ......................... 290

ceil ..................................... 275

char commands:
\l_char_active_seq .................. 90, 199
\char_generate:nn .................... 124, 195
\char_get_active_eq:NN ............ 195
\char_get_active_eq:nN ............ 195
\char_set_active_eq:NN ............ 195
\char_set_active_eq:nN ............ 195
\char_set_catcode_active:N ........ 196
\char_set_catcode_alignment:N .... 196
\char_set_catcode_alignment:n .... 197
\char_set_catcode_comment:N ....... 196
\char_set_catcode_comment:n ....... 197
\char_set_catcode_end_line:N ...... 196
\char_set_catcode_end_line:n ...... 197
\char_set_catcode_escape:N ........ 196
\char_set_catcode_escape:n ........ 197
\char_set_catcode_group_begin:N ... 196
\char_set_catcode_group_begin:n ... 197
\char_set_catcode_group_end:N ...... 196
\char_set_catcode_group_end:n ...... 197
\char_set_catcode_ignore:N ........ 196
\char_set_catcode_ignore:n ........ 197
\char_set_catcode_invalid:N ....... 196
\char_set_catcode_invalid:n ....... 197
\char_set_catcode_letter:N ........ 196
\char_set_catcode_letter:n ........ 197
\char_set_catcode_math_subscript:N ........................................ 196
\char_set_catcode_math_subscript:n ........................................ 196

\char_set_catcode_math_superscript:n ........................................ 197
\char_set_catcode_math_toggle:N ............................................ 196
\char_set_catcode_math_toggle:n ............................................ 197
\char_set_catcode_other:N ........ 196
\char_set_catcode_other:n ........ 197
\char_set_catcode_parameter:N .... 196
\char_set_catcode_parameter:n .... 197
\char_set_catcode_space:N .......... 196
\char_set_catcode_space:n .......... 197
\char_set_catcode_math_superscript:n ........................................ 197
\char_set_catcode_math_toggle:N ............................................ 197
\char_set_catcode_math_toggle:n ............................................ 197
\char_set_catcode_math_subscript:n ........................................ 197
\char_set_catcode_math_subscript:N ........................................ 197
\char_set_catcode_math_toggle:n ............................................ 196
\char_set_catcode_math_subscript:n ........................................ 197
\char_set_catcode_math_subscript:N ........................................ 196
\char_set_catcode_math_subscript:n ........................................ 196
\file_get_timestamp:nNTF \ 101 .fp_gset:N \ 241
\file_hex_dump:n \ 100 .fp_gset:Nn \ 259
\file_hex_dump:nNN \ 100 .fp_gset_eq:NN \ 259
\file_if_exist:nTF \ 100, 102, 103 .fp_gsub:Nn \ 260
\file_if_exist_input:n \ 103 .fp_gzero:N \ 259
\file_if_exist_input:nTF \ 103 .fp_gzero_new:N \ 259
\file_if_exist:p:n \ 100 .fp_if_exist:NTF \ 261
\file_input:n \ 103, 104 .fp_if_exist_p:N \ 261
\file_input_raw:n \ 103 .fp_if_nan:NTF \ 263, 280
\file_input_stop: \ 104 .fp_if_nan:p:n \ 263
\file_log_list: \ 104 \l_fp_invalid_operation_flag \ 269
\file_mdfive_hash:n \ 101 .fp_log:N \ 270
\file_parse_full_name:n \ 103 .fp_log:n \ 270
\file_parse_full_name:nNN \ 102, 103 .fp_max:nn \ 279
\file_parse_full_name_apply:nN \ 103 .fp_min:nn \ 279
\l_file_search_path_seq \ 100, 101, 103 .fp_new:N \ 259
\file_show_list: \ 104 .fp_new_function:n \ 267
\file_size:n \ 101 .fp_new_variable:n \ 265–267
\file_timestamp:n \ 74, 101 \l_fp_overflow_flag \ 269
flag commands:
\flag_clear:N \ 181 .fp_set:N \ 241
\flag_clear_new:N \ 181 \fp_set_eq:Nn \ 259
\flag_ensure_raised:N \ 181 \fp_set_eq:nn \ 267
\flag_height:N \ 181 \fp_set_variable:nn \ 265–267
\flag_if_exist:NTF \ 181 \fp_show:N \ 266, 270
\flag_if_exist_p:N \ 181 \fp_show:n \ 265–267, 270
\flag_if_raised:NTF \ 181 \fp_sign:n \ 260
\flag_if_raised_p:N \ 181 \fp_step_function:nnn \ 265
\flag_log:N \ 181 \fp_step_inline:nnnn \ 265
\flag_new:N \ 180, 181 \fp_step_variable:nnn \ 265
\flag_new:N \ 181 \fp_step_variable:nnnn \ 265
\flag_raise:N \ 181 .fp_sub:Nn \ 260
\flag_show:N \ 181 .fp_to_decimal:N \ 260, 261
\l_tampa_flag \ 182 \fp_to_decimal:n \ 260, 261
\l_tamb_flag \ 182 \fp_to_dim:N \ 260
floor commands:
\c_e_fp \ 268, 271 .fp_to_int:N \ 260
\fp_abs:n \ 273, 279 .fp_to_int:n \ 260
\fp_add:Nn \ 259 \fp_to_decimal: \ 260
\fp_clear_function:n \ 267 \fp_to_decimal:n \ 261
\fp_clear_variable:n \ 267 \fp_to_tl:n \ 261
\fp_compare:nNnTF \ 262–264 \fp_trap:nn \ 269
\fp_compare:nTF \ 262–264, 273 \l_fp_underflow_flag \ 269
\fp_compare_p:n \ 263 \fp_until_do:nn \ 264
\fp_compare_p:Nn \ 262 \fp_until_do:nNn \ 264
\fp_const:Nn \ 259 .fp_use:N \ 261, 283
\l_fp_division_by_zero_flag \ 269 \fp_while_do:nn \ 264
\fp_do_until:nn \ 264 \fp_while_do:nNn \ 264
\fp_do_until:Nnn \ 263 \fp_zero:N \ 259
\fp_do_while:nn \ 264 \fp_zero_new:N \ 259
\fp_do_while:Nnn \ 264 \c_inf_fp \ 268, 278
\fp_eval:n \ 260, 263, 267, 272–279, 287 \c_minus_inf_fp \ 268, 278
\fp_format:n \ 280 \c_minus_zero_fp \ 268
\fp_gadd:Nn \ 259 \c_nan_fp \ 268, 278
\c_one_degree_fp \dots \quad 268, 278
\c_one_fp \quad 268
\c_pi_fp \quad 268, 278
\g_tmpa_fp \quad 268
\l_tmpa_fp \quad 268
\g_tmpb_fp \quad 268
\l_tmpb_fp \quad 265, 266, 268
\c_zero_fp \quad 268

fparray commands:
\fparray_count:N \quad 282, 283
\fparray_get:Nn \quad 282
\fparray_gzero:N \quad 282
\fparray_if_exist:NTF \quad 283
\fparray_if_exist_p:N \quad 283
\fparray_item:Nn \quad 283
\fparray_item_to_tl:Nn \quad 283
\fparray_new:Nn \quad 282

hcoffin commands:
\hcoffin_get:Nn \quad 315
\hcoffin_get:Wn \quad 315
\hcoffin_get_end: \quad 315
\hcoffin_set:N \quad 315, 316
\hcoffin_set:Wn \quad 315
\hcoffin_set_end: \quad 315

if commands:
\if:w \quad 28, 29, 194
\if_bool:N \quad 72
\if_box_empty:N \quad 313
\if_case:w \quad 178
\if_catcode:w \quad 29
\if_charcode:w \quad 29, 194
\if_cs_exist:N \quad 29
\if_cs_exist:w \quad 29
\if_dim:w \quad 237
\if_eof:w \quad 99
\if_false: \quad 28, 65, 207
\if_hbox:N \quad 313
\if_int_compare:w \quad 28, 178
\if_int_odd:w \quad 179
\if_meaning:w \quad 29
\if_mode_horizontal: \quad 29
\if_mode_inner: \quad 29
\if_mode_math: \quad 29
\if_mode_vertical: \quad 29
\if_predicate:w \quad 63, 65, 72
\if_true: \quad 28, 66
\if_vbox:N \quad 313
\in \quad 279
\inf \quad 278

inherit commands:
.inherit:n \quad 242

initial commands:
.initial:n \quad 242

int commands:
\int_abs:n \quad 166
\int_add:N \quad 168
\int_case:nn \quad 171
\int_case:ntn \quad 171
\int_compare:nNn \quad 169-172, 262
\int_compare:nTF \quad 169, 170, 172, 263
\int_compare:p:n \quad 170
\int_compare:p:Nn \quad 28, 169
\int_const:Nn \quad 167
\int_dec:N \quad 168
\int_div_round:nn \quad 167
\int_div_truncate:nn \quad 167
\int_do_until:nn \quad 172
\int_do_until:Nn \quad 171
\int_do_until:NN \quad 172

G
\GetIdInfo \quad 10
group commands:
\group_align_safe_begin: \quad 73
\group_align_safe_end: \quad 73
\group_begin: \quad 13
\c_group_begin_token : \quad 114, 199, 207
\group_end: \quad 13, 14
\c_group_end_token \quad 199
\group_insert_after:N \quad 14
\group_log_list: \quad 14
\group_show_list: \quad 14
groups commands:
.groups:n \quad 242

H
\hbox:n \quad 301, 305
\hbox_get:Nn \quad 306
\hbox_get:Wn \quad 306
\hbox_get_end: \quad 306
\hbox_get_to_wd:Nn \quad 306
\hbox_get_to_wd:Wn \quad 306
\hbox_overlap_center:n \quad 306
\hbox_overlap_left:n \quad 306
\hbox_overlap_right:n \quad 306
\hbox_set:Nn \quad 301, 305, 306, 320
\hbox_set:Wn \quad 306
\hbox_set_end: \quad 306
\hbox_set_to_wd:Nn \quad 306
\hbox_set_to_wd:Wn \quad 306
\hbox_to_wd:nn \quad 305
\hbox_to_zero:n \quad 305
\hbox_unpack:N \quad 306
\hbox_unpack_drop:N \quad 309

\hbox_set:Wn \quad 306
\hbox_set:Wn \quad 306
\c_one_degree_fp \quad 268, 278
\c_one_fp \quad 268
\c_pi_fp \quad 268, 278
\c_zero_fp \quad 268
\int_do_while:nNn ................. 172 \int_to_Base:n ...................... 175
\int_eval:n .................. 19, 34, 166–171, 178 \int_to_base:n .................. 175
\int_eval:w .................. 166 \int_to_Base:nn .................. 175, 176
\int_from_alph:n .............. 175 \int_to_base:nn .................. 175, 176
\int_from_base:nn ............ 176 \int_to_bin:n .................. 175
\int_from_bin:n .............. 175, 287 \int_to_Hex:n .................. 175, 176
\int_from_hex:n .............. 176 \int_to_hex:n .................. 175, 176
\int_from_oct:n .............. 176 \int_to_oct:n .................. 175, 176
\int_from_roman:n ............ 176 \int_to_Roman:n .................. 175, 176
\int_gadd:NN .................. 168 \int_to_roman:n .................. 175, 176
\int_gadd:NN .................. 168 \int_to_symbols:nnn ............... 174
\int_gincr:N ................. 168 \int_until:do:NNn ................. 172
\int_gset:N .................. 242 \int_until:do:NNnn ............... 172
\int_gset:NN .................. 168 \int_use:N ...................... 165, 169
\int_gset_eq:NN .............. 168 \int_value:w .................... 166, 178
\int_gzero:NN ................. 167 \int_while:do:NNn ............... 172
\int_gzero_new:N ............. 167 \int_zero:N ..................... 167
\int_if_even:nTF .............. 171 \int_zero_new:N .................. 172
\int_if_even:p:n ............. 171 \int_zero:N ..................... 167
\int_if_exist:NTF ............. 168 \c_max_char_int ................ 177
\int_if_exist:p:N ............. 168 \c_max_register_int ............. 177
\int_if_odd:nTF .............. 171 \c_one_int ..................... 177
\int_if_odd:p:n .............. 171 \int_if_odd:p:n .................. 177
\int_iiif_odd:nTF ............ 171 \g_temp_int .................. 177
\int_if_zero:p:n ............. 171 \l_temp_int .................. 4, 53, 177
\int_inc:n .................. 175 \l_temp_int .................. 177
\int_log:N .................. 177 \l_temp_int .................. 177
\int_log:n .................. 177 \c_zero_int .................. 177
\int_max:nn .................. 167 \intarray_jump:cN................. 254
\int_min:nn .................. 167 \intarray_jump:NN................. 255
\int_mod:nn .................. 167 \intarray_jump:NNn................. 255
\int_new:N .................. 167 \intarray_rand:n ................. 255
\int_rand:nn ................. 77, 176 \intarray_rand:NN................. 255
\int_set:N .................. 242 \intarray_rand:NNn................. 255
\int_set:NN .................. 168 \intarray_rand_item:NN............. 255
\int_set_eq:NN ............... 168 \intarray_show:NN................. 255
\int_show:N .................. 176 \intarray_show:NNn................. 255
\int_show:n .................. 177 \intarray_walk:NN................ 255
\int_sign:n .................. 166 \ior_close:N .................. 91, 92
\int_step_function:nN ........ 173 \ior_get:NN .................. 92–94, 96
\int_step_function:nnN ......... 173 \ior_get:NNTF .................. 93
\int_step_function:nnnn ........ 73, 173 \ior_get_term:N .................. 96
\int_step_inline:nn .......... 173 \ior_if:EOF:NTF .................. 95
\int_step_inline:nnn .......... 173 \ior_if:EOF:p ................. 95
\int_step_variable:nn ........ 173 \ior_log:N .................. 92
\int_step_variable:nnn ......... 173 \ior_map_break:................ 95
\int_step_variable:nnnn ......... 173 \ior_map_break:nn................. 95
\int_sub:NN .................. 168 \ior_map_inlinen ................. 94
\int_to_Alph:n .............. 174, 175 \ior_map_inlinen ................. 94
\int_to_alph:n .............. 174, 175 \ior_map_variable:NNn............. 94
\int_to_arabic:n ............ 174 \ior_new:NN .................. 91