LuaTEX Reference

beta 0.80.0
Contents

1 Introduction 15
2 Basic \TeX{} enhancements 17
  2.1 Introduction 17
  2.2 Version information 17
  2.3 UNICODE text support 18
  2.4 Extended tables 18
  2.5 Attribute registers 19
    2.5.1 Box attributes 19
  2.6 \LUA{} related primitives 20
    2.6.1 \texttt{\directlua} 20
    2.6.2 \texttt{\luafunction} 21
    2.6.3 \texttt{\latelua} 22
    2.6.4 \texttt{\luaescapestring} 22
  2.7 New \texttt{\epsilon-\TeX} primitives 23
    2.7.1 \texttt{\clearmarks} 23
    2.7.2 \texttt{\noligs} and \texttt{\nokerns} 23
    2.7.3 \texttt{\formatname} 23
    2.7.4 \texttt{\scantextokens} 23
    2.7.5 Verbose versions of single-character alignments commands (0.45) 24
    2.7.6 Catcode tables 24
      2.7.6.1 \texttt{\catcodetable} 24
      2.7.6.2 \texttt{\initcatcodetable} 24
      2.7.6.3 \texttt{\savecatcodetable} 25
    2.7.7 \texttt{\suppressfontnotfounderror} (0.11) 25
    2.7.8 \texttt{\suppresslongerror} (0.36) 25
    2.7.9 \texttt{\suppressifcsnameerror} (0.36) 25
    2.7.10 \texttt{\suppressoutererror} (0.36) 25
    2.7.11 \texttt{\suppressmathparerror} (0.80) 26
    2.7.12 \texttt{\outputbox} (0.37) 26
    2.7.13 Font syntax 26
2.7.14 File syntax (0.45) 26
2.7.15 Images and Forms 27
2.8 Debugging 27
2.9 Global leaders 27
2.10 Expandable character codes (0.75) 27

3 LUA general 29
3.1 Initialization 29
3.1.1 LUATeX as a LUA interpreter 29
3.1.2 LUATeX as a LUA byte compiler 29
3.1.3 Other commandline processing 29
3.2 LUA changes 32
3.3 LUA modules 35

4 LUATeX LUA Libraries 37
4.1 The callback library 37
4.1.1 File discovery callbacks 38
4.1.1.1 find_read_file and find_write_file 38
4.1.1.2 find_font_file 38
4.1.1.3 find_output_file 39
4.1.1.4 find_format_file 39
4.1.1.5 find_vf_file 39
4.1.1.6 find_map_file 39
4.1.1.7 find_enc_file 39
4.1.1.8 find_sfd_file 39
4.1.1.9 find_pk_file 39
4.1.1.10 find_data_file 40
4.1.1.11 find_opentype_file 40
4.1.1.12 find_truetype_file and find_type1_file 40
4.1.1.13 find_image_file 40
4.1.2 File reading callbacks 40
4.1.2.1 open_read_file 40
4.1.2.1.1 reader 41
4.1.2.1.2 close 41
4.1.2.2 General file readers

4.1.3 Data processing callbacks
4.1.3.1 process_input_buffer
4.1.3.2 process_output_buffer
4.1.3.3 process_jobname
4.1.3.4 token_filter

4.1.4 Node list processing callbacks
4.1.4.1 buildpage_filter
4.1.4.2 pre_linebreak_filter
4.1.4.3 linebreak_filter
4.1.4.4 post_linebreak_filter
4.1.4.5 hpack_filter
4.1.4.6 vpack_filter
4.1.4.7 pre_output_filter
4.1.4.8 hyphenate
4.1.4.9 ligaturing
4.1.4.10 kerning
4.1.4.11 mlist_to_hlist

4.1.5 Information reporting callbacks
4.1.5.1 pre_dump
4.1.5.2 start_run
4.1.5.3 stop_run
4.1.5.4 start_page_number
4.1.5.5 stop_page_number
4.1.5.6 show_error_hook
4.1.5.7 show_error_message
4.1.5.8 show_lua_error_hook
4.1.5.9 start_file
4.1.5.10 stop_file

4.1.6 PDF-related callbacks
4.1.6.1 finish_pdfpage
4.1.6.2 finish_pdfpage

4.1.7 Font-related callbacks
4.1.7.1 define_font

4.2 The epdf library

4.3 The font library

4.3.1 Loading a TFM file

4.3.2 Loading a VF file

4.3.3 The fonts array

4.3.4 Checking a font’s status

4.3.5 Defining a font directly

4.3.6 Projected next font id

4.3.7 Font id (0.47)

4.3.8 Currently active font

4.3.9 Maximum font id

4.3.10 Iterating over all fonts

4.4 The fontloader library (0.36)

4.4.1 Getting quick information on a font

4.4.2 Loading an OPENTYPE or TRUETYPE file

4.4.3 Applying a ‘feature file’

4.4.4 Applying an ‘AFM file’

4.4.5 Fontloader font tables

4.4.5.1 Table types

4.4.5.1.1 Top-level

4.4.5.1.2 Glyph items

4.4.5.1.3 map table

4.4.5.1.4 private table

4.4.5.1.5 cidinfo table

4.4.5.1.6 pfminfo table

4.4.5.1.7 names table

4.4.5.1.8 anchor_classes table

4.4.5.1.9 gpos table

4.4.5.1.10 gsub table

4.4.5.1.11 ttf_tables and ttf_tab_saved tables

4.4.5.1.12 mm table

4.4.5.1.13 mark_classes table (0.44)
4.4.5.1.14 math table
4.4.5.1.15 validation_state table
4.4.5.1.16 horiz_base and vert_base table
4.4.5.1.17 altuni table
4.4.5.1.18 vert_variants and horiz_variants table
4.4.5.1.19 mathkern table
4.4.5.1.20 kerns table
4.4.5.1.21 vkerns table
4.4.5.1.22 texdata table
4.4.5.1.23 lookups table

4.5 The img library
4.5.1 img.new
4.5.2 img.keys
4.5.3 img.scan
4.5.4 img.copy
4.5.5 img.write
4.5.6 img.immediatewrite
4.5.7 img.node
4.5.8 img.types
4.5.9 img.boxes

4.6 The kpse library
4.6.1 kpse.set_program_name and kpse.new
4.6.2 find_file
4.6.3 lookup
4.6.4 init_prog
4.6.5 readable_file
4.6.6 expand_path
4.6.7 expand_var
4.6.8 expand_braces
4.6.9 show_path
4.6.10 var_value
4.6.11 version

4.7 The lang library
4.8 The lua library
4.8.1 LUA bytecode registers
4.8.2 LUA chunk name registers
4.9 The mplib library
4.9.1 mplib.new
4.9.2 mp:statistics
4.9.3 mp:execute
4.9.4 mp:finish
4.9.5 Result table
4.9.5.1 fill
4.9.5.2 outline
4.9.5.3 text
4.9.5.4 special
4.9.5.5 start_bounds, start_clip
4.9.5.6 stop_bounds, stop_clip
4.9.6 Subsidiary table formats
4.9.6.1 Paths and pens
4.9.6.2 Colors
4.9.6.3 Transforms
4.9.6.4 Dashes
4.9.7 Character size information
4.9.7.1 mp:char_width
4.9.7.2 mp:char_height
4.9.7.3 mp:char_depth
4.10 The node library
4.10.1 Node handling functions
4.10.1.1 node.is_node
4.10.1.2 node.types
4.10.1.3 node.whatsits
4.10.1.4 node.id
4.10.1.5 node.subtype
4.10.1.6 node.type
4.10.1.7 node.fields
4.10.1.8 node.has_field
4.10.1.9 node.new
4.10.1.10 node.free
4.10.1.11 node.flush_list
4.10.1.12 node.copy
4.10.1.13 node.copy_list
4.10.1.14 node.next (0.65)
4.10.1.15 node.prev (0.65)
4.10.1.16 node.current_attr (0.66)
4.10.1.17 node.hpack
4.10.1.18 node.vpack (since 0.36)
4.10.1.19 node.dimensions (0.43)
4.10.1.20 node.mlist_to_hlist
4.10.1.21 node.slide
4.10.1.22 node.tail
4.10.1.23 node.length
4.10.1.24 node.count
4.10.1.25 node.traverse
4.10.1.26 node.traverse_id
4.10.1.27 node.end_of_math (0.76)
4.10.1.28 node.remove
4.10.1.29 node.insert_before
4.10.1.30 node.insert_after
4.10.1.31 node.first_glyph (0.65)
4.10.1.32 node.ligaturing
4.10.1.33 node.kerning
4.10.1.34 node.unprotect_glyphs
4.10.1.35 node.protect_glyphs
4.10.1.36 node.last_node
4.10.1.37 node.write
4.10.1.38 node.protrusion_skippable (0.60.1)
4.10.2 Attribute handling
4.10.2.1 node.has_attribute
4.10.2.2 node.set_attribute
4.10.2.3 node.unset_attribute

4.11 The pdf library

4.11.1 pdf.mapfile, pdf.mapline (new in 0.53.0)
4.11.2 pdf.catalog, pdf.info, pdf.names, pdf.trailer (new in 0.53.0)
4.11.3 pdf.<set/get>pageattributes, pdf.<set/get>pageresources, pdf.<set/get>pagesattributes

4.11.4 pdf.h, pdf.v
4.11.5 pdf.getpos, pdf.gethpos, pdf.getvpos
4.11.6 pdf.hasmatrix, pdf.getmatrix
4.11.7 pdf.print
4.11.8 pdf.immediateobj
4.11.9 pdf.obj
4.11.10 pdf.refobj
4.11.11 pdf.reserveobj
4.11.12 pdf.registerannot (new in 0.47.0)

4.12 The pdfscanner library (new in 0.72.0)

4.13 The status library

4.14 The tex library

4.14.1 Internal parameter values
4.14.1.1 Integer parameters
4.14.1.2 Dimension parameters
4.14.1.3 Direction parameters
4.14.1.4 Glue parameters
4.14.1.5 Muglue parameters
4.14.1.6 Tokenlist parameters
4.14.2 Convert commands
4.14.3 Last item commands
4.14.4 Attribute, count, dimension, skip and token registers
4.14.5 Character code registers (0.63)
4.14.6 Box registers
4.14.7 Math parameters
4.14.8 Special list heads
4.14.9  Semantic nest levels (0.51) 125
4.14.10 Print functions 125
4.14.10.1 \texttt{tex.print} 126
4.14.10.2 \texttt{tex.sprint} 126
4.14.10.3 \texttt{tex.tprint} 127
4.14.10.4 \texttt{tex.write} 127
4.14.11 Helper functions 127
4.14.11.1 \texttt{tex.round} 127
4.14.11.2 \texttt{tex.scale} 127
4.14.11.3 \texttt{tex.sp} (0.51) 128
4.14.11.4 \texttt{tex.definefont} 128
4.14.11.5 \texttt{tex.error} (0.61) 128
4.14.11.6 \texttt{tex.hashtokens} (0.25) 128
4.14.12 Functions for dealing with primitives 129
4.14.12.1 \texttt{tex.enableprimitives} 129
4.14.12.2 \texttt{tex.extraprimitives} 130
4.14.12.3 \texttt{tex.primitives} 133
4.14.13 Core functionality interfaces 133
4.14.13.1 \texttt{tex.badness} (0.53) 133
4.14.13.2 \texttt{tex.linebreak} (0.53) 134
4.14.13.3 \texttt{tex.shipout} (0.51) 135
4.15 The \texttt{texconfig} table 135
4.16 The \texttt{texio} library 136
4.16.1 Printing functions 136
4.16.1.1 \texttt{texio.write} 137
4.16.1.2 \texttt{texio.write_nI} 137
4.17 The \texttt{token} library 137
4.17.1 \texttt{token.get_next} 137
4.17.2 \texttt{token.is_expandable} 137
4.17.3 \texttt{token.expand} 138
4.17.4 \texttt{token.is_activechar} 138
4.17.5 \texttt{token.create} 138
4.17.6 \texttt{token.command_name} 138
4.17.7  \texttt{token.command\_id}  
4.17.8  \texttt{token.csname\_name}  
4.17.9  \texttt{token.csname\_id}  
4.17.10 The \texttt{newtoken} library

5  Math
5.1  The current math style
5.1.1  \texttt{\mathstyle}  
5.1.2  \texttt{\Ustack}  
5.2  Unicode math characters  
5.3  Cramped math styles  
5.4  Math parameter settings  
5.5  Font-based Math Parameters  
5.6  Math spacing setting  
5.7  Math accent handling  
5.8  Math root extension  
5.9  Math kerning in super- and subscripts  
5.10  Scripts on horizontally extensible items like arrows  
5.11  Extensible delimiters  
5.12  Other Math changes
5.12.1  Verbose versions of single-character math commands  
5.12.2  Allowed math commands in non-math modes  
5.13  Math todo  

6  Languages and characters, fonts and glyphs
6.1  Characters and glyphs  
6.2  The main control loop  
6.3  Loading patterns and exceptions  
6.4  Applying hyphenation  
6.5  Applying ligatures and kerning  
6.6  Breaking paragraphs into lines
7  Font structure 163
7.1  Real fonts 168
7.2  Virtual fonts 170
7.2.1  Artificial fonts 172
7.2.2  Example virtual font 172

8  Nodes 175
8.1  LUA node representation 175
8.1.1  Auxiliary items 175
8.1.1.1  glue_spec items 175
8.1.1.2  attribute_list and attribute items 176
8.1.1.3  action item 176
8.1.2  Main text nodes 177
8.1.2.1  hlist nodes 177
8.1.2.2  vlist nodes 178
8.1.2.3  rule nodes 178
8.1.2.4  ins nodes 178
8.1.2.5  mark nodes 178
8.1.2.6  adjust nodes 179
8.1.2.7  disc nodes 179
8.1.2.8  math nodes 179
8.1.2.9  glue nodes 180
8.1.2.10  kern nodes 180
8.1.2.11  penalty nodes 181
8.1.2.12  glyph nodes 181
8.1.2.13  margin_kern nodes 182
8.1.3  Math nodes 182
8.1.3.1  Math kernel subnodes 182
8.1.3.1.1  math_char and math_text_char subnodes 182
8.1.3.1.2  sub_box and sub_mlist subnodes 183
8.1.3.2  Math delimiter subnode 183
8.1.3.2.1  delim subnodes 183
8.1.3.3  Math core nodes 184
8.1.3.3.1  simple nodes 184
8.1.3.3.2 accent nodes 184
8.1.3.3.3 style nodes 185
8.1.3.3.4 choice nodes 185
8.1.3.3.5 radical nodes 185
8.1.3.3.6 fraction nodes 186
8.1.3.3.7 fence nodes 186
8.1.4 whatsit nodes 186
8.1.4.1 open nodes 186
8.1.4.2 write nodes 187
8.1.4.3 close nodes 187
8.1.4.4 special nodes 187
8.1.4.5 language nodes 187
8.1.4.6 local_par nodes 187
8.1.4.7 dir nodes 188
8.1.4.8 pdf_literal nodes 188
8.1.4.9 pdf_refobj nodes 189
8.1.4.10 pdf_refxform nodes 189
8.1.4.11 pdf_refximage nodes 189
8.1.4.12 pdf_annot nodes 190
8.1.4.13 pdf_start_link nodes 190
8.1.4.14 pdf_end_link nodes 190
8.1.4.15 pdf_dest nodes 190
8.1.4.16 pdf_thread nodes 191
8.1.4.17 pdf_start_thread nodes 191
8.1.4.18 pdf_end_thread nodes 191
8.1.4.19 pdf_save_pos nodes 192
8.1.4.20 late_lua nodes 192
8.1.4.21 pdf_colorstack nodes 192
8.1.4.22 pdf_setmatrix nodes 192
8.1.4.23 pdf_save nodes 193
8.1.4.24 pdf_restore nodes 193
8.1.4.25 user_defined nodes 193
8.2 Two access models 194
9 Modifications 199
9.1 Changes from TeX 3.1415926 199
9.2 Changes from $\varepsilon$-TeX 2.2 199
9.3 Changes from PDFTeX 1.40 199
9.4 Changes from ALEPH RC4 200
9.5 Changes from standard WEB2C 201

10 Implementation notes 203
10.1 Primitives overlap 203
10.2 Memory allocation 203
10.3 Sparse arrays 203
10.4 Simple single-character csnames 204
10.5 Compressed format 204
10.6 Binary file reading 204

11 Known bugs and limitations, TODO 205
1 Introduction

This book will eventually become the reference manual of LuaTeX. At the moment, it simply reports the behavior of the executable matching the snapshot or beta release date in the title page.

Features may come and go. The current version of LuaTeX is not meant for production and users cannot depend on stability, nor on functionality staying the same.

Nothing is considered stable just yet. This manual therefore simply reflects the current state of the executable. Absolutely nothing on the following pages is set in stone. When the need arises, anything can (and will) be changed.

LuaTeX consists of a number of interrelated but (still) distinguishable parts:

- PDF\TeX\ version 1.40.9, converted to C (with patches from later releases).
- The direction model and some other bits from Aleph RC4 converted to C.
- Lua 5.2.1
- dedicated Lua libraries
- various \TeX\ extensions
- parts of FontForge 2008.11.17
- the MetaPost library
- newly written compiled source code to glue it all together

Neither Aleph’s I/O translation processes, nor tcx files, nor Enc\TeX\ can be used, these encoding-related functions are superseded by a Lua-based solution (reader callbacks). Also, some experimental PDF\TeX\ features are removed. These can be implemented in Lua instead.
2 Basic \TeX{} enhancements

2.1 Introduction

From day one, Lua\TeX{} has offered extra functionality when compared to the superset of PDF\TeX{} and \Alph. That has not been limited to the possibility to execute \La{} via \texttt{\directlua}, but Lua\TeX{} also adds functionality via new \TeX{}-side primitives.

However, starting with beta 0.39.0, most of that functionality is hidden by default. When Lua\TeX{} 0.40.0 starts up in 'iniluatex' mode (\texttt{luatex -ini}), it defines only the primitive commands known by \TeX{}82 and the one extra command \texttt{\directlua}.

As is fitting, a LUA function has to be called to add the extra primitives to the user environment. The simplest method to get access to all of the new primitive commands is by adding this line to the format generation file:

\begin{verbatim}
\directlua { tex.enableprimitives('',tex.extraprimitives()) }
\end{verbatim}

But be aware that the curly braces may not have the proper \texttt{\catcode} assigned to them at this early time (giving a 'Missing number' error), so it may be needed to put these assignments

\begin{verbatim}
\catcode `\{=1
\catcode `\} =2
\end{verbatim}

before the above line. More fine-grained primitives control is possible, you can look up the details in section 4.14.12. For simplicity's sake, this manual assumes that you have executed the \texttt{\directlua} command as given above.

The startup behavior documented above is considered stable in the sense that there will not be backward-incompatible changes any more.

2.2 Version information

There are three new primitives to test the version of Lua\TeX{}:

<table>
<thead>
<tr>
<th>primitive</th>
<th>explanation</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\luatexbanner}</td>
<td>the banner as reported on the command line</td>
<td>This is Lua\TeX{}, Version beta-0.80.1 (TeX Live 2015)</td>
</tr>
<tr>
<td>\texttt{\luatexversion}</td>
<td>a combination of major and minor number</td>
<td>80</td>
</tr>
<tr>
<td>\texttt{\luatexrevision}</td>
<td>the revision number, the current value is</td>
<td>1</td>
</tr>
</tbody>
</table>

The official Lua\TeX{} version is defined as follows:
The major version is the integer result of `\luatexversion` divided by 100. The primitive is an 'internal variable', so you may need to prefix its use with `\the` depending on the context.

- The minor version is the two-digit result of `\luatexversion` modulo 100.
- The revision is given by `\luatexrevision`. This primitive expands to a positive integer.
- The full version number consists of the major version, minor version and revision, separated by dots.

### 2.3 UNICODE text support

Text input and output is now considered to be UNICODE text, so input characters can use the full range of UNICODE \(2^{20} + 2^{16} - 1 = \text{0x10FFFF}\).

Later chapters will talk of characters and glyphs. Although these are not interchangeable, they are closely related. During typesetting, a character is always converted to a suitable graphic representation of that character in a specific font. However, while processing a list of to-be-typeset nodes, its contents may still be seen as a character. Inside LuaTEX there is not yet a clear separation between the two concepts. Until this is implemented, please do not be too harsh on us if we make errors in the usage of the terms.

A few primitives are affected by this, all in a similar fashion: each of them has to accommodate for a larger range of acceptable numbers. For instance, `\char` now accepts values between 0 and 1,114,111.

This should not be a problem for well-behaved input files, but it could create incompatibilities for input that would have generated an error when processed by older \TeX-based engines. The affected commands with an altered initial (left of the equals sign) or secondary (right of the equals sign) value are: `\char`, `\lccode`, `\uccode`, `\catcode`, `\sfcode`, `\efcode`, `\lpcode`, `\rpcode`, `\chardef`.

As far as the core engine is concerned, all input and output to text files is UTF-8 encoded. Input files can be pre-processed using the `reader` callback. This will be explained in a later chapter.

Output in byte-sized chunks can be achieved by using characters just outside of the valid UNICODE range, starting at the value 1,114,112 (0x110000). When the time comes to print a character \(c \geq 1,114,112\), LuaTEX will actually print the single byte corresponding to \(c\) minus 1,114,112.

Output to the terminal uses `\^\text{\char}` notation for the lower control range \((c < 32)\), with the exception of `\^\text{I}`, `\^\text{J}` and `\^\text{M}`. These are considered 'safe' and therefore printed as-is.

Normalization of the UNICODE input can be handled by a macro package during callback processing (this will be explained in section 4.1.2).

### 2.4 Extended tables

All traditional \TeX and \$\epsilon$-\TeX registers can be 16-bit numbers as in \textsc{Aleph}. The affected commands are:

```
\count \toks \toksdef \unhcopy
\dimen \countdef \box \unvcopy
\skip \dimendef \unhbox \wd
\muskip \skipdef \unvbox \ht
\marks \muskipdef \copy \dp
```
The glyph properties (like \efcode) introduced in pdFTeX that deal with font expansion (hz) and character protruding are also 16-bit. Because font memory management has been rewritten, these character properties are no longer shared among fonts instances that originate from the same metric file.

The behavior documented in the above section is considered stable in the sense that there will not be backward-incompatible changes any more.

2.5 Attribute registers

Attributes are a completely new concept in LuaTeX. Syntactically, they behave a lot like counters: attributes obey \TeX's nesting stack and can be used after \the etc. just like the normal \count registers.

\attribute \{16-bit number\} \{optional equals\} \{32-bit number\}
\attributedef \{csname\} \{optional equals\} \{16-bit number\}

Conceptually, an attribute is either 'set' or 'unset'. Unset attributes have a special negative value to indicate that they are unset, that value is the lowest legal value: \texttt{-7FFFFFFF} in hexadecimal, a.k.a. \texttt{-2147483647} in decimal. It follows that the value \texttt{-7FFFFFFF} cannot be used as a legal attribute value, but you can assign \texttt{-7FFFFFFF} to 'unset' an attribute. All attributes start out in this 'unset' state in iniTeX (prior to 0.37, there could not be valid negative attribute values, and the 'unset' value was \texttt{-1}).

Attributes can be used as extra counter values, but their usefulness comes mostly from the fact that the numbers and values of all 'set' attributes are attached to all nodes created in their scope. These can then be queried from any Lua code that deals with node processing. Further information about how to use attributes for node list processing from Lua is given in chapter 8.

The behavior documented in the above subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

2.5.1 Box attributes

Nodes typically receive the list of attributes that is in effect when they are created. This moment can be quite asynchronous. For example: in paragraph building, the individual line boxes are created after the \par command has been processed, so they will receive the list of attributes that is in effect then, not the attributes that were in effect in, say, the first or third line of the paragraph.

Similar situations happen in LuaTeX regularly. A few of the more obvious problematic cases are dealt with: the attributes for nodes that are created during hyphenation, kerning and ligaturing borrow their attributes from their surrounding glyphs, and it is possible to influence box attributes directly.

When you assemble a box in a register, the attributes of the nodes contained in the box are unchanged when such a box is placed, unboxed, or copied. In this respect attributes act the same as characters that have been converted to references to glyphs in fonts. For instance, when you use attributes to implement color support, each node carries information about its eventual color. In that case, unless you implement mechanisms that deal with it, applying a color to already boxed material will have no effect. Keep in
mind that this incompatibility is mostly due to the fact that separate specials and literals are a more unnatural approach to colors than attributes.

It is possible to fine-tune the list of attributes that are applied to a \hbox, \vbox or \vtop by the use of the keyword \texttt{attr}. An example:

\begin{verbatim}
\attribute2=5
\setbox0=\hbox {Hello}
\setbox2=\hbox attr1=12 attr2=-"7FFFFFFF{Hello}
\end{verbatim}

This will set the attribute list of box 2 to $1 = 12$, and the attributes of box 0 will be $2 = 5$. As you can see, assigning the maximum negative value causes an attribute to be ignored.

The \texttt{attr} keyword(s) should come before a \texttt{to} or \texttt{spread}, if that is also specified.

\section*{2.6 LUA related primitives}

In order to merge LUA code with TeX input, a few new primitives are needed.

\subsection*{2.6.1 \texttt{directlua}}

The primitive \texttt{directlua} is used to execute LUA code immediately. The syntax is

\begin{verbatim}
\directlua \langle general text \rangle
\directlua name \langle general text \rangle \langle general text \rangle
\directlua \langle 16-bit number \rangle \langle general text \rangle
\end{verbatim}

The last \langle general text \rangle is expanded fully, and then fed into the LUA interpreter. After reading and expansion has been applied to the \langle general text \rangle, the resulting token list is converted to a string as if it was displayed using \texttt{\the\toks}. On the LUA side, each \texttt{directlua} block is treated as a separate chunk. In such a chunk you can use the \texttt{local} directive to keep your variables from interfering with those used by the macro package.

The conversion to and from a token list means that you normally can not use LUA line comments (starting with \texttt{--}) within the argument. As there typically will be only one ‘line’ the first line comment will run on until the end of the input. You will either need to use TeX-style line comments (starting with \texttt{\%}), or change the \TeX{} category codes locally. Another possibility is to say:

\begin{verbatim}
\begingroup
\endlinechar=10
\directlua ...
\endgroup
\end{verbatim}

Then LUA line comments can be used, since \TeX{} does not replace line endings with spaces.

The \texttt{name} \langle general text \rangle specifies the name of the LUA chunk, mainly shown in the stack backtrace of error messages created by LUA code. The \langle general text \rangle is expanded fully, thus macros can be used to generate the chunk name, i.e.
\directlua name{\jobname:\the\inputlineno} ...

to include the name of the input file as well as the input line into the chunk name.
Likewise, the ⟨16-bit number⟩ designates a name of a LUA chunk, but in this case the name will be taken from the \lua.name array (see the documentation of the \lua table further in this manual). This syntax is new in version 0.36.0.
The chunk name should not start with a @, or it will be displayed as a file name (this is a quirk in the current LUA implementation).
The \directlua command is expandable. Since it passes LUA code to the LUA interpreter its expansion from the \TeX viewpoint is usually empty. However, there are some LUA functions that produce material to be read by \TeX, the so called print functions. The most simple use of these is \texttt{tex.print(<string> s)}. The characters of the string \texttt{s} will be placed on the \TeX input buffer, that is, 'before \TeX's eyes' to be read by \TeX immediately. For example:
\begin{verbatim}
\count10=20
a\directlua{tex.print(tex.count[10]+5)}b
\end{verbatim}
expands to

a25b

Here is another example:
\begin{verbatim}
$\pi = \directlua{tex.print(math.pi)}$
\end{verbatim}
will result in
\[\pi = 3.1415926535898\]
Note that the expansion of \directlua is a sequence of characters, not of tokens, contrary to all \TeX commands. So formally speaking its expansion is null, but it places material on a pseudo-file to be immediately read by \TeX, as e-\TeX's \texttt{\scantokens}.

For a description of print functions look at section \texttt{section 4.14.10}.

Because the ⟨general text⟩ is a chunk, the normal LUA error handling is triggered if there is a problem in the included code. The LUA error messages should be clear enough, but the contextual information is still pretty bad. Often, you will only see the line number of the right brace at the end of the code.

While on the subject of errors: some of the things you can do inside LUA code can break up LUA\TeX pretty bad. If you are not careful while working with the node list interface, you may even end up with assertion errors from within the \TeX portion of the executable.

The behavior documented in the above subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

\section*{2.6.2 \texttt{\luafunction}}

The \directlua commands involves tokenization of its argument (after picking up an optional name or number specification). The tokenlist is then converted into a string and given to LUA to turn into a function
that is called. The overhead is rather small but when you use this primitive hundreds or thousands of
times, it can become noticeable. For this reason there is a variant call available: `{\texttt{\textbackslash luafunction}}`. This
command is used as follows:

\begin{verbatim}
\directlua{
local t = lua.get_functions_table()
t[1] = function() tex.print("!") end
  t[2] = function() tex.print("?") end
}
\end{verbatim}

Of course the functions can also be defined in a separate file. There is no limit on the number of functions
apart from normal Lua limitations. Of course there is the limitation of no arguments but that would involve
parsinging and thereby give no gain. The function, when called in fact gets one argument, being the index,
so in:

\begin{verbatim}
\directlua{
local t = lua.get_functions_table()
t[8] = function(slot) tex.print(slot) end
}
\end{verbatim}

the number 8 gets typeset.

### 2.6.3 \texttt{\textbackslash latelua}

`{\texttt{\textbackslash latelua}}` stores Lua code in a whatsit that will be processed at the time of shipping out. Its intended
use is a cross between `{\texttt{\textbackslash pdfliteral}}` and `{\texttt{\textbackslash write}}`. Within the Lua code you can print PDF statements
directly to the PDF file via `{\texttt{\textbackslash pdf.print}}`, or you can write to other output streams via `{\texttt{\textbackslash texio.write}}`
or simply using Lua’s I/O routines.

\begin{verbatim}
{\texttt{\textbackslash latelua}} \langle general text \rangle
{\texttt{\textbackslash latelua name}} \langle general text \rangle \langle general text \rangle
{\texttt{\textbackslash latelua}} \langle 16-bit number \rangle \langle general text \rangle
\end{verbatim}

Expansion of macros etcetera in the final `<general text>` is delayed until just before the whatsit is executed (like in `{\texttt{\textbackslash write}}`). With regard to PDF output stream `{\texttt{\textbackslash latelua}}` behaves as `{\texttt{\textbackslash pdfliteral page}}.

The name `<general text>` and `<16-bit number>` behave in the same way as they do for `{\texttt{\textbackslash directlua}}`.

### 2.6.4 \texttt{\textbackslash luaescapestring}

This primitive converts a TeX token sequence so that it can be safely used as the contents of a Lua string:
embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This

![image](image-url)
is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to \texttt{n} and \texttt{r} respectively. The token sequence is fully expanded.

\texttt{\luaescapestring{⟨general text⟩}}

Most often, this command is not actually the best way to deal with the differences between the \TeX{} and Lua. In very short bits of Lua code it is often not needed, and for longer stretches of Lua code it is easier to keep the code in a separate file and load it using Lua’s \texttt{dofile}:

\texttt{\directlua{dofile('mysetups.lua')}}

\section*{2.7 New $\epsilon$-\TeX{} primitives}

\subsection*{2.7.1 \texttt{\clearmarks}}

This primitive clears a mark class completely, resetting all three connected mark texts to empty.

\texttt{\clearmarks⟨16-bit number⟩}

\subsection*{2.7.2 \texttt{\noligs and \nokerns}}

These primitives prohibit ligature and kerning insertion at the time when the initial node list is built by Lua\TeX{}'s main control loop. They are part of a temporary trick and will be removed in the near future. For now, you need to enable these primitives when you want to do node list processing of ‘characters’, where \TeX{}’s normal processing would get in the way.

\texttt{\noligs⟨integer⟩}
\texttt{\nokerns⟨integer⟩}

These primitives can now be implemented by overloading the ligature building and kerning functions, i.e. by assigning dummy functions to their associated callbacks.

\subsection*{2.7.3 \texttt{\formatname}}

\texttt{\formatname}’s syntax is identical to \texttt{\jobname}.

In ini\TeX{}, the expansion is empty. Otherwise, the expansion is the value that \texttt{\jobname} had during the ini\TeX{} run that dumped the currently loaded format.

\subsection*{2.7.4 \texttt{\scantextokens}}

The syntax of \texttt{\scantextokens} is identical to \texttt{\scantokens}. This primitive is a slightly adapted version of $\epsilon$-\TeX{}’s \texttt{\scantokens}. The differences are:
• The last (and usually only) line does not have a \endlinechar appended
• \scantextokens never raises an EOF error, and it does not execute \everyeof tokens.
• The ‘... while end of file ...’ error tests are not executed, allowing the expansion to end on a
different grouping level or while a conditional is still incomplete.

2.7.5 Verbose versions of single-character alignents commands (0.45)

Lua\TeX{} defines two new primitives that have the same function as \# and & in alignments:

<table>
<thead>
<tr>
<th>primitive</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\alignmark</td>
<td>Duplicates the functionality of # inside alignment preambles</td>
</tr>
<tr>
<td>\aligntab</td>
<td>Duplicates the functionality of &amp; inside alignments (and preambles)</td>
</tr>
</tbody>
</table>

2.7.6 Catcode tables

Catcode tables are a new feature that allows you to switch to a predefined catcode regime in a single
statement. You can have a practically unlimited number of different tables.

The subsystem is backward compatible: if you never use the following commands, your document will
not notice any difference in behavior compared to traditional \TeX{}.

The contents of each catcode table is independent from any other catcode tables, and their contents is
stored and retrieved from the format file.

2.7.6.1 \catcodetable

\catcodetable ⟨15-bit number⟩

The primitive \catcodetable switches to a different catcode table. Such a table has to be previously
created using one of the two primitives below, or it has to be zero. Table zero is initialized by ini\TeX{}.

2.7.6.2 \initcatcodetable

\initcatcodetable ⟨15-bit number⟩

The primitive \initcatcodetable creates a new table with catcodes identical to those defined by
ini\TeX{}:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 \</td>
<td>escape</td>
</tr>
<tr>
<td>5 ^M</td>
<td>return</td>
</tr>
<tr>
<td>9 ^@</td>
<td>null</td>
</tr>
<tr>
<td>10 &lt;space&gt;</td>
<td>space</td>
</tr>
<tr>
<td>11 a–z</td>
<td>letter</td>
</tr>
<tr>
<td>11 A–Z</td>
<td>letter</td>
</tr>
<tr>
<td>12 everything else</td>
<td>other</td>
</tr>
</tbody>
</table>
The new catcode table is allocated globally: it will not go away after the current group has ended. If the supplied number is identical to the currently active table, an error is raised.

2.7.6.3 \savecatcodetable

\savecatcodetable (15-bit number)

\savecatcodetable copies the current set of catcodes to a new table with the requested number. The definitions in this new table are all treated as if they were made in the outermost level.

The new table is allocated globally: it will not go away after the current group has ended. If the supplied number is the currently active table, an error is raised.

2.7.7 \suppressfontnotfounderror (0.11)

\suppressfontnotfounderror = 1

If this new integer parameter is non-zero, then LuaTeX will not complain about font metrics that are not found. Instead it will silently skip the font assignment, making the requested csname for the font \ifx equal to \nullfont, so that it can be tested against that without bothering the user.

2.7.8 \suppresslongerror (0.36)

\suppresslongerror = 1

If this new integer parameter is non-zero, then LuaTeX will not complain about \par commands encountered in contexts where that is normally prohibited (most prominently in the arguments of non-long macros).

2.7.9 \suppressifcsnameerror (0.36)

\suppressifcsnameerror = 1

If this new integer parameter is non-zero, then LuaTeX will not complain about non-expandable commands appearing in the middle of a \ifcsname expansion. Instead, it will keep getting expanded tokens from the input until it encounters an \endcsname command. Use with care! This command is experimental: if the input expansion is unbalanced wrt. \csname \endcsname pairs, the LuaTeX process may hang indefinitely.

2.7.10 \suppressoutererror (0.36)

\suppressoutererror = 1
If this new integer parameter is non-zero, then \texttt{Lua\TeX} will not complain about \texttt{outer} commands encountered in contexts where that is normally prohibited.

The addition of this command coincides with a change in the \texttt{Lua\TeX} engine: ever since the snapshot of 20060915, \texttt{outer} was simply ignored. That behavior has now reverted back to be \texttt{TeX82}-compatible by default.

### 2.7.11 \texttt{\textbackslash suppressmathparerror} (0.80)

The following setting will permit tokens in a math formula:

\[
\texttt{\textbackslash suppressmathparerror = 1}
\]

So, the next code is valid then:

\[
\begin{equation*}
x + 1 = a
\end{equation*}
\]

### 2.7.12 \texttt{\textbackslash outputbox} (0.37)

\[
\texttt{\textbackslash outputbox = 65535}
\]

This new integer parameter allows you to alter the number of the box that will be used to store the page sent to the output routine. Its default value is 255, and the acceptable range is from 0 to 65535.

### 2.7.13 Font syntax

\texttt{Lua\TeX} will accept a braced argument as a font name:

\[
\texttt{\textbackslash \textbf{\textbackslash font} \textbackslash myfont = \{\texttt{cmr10}\}}
\]

This allows for embedded spaces, without the need for double quotes. Macro expansion takes place inside the argument.

### 2.7.14 File syntax (0.45)

\texttt{Lua\TeX} will accept a braced argument as a file name:

\[
\texttt{\textbackslash \textbf{\textbackslash input} \{\texttt{plain}\}}
\]

\[
\texttt{\textbackslash openin 0 \{\texttt{plain}\}}
\]

This allows for embedded spaces, without the need for double quotes. Macro expansion takes place inside the argument.
2.7.15 Images and Forms

\LaTeX{} accepts optional dimension parameters for \texttt{\pdfformximage} and \texttt{\pdfformxform} in the same format as for \texttt{\pdfximage}. With images, these dimensions are then used instead of the ones given to \texttt{\pdfximage}; but the original dimensions are not overwritten, so that a \texttt{\pdfformximage} without dimensions still provides the image with dimensions defined by \texttt{\pdfximage}. These optional parameters are not implemented for \texttt{\pdfformxform}.

\pdfximage width 20mm height 10mm depth 5mm \pdflastximage
\pdfxform width 20mm height 10mm depth 5mm \pdflastxform

2.8 Debugging

If \texttt{\tracingonline} is larger than 2, the node list display will also print the node number of the nodes.

2.9 Global leaders

There is a new experimental primitive: \texttt{\gleaders} (a \LaTeX{} extension, added in 0.43). This type of leaders is anchored to the origin of the box to be shipped out. So they are like normal \texttt{\leaders} in that they align nicely, except that the alignment is based on the largest enclosing box instead of the smallest.

2.10 Expandable character codes (0.75)

The new expandable command \texttt{\Uchar} reads a number between 0 and 1,114,111 and expands to the associated Unicode character.
3 LUA general

3.1 Initialization

3.1.1 LuaTeX as a Lua interpreter

There are some situations that make LuaTeX behave like a standalone Lua interpreter:

- if a `--luaonly` option is given on the commandline, or
- if the executable is named `texlua` (or `luatexlua`), or
- if the only non-option argument (file) on the commandline has the extension `lua` or `luc`.

In this mode, it will set Lua’s `arg[0]` to the found script name, pushing preceding options in negative values and the rest of the commandline in the positive values, just like the Lua interpreter.

LuaTeX will exit immediately after executing the specified Lua script and is, in effect, a somewhat bulky standalone Lua interpreter with a bunch of extra preloaded libraries.

3.1.2 LuaTeX as a Lua byte compiler

There are two situations that make LuaTeX behave like the Lua byte compiler:

- if a `--luaonly` option is given on the commandline, or
- if the executable is named `texluac`

In this mode, LuaTeX is exactly like `luac` from the standalone Lua distribution, except that it does not have the `-l` switch, and that it accepts (but ignores) the `--luaonly` switch.

3.1.3 Other commandline processing

When the LuaTeX executable starts, it looks for the `--lua` commandline option. If there is no `--lua` option, the commandline is interpreted in a similar fashion as in traditional PDFTeX and ALEPH.

The following command-line switches are understood.

```
--fmt=FORMAT        load the format file FORMAT
--lua=FILE          load and execute a Lua initialization script
--safer             disable easily exploitable Lua commands
--nosocket          disable the Lua socket library
--help              display help and exit
--ini               be iniLuatex, for dumping formats
--interaction=STRING set interaction mode (STRING=batchmode/nonstopmode/
                     scrollmode/errorstopmode)
```
--halt-on-error  stop processing at the first error
--kpathsea-debug=NUMBER  set path searching debugging flags according to the bits of NUMBER
--progname=STRING  set the program name to STRING
--version  display version and exit
--credits  display credits and exit
--recorder  enable filename recorder
--etex  ignored
--output-comment=STRING  use STRING for DVI file comment instead of date (no effect for PDF)
--output-directory=DIR  use DIR as the directory to write files to
--draftmode  switch on draft mode (generates no output PDF)
--output-format=FORMAT  use FORMAT for job output; FORMAT is ’dvi’ or ’pdf’
--[no-]shell-escape  disable/enable \write 18\{SHELL COMMAND\}
--enable-write18  enable \write 18\{SHELL COMMAND\}
--disable-write18  disable \write 18\{SHELL COMMAND\}
--shell-restricted  restrict \write 18 to a list of commands given in texmf.cnf
--debug-format  enable format debugging
--[no-]file-line-error  disable/enable file:line:error style messages
--[no-]file-line-error-style  aliases of --[no-]file-line-error
--jobname=STRING  set the job name to STRING
--[no-]parse-first-line  disable/enable parsing of the first line of the input file
--translate-file=  ignored
--default-translate-file=  ignored
--8bit  ignored
--[no-]mktex=FMT  disable/enable mktexFMT generation (FMT=tex/tfm)
--synctex=NUMBER  enable synctex

A note on the creation of the various temporary files and the \jobname. The value to use for \jobname is decided as follows:

- If --jobname is given on the command line, its argument will be the value for \jobname, without any changes. The argument will not be used for actual input so it need not exist. The --jobname switch only controls the \jobname setting.
- Otherwise, \jobname will be the name of the first file that is read from the file system, with any path components and the last extension (the part following the last .) stripped off.
- An exception to the previous point: if the command line goes into interactive mode (by starting with a command) and there are no files input via \everyjob either, then the \jobname is set to texput as a last resort.

The file names for output files that are generated automatically are created by attaching the proper extension (.log, .pdf, etc.) to the found \jobname. These files are created in the directory pointed to by --output-directory, or in the current directory, if that switch is not present.

Without the --lua option, command line processing works like it does in any other web2c-based type-setting engine, except that LuaTEX has a few extra switches.
If the `--lua` option is present, LuaTeX will enter an alternative mode of commandline processing in comparison to the standard web2c programs.

In this mode, a small series of actions is taken in order. First, it will parse the commandline as usual, but it will only interpret a small subset of the options immediately: `--safer`, `--nosocket`, `--[no-]shell-escape`, `--enable-write18`, `--disable-write18`, `--shell-restricted`, `--help`, `--version`, and `--credits`.

Now it searches for the requested Lua initialization script. If it cannot be found using the actual name given on the commandline, a second attempt is made by prepending the value of the environment variable `LUATEXDIR`, if that variable is defined in the environment.

Then it checks the various safety switches. You can use those to disable some Lua commands that can easily be abused by a malicious document. At the moment, `--safer` nils the following functions:

- library functions
  - os: execute, exec, setenv, rename, remove, tmpdir
  - io: popen, output, tmpfile
  - lfs: rmdir, mkdir, chdir, lock, touch

Furthermore, it disables loading of compiled Lua libraries (support for these was added in 0.46.0), and it makes `io.open()` fail on files that are opened for anything besides reading.

`--nosocket` makes the socket library unavailable, so that Lua cannot use networking.

The switches `--[no-]shell-escape`, `--[enable|disable]-write18`, and `--shell-restricted` have the same effects as in PDFTeX, and additionally make `io.popen()`, `os.execute`, `os.exec` and `os.spawn` adhere to the requested option.

Next the initialization script is loaded and executed. From within the script, the entire commandline is available in the Lua table `arg`, beginning with `arg[0]`, containing the name of the executable.

Commandline processing happens very early on. So early, in fact, that none of TeX's initializations have taken place yet. For that reason, the tables that deal with typesetting, like `tex`, `token`, `node` and `pdf`, are off-limits during the execution of the startup file (they are nilled). Special care is taken that `texio.write` and `texio.write_nl` function properly, so that you can at least report your actions to the log file when (and if) it eventually becomes opened (note that TeX does not even know its `\jobname` yet at this point). See chapter 4 for more information about the LuaTeX-specific Lua extension tables.

Everything you do in the Lua initialization script will remain visible during the rest of the run, with the exception of the aforementioned `tex`, `token`, `node` and `pdf` tables: those will be initialized to their documented state after the execution of the script. You should not store anything in variables or within tables with these four global names, as they will be overwritten completely.

We recommend you use the startup file only for your own TeX-independent initializations (if you need any), to parse the commandline, set values in the `texconfig` table, and register the callbacks you need.

LuaTeX allows some of the commandline options to be overridden by reading values from the `texconfig` table at the end of script execution (see the description of the `texconfig` table later on in this document for more details on which ones exactly).
Unless the `texconfig` table tells \LaTeX{} not to initialize `kpathsea` at all (set `texconfig.kpse_init` to `false` for that), \LaTeX{} acts on some more commandline options after the initialization script is finished: in order to initialize the built-in `kpathsea` library properly, \LaTeX{} needs to know the correct program name to use, and for that it needs to check `--progname`, or `--ini` and `--fmt`, if `--progname` is missing.

### 3.2 LUA changes

**NOTE:** \LaTeX{} 0.74.0 is the first version with Lua 5.2, and this is used without any patches to the core, which has some side effects. In particular, Lua’s `tonumber()` may return values in scientific notation, thereby confusing the \TeX{} end of things when it is used as the right-hand side of an assignment to a `\dimen` or `\count`.

**NOTE:** Also in \LaTeX{} 0.74.0 (this is a change in Lua 5.2), loading dynamic Lua libraries will fail if there are two Lua libraries loaded at the same time (which will typically happen on Win32, because there is one Lua 5.2 inside luatex, and another will likely be linked to the `dll` file of the module itself). We plan to fix that later by switching \LaTeX{} itself to using de DLL version of Lua 5.2 inside \LaTeX{} instead of including a static version in the binary.

Starting from version 0.45, \LaTeX{} is able to use the `kpathsea` library to find `require()`d modules. For this purpose, `package.searchers[2]` is replaced by a different loader function, that decides at runtime whether to use `kpathsea` or the built-in core lua function. It uses `kpathsea` when that is already initialized at that point in time, otherwise it reverts to using the normal `package.path` loader.

Initialization of `kpathsea` can happen either implicitly (when \LaTeX{} starts up and the startup script has not set `texconfig.kpse_init` to `false`), or explicitly by calling the Lua function `kpse.set_program_name()`.

Starting from version 0.46.0 \LaTeX{} is also able to use dynamically loadable Lua libraries, unless `--safer` was given as an option on the command line.

For this purpose, `package.searchers[3]` is replaced by a different loader function, that decides at runtime whether to use `kpathsea` or the build-in core lua function. As in the previous paragraph, it uses `kpathsea` when that is already initialized at that point in time, otherwise it reverts to using the normal `package.cpath` loader.

This functionality required an extension to `kpathsea`:

There is a new `kpathsea` file format: `kpse_clua_format` that searches for files with extension `.dll` and `.so`. The `texmf.cnf` setting for this variable is `CLUAINPUTS`, and by default it has this value:

```
CLUAINPUTS=.:$SELFAUTOLOC/lib/{$progname,$engine,}/lua/
```

This path is imperfect (it requires a TDS subtree below the binaries directory), but the architecture has to be in the path somewhere, and the currently simplest way to do that is to search below the binaries directory only.

One level up (a `lib` directory parallel to `bin`) would have been nicer, but that is not doable because \TeX{}Live uses a `bin/<arch>` structure.
In keeping with the other \TeX{}-like programs in \TeX{}Live, the two Lua functions \texttt{os.execute} and \texttt{io.popen} (as well as the two new functions \texttt{os.exec} and \texttt{os.spawn} that are explained below) take the value of \texttt{shell_escape} and/or \texttt{shell_escape_commands} in account. Whenever \LaTeX{} is run with the assumed intention to typeset a document (and by that I mean that it is called as \texttt{latex}, as opposed to \texttt{texlua}, and that the commandline option \texttt{--luaonly} was not given), it will only run the four functions above if the matching \texttt{texmf.cnf} variable(s) or their \texttt{texconfig} (see section 4.15) counterparts allow execution of the requested system command. In ‘script interpreter’ runs of \LaTeX{}, these settings have no effect, and all four functions function as normal. This change is new in 0.37.0.

The \texttt{f:read("*line")] and \texttt{f:lines()} functions from the \texttt{io} library have been adjusted so that they are line-ending neutral: any of LF, CR or CR+LF are acceptable line endings.

\texttt{luafilesystem} has been extended: there are two extra boolean functions (\texttt{lfs.isdir(filename)} and \texttt{lfs.isfile(filename)}) and one extra string field in its attributes table \texttt{(permissions)}. There is an additional function (added in 0.51) \texttt{lfs.shortname()} which takes a file name and returns its short name on WIN32 platforms. On other platforms, it just returns the given argument. The file name is not tested for existence. Finally, for non-WIN32 platforms only, there is the new function \texttt{lfs.readlink()} (added in 0.51) that takes an existing symbolic link as argument and returns its content. It returns an error on WIN32.

The \texttt{string} library has an extra function: \texttt{string.explode(s[,m])}. This function returns an array containing the string argument \texttt{s} split into sub-strings based on the value of the string argument \texttt{m}. The second argument is a string that is either empty (this splits the string into characters), a single character (this splits on each occurrence of that character, possibly introducing empty strings), or a single character followed by the plus sign \texttt{+} (this special version does not create empty sub-strings). The default value for \texttt{m} is ‘ \texttt{+} ‘(multiple spaces).

Note: \texttt{m} is not hidden by surrounding braces (as it would be if this function was written in \TeX{} macros).

The \texttt{string} library also has six extra iterators that return strings piecemeal:

- \texttt{string.utfvalues(s)} (returns an integer value in the UNICODE range)
- \texttt{string.utfcharacters(s)} (returns a string with a single UTF-8 token in it)
- \texttt{string.characters(s)} (a string containing one byte)
- \texttt{string.characterpairs(s)} (two strings each containing one byte) will produce an empty second string if the string length was odd.
- \texttt{string.bytes(s)} (a single byte value)
- \texttt{string.bytepairs(s)} (two byte values) Will produce nil instead of a number as its second return value if the string length was odd.

The \texttt{string.characterpairs()} and \texttt{string.bytepairs()} are useful especially in the conversion of UTF-16 encoded data into UTF-8.

Starting with \LuaTeX{} 0.74, there is also a two-argument form of \texttt{string.dump()}. The second argument is a boolean which, if true, strips the symbols from the dumped data. This matches an extension made in \texttt{luajit}.

Note: The \texttt{string} library functions \texttt{len}, \texttt{lower}, \texttt{sub} etc. are not UNICODE-aware. For strings in the UTF-8 encoding, i.e., strings containing characters above code point 127, the corresponding functions from the \texttt{slunicode} library can be used, e.g., \texttt{unicode.utf8.len}, \texttt{unicode.utf8.lower}
etc. The exceptions are `unicode.utf8.find`, that always returns byte positions in a string, and `unicode.utf8.match` and `unicode.utf8.gmatch`. While the latter two functions in general are Unicode-aware, they fall-back to non-Unicode-aware behavior when using the empty capture () (other captures work as expected). For the interpretation of character classes in `unicode.utf8` functions refer to the library sources at http://luaforge.net/projects/sln. The `slnunicode` library will be replaced by an internal Unicode library in a future LuaTeX version.

The os library has a few extra functions and variables:

- **os.selfdir** is a variable that holds the directory path of the actual executable. For example: /opt/luatex/standalone-mkiv-new/tex/texmf-linux-64/bin (present since 0.27.0).
- **os.exec(commandline)** is a variation on os.execute. The `commandline` can be either a single string or a single table.
  - If the argument is a table: LuaTeX first checks if there is a value at integer index zero. If there is, this is the command to be executed. Otherwise, it will use the value at integer index one. (if neither are present, nothing at all happens).
  - The set of consecutive values starting at integer 1 in the table are the arguments that are passed on to the command (the value at index 1 becomes `arg[0]`). The command is searched for in the execution path, so there is normally no need to pass on a fully qualified pathname.
  - If the argument is a string, then it is automatically converted into a table by splitting on whitespace.
  - In this case, it is impossible for the command and first argument to differ from each other.
  - In the string argument format, whitespace can be protected by putting (part of) an argument inside single or double quotes. One layer of quotes is interpreted by LuaTeX, and all occurrences of \" or \' within the quoted text are un-escaped. In the table format, there is no string handling taking place.
  - This function normally does not return control back to the Lua script: the command will replace the current process. However, it will return the two values nil and 'error' if there was a problem while attempting to execute the command.
  - On Windows, the current process is actually kept in memory until after the execution of the command has finished. This prevents crashes in situations where \TeX{} scripts are run inside integrated \TeX{} environments.
  - The original reason for this command is that it cleans out the current process before starting the new one, making it especially useful for use in \TeX{} scripts.
- **os.spawn(commandline)** is a returning version of os.exec, with otherwise identical calling conventions. If the command ran ok, then the return value is the exit status of the command. Otherwise, it will return the two values nil and 'error'.
- **os.setenv('key','value')** This sets a variable in the environment. Passing nil instead of a value string will remove the variable.
- **os.env** This is a hash table containing a dump of the variables and values in the process environment at the start of the run. It is writeable, but the actual environment is not updated automatically.
- **os.gettimeofday()** Returns the current 'Unix time', but as a float. This function is not available on the SUNOS platforms, so do not use this function for portable documents.
• **os.times()** Returns the current process times according to the UNIX C library function ‘times’. This function is not available on the MS Windows and SunOS platforms, so do not use this function for portable documents.

• **os.tmpdir()** This will create a directory in the ‘current directory’ with the name _luatex.XXXXXX_ where the X-es are replaced by a unique string. The function also returns this string, so you can _lfs.chdir()_ into it, or _nil_ if it failed to create the directory. The user is responsible for cleaning up at the end of the run, it does not happen automatically.

• **os.type** This is a string that gives a global indication of the class of operating system. The possible values are currently _windows_, _unix_, and _msdos_ (you are unlikely to find this value ‘in the wild’).

• **os.name** This is a string that gives a more precise indication of the operating system. These possible values are not yet fixed, and for _os.type_ values _windows_ and _msdos_, the _os.name_ values are simply _windows_ and _msdos_.

  The list for the type _unix_ is more precise: _linux_, _freebsd_, _kfreebsd_ (since 0.51), _cygwin_ (since 0.53), _openbsd_, _solaris_, _sunos_ (pre-solaris), _hpux_, _irix_, _macosx_, _gnu_ (hurd), _bsd_ (unknown, but _bsd_-like), _sysv_ (unknown, but _sysv_-like), _generic_ (unknown). (_os.version_ is planned as a future extension)

• **os.uname()** This function returns a table with specific operating system information acquired at runtime. The keys in the returned table are all string valued, and their names are: _sysname_, _machine_, _release_, _version_, and _nodename_.

In stock Lua, many things depend on the current locale. In _LUAEX_, we can’t do that, because it makes documents unportable. While _LUAEX_ is running if forces the following locale settings:

```
LC_CTYPE=C
LC_COLLATE=C
LC_NUMERIC=C
```

### 3.3 LUA modules

**NOTE:** Starting with _LUAEX_ 0.74, the implied use of the built-in Lua modules in this section is deprecated. If you want to use one of these libraries, please start your source file with a proper _require_ line. In the near future, _LUAEX_ will switch to loading these modules on demand.

Some modules that are normally external to _Lua_ are statically linked in with _LUAEX_, because they offer useful functionality:

• **slnunicode**, from the Selene libraries, http://luaforge.net/projects/sln. (version 1.1)
  This library has been slightly extended so that the _unicode.utf8.*_ functions also accept the first 256 values of plane 18. This is the range _LUAEX_ uses for raw binary output, as explained above.

• **luazip**, from the kepler project, http://www.keplerproject.org/luazip/ . (version 1.2.1, but patched for compilation with _Lua_ 5.2)

• **luafilesystem**, also from the kepler project, http://www.keplerproject.org/luafilesystem/ . (version 1.5.0)

• **lpeg**, by Roberto Ierusalimschy, http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html. (version 0.10.2)
Note: `lpeg` is not UNICODE-aware, but interprets strings on a byte-per-byte basis. This mainly means that `lpeg.S` cannot be used with characters above code point 127, since those characters are encoded using two bytes, and thus `lpeg.S` will look for one of those two bytes when matching, not the combination of the two.

The same is true for `lpeg.R`, although the latter will display an error message if used with characters above code point 127: i.e. `lpeg.R('aâ')` results in the message `bad argument #1 to 'R' (range must have two characters)`, since to `lpeg`, `â` is two 'characters' (bytes), so `aâ` totals three.

- **lzlib**, by Tiago Dionizio, http://luaforge.net/projects/lzlib/. (version 0.2)
- **luasocket**, by Diego Nehab http://w3.impa.br/~diego/software/luasocket/ (version 2.0.2).

Note: the `.lua` support modules from `luasocket` are also preloaded inside the executable, there are no external file dependencies.
4 LUATEX LUA Libraries

NOTE: Starting with LUATEX 0.74, the implied use of the built-in Lua modules epdf, fontloader, mplib, and pdfscanner is deprecated. If you want to use these, please start your source file with a proper require line. In the near future, LUATEX will switch to loading these modules on demand.

The interfacing between TEX and Lua is facilitated by a set of library modules. The Lua libraries in this chapter are all defined and initialized by the LUATEX executable. Together, they allow Lua scripts to query and change a number of TEX's internal variables, run various internal TEX functions, and set up LUATEX's hooks to execute Lua code.

The following sections are in alphabetical order.

4.1 The callback library

This library has functions that register, find and list callbacks.

A quick note on what callbacks are (thanks, Paul!):

Callbacks are entry points to LUATEX's internal operations, which can be interspersed with additional Lua code, and even replaced altogether. In the first case, TEX is simply augmented with new operations (for instance, a manipulation of the nodes resulting from the paragraph builder); in the second case, its hard-coded behavior (for instance, the paragraph builder itself) is ignored and processing relies on user code only.

More precisely, the code to be inserted at a given callback is a function (an anonymous function or the name of a function variable); it will receive the arguments associated with the callback, if any, and must frequently return some other arguments for TEX to resume its operations.

The first task is registering a callback:

\[
\text{id, error} = \text{callback.register (<string> callback_name, <function> func)}
\]
\[
\text{id, error} = \text{callback.register (<string> callback_name, nil)}
\]
\[
\text{id, error} = \text{callback.register (<string> callback_name, false)}
\]

where the callback_name is a predefined callback name, see below. The function returns the internal id of the callback or nil, if the callback could not be registered. In the latter case, error contains an error message, otherwise it is nil.

LUATEX internalizes the callback function in such a way that it does not matter if you redefine a function accidentally.

Callback assignments are always global. You can use the special value nil instead of a function for clearing the callback.

For some minor speed gain, you can assign the boolean false to the non-file related callbacks, doing so will prevent LUATEX from executing whatever it would execute by default (when no callback function is registered at all). Be warned: this may cause all sorts of grief unless you know exactly what you are doing! This functionality is present since version 0.38.
Currently, callbacks are not dumped into the format file.

\begin{table}
\begin{tabular}{|l|l|}
\hline
\textbf{info} & \texttt{callback.list()} \\
\hline
\end{tabular}
\end{table}

The keys in the table are the known callback names, the value is a boolean where \texttt{true} means that the callback is currently set (active).

\begin{function}
f = \texttt{callback.find(callback_name)}
\end{function}

If the callback is not set, \texttt{callback.find} returns \texttt{nil}.

## 4.1.1 File discovery callbacks

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

### 4.1.1.1 find\_read\_file and find\_write\_file

Your callback function should have the following conventions:

\begin{function}
actual_name = function (id_number, asked_name)
\end{function}

\textbf{Arguments}:

- \texttt{id\_number}
  
  This number is zero for the log or \texttt{\textbackslash input} files. For \TeX{}\texttt{\textbackslash read} or \texttt{\textbackslash write} the number is incremented by one, so \texttt{\textbackslash read0} becomes 1.

- \texttt{asked\_name}
  
  This is the user-supplied filename, as found by \texttt{\textbackslash input}, \texttt{\textbackslash openin} or \texttt{\textbackslash openout}.

\textbf{Return value}:

- \texttt{actual\_name}
  
  This is the filename used. For the very first file that is read in by \TeX{}, you have to make sure you return an \texttt{actual\_name} that has an extension and that is suitable for use as \texttt{jobname}. If you don’t, you will have to manually fix the name of the log file and output file after \luatex{} is finished, and an eventual format filename will become mangled. That is because these file names depend on the \texttt{jobname}. You have to return \texttt{nil} if the file cannot be found.

### 4.1.1.2 find\_font\_file

Your callback function should have the following conventions:

\begin{function}
actual_name = function (asked_name)
\end{function}

The \texttt{asked\_name} is an \texttt{OTF} or \texttt{TFM} font metrics file.
Return **nil** if the file cannot be found.

### 4.1.1.3 find_output_file

Your callback function should have the following conventions:

```lua
<string> actual_name = function (<string> asked_name)
```

The *asked_name* is the PDF or DVI file for writing.

### 4.1.1.4 find_format_file

Your callback function should have the following conventions:

```lua
<string> actual_name = function (<string> asked_name)
```

The *asked_name* is a format file for reading (the format file for writing is always opened in the current directory).

### 4.1.1.5 find_vf_file

Like `find_font_file`, but for virtual fonts. This applies to both ALEPH's OVF files and traditional Knuthian VF files.

### 4.1.1.6 find_map_file

Like `find_font_file`, but for map files.

### 4.1.1.7 find_enc_file

Like `find_font_file`, but for enc files.

### 4.1.1.8 find_sfd_file

Like `find_font_file`, but for subfont definition files.

### 4.1.1.9 find_pk_file

Like `find_font_file`, but for pk bitmap files. The argument *asked_name* is a bit special in this case. Its form is

```lua
<base res>dpi/<fontname>.<actual res>pk
```

So you may be asked for `600dpi/manfnt.720pk`. It is up to you to find a ‘reasonable’ bitmap file to go with that specification.
4.1.1.10 find_data_file
Like find_font_file, but for embedded files (\pdfobj file '...').

4.1.1.11 find_opentype_file
Like find_font_file, but for OpenType font files.

4.1.1.12 find_truetype_file and find_type1_file
Your callback function should have the following conventions:

<string> actual_name = function (<string> asked_name)

The asked_name is a font file. This callback is called while LuaTeX is building its internal list of needed
font files, so the actual timing may surprise you. Your return value is later fed back into the matching
read_file callback.

Strangely enough, find_type1_file is also used for OpenType (otf) fonts.

4.1.1.13 find_image_file
Your callback function should have the following conventions:

<string> actual_name = function (<string> asked_name)

The asked_name is an image file. Your return value is used to open a file from the harddisk, so make
sure you return something that is considered the name of a valid file by your operating system.

4.1.2 File reading callbacks
The behavior documented in this subsection is considered stable in the sense that there will not be
backward-incompatible changes any more.

4.1.2.1 open_read_file
Your callback function should have the following conventions:

<table> env = function (<string> file_name)

Argument:

file_name
The filename returned by a previous find_read_file or the return value of kpse.find_file()
if there was no such callback defined.
Return value:

`env`

This is a table containing at least one required and one optional callback function for this file. The required field is `reader` and the associated function will be called once for each new line to be read, the optional one is `close` that will be called once when LuaTeX is done with the file. LuaTeX never looks at the rest of the table, so you can use it to store your private per-file data. Both the callback functions will receive the table as their only argument.

### 4.1.2.1.1 reader

LuaTeX will run this function whenever it needs a new input line from the file.

```lua
function(<table> env)
    return <string> line
end
```

Your function should return either a string or `nil`. The value `nil` signals that the end of file has occurred, and will make TeX call the optional `close` function next.

### 4.1.2.1.2 close

LuaTeX will run this optional function when it decides to close the file.

```lua
function(<table> env)
end
```

Your function should not return any value.

### 4.1.2.2 General file readers

There is a set of callbacks for the loading of binary data files. These all use the same interface:

```lua
function(<string> name)
    return <boolean> success, <string> data, <number> data_size
end
```

The `name` will normally be a full path name as it is returned by either one of the file discovery callbacks or the internal version of `kpse.find_file()`.

- `success`
  - Return `false` when a fatal error occurred (e.g. when the file cannot be found, after all).
- `data`
  - The bytes comprising the file.
- `data_size`
  - The length of the `data`, in bytes.
Return an empty string and zero if the file was found but there was a reading problem.

The list of functions is as follows:

- `read_font_file` ofm or tfm files
- `read_vf_file` virtual fonts
- `read_map_file` map files
- `read_enc_file` encoding files
- `read_sfd_file` subfont definition files
- `read_pk_file` pk bitmap files
- `read_data_file` embedded files (`\pdfobj file` ...)
- `read_truetype_file` TrueType font files
- `read_type1_file` Type1 font files
- `read_opentype_file` OpenType font files

### 4.1.3 Data processing callbacks

#### 4.1.3.1 process_input_buffer

This callback allows you to change the contents of the line input buffer just before \LaTeX actually starts looking at it.

```lua
function(<string> buffer)
    return <string> adjusted_buffer
end
```

If you return `nil`, \LaTeX will pretend like your callback never happened. You can gain a small amount of processing time from that.

This callback does not replace any internal code.

#### 4.1.3.2 process_output_buffer (0.43)

This callback allows you to change the contents of the line output buffer just before \LaTeX actually starts writing it to a file as the result of a `\write` command. It is only called for output to an actual file (that is, excluding the log, the terminal, and `\write18` calls).

```lua
function(<string> buffer)
    return <string> adjusted_buffer
end
```

If you return `nil`, \LaTeX will pretend like your callback never happened. You can gain a small amount of processing time from that.

This callback does not replace any internal code.
4.1.3.3 process_jobname (0.71)

This callback allows you to change the jobname given by `\jobname` in TeX and `tex.jobname` in Lua. It does not affect the internal job name or the name of the output or log files.

```lua
function(<string> jobname)
    return <string> adjusted_jobname
end
```

The only argument is the actual job name; you should not use `tex.jobname` inside this function or infinite recursion may occur. If you return `nil`, LuaTeX will pretend your callback never happened. This callback does not replace any internal code.

4.1.3.4 token_filter

This callback allows you to replace the way LuaTeX fetches lexical tokens.

```lua
function()
    return <table> token
end
```

The calling convention for this callback is a bit more complicated than for most other callbacks. The function should either return a Lua table representing a valid to-be-processed token or tokenlist, or something else like `nil` or an empty table.

If your Lua function does not return a table representing a valid token, it will be immediately called again, until it eventually does return a useful token or tokenlist (or until you reset the callback value to `nil`). See the description of `token` for some handy functions to be used in conjunction with this callback.

If your function returns a single usable token, then that token will be processed by LuaTeX immediately. If the function returns a token list (a table consisting of a list of consecutive token tables), then that list will be pushed to the input stack at a completely new token list level, with its token type set to 'inserted'. In either case, the returned token(s) will not be fed back into the callback function.

Setting this callback to `false` has no effect (because otherwise nothing would happen, forever).

4.1.4 Node list processing callbacks

The description of nodes and node lists is in chapter 8.

4.1.4.1 buildpage_filter

This callback is called whenever LuaTeX is ready to move stuff to the main vertical list. You can use this callback to do specialized manipulation of the page building stage like imposition or column balancing.

```lua
function(<string> extrainfo)
```
end

The string `extrainfo` gives some additional information about what \TeX{}'s state is with respect to the 'current page'. The possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment</td>
<td>a (partial) alignment is being added</td>
</tr>
<tr>
<td>after_output</td>
<td>an output routine has just finished</td>
</tr>
<tr>
<td>box</td>
<td>a typeset box is being added</td>
</tr>
<tr>
<td>new_graf</td>
<td>the beginning of a new paragraph</td>
</tr>
<tr>
<td>vmode_par</td>
<td>\par was found in vertical mode</td>
</tr>
<tr>
<td>hmode_par</td>
<td>\par was found in horizontal mode</td>
</tr>
<tr>
<td>insert</td>
<td>an insert is added</td>
</tr>
<tr>
<td>penalty</td>
<td>a penalty (in vertical mode)</td>
</tr>
<tr>
<td>before_display</td>
<td>immediately before a display starts</td>
</tr>
<tr>
<td>after_display</td>
<td>a display is finished</td>
</tr>
<tr>
<td>end</td>
<td>\LuaTeX{} is terminating (it's all over)</td>
</tr>
</tbody>
</table>

This callback does not replace any internal code.

### 4.1.4.2 pre_linebreak_filter

This callback is called just before \LuaTeX{} starts converting a list of nodes into a stack of \hbox{}es, after the addition of \parfillskip.

```lua
function(<node> head, <string> groupcode)
    return true | false | <node> newhead
end
```

The string called `groupcode` identifies the nodelist's context within \TeX{}'s processing. The range of possibilities is given in the table below, but not all of those can actually appear in `pre_linebreak_filter`, some are for the `hpack_filter` and `vpack_filter` callbacks that will be explained in the next two paragraphs.

<table>
<thead>
<tr>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;empty&gt;</td>
<td>main vertical list</td>
</tr>
<tr>
<td>hbox</td>
<td>\hbox in horizontal mode</td>
</tr>
<tr>
<td>adjusted hbox</td>
<td>\hbox in vertical mode</td>
</tr>
<tr>
<td>vbox</td>
<td>\vbox</td>
</tr>
<tr>
<td>vtop</td>
<td>\vtop</td>
</tr>
<tr>
<td>align</td>
<td>\halign or \align</td>
</tr>
<tr>
<td>disc</td>
<td>discretionaries</td>
</tr>
<tr>
<td>insert</td>
<td>packaging an insert</td>
</tr>
<tr>
<td>vcenter</td>
<td>\vcenter</td>
</tr>
<tr>
<td>local_box</td>
<td>\localleftbox or \localrightbox</td>
</tr>
<tr>
<td>split_off</td>
<td>top of a \vsplit</td>
</tr>
</tbody>
</table>
split_keep  remainder of a \vsplit
align_set   alignment cell
fin_row     alignment row

As for all the callbacks that deal with nodes, the return value can be one of three things:

- boolean true signals successful processing
- \texttt{<node>} signals that the 'head' node should be replaced by the returned node
- boolean false signals that the 'head' node list should be ignored and flushed from memory

This callback does not replace any internal code.

### 4.1.4.3 linebreak_filter

This callback replaces \LaTeX{}'s line breaking algorithm.

\begin{verbatim}
function(<node> head, <boolean> is_display)
    return <node> newhead
end
\end{verbatim}

The returned node is the head of the list that will be added to the main vertical list, the boolean argument is true if this paragraph is interrupted by a following math display.

If you return something that is not a \texttt{<node>}, \LaTeX{} will apply the internal linebreak algorithm on the list that starts at \texttt{<head>}. Otherwise, the \texttt{<node>} you return is supposed to be the head of a list of nodes that are all allowed in vertical mode, and at least one of those has to represent a hbox. Failure to do so will result in a fatal error.

Setting this callback to false is possible, but dangerous, because it is possible you will end up in an unfixable 'deadcycles loop'.

### 4.1.4.4 post_linebreak_filter

This callback is called just after \LaTeX{} has converted a list of nodes into a stack of \texttt{\textbackslash\hbox{es}}.

\begin{verbatim}
function(<node> head, <string> groupcode)
    return true | false | <node> newhead
end
\end{verbatim}

This callback does not replace any internal code.

### 4.1.4.5 hpack_filter

This callback is called when \TeX{} is ready to start boxing some horizontal mode material. Math items and line boxes are ignored at the moment.

\begin{verbatim}
function(<node> head, <string> groupcode, <number> size,
\end{verbatim}
The `packtype` is either `additional` or `exactly`. If `additional`, then the `size` is a `\hbox spread ...` argument. If `exactly`, then the `size` is a `\hbox to ....` In both cases, the number is in scaled points.

The `direction` is either one of the three-letter direction specifier strings, or `nil` (added in 0.45).

This callback does not replace any internal code.

### 4.1.4.6 `vpack_filter`

This callback is called when \TeX{} is ready to start boxing some vertical mode material. Math displays are ignored at the moment.

This function is very similar to the `hpack_filter`. Besides the fact that it is called at different moments, there is an extra variable that matches \TeX{}’s `\maxdepth` setting.

```lua
function(<node> head, <string> groupcode, <number> size, <string> packtype, <number> maxdepth [, <string> direction])
    return true | false | <node> newhead
end
```

This callback does not replace any internal code.

### 4.1.4.7 `pre_output_filter`

This callback is called when \TeX{} is ready to start boxing the box 255 for `\output`.

```lua
function(<node> head, <string> groupcode, <number> size, <string> packtype, <number> maxdepth [, <string> direction])
    return true | false | <node> newhead
end
```

This callback does not replace any internal code.

### 4.1.4.8 `hyphenate`

```lua
function(<node> head, <node> tail)
    end
```

No return values. This callback has to insert discretionary nodes in the node list it receives. Setting this callback to `false` will prevent the internal discretionary insertion pass.
4.1.4.9 **ligaturing**

```latex
function(<node> head, <node> tail)
end
```

No return values. This callback has to apply ligaturing to the node list it receives.

You don’t have to worry about return values because the `head` node that is passed on to the callback is guaranteed not to be a `glyph_node` (if need be, a temporary node will be prepended), and therefore it cannot be affected by the mutations that take place. After the callback, the internal value of the ‘tail of the list’ will be recalculated.

The `next` of `head` is guaranteed to be non-nil.

The `next` of `tail` is guaranteed to be nil, and therefore the second callback argument can often be ignored. It is provided for orthogonality, and because it can sometimes be handy when special processing has to take place.

Setting this callback to `false` will prevent the internal ligature creation pass.

4.1.4.10 **kerning**

```latex
function(<node> head, <node> tail)
end
```

No return values. This callback has to apply kerning between the nodes in the node list it receives. See **ligaturing** for calling conventions.

Setting this callback to `false` will prevent the internal kern insertion pass.

4.1.4.11 **mlist_to_hlist**

This callback replaces LuaTeX’s math list to node list conversion algorithm.

```latex
function(<node> head, <string> display_type, <boolean> need_penalties)
    return <node> newhead
end
```

The returned node is the head of the list that will be added to the vertical or horizontal list, the string argument is either ‘text’ or ‘display’ depending on the current math mode, the boolean argument is `true` if penalties have to be inserted in this list, `false` otherwise.

Setting this callback to `false` is bad, it will almost certainly result in an endless loop.

4.1.5 **Information reporting callbacks**
4.1.5.1 pre_dump (0.61)

function()
end

This function is called just before dumping to a format file starts. It does not replace any code and there are neither arguments nor return values.

4.1.5.2 start_run

function()
end

This callback replaces the code that prints \LaTeX{}'s banner. Note that for successful use, this callback has to be set in the lua initialization script, otherwise it will be seen only after the run has already started.

4.1.5.3 stop_run

function()
end

This callback replaces the code that prints \LaTeX{}'s statistics and 'output written to' messages.

4.1.5.4 start_page_number

function()
end

Replaces the code that prints the \[ and the page number at the begin of \shipout. This callback will also override the printing of box information that normally takes place when \texttt{tracingoutput} is positive.

4.1.5.5 stop_page_number

function()
end

Replaces the code that prints the \] at the end of \shipout.

4.1.5.6 show_error_hook

function()
end
This callback is run from inside the TeX error function, and the idea is to allow you to do some extra reporting on top of what TeX already does (none of the normal actions are removed). You may find some of the values in the status table useful.

This callback does not replace any internal code.

### 4.1.5.7 show_error_message

```latex
function()
end
```

This callback replaces the code that prints the error message. The usual interaction after the message is not affected.

### 4.1.5.8 show_lua_error_hook

```latex
function()
end
```

This callback replaces the code that prints the extra lua error message.

### 4.1.5.9 start_file

```latex
function(category, filename)
end
```

This callback replaces the code that prints LuaTeX's when a file is opened like (filename) for regular files. The category is a number:

1. a normal data file, like a TeX source
2. a font map coupling font names to resources
3. an image file (png, pdf, etc)
4. an embedded font subset
5. a fully embedded font

### 4.1.5.10 stop_file

```latex
function(category)
end
```

This callback replaces the code that prints LuaTeX's when a file is closed like ) for regular files.

### 4.1.6 PDF-related callbacks
4.1.6.1 finish_pdffile

function()
end

This callback is called when all document pages are already written to the PDF file and \LaTeX{} is about to finalize the output document structure. Its intended use is final update of PDF dictionaries such as /Catalog or /Info. The callback does not replace any code. There are neither arguments nor return values.

4.1.6.2 finish_pdfpage

function(shippingout)
end

This callback is called after the pdf page stream has been assembled and before the page object gets finalized. This callback is available in \LaTeX{} 0.78.4 and later.

4.1.7 Font-related callbacks

4.1.7.1 define_font

function(<string> name, <number> size, <number> id)
   return <table> font
end

The string \texttt{name} is the filename part of the font specification, as given by the user. The number \texttt{size} is a bit special:

- if it is positive, it specifies an ‘at size’ in scaled points.
- if it is negative, its absolute value represents a ‘scaled’ setting relative to the designsize of the font.

The \texttt{id} is the internal number assigned to the font.

The internal structure of the \texttt{font} table that is to be returned is explained in chapter 7. That table is saved internally, so you can put extra fields in the table for your later Lua code to use.

Setting this callback to \texttt{false} is pointless as it will prevent font loading completely but will nevertheless generate errors.

4.2 The epdf library

The \texttt{epdf} library provides Lua bindings to many PDF access functions that are defined by the poppler pdf viewer library (written in C++ by Kristian Høgsberg, based on xpdf by Derek Noonburg). Within
LuATeX (and PDFTeX), xpdf functionality is being used since long time to embed PDF files. The \texttt{epdf} library shall allow to scrutinize an external PDF file. It gives access to its document structure, e.g., catalog, cross-reference table, individual pages, objects, annotations, info, and metadata. The LuATeX team is evaluating the possibility of reducing the binding to a basic low level PDF primitives and delegate the complete set of functions to an external shared object module.

The \texttt{epdf} library is still in alpha state: PDF access is currently read-only (it’s not yet possible to alter a PDF file or to assemble it from scratch), and many function bindings are still missing.

For a start, a PDF file is opened by \texttt{epdf.open()} with file name, e.g.:

\begin{verbatim}
doc = epdf.open("foo.pdf")
\end{verbatim}

This normally returns a \texttt{PDFDoc} userdata variable; but if the file could not be opened successfully, instead of a fatal error just the value \texttt{nil} is returned.

All Lua functions in the \texttt{epdf} library are named after the poppler functions listed in the poppler header files for the various classes, e.g., files \texttt{PDFDoc.h}, \texttt{Dict.h}, and \texttt{Array.h}. These files can be found in the poppler subdirectory within the LuATeX sources. Which functions are already implemented in the \texttt{epdf} library can be found in the LuATeX source file \texttt{lepdflib.cc}. For using the \texttt{epdf} library, knowledge of the PDF file architecture is indispensable.

There are many different userdata types defined by the \texttt{epdf} library, currently these are \texttt{AnnotBorderStyle}, \texttt{AnnotBorder}, \texttt{Annots}, \texttt{Annot}, \texttt{Array}, \texttt{Attribute}, \texttt{Catalog}, \texttt{Dict}, \texttt{EmbFile}, \texttt{GString}, \texttt{LinkDest}, \texttt{Links}, \texttt{Link}, \texttt{ObjectStream}, \texttt{Object}, \texttt{PDFDoc}, \texttt{PDFRectangle}, \texttt{Page}, \texttt{Ref}, \texttt{Stream}, \texttt{StructElement}, \texttt{StructTreeRoot}, \texttt{TextSpan}, \texttt{XRefEntry}, and \texttt{XRef}.

All these userdata names and the Lua access functions closely resemble the classes naming from the poppler header files, including the choice of mixed upper and lower case letters. The Lua function calls use object-oriented syntax, e.g., the following calls return the \texttt{Page} object for page 1:

\begin{verbatim}
pageref = doc:getCatalog():getPageRef(1)
pageobj = doc:getXRef():fetch(pageref.num, pageref.gen)
\end{verbatim}

But writing such chained calls is risky, as an intermediate function may return \texttt{nil} on error; therefore between function calls there should be Lua type checks (e.g., against \texttt{nil}) done. If a non-object item is requested (e.g., a \texttt{Dict} item by calling \texttt{page:getPieceInfo()}, cf. \texttt{Page.h}) but not available, the Lua functions return \texttt{nil} (without error). If a function should return an \texttt{Object}, but it’s not existing, a \texttt{Null} object is returned instead (also without error; this is in-line with poppler behavior).

All library objects have a \texttt{__gc} metamethod for garbage collection. The \texttt{__tostring} metamethod gives the type name for each object.

All object constructors:

\begin{verbatim}
<PDFDoc> = epdf.open(<string> PDF filename)
<Annot> = epdf.Annot(<XRef>, <Dict>, <Catalog>, <Ref>)
<Annots> = epdf.Annots(<XRef>, <Catalog>, <Object>)
<Array> = epdf.Array(<XRef>)
\end{verbatim}
The functions `StructElement_Type`, `Attribute_Type` and `AttributeOwner_Type` return a hash table `{<string>,<integer>}`.

**Annot** methods:

- `<boolean> = <Annot>:isOk()`
- `<Object> = <Annot>:getAppearance()`
- `<AnnotBorder> = <Annot>:getBorder()`
- `<boolean> = <Annot>:match(<Ref>)`

**AnnotBorderStyle** methods:

- `<number> = <AnnotBorderStyle>:getWidth()`

**Annots** methods:

- `<integer> = <Annots>:getNumAnnots()`
- `<Annot> = <Annots>:getAnnot(<integer>)`

**Array** methods:

- `<Array>:incRef()`
- `<Array>:decRef()`
- `<integer> = <Array>:getLength()`
- `<Array>:add(<Object>)`
- `<Object> = <Array>:get(<integer>)`
- `<Object> = <Array>:getNF(<integer>)`
- `<string> = <Array>:getString(<integer>)`

**Attribute** methods:

- `<boolean> = <Attribute>:isOk()`
- `<integer> = <Attribute>:getType()`
- `<integer> = <Attribute>:getOwner()`
- `<string> = <Attribute>:getTypeName()`
- `<string> = <Attribute>:getOwnerName()`
- `<Object> = <Attribute>:getValue()`
- `<Object> = <Attribute>:getDefaultValue`
- `<string> = <Attribute>:getName()`
- `<integer> = <Attribute>:getRevision()`
- `<boolean> = <Attribute>:isHidden()`
<Attribute>:setHidden(<boolean>)
<string> = <Attribute>:getFormattedValue()
<string> = <Attribute>:setFormattedValue(<string>)

Catalog methods:
<boolean> = <Catalog>:isOk()
<integer> = <Catalog>:getNumPages()
<Page> = <Catalog>:getPage(<integer>)
<Ref> = <Catalog>:getPageRef(<integer>)
<string> = <Catalog>:getBaseURI()
<string> = <Catalog>:readMetadata()
<Object> = <Catalog>:getStructTreeRoot()
<integer> = <Catalog>:findPage(<integer> object number, <integer> object generation)
<LinkDest> = <Catalog>:findDest(<string> name)
<Object> = <Catalog>:getDests()
<integer> = <Catalog>:numEmbeddedFiles()
<EmbFile> = <Catalog>:embeddedFile(<integer>)
<integer> = <Catalog>:numJS()
<string> = <Catalog>:getJS(<integer>)
<Object> = <Catalog>:getOutline()
<Object> = <Catalog>:getAcroForm()

EmbFile methods:
<string> = <EmbFile>:name()
<string> = <EmbFile>:description()
<integer> = <EmbFile>:size()
<string> = <EmbFile>:modDate()
<string> = <EmbFile>:createDate()
<string> = <EmbFile>:checksum()
<string> = <EmbFile>:mimeType()
<Object> = <EmbFile>:streamObject()
<boolean> = <EmbFile>:isOk()

Dict methods:
<Dict>:incRef()
<Dict>:decRef()
<integer> = <Dict>:getLength()
<Dict>:add(<string>, <Object>)
<Dict>:set(<string>, <Object>)
<Dict>:remove(<string>)
<boolean> = <Dict>:is(<string>)
<Object> = <Dict>:lookup(<string>)
<Object> = <Dict>:lookupNF(<string>)
<integer> = <Dict>:lookupInt(<string>, <string>)
<string> = <Dict>:getKey(<integer>)
<Object> = <Dict>:getVal(<integer>)
<Object> = <Dict>:getValNF(<integer>)
<boolean> = <Dict>:hasKey(<string>)

Link methods:
<boolean> = <Link>:isOk()
<boolean> = <Link>:inRect(<number>, <number>)

LinkDest methods:
<boolean> = <LinkDest>:isOk()
<integer> = <LinkDest>:getKind()
<string> = <LinkDest>:getKindName()
<boolean> = <LinkDest>:isPageRef()
<integer> = <LinkDest>:getPageNum()
<Ref> = <LinkDest>:getPageRef()
<number> = <LinkDest>:getLeft()
<number> = <LinkDest>:getBottom()
<number> = <LinkDest>:getRight()
<number> = <LinkDest>:getTop()
<number> = <LinkDest>:getZoom()
<boolean> = <LinkDest>:getChangeLeft()
<boolean> = <LinkDest>:getChangeTop()
<boolean> = <LinkDest>:getChangeZoom()

Links methods:
<integer> = <Links>:getNumLinks()
<Link> = <Links>:getLink(<integer>)

Object methods:
<Object>:initBool(<boolean>)
<Object>:initInt(<integer>)
<Object>:initReal(<number>)
<Object>:initString(<string>)
<Object>:initName(<string>)
<Object>:initNull()
<Object>:initArray(<XRef>)
<Object>:initDict(<XRef>)
<Object>:initStream(<Stream>)
<Object>:initRef(<integer> object number, <integer> object generation)
<Object>::initCmd(<string>)
<Object>::initError()
<Object>::initEOF()
<Object> = <Object>::fetch(<XRef>)
<Integer> = <Object>::getType()
<String> = <Object>::getTypeName()
<Boolean> = <Object>::isBool()
<Boolean> = <Object>::isInt()
<Boolean> = <Object>::isReal()
<Boolean> = <Object>::isNum()
<Boolean> = <Object>::isString()
<Boolean> = <Object>::isName()
<Boolean> = <Object>::isNull()
<Boolean> = <Object>::isArray()
<Boolean> = <Object>::isDict()
<Boolean> = <Object>::isStream()
<Boolean> = <Object>::isRef()
<Boolean> = <Object>::isCmd()
<Boolean> = <Object>::isError()
<Boolean> = <Object>::isEOF()
<Boolean> = <Object>::isNone()
<Boolean> = <Object>::getBool()
<Integer> = <Object>::getInt()
<number> = <Object>::getReal()
<number> = <Object>::getNum()
<string> = <Object>::getString()
<string> = <Object>::getName()
<Array> = <Object>::getArray()
<Dict> = <Object>::getDict()
<Stream> = <Object>::getStream()
<Ref> = <Object>::getRef()
<Integer> = <Object>::getRefNum()
<Integer> = <Object>::getRefGen()
<string> = <Object>::getCmd()
<Integer> = <Object>::arrayGetLength()
 = <Object>::arrayAdd(<Object>)
<Object> = <Object>::arrayGet(<integer>)
<Object> = <Object>::arrayGetNF(<integer>)
<Integer> = <Object>::dictGetLength(<integer>)
 = <Object>::dictAdd(<string>, <Object>)
 = <Object>::dictSet(<string>, <Object>)
<Object> = <Object>::dictLookup(<string>)
<Object> = <Object>::dictLookupNF(<string>)
<string> = <Object>::dictgetKey(<integer>)
<Object> = <Object>:dictgetVal(<integer>)
<Object> = <Object>:dictgetValNF(<integer>)
<boolean> = <Object>:streamIs(<string>)
 = <Object>:streamReset()
<integer> = <Object>:streamGetChar()
<integer> = <Object>:streamLookChar()
<integer> = <Object>:streamGetPos()
 = <Object>:streamSetPos(<integer>)
\<Dict\> = <Object>:streamGetDict()

Page methods:
<boolean> = <Page>:isOk()
<integer> = <Page>:getNum()
\<PDFRectangle\> = <Page>:getMediaBox()
\<PDFRectangle\> = <Page>:getCropBox()
<boolean> = <Page>:isCropped()
<number> = <Page>:getMediaWidth()
<number> = <Page>:getMediaHeight()
<number> = <Page>:getCropWidth()
<number> = <Page>:getCropHeight()
\<PDFRectangle\> = <Page>:getBleedBox()
\<PDFRectangle\> = <Page>:getTrimBox()
\<PDFRectangle\> = <Page>:getArtBox()
<integer> = <Page>:getRotate()
<string> = <Page>:getLastModified()
\<Dict\> = <Page>:getBoxColorInfo()
\<Dict\> = <Page>:getGroup()
\<Stream\> = <Page>:getMetadata()
\<Dict\> = <Page>:getPieceInfo()
\<Dict\> = <Page>:getSeparationInfo()
\<Dict\> = <Page>:getResourceDict()
<Object> = <Page>:getAnnots()
\<Links\> = <Page>:getLinks(<Catalog>)
<Object> = <Page>:getContents()

PDFDoc methods:
<boolean> = <PDFDoc>:isOk()
<integer> = <PDFDoc>:getErrorCode()
<string> = <PDFDoc>:getErrorCodeName()
<string> = <PDFDoc>:getFileName()
\<XRef\> = <PDFDoc>:getXRef()
\<Catalog\> = <PDFDoc>:getCatalog()
<number> = <PDFDoc>:getPageMediaWidth()
<number> = <PDFDoc>:getPageMediaHeight()
<number> = <PDFDoc>:getPageCropWidth()
<number> = <PDFDoc>:getPageCropHeight()
<Integer> = <PDFDoc>:getNumPages()
<String> = <PDFDoc>:readMetadata()
<Object> = <PDFDoc>:getStructTreeRoot()
<Integer> = <PDFDoc>:findPage(<integer> object number, <integer> object generation)
<Link> = <PDFDoc>:getLinks(<integer>)
<LinkDest> = <PDFDoc>:findDest(<string>)
<boolean> = <PDFDoc>:isEncrypted()
<boolean> = <PDFDoc>:okToPrint()
<boolean> = <PDFDoc>:okToChange()
<boolean> = <PDFDoc>:okToCopy()
<boolean> = <PDFDoc>:okToAddNotes()
<boolean> = <PDFDoc>:isLinearized()
<Object> = <PDFDoc>:getDocInfo()
<Object> = <PDFDoc>:getDocInfoNF()
<Integer> = <PDFDoc>:getPDFMajorVersion()
<Integer> = <PDFDoc>:getPDFMinorVersion()

PDFRectangle methods:
<boolean> = <PDFRectangle>:isValid()

Stream methods:
<Integer> = <Stream>:getKind()
<String> = <Stream>:getKindName()
 = <Stream>:reset()
 = <Stream>:close()
<Integer> = <Stream>:getChar()
<Integer> = <Stream>:lookChar()
<Integer> = <Stream>:getRawChar()
<Integer> = <Stream>:getUnfilteredChar()
 = <Stream>:unfilteredReset()
<Integer> = <Stream>:getPos()
<boolean> = <Stream>:isBinary()
<Object> = <Stream>:getUndecodedStream()
<Object> = <Stream>:getDict()

StructElement methods:
<String> = <StructElement>:getTypeName()
<Integer> = <StructElement>:getType()
<boolean> = <StructElement>:isOk()
<boolean> = <StructElement>:isBlock()
<boolean> = <StructElement>:isInline()
<boolean> = <StructElement>:isGrouping()
<boolean> = <StructElement>:isContent()
<boolean> = <StructElement>:isObjectRef()
<integer> = <StructElement>:getMCID()
<Ref> = <StructElement>:getObjectRef()
<Ref> = <StructElement>:getParentRef()
<boolean> = <StructElement>:hasPageRef()
<Ref> = <StructElement>:getPageRef()
<StructTreeRoot> = <StructElement>:getStructTreeRoot()
<string> = <StructElement>:getID()
<string> = <StructElement>:getLanguage()
<integer> = <StructElement>:getRevision()
    <StructElement>:setRevision(<unsigned integer>)
<string> = <StructElement>:getTitle()
<string> = <StructElement>:getExpandedAbbr()
<integer> = <StructElement>:getNumChildren()
<StructElement> = <StructElement>:getChild()
    = <StructElement>:appendChild<StructElement>)
<integer> = <StructElement>:getNumAttributes()
<Attribute> = <StructElement>:getAttribute(<integer>)
<string> = <StructElement>:appendAttribute(<Attribute>)
<Attribute> = <StructElement>:findAttribute(<Attribute::Type>,boolean,Attribute::Owner)
<string> = <StructElement>:getAltText()
<string> = <StructElement>:getActualText()
<string> = <StructElement>:getText(<boolean>)
<table> = <StructElement>:getTextSpans()

StructTreeRoot methods:

<StructElement> = <StructTreeRoot>:findParentElement
<PDFDoc> = <StructTreeRoot>:getDoc
<Dict> = <StructTreeRoot>:getRoleMap
<Dict> = <StructTreeRoot>:getClassMap
<StructElement> = <StructTreeRoot>:getNumChildren
<StructElement> = <StructTreeRoot>:getChild
    = <StructTreeRoot>:appendChild
<StructElement> = <StructTreeRoot>:findParentElement

TextSpan han only one method:

<string> = <TextSpan>:getText()

XRef methods:
4.3 The font library

The font library provides the interface into the internals of the font system, and also it contains helper functions to load traditional \TeX\ font metrics formats. Other font loading functionality is provided by the \texttt{fontloader} library that will be discussed in the next section.

4.3.1 Loading a TFM file

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

\begin{verbatim}
<\texttt{table}> fnt = font.read_tfm(<\texttt{string}> name, <\texttt{number}> s)
\end{verbatim}

The number is a bit special:

- if it is positive, it specifies an ‘at size’ in scaled points.
- if it is negative, its absolute value represents a ‘scaled’ setting relative to the designsize of the font.

The internal structure of the metrics font table that is returned is explained in \texttt{chapter 7}.
4.3.2 Loading a VF file

The behavior documented in this subsection is considered stable in the sense that there will not be backward-incompatible changes any more.

\[ \texttt{vf\_fnt = font.read\_vf(<string> name, <number> s)} \]

The meaning of the number \( s \) and the format of the returned table are similar to the ones in the \texttt{read\_tfm()} function.

4.3.3 The fonts array

The whole table of \TeX{} fonts is accessible from \texttt{LUA} using a virtual array.

\[ \texttt{font.fonts[n] = \{ ... \} } \]
\[ \texttt{f = font.fonts[n]} \]

See chapter 7 for the structure of the tables. Because this is a virtual array, you cannot call \texttt{pairs} on it, but see below for the \texttt{font.each} iterator.

The two metatable functions implementing the virtual array are:

\[ \texttt{f = font.getfont(<number> n)} \]
\[ \texttt{font.setfont(<number> n, <table> f)} \]

Note that at the moment, each access to the \texttt{font.fonts} or call to \texttt{font.getfont} creates a lua table for the whole font. This process can be quite slow. In a later version of \texttt{LUATEX}, this interface will change (it will start using userdata objects instead of actual tables).

Also note the following: assignments can only be made to fonts that have already been defined in \TeX{}, but have not been accessed at all since that definition. This limits the usability of the write access to \texttt{font.fonts} quite a lot, a less stringent ruleset will likely be implemented later.

4.3.4 Checking a font’s status

You can test for the status of a font by calling this function:

\[ \texttt{f = font.frozen(<number> n)} \]

The return value is one of \texttt{true} (unassignable), \texttt{false} (can be changed) or \texttt{nil} (not a valid font at all).

4.3.5 Defining a font directly

You can define your own font into \texttt{font.fonts} by calling this function:

\[ \texttt{i = font.define(<table> f)} \]
The return value is the internal id number of the defined font (the index into \texttt{font.fonts}). If the font creation fails, an error is raised. The table is a font structure, as explained in chapter 7.

### 4.3.6 Projected next font id

\begin{verbatim}
<number> i = font.nextid()
\end{verbatim}

This returns the font id number that would be returned by a \texttt{font.define} call if it was executed at this spot in the code flow. This is useful for virtual fonts that need to reference themselves.

### 4.3.7 Font id (0.47)

\begin{verbatim}
<number> i = font.id(<string> csname)
\end{verbatim}

This returns the font id associated with \texttt{csname} string, or $-1$ if \texttt{csname} is not defined; new in 0.47.

### 4.3.8 Currently active font

\begin{verbatim}
<number> i = font.current()
font.current(<number> i)
\end{verbatim}

This gets or sets the currently used font number.

### 4.3.9 Maximum font id

\begin{verbatim}
<number> i = font.max()
\end{verbatim}

This is the largest used index in \texttt{font.fonts}.

### 4.3.10 Iterating over all fonts

\begin{verbatim}
for i,v in font.each() do ...
end
\end{verbatim}

This is an iterator over each of the defined \TeX fonts. The first returned value is the index in \texttt{font.fonts}, the second the font itself, as a \texttt{Lua} table. The indices are listed incrementally, but they do not always form an array of consecutive numbers: in some cases there can be holes in the sequence.

### 4.4 The \texttt{fontloader} library (0.36)
4.4.1 Getting quick information on a font

```lua
info = fontloader.info(<string> filename)
```

This function returns either `nil`, or a `table`, or an array of small tables (in the case of a TrueType collection). The returned table(s) will contain some fairly interesting information items from the font(s) defined by the file:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fontname</td>
<td>string</td>
<td>the PostScript name of the font</td>
</tr>
<tr>
<td>fullname</td>
<td>string</td>
<td>the formal name of the font</td>
</tr>
<tr>
<td>familyname</td>
<td>string</td>
<td>the family name this font belongs to</td>
</tr>
<tr>
<td>weight</td>
<td>string</td>
<td>a string indicating the color value of the font</td>
</tr>
<tr>
<td>version</td>
<td>string</td>
<td>the internal font version</td>
</tr>
<tr>
<td>italicangle</td>
<td>float</td>
<td>the slant angle</td>
</tr>
<tr>
<td>units_per_em</td>
<td>number</td>
<td>(since 0.78.2) 1000 for PostScript-based fonts, usually 2048 for TrueType</td>
</tr>
<tr>
<td>pfminfo</td>
<td>table</td>
<td>(since 0.78.2) (see section 4.4.5.1.6)</td>
</tr>
</tbody>
</table>

Getting information through this function is (sometimes much) more efficient than loading the font properly, and is therefore handy when you want to create a dictionary of available fonts based on a directory contents.

4.4.2 Loading an OPENTYPE or TRUETYPE file

If you want to use an OpenType font, you have to get the metric information from somewhere. Using the `fontloader` library, the simplest way to get that information is thus:

```lua
function load_font (filename)
    local metrics = nil
    local font = fontloader.open(filename)
    if font then
        metrics = fontloader.to_table(font)
        fontloader.close(font)
    end
    return metrics
end
```

```lua
myfont = load_font('/opt/tex/texmf/fonts/data/arial.ttf')
```

The main function call is

```lua
userdata f, table w = fontloader.open(<string> filename)
```

```lua
userdata f, table w = fontloader.open(<string> filename, <string> font-name)
```
The first return value is a userdata representation of the font. The second return value is a table containing any warnings and errors reported by fontloader while opening the font. In normal typesetting, you would probably ignore the second argument, but it can be useful for debugging purposes.

For TrueType collections (when filename ends in ‘ttc’) and DFont collections, you have to use a second string argument to specify which font you want from the collection. Use the fontname strings that are returned by fontloader.info for that.

To turn the font into a table, fontloader.to_table is used on the font returned by fontloader.open.

\[ f = \text{fontloader.to.table}(<\text{userdata}> \text{font}) \]

This table cannot be used directly by \LaTeX{} and should be turned into another one as described in chapter 7. Do not forget to store the fontname value in the psname field of the metrics table to be returned to \LaTeX{}, otherwise the font inclusion backend will not be able to find the correct font in the collection.

See section 4.4.5 for details on the userdata object returned by fontloader.open() and the layout of the metrics table returned by fontloader.to_table().

The font file is parsed and partially interpreted by the font loading routines from FontForge. The file format can be OpenType, TrueType, TrueType Collection, CFF, or Type1.

There are a few advantages to this approach compared to reading the actual font file ourselves:

- The font is automatically re-encoded, so that the metrics table for TrueType and OpenType fonts is using Unicode for the character indices.
- Many features are pre-processed into a format that is easier to handle than just the bare tables would be.
- PostScript-based OpenType fonts do not store the character height and depth in the font file, so the character bounding box has to be calculated in some way.
- In the future, it may be interesting to allow Lua scripts access to the font program itself, perhaps even creating or changing the font.

A loaded font is discarded with:

\[ \text{fontloader.close}(<\text{userdata}> \text{font}) \]

### 4.4.3 Applying a ‘feature file’

You can apply a ‘feature file’ to a loaded font:

\[ \text{errors} = \text{fontloader.apply_featurefile}(<\text{userdata}> \text{font}, <\text{string}> \text{filename}) \]

A ‘feature file’ is a textual representation of the features in an OpenType font. See http://www.adobe.com/devnet/opentype/afdko/topic_feature_file_syntax.html and
http://fontforge.sourceforge.net/featurefile.html
for a more detailed description of feature files.

If the function fails, the return value is a table containing any errors reported by fontloader while applying the feature file. On success, nil is returned. (the return value is new in 0.65)

4.4.4 Applying an ‘AFM file’

You can apply an ‘AFM file’ to a loaded font:

<table> errors = fontloader.apply_afmfile(<userdata> font, <string> filename)</table>

An AFM file is a textual representation of (some of) the meta information in a Type1 font. See ftp://ftp.math.utah.edu/u/ma/hohn/linux/postscript/5004.AFM_Spec.pdf for more information about afm files.

Note: If you fontloader.open() a Type1 file named font.pfb, the library will automatically search for and apply font.afm if it exists in the same directory as the file font.pfb. In that case, there is no need for an explicit call to apply_afmfile().

If the function fails, the return value is a table containing any errors reported by fontloader while applying the AFM file. On success, nil is returned. (the return value is new in 0.65)

4.4.5 Fontloader font tables

As mentioned earlier, the return value of fontloader.open() is a userdata object. In LuaTEX versions before 0.63, the only way to have access to the actual metrics was to call fontloader.to_table() on this object, returning the table structure that is explained in the following subsections.

However, it turns out that the result from fontloader.to_table() sometimes needs very large amounts of memory (depending on the font’s complexity and size) so starting with LuaTEX 0.63, it is possible to access the userdata object directly.

In the LuaTEX 0.63.0, the following is implemented:

- all top-level keys that would be returned by to_table() can also be accessed directly.
- the top-level key ‘glyphs’ returns a virtual array that allows indices from 0 to (f.glyphmax - 1).
- the items in that virtual array (the actual glyphs) are themselves also userdata objects, and each has accessors for all of the keys explained in the section ‘Glyph items’ below.
- the top-level key ‘subfonts’ returns an actual array of userdata objects, one for each of the subfonts (or nil, if there are no subfonts).

A short example may be helpful. This code generates a printout of all the glyph names in the font PunkNova.kern.otf:

```lua
local f = fontloader.open('PunkNova.kern.otf')
print (f.fontname)
```
local i = 0
while (i < f.glyphmax) do
    local g = f.glyphs[i]
    if g then
        print(g.name)
    end
    i = i + 1
end
fontloader.close(f)

In this case, the \texttt{LuaTEX} memory requirement stays below 100MB on the test computer, while the internal structure generated by \texttt{to_table()} needs more than 2GB of memory (the font itself is 6.9MB in disk size).

In \texttt{LuaTEX} 0.63 only the top-level font, the subfont table entries, and the glyphs are virtual objects, everything else still produces normal lua values and tables. In future versions, more return values may be replaced by userdata objects (as much as needed to keep the memory requirements in check).

If you want to know the valid fields in a font or glyph structure, call the \texttt{fields} function on an object of a particular type (either glyph or font for now, more will be implemented later):

\begin{verbatim}
<table> fields = fontloader.fields(<userdata> font)
<table> fields = fontloader.fields(<userdata> font_glyph)
\end{verbatim}

For instance:

\begin{verbatim}
local fields = fontloader.fields(f)
local fields = fontloader.fields(f.glyphs[0])
\end{verbatim}

### 4.4.5.1 Table types

#### 4.4.5.1.1 Top-level

The top-level keys in the returned table are (the explanations in this part of the documentation are not yet finished):

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_version</td>
<td>number</td>
<td>indicates the metrics version (currently 0.3)</td>
</tr>
<tr>
<td>fontname</td>
<td>string</td>
<td>PostSCRIPT font name</td>
</tr>
<tr>
<td>fullname</td>
<td>string</td>
<td>official (human-oriented) font name</td>
</tr>
<tr>
<td>familyname</td>
<td>string</td>
<td>family name</td>
</tr>
<tr>
<td>weight</td>
<td>string</td>
<td>weight indicator</td>
</tr>
<tr>
<td>copyright</td>
<td>string</td>
<td>copyright information</td>
</tr>
<tr>
<td>filename</td>
<td>string</td>
<td>the file name</td>
</tr>
<tr>
<td>version</td>
<td>string</td>
<td>font version</td>
</tr>
<tr>
<td>italicangle</td>
<td>float</td>
<td>slant angle</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>units_per_em</td>
<td>number</td>
<td>1000 for PostScript-based fonts, usually 2048 for TrueType</td>
</tr>
<tr>
<td>ascent</td>
<td>number</td>
<td>height of ascender in units_per_em</td>
</tr>
<tr>
<td>descent</td>
<td>number</td>
<td>depth of descender in units_per_em</td>
</tr>
<tr>
<td>upos</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>uwidth</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>uniqueid</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>glyphcnt</td>
<td>number</td>
<td>number of included glyphs</td>
</tr>
<tr>
<td>glyphs</td>
<td>array</td>
<td></td>
</tr>
<tr>
<td>glyphmax</td>
<td>number</td>
<td>maximum used index the glyphs array</td>
</tr>
<tr>
<td>hasvmetrics</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>onlybitmaps</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>serifcheck</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>isserif</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>issans</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>encodingchanged</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>strokedfont</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>use_typo_metrics</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>weight_width_slope_only</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>head_optimized_for_cleartype</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>uni_interp</td>
<td>enum</td>
<td>unset, none, adobe, greek, japanese, trad_chinese, simp_chinese, korean, ams</td>
</tr>
<tr>
<td>origname</td>
<td>string</td>
<td>the file name, as supplied by the user</td>
</tr>
<tr>
<td>map</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>xuid</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>pfminfo</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>names</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>cidinfo</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>subfonts</td>
<td>array</td>
<td></td>
</tr>
<tr>
<td>comments</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>fontlog</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cvt_names</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>anchor_classes</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>ttf_tables</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>ttf_tab_saved</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>kerns</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>vkerns</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>texdata</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>lookups</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>gpos</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>gsub</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>mm</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>chosenname</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>
The \texttt{glyphs} is an array containing the per-character information (quite a few of these are only present if nonzero).

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>the glyph name</td>
</tr>
<tr>
<td>unicode</td>
<td>number</td>
<td>unicode code point, or -1</td>
</tr>
<tr>
<td>boundingbox</td>
<td>array</td>
<td>array of four numbers, see note below</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>only for horizontal fonts</td>
</tr>
<tr>
<td>vwidth</td>
<td>number</td>
<td>only for vertical fonts</td>
</tr>
<tr>
<td>tsidebearing</td>
<td>number</td>
<td>only for vertical ttf/otf fonts, and only if nonzero (0.79.0)</td>
</tr>
<tr>
<td>lsidebearing</td>
<td>number</td>
<td>only if nonzero and not equal to boundingbox[1]</td>
</tr>
<tr>
<td>class</td>
<td>string</td>
<td>one of &quot;none&quot;, &quot;base&quot;, &quot;ligature&quot;, &quot;mark&quot;, &quot;component&quot; (if not present, the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>glyph class is 'automatic')</td>
</tr>
<tr>
<td>kerns</td>
<td>array</td>
<td>only for horizontal fonts, if set</td>
</tr>
<tr>
<td>vkerns</td>
<td>array</td>
<td>only for vertical fonts, if set</td>
</tr>
<tr>
<td>dependents</td>
<td>array</td>
<td>linear array of glyph name strings, only if nonempty</td>
</tr>
<tr>
<td>lookups</td>
<td>table</td>
<td>only if nonempty</td>
</tr>
<tr>
<td>ligatures</td>
<td>table</td>
<td>only if nonempty</td>
</tr>
<tr>
<td>anchors</td>
<td>table</td>
<td>only if set</td>
</tr>
<tr>
<td>comment</td>
<td>string</td>
<td>only if set</td>
</tr>
<tr>
<td>tex_height</td>
<td>number</td>
<td>only if set</td>
</tr>
<tr>
<td>tex_depth</td>
<td>number</td>
<td>only if set</td>
</tr>
<tr>
<td>italic_correction</td>
<td>number</td>
<td>only if set</td>
</tr>
<tr>
<td>top_accent</td>
<td>number</td>
<td>only if set</td>
</tr>
<tr>
<td>is_extended_shape</td>
<td>number</td>
<td>only if this character is part of a math extension list</td>
</tr>
<tr>
<td>altuni</td>
<td>table</td>
<td>alternate \texttt{UNICODE} items</td>
</tr>
</tbody>
</table>
vert_variants    table
horiz_variants   table
mathkern         table

On boundingbox: The boundingbox information for TrueType fonts and TrueType-based otf fonts is read directly from the font file. PostScript-based fonts do not have this information, so the boundingbox of traditional PostScript fonts is generated by interpreting the actual bezier curves to find the exact boundingbox. This can be a slow process, so starting from LuATeX 0.45, the boundingboxes of PostScript-based otf fonts (and raw cff fonts) are calculated using an approximation of the glyph shape based on the actual glyph points only, instead of taking the whole curve into account. This means that glyphs that have missing points at extrema will have a too-tight boundingbox, but the processing is so much faster that in our opinion the tradeoff is worth it.

The kerns and vkerns are linear arrays of small hashes:

key     type     explanation
char    string   
off     number   
lookup  string   

The lookups is a hash, based on lookup subtable names, with the value of each key inside that a linear array of small hashes:

key     type     explanation
position, pair, substitution, alternate, multiple, ligature, lcaret, kerning, vkerning, anchors, contextpos, contextsub, chainpos, chainsub, reversesub, max, kernback, vkernback

For the first seven values of type, there can be additional sub-information, stored in the sub-table specification:

value     type     explanation
position   table    a table of the offset_specs type
pair       table    one string: paired, and an array of one or two offset_specs tables: offsets
substitution   table    one string: variant
alternate    table    one string: components
multiple     table    one string: components
ligature     table    two strings: components, char
lcaret       array    linear array of numbers

Tables for offset_specs contain up to four number-valued fields: x (a horizontal offset), y (a vertical offset), h (an advance width correction) and v (an advance height correction).

The ligatures is a linear array of small hashes:
The anchor table is indexed by a string signifying the anchor type, which is one of

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>lig</td>
<td>table</td>
<td>uses the same substructure as a single item in the lookups table explained above</td>
</tr>
<tr>
<td>char</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>components</td>
<td>array</td>
<td>linear array of named components</td>
</tr>
<tr>
<td>ccnt</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

For all except baselig, the value is a single table with this definition:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>number</td>
<td>x location</td>
</tr>
<tr>
<td>y</td>
<td>number</td>
<td>y location</td>
</tr>
<tr>
<td>ttf_pt_index</td>
<td>number</td>
<td>truetype point index, only if given</td>
</tr>
</tbody>
</table>

For clarification, an anchor table could for example look like this:

```lua
['anchor'] = {
    ['basemark'] = {
        ['Anchor-7'] = { ['x']=170, ['y']=1080 }
    },
    ['mark'] ={
        ['Anchor-1'] = { ['x']=160, ['y']=810 },
        ['Anchor-4'] = { ['x']=160, ['y']=800 }
    },
    ['baselig'] = {
        [1] = { ['Anchor-2'] = { ['x']=160, ['y']=650 } },
        [2] = { ['Anchor-2'] = { ['x']=460, ['y']=640 } }
    }
}
```

Note: The baselig table can be sparse!
### 4.4.5.1.3 map table

The top-level map is a list of encoding mappings. Each of those is a table itself.

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>enccount</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>encmax</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>backmax</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>remap</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>map</td>
<td>array</td>
<td>non-linear array of mappings</td>
</tr>
<tr>
<td>backmap</td>
<td>array</td>
<td>non-linear array of backward mappings</td>
</tr>
<tr>
<td>enc</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The `remap` table is very small:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstenc</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lastenc</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>infont</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

The `enc` table is a bit more verbose:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>enc_name</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>char_cnt</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>char_max</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>unicode</td>
<td>array</td>
<td>of UNICODE position numbers</td>
</tr>
<tr>
<td>psnames</td>
<td>array</td>
<td>of PostSCRIPT glyph names</td>
</tr>
<tr>
<td>builtin</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hidden</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>only_1byte</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>has_1byte</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>has_2byte</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>is_unicharbmp</td>
<td>number</td>
<td>only if nonzero</td>
</tr>
<tr>
<td>is_unicodefull</td>
<td>number</td>
<td>only if nonzero</td>
</tr>
<tr>
<td>is_custom</td>
<td>number</td>
<td>only if nonzero</td>
</tr>
<tr>
<td>is_original</td>
<td>number</td>
<td>only if nonzero</td>
</tr>
<tr>
<td>is_compact</td>
<td>number</td>
<td>only if nonzero</td>
</tr>
<tr>
<td>is_japanese</td>
<td>number</td>
<td>only if nonzero</td>
</tr>
<tr>
<td>is_korean</td>
<td>number</td>
<td>only if nonzero</td>
</tr>
<tr>
<td>is_tradchinese</td>
<td>number</td>
<td>only if nonzero [name?]</td>
</tr>
<tr>
<td>is_simplechinese</td>
<td>number</td>
<td>only if nonzero</td>
</tr>
<tr>
<td>low_page</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>high_page</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>iconv_name</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>iso_2022_escape</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>
### 4.4.5.1.4 private table

This is the font’s private PostScript dictionary, if any. Keys and values are both strings.

### 4.4.5.1.5 cidinfo table

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>registry</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>ordering</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>supplement</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>version</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.5.1.6 pfminfo table

The pfminfo table contains most of the OS/2 information:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfmset</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>winascent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>windescent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hheadascent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hheaddescent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>typoascent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>typodescent_add</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>subsuper_set</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>panose_set</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hheadset</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>vheadset</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>pfmfamily</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>avgwidth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>firstchar</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lastchar</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>fstype</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>linegap</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>vlinegap</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hhead_ascent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hhead_descent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>os2_typoascent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>os2_typodescent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>os2_typolinegap</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>os2_winascent</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>os2_windescent</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>
os2_subxsize number
os2_subysize number
os2_subxoff number
os2_subyoff number
os2_supxsize number
os2_supysize number
os2_supxoff number
os2_supyoff number
os2_strikeysizesize number
os2_strikeypos number
os2_family_class number
os2_xheight number
os2_capheight number
os2_defaultchar number
os2_breakchar number
os2_vendor string
codepages table A two-number array of encoded code pages
unicoderanges table A four-number array of encoded unicode ranges
panose table

The `panose` subtable has exactly 10 string keys:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>familytype</td>
<td>string</td>
<td>Values as in the OpenType font specification: Any, No Fit, Text and Display, Script, Decorative, Pictorial</td>
</tr>
<tr>
<td>serifstyle</td>
<td>string</td>
<td>See the OpenType font specification for values</td>
</tr>
<tr>
<td>weight</td>
<td>string</td>
<td>id.</td>
</tr>
<tr>
<td>proportion</td>
<td>string</td>
<td>id.</td>
</tr>
<tr>
<td>contrast</td>
<td>string</td>
<td>id.</td>
</tr>
<tr>
<td>strokevariation</td>
<td>string</td>
<td>id.</td>
</tr>
<tr>
<td>armstyle</td>
<td>string</td>
<td>id.</td>
</tr>
<tr>
<td>letterform</td>
<td>string</td>
<td>id.</td>
</tr>
<tr>
<td>midline</td>
<td>string</td>
<td>id.</td>
</tr>
<tr>
<td>xheight</td>
<td>string</td>
<td>id.</td>
</tr>
</tbody>
</table>

### 4.4.5.1.7 names table

Each item has two top-level keys:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>lang</td>
<td>string</td>
<td>language for this entry</td>
</tr>
<tr>
<td>names</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The `names` keys are the actual TrueType name strings. The possible keys are:
key explanation
copyright
family
subfamily
uniqueid
fullname
version
postscriptname
trademark
manufacturer
designer
descriptor
venderurl
designerurl
license
licenseurl
idontknow
preffamilyname
prefmodifiers
compatfull
sampletext
cidfondfontname
wwsfamily
wwssubfamily

4.4.5.1.8 anchor_classes table

The anchor_classes classes:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>a descriptive id of this anchor class</td>
</tr>
<tr>
<td>lookup</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>string</td>
<td>one of mark, mkmk, curs, mklg</td>
</tr>
</tbody>
</table>

4.4.5.1.9 gpos table

The gpos table has one array entry for each lookup. (The gpos_ prefix is somewhat redundant.)

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>string</td>
<td>one of gpos_single, gpos_pair, gpos_cursive, gpos_mark2base, gpos_mark2ligature, gpos_mark2mark, gpos_context, gpos_contextchain</td>
</tr>
<tr>
<td>flags</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>
features array
subtables array

The flags table has a true value for each of the lookup flags that is actually set:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2l</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>ignorebaseglyphs</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>ignoreligatures</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>ignorecombiningmarks</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>mark_class</td>
<td>string</td>
<td>(new in 0.44)</td>
</tr>
</tbody>
</table>

The features subtable items of gpos have:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>scripts</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The scripts table within features has:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>script</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>langs</td>
<td>array of strings</td>
<td></td>
</tr>
</tbody>
</table>

The subtables table has:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>suffix</td>
<td>string</td>
<td>(only if used)</td>
</tr>
<tr>
<td>anchor_classes</td>
<td>number</td>
<td>(only if used)</td>
</tr>
<tr>
<td>vertical_kerning</td>
<td>number</td>
<td>(only if used)</td>
</tr>
<tr>
<td>kernclass</td>
<td>table</td>
<td>(only if used)</td>
</tr>
</tbody>
</table>

The kernclass with subtables table has:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>firsts</td>
<td>array of strings</td>
<td></td>
</tr>
<tr>
<td>seconds</td>
<td>array of strings</td>
<td></td>
</tr>
<tr>
<td>lookup</td>
<td>string or array of strings</td>
<td>associated lookup(s)</td>
</tr>
<tr>
<td>offsets</td>
<td>array of numbers</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.5.1.10 gsub table

This has identical layout to the gpos table, except for the type:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>string</td>
<td>one of gsub_single, gsub_multiple, gsub_alternate, gsub_ligature, gsub_context, gsub_contextchain, gsub_reversecontextchain</td>
</tr>
</tbody>
</table>
### 4.4.5.1.11 ttf_tables and ttf_tab_saved tables

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>len</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>maxlen</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.5.1.12 mm table

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>axes</td>
<td>table</td>
<td>array of axis names</td>
</tr>
<tr>
<td>instance_count</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>positions</td>
<td>table</td>
<td>array of instance positions (#axes * instances )</td>
</tr>
<tr>
<td>defweights</td>
<td>table</td>
<td>array of default weights for instances</td>
</tr>
<tr>
<td>cdv</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>ndv</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>axismaps</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The **axismaps**:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>blends</td>
<td>table</td>
<td>an array of blend points</td>
</tr>
<tr>
<td>designs</td>
<td>table</td>
<td>an array of design values</td>
</tr>
<tr>
<td>min</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>def</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.5.1.13 mark_classes table (0.44)

The keys in this table are mark class names, and the values are a space-separated string of glyph names in this class.

Note: This table is indeed new in 0.44. The manual said it existed before then, but in practise it was missing due to a bug.

### 4.4.5.1.14 math table

ScriptPercentScaleDown
ScriptScriptPercentScaleDown
DelimitedSubFormulaMinHeight
DisplayOperatorMinHeight
MathLeading
AxisHeight
AccentBaseHeight
FlattenedAccentBaseHeight
SubscriptShiftDown
SubscriptTopMax
SubscriptBaselineDropMin
SuperscriptShiftUp
SuperscriptShiftUpCramped
SuperscriptBottomMin
SuperscriptBaselineDropMax
SubSuperscriptGapMin
SuperscriptBottomMaxWithSubscript
SpaceAfterScript
UpperLimitGapMin
UpperLimitBaselineRiseMin
LowerLimitGapMin
LowerLimitBaselineDropMin
StackTopShiftUp
StackTopDisplayStyleShiftUp
StackBottomShiftDown
StackBottomDisplayStyleShiftDown
StackGapMin
StackDisplayStyleGapMin
StretchStackTopShiftUp
StretchStackBottomShiftDown
StretchStackGapAboveMin
StretchStackGapBelowMin
FractionNumeratorShiftUp
FractionNumeratorDisplayStyleShiftUp
FractionDenominatorShiftDown
FractionDenominatorDisplayStyleShiftDown
FractionNumeratorGapMin
FractionNumeratorDisplayStyleGapMin
FractionRuleThickness
FractionDenominatorGapMin
FractionDenominatorDisplayStyleGapMin
SkewedFractionHorizontalGap
SkewedFractionVerticalGap
OverbarVerticalGap
OverbarRuleThickness
OverbarExtraAscender
UnderbarVerticalGap
UnderbarRuleThickness
UnderbarExtraDescender
RadicalVerticalGap
RadicalDisplayStyleVerticalGap
RadicalRuleThickness
RadicalExtraAscender
RadicalKernBeforeDegree
RadicalKernAfterDegree
RadicalDegreeBottomRaisePercent
MinConnectorOverlap
FractionDelimiterSize \text{(new in 0.47.0)}
FractionDelimiterDisplayStyleSize \text{(new in 0.47.0)}

4.4.5.1.15 \textit{validation\_state table}

\begin{verbatim}
key explanation
bad_ps_fontname
bad_glyph_table
bad_cff_table
bad_metrics_table
bad_cmap_table
bad_bitmaps_table
bad_gx_table
bad_ot_table
bad_os2_version
bad_sfnt_header
\end{verbatim}

4.4.5.1.16 \textit{horiz\_base and vert\_base table}

\begin{verbatim}
key type explanation
tags table an array of script list tags
scripts table

The \textit{scripts} subtable:

key type explanation
baseline table
default_baseline number
lang table

The \textit{lang} subtable:

key type explanation
tag string a script tag
ascent number
descent number
features table
\end{verbatim}
The **features** points to an array of tables with the same layout except that in those nested tables, the tag represents a language.

### 4.4.5.1.17 altuni table

An array of alternate Unicode values. Inside that array are hashes with:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unicode</td>
<td>number</td>
<td>this glyph is also used for this unicode</td>
</tr>
<tr>
<td>variant</td>
<td>number</td>
<td>the alternative is driven by this unicode selector</td>
</tr>
</tbody>
</table>

### 4.4.5.1.18 vert_variants and horiz_variants table

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>variants</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>italic_corrections</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>parts</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

The **parts** table is an array of smaller tables:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>extender</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>start</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>advance</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.5.1.19 mathkern table

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_right</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>bottom_right</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>top_left</td>
<td>table</td>
<td></td>
</tr>
<tr>
<td>bottom_left</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

Each of the subtables is an array of small hashes with two keys:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>kern</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.5.1.20 kerns table

Substructure is identical to the per-glyph subtable.
4.4.5.1.21 vkerns table

Substructure is identical to the per-glyph subtable.

4.4.5.1.22 texdata table

key  type  explanation
---  ----  ----------------
type  string  unset, text, math, mathext
params array  22 font numeric parameters

4.4.5.1.23 lookups table

Top-level lookups is quite different from the ones at character level. The keys in this hash are strings, the values the actual lookups, represented as dictionary tables.

key  type  explanation
---  ----  ----------------
type  string  
format  enum  one of glyphs, class, coverage, reversecoverage
tag  string  
current_class  array  
before_class  array  
after_class  array  
rules  array  an array of rule items

Rule items have one common item and one specialized item:

key  type  explanation
---  ----  ----------------
lookups  array  a linear array of lookup names
glyphs  array  only if the parent’s format is glyphs
class  array  only if the parent’s format is class
coverage  array  only if the parent’s format is coverage
reversecoverage  array  only if the parent’s format is reversecoverage

A glyph table is:

key  type  explanation
---  ----  ----------------
names  string  
back  string  
fore  string  

A class table is:

key  type  explanation
---  ----  ----------------
current  array  of numbers
before  array  of numbers
after  array  of numbers
coverage:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>before</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>after</td>
<td>array</td>
<td>of strings</td>
</tr>
</tbody>
</table>

reversecoverage:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>before</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>after</td>
<td>array</td>
<td>of strings</td>
</tr>
<tr>
<td>replacements</td>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

4.5 The img library

The img library can be used as an alternative to \pdfximage and \pdfrefximage, and the associated 'satellite' commands like \pdfximagebbox. Image objects can also be used within virtual fonts via the image command listed in section 7.2.

4.5.1 img.new

\texttt{\textless image\textgreater{} var = img.new()}

\texttt{\textless image\textgreater{} var = img.new(<table> image_spec)}

This function creates a userdata object of type 'image'. The image_spec argument is optional. If it is given, it must be a table, and that table must contain a filename key. A number of other keys can also be useful, these are explained below.

You can either say

\texttt{a = img.new()}

followed by

\texttt{a.filename = "foo.png"}

or you can put the file name (and some or all of the other keys) into a table directly, like so:

\texttt{a = img.new({filename='foo.pdf', page=1})}

The generated \texttt{\textless image\textgreater{}} userdata object allows access to a set of user-specified values as well as a set of values that are normally filled in and updated automatically by \LaTeX{} itself. Some of those are derived from the actual image file, others are updated to reflect the PDF output status of the object.
There is one required user-specified field: the file name (filename). It can optionally be augmented by the requested image dimensions (width, depth, height), user-specified image attributes (attr), the requested PDF page identifier (page), the requested bounding box (pagebox) for PDF inclusion, the requested color space object (colorspace).

The function **img.new** does not access the actual image file, it just creates the <image> userdata object and initializes some memory structures. The <image> object and its internal structures are automatically garbage collected.

Once the image is scanned, all the values in the <image> except width, height and depth, become frozen, and you cannot change them any more.

### 4.5.2 img.keys

<table> keys = img.keys()

This function returns a list of all the possible image_spec keys, both user-supplied and automatic ones.

<table>
<thead>
<tr>
<th>field name</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>string</td>
<td>the image attributes for \LaTeX{}</td>
</tr>
<tr>
<td>bbox</td>
<td>table</td>
<td>table with 4 boundingbox dimensions llx, lly, urx, and ury overruling the pagebox entry</td>
</tr>
<tr>
<td>colordepth</td>
<td>number</td>
<td>the number of bits used by the color space</td>
</tr>
<tr>
<td>colorspace</td>
<td>number</td>
<td>the color space object number</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>the image depth for \LaTeX{} (in scaled points)</td>
</tr>
<tr>
<td>filename</td>
<td>string</td>
<td>the image file name</td>
</tr>
<tr>
<td>filepath</td>
<td>string</td>
<td>the full (expanded) file name of the image</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>the image height for \LaTeX{} (in scaled points)</td>
</tr>
<tr>
<td>imagetype</td>
<td>string</td>
<td>one of pdf, png, jpg, jp2, jbig2, or nil</td>
</tr>
<tr>
<td>index</td>
<td>number</td>
<td>the PDF image name suffix</td>
</tr>
<tr>
<td>objnum</td>
<td>number</td>
<td>the PDF image object number</td>
</tr>
<tr>
<td>page</td>
<td>??</td>
<td>the identifier for the requested image page (type is number or string, default is the number 1)</td>
</tr>
<tr>
<td>pagebox</td>
<td>string</td>
<td>the requested bounding box, one of none, media, crop, bleed, trim, art</td>
</tr>
<tr>
<td>pages</td>
<td>number</td>
<td>the total number of available pages</td>
</tr>
<tr>
<td>rotation</td>
<td>number</td>
<td>the image rotation from included PDF file, in multiples of 90 deg.</td>
</tr>
<tr>
<td>stream</td>
<td>string</td>
<td>the raw stream data for an /XObject /Form object</td>
</tr>
<tr>
<td>transform</td>
<td>number</td>
<td>the image transform, integer number 0..7</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the image width for \LaTeX{} (in scaled points)</td>
</tr>
<tr>
<td>xres</td>
<td>number</td>
<td>the horizontal natural image resolution (in DPI)</td>
</tr>
<tr>
<td>xsize</td>
<td>number</td>
<td>the natural image width</td>
</tr>
<tr>
<td>yres</td>
<td>number</td>
<td>the vertical natural image resolution (in DPI)</td>
</tr>
<tr>
<td>ysize</td>
<td>number</td>
<td>the natural image height</td>
</tr>
</tbody>
</table>

A running (undefined) dimension in width, height, or depth is represented as nil in Lua, so if you want to load an image at its 'natural' size, you do not have to specify any of those three fields.
The **stream** parameter allows to fabricate an /XObject /Form object from a string giving the stream contents, e.g., for a filled rectangle:

```lua
a.stream = "0 0 20 10 re f"
```

When writing the image, an /XObject /Form object is created, like with embedded PDF file writing. The object is written out only once. The **stream** key requires that also the **bbox** table is given. The **stream** key conflicts with the **filename** key. The **transform** key works as usual also with **stream**.

The **bbox** key needs a table with four boundingbox values, e.g.:

```lua
a.bbox = {"30bp", 0, "225bp", "200bp"}
```

This replaces and overrules any given **pagebox** value; with given **bbox** the box dimensions coming with an embedded PDF file are ignored. The **xsize** and **ysize** dimensions are set accordingly, when the image is scaled. The **bbox** parameter is ignored for non-PDF images.

The **transform** allows to mirror and rotate the image in steps of 90 deg. The default value 0 gives an unmirrored, unrotated image. Values 1–3 give counterclockwise rotation by 90, 180, or 270 degrees, whereas with values 4–7 the image is first mirrored and then rotated counterclockwise by 90, 180, or 270 degrees. The **transform** operation gives the same visual result as if you would externally preprocess the image by a graphics tool and then use it by LuaTeX. If a PDF file to be embedded already contains a /Rotate specification, the rotation result is the combination of the /Rotate rotation followed by the **transform** operation.

### 4.5.3 **img.scan**

```lua
<image> var = img.scan(<image> var)
<image> var = img.scan(<table> image_spec)
```

When you say **img.scan(a)** for a new image, the file is scanned, and variables such as **xsize**, **ysize**, image **type**, number of **pages**, and the resolution are extracted. Each of the **width**, **height**, **depth** fields are set up according to the image dimensions, if they were not given an explicit value already. An image file will never be scanned more than once for a given image variable. With all subsequent **img.scan(a)** calls only the dimensions are again set up (if they have been changed by the user in the meantime).

For ease of use, you can do right-away a

```lua
<image> a = img.scan ({ filename = "foo.png" })
```

without a prior **img.new**.

Nothing is written yet at this point, so you can do **a=img.scan**, retrieve the available info like image width and height, and then throw away **a** again by saying **a=nil**. In that case no image object will be reserved in the PDF, and the used memory will be cleaned up automatically.
4.5.4 **img.copy**

```latex
<image> var = img.copy(<image> var)
<image> var = img.copy(<table> image_spec)
```

If you say \`a = b\', then both variables point to the same `image` object. If you want to write out an image with different sizes, you can do a `b=img.copy(a)`.

Afterwards, `a` and `b` still reference the same actual image dictionary, but the dimensions for `b` can now be changed from their initial values that were just copies from `a`.

4.5.5 **img.write**

```latex
<image> var = img.write(<image> var)
<image> var = img.write(<table> image_spec)
```

By `img.write(a)` a PDF object number is allocated, and a whatsit node of subtype `pdf_refximage` is generated and put into the output list. By this the image `a` is placed into the page stream, and the image file is written out into an image stream object after the shipping of the current page is finished.

Again you can do a terse call like

```latex
img.write ({ filename = "foo.png" })
```

The `<image>` variable is returned in case you want it for later processing.

4.5.6 **img.immediatewrite**

```latex
<image> var = img.immediatewrite(<image> var)
<image> var = img.immediatewrite(<table> image_spec)
```

By `img.immediatewrite(a)` a PDF object number is allocated, and the image file for image `a` is written out immediately into the PDF file as an image stream object (like with `\immediate\pdfximage`). The object number of the image stream dictionary is then available by the `objnum` key. No `pdf_refximage` whatsit node is generated. You will need an `img.write(a)` or `img.node(a)` call to let the image appear on the page, or reference it by another trick; else you will have a dangling image object in the PDF file.

Also here you can do a terse call like

```latex
a = img.immediatewrite ({ filename = "foo.png" })
```

The `<image>` variable is returned and you will most likely need it.

4.5.7 **img.node**

```latex
<node> n = img.node(<image> var)
```
This function allocates a PDF object number and returns a whatsis node of subtype pdf_refximage, filled with the image parameters width, height, depth, and objnum. Also here you can do a terse call like:

```
n = img.node({ filename = "foo.png" })
```

This example outputs an image:

```latex
node.write(img.node{filename="foo.png"})
```

### 4.5.8 `img.types`

This function returns a list with the supported image file type names, currently these are pdf, png, jpg, jp2 (JPEG 2000), and jbig2.

### 4.5.9 `img.boxes`

This function returns a list with the supported PDF page box names, currently these are media, crop, bleed, trim, and art (all in lowercase letters).

### 4.6 The kpse library

This library provides two separate, but nearly identical interfaces to the KPATHSEA file search functionality: there is a 'normal' procedural interface that shares its kpathsea instance with LuaTeX itself, and an object oriented interface that is completely on its own. The object oriented interface and `kpse.new` have been added in LuaTeX 0.37.

#### 4.6.1 `kpse.set_program_name` and `kpse.new`

Before the search library can be used at all, its database has to be initialized. There are three possibilities, two of which belong to the procedural interface.

First, when LuaTeX is used to typeset documents, this initialization happens automatically and the KPATHSEA executable and program names are set to luatex (that is, unless explicitly prohibited by the user’s startup script. See section 3.1 for more details).

Second, in TExLAU mode, the initialization has to be done explicitly via the `kpse.set_program_name` function, which sets the KPATHSEA executable (and optionally program) name.
kpse.set_program_name(<string> name)
kpse.set_program_name(<string> name, <string> progname)

The second argument controls the use of the 'dotted' values in the texmf.cnf configuration file, and defaults to the first argument.

Third, if you prefer the object oriented interface, you have to call a different function. It has the same arguments, but it returns a userdata variable.

local kpathsea = kpse.new(<string> name)
local kpathsea = kpse.new(<string> name, <string> progname)

Apart from these two functions, the calling conventions of the interfaces are identical. Depending on the chosen interface, you either call kpse.find_file() or kpathsea:find_file(), with identical arguments and return values.

### 4.6.2 find_file

The most often used function in the library is find_file:

<string> f = kpse.find_file(<string> filename)
<string> f = kpse.find_file(<string> filename, <string> ftype)
<string> f = kpse.find_file(<string> filename, <boolean> mustexist)
<string> f = kpse.find_file(<string> filename, <string> ftype, <boolean> mustexist)
<string> f = kpse.find_file(<string> filename, <string> ftype, <number> dpi)

Arguments:

filename
  the name of the file you want to find, with or without extension.
ftype
  maps to the \texttt{-format} argument of kpsewhich. The supported \texttt{ftype} values are the same as the ones supported by the standalone kpsewhich program:
The default type is \texttt{tex}. Note: this is different from \texttt{kpsewhich}, which tries to deduce the file type itself from looking at the supplied extension. The last four types: ‘font feature files’, ‘cid maps’, ‘mlbib’, ‘mlbst’ were new additions in \LaTeX{} 0.40.2.

\texttt{mustexist} is similar to \texttt{kpsewhich}'s \texttt{-must-exist}, and the default is \texttt{false}. If you specify \texttt{true} (or a non-zero integer), then the \texttt{kpse} library will search the disk as well as the \texttt{ls-R} databases.

\texttt{dpi}

This is used for the size argument of the formats \texttt{pk}, \texttt{gf}, and \texttt{bitmap font}.

### 4.6.3 \texttt{lookup}

A more powerful (but slower) generic method for finding files is also available (since 0.51). It returns a string for each found file.

\begin{verbatim}
<string> f, ... = kpse.lookup(<string> filename, <table> options)
\end{verbatim}
The options match commandline arguments from `kpsewhich`:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>number</td>
<td>set debugging flags for this lookup</td>
</tr>
<tr>
<td>format</td>
<td>string</td>
<td>use specific file type (see list above)</td>
</tr>
<tr>
<td>dpi</td>
<td>number</td>
<td>use this resolution for this lookup; default 600</td>
</tr>
<tr>
<td>path</td>
<td>string</td>
<td>search in the given path</td>
</tr>
<tr>
<td>all</td>
<td>boolean</td>
<td>output all matches, not just the first</td>
</tr>
<tr>
<td>mustexist</td>
<td>boolean</td>
<td>(0.65 and higher) search the disk as well as ls-R if necessary</td>
</tr>
<tr>
<td>must-exist</td>
<td>boolean</td>
<td>(0.64 and lower) search the disk as well as ls-R if necessary</td>
</tr>
<tr>
<td>mk texpk</td>
<td>boolean</td>
<td>disable/enable mktexpk generation for this lookup</td>
</tr>
<tr>
<td>mk textex</td>
<td>boolean</td>
<td>disable/enable mktextex generation for this lookup</td>
</tr>
<tr>
<td>mk textmf</td>
<td>boolean</td>
<td>disable/enable mktexmf generation for this lookup</td>
</tr>
<tr>
<td>mk textfm</td>
<td>boolean</td>
<td>disable/enable mktextfm generation for this lookup</td>
</tr>
<tr>
<td>subdir</td>
<td>string or table</td>
<td>only output matches whose directory part ends with the given string(s)</td>
</tr>
</tbody>
</table>

### 4.6.4 init prog

Extra initialization for programs that need to generate bitmap fonts.

```lua
kpse.init_prog(<string> prefix, <number> base_dpi, <string> mfmode)
kps.init_prog(<string> prefix, <number> base_dpi, <string> mfmode, <string> fallback)
```

### 4.6.5 readable_file

Test if an (absolute) file name is a readable file.

```lua
<string> f = kpse.readable_file(<string> name)
```

The return value is the actual absolute filename you should use, because the disk name is not always the same as the requested name, due to aliases and system-specific handling under e.g. MSDOS.

Returns `nil` if the file does not exist or is not readable.

### 4.6.6 expand_path

Like kpsewhich's `-expand-path`:

```lua
<string> r = kpse.expand_path(<string> s)
```

### 4.6.7 expand_var

Like kpsewhich's `-expand-var`:
<string> r = kpse.expand_var(<string> s)

4.6.8 expand_braces
Like kpsewhich’s -expand-braces:
<string> r = kpse.expand_braces(<string> s)

4.6.9 show_path
Like kpsewhich’s -show-path:
<string> r = kpse.show_path(<string> ftype)

4.6.10 var_value
Like kpsewhich’s -var-value:
<string> r = kpse.var_value(<string> s)

4.6.11 version
Returns the kpathsea version string (new in 0.51)
<string> r = kpse.version()

4.7 The lang library
This library provides the interface to LuaTeX’s structure representing a language, and the associated functions.

<language> l = lang.new()
<language> l = lang.new(<number> id)

This function creates a new userdata object. An object of type <language> is the first argument to most of the other functions in the lang library. These functions can also be used as if they were object methods, using the colon syntax.

Without an argument, the next available internal id number will be assigned to this object. With argument, an object will be created that links to the internal language with that id number.

<number> n = lang.id(<language> l)
returns the internal \language id number this object refers to.
<string> n = lang.hyphenation(<language> l)
lang.hyphenation(<language> l, <string> n)

Either returns the current hyphenation exceptions for this language, or adds new ones. The syntax of the string is explained in section 6.3.

lang.clear_hyphenation(<language> l)

Clears the exception dictionary for this language.

<string> n = lang.clean(<string> o)

Creates a hyphenation key from the supplied hyphenation value. The syntax of the argument string is explained in section 6.3. This function is useful if you want to do something else based on the words in a dictionary file, like spell-checking.

<string> n = lang.patterns(<language> l)
lang.patterns(<language> l, <string> n)

Adds additional patterns for this language object, or returns the current set. The syntax of this string is explained in section 6.3.

lang.clear_patterns(<language> l)

Clears the pattern dictionary for this language.

<number> n = lang.prehyphenchar(<language> l)
lang.prehyphenchar(<language> l, <number> n)

Gets or sets the 'pre-break' hyphen character for implicit hyphenation in this language (initially the hyphen, decimal 45).

<number> n = lang.posthyphenchar(<language> l)
lang.posthyphenchar(<language> l, <number> n)

Gets or sets the 'post-break' hyphen character for implicit hyphenation in this language (initially null, decimal 0, indicating emptiness).

<number> n = lang.preexhyphenchar(<language> l)
lang.preexhyphenchar(<language> l, <number> n)

Gets or sets the ‘pre-break’ hyphen character for explicit hyphenation in this language (initially null, decimal 0, indicating emptiness).

<number> n = lang.postexhyphenchar(<language> l)
lang.postexhyphenchar(<language> l, <number> n)

Gets or sets the ‘post-break’ hyphen character for explicit hyphenation in this language (initially null, decimal 0, indicating emptiness).
\begin{verbatim}
<boolean> success = lang.hyphenate(<node> head)
<boolean> success = lang.hyphenate(<node> head, <node> tail)

Inserts hyphenation points (discretionary nodes) in a node list. If \texttt{tail} is given as argument, processing stops on that node. Currently, \texttt{success} is always true if \texttt{head} (and \texttt{tail}, if specified) are proper nodes, regardless of possible other errors.

Hyphenation works only on ‘characters’, a special subtype of all the glyph nodes with the node subtype having the value 1. Glyph modes with different subtypes are not processed. See section 6.1 for more details.
\end{verbatim}

4.8 The \texttt{lua} library

This library contains one read-only item:

\begin{verbatim}
<string> s = lua.version
\end{verbatim}

This returns the LUA version identifier string. The value is currently Lua 5.2.

4.8.1 LUA bytecode registers

Lua registers can be used to communicate Lua functions across LUA chunks. The accepted values for assignments are functions and \texttt{nil}. Likewise, the retrieved value is either a function or \texttt{nil}.

\begin{verbatim}
lua.bytecode[<number> n] = <function> f
lua.bytecode[<number> n]()
\end{verbatim}

The contents of the \texttt{lua.bytecode} array is stored inside the format file as actual LUA bytecode, so it can also be used to preload LUA code.

Note: The function must not contain any upvalues. Currently, functions containing upvalues can be stored (and their upvalues are set to \texttt{nil}), but this is an artifact of the current LUA implementation and thus subject to change.

The associated function calls are

\begin{verbatim}
<function> f = lua.getbytecode(<number> n)
lua.setbytecode(<number> n, <function> f)
\end{verbatim}

Note: Since a LUA file loaded using \texttt{loadfile(filename)} is essentially an anonymous function, a complete file can be stored in a bytecode register like this:

\begin{verbatim}
lua.bytecode[n] = loadfile(filename)
\end{verbatim}

Now all definitions (functions, variables) contained in the file can be created by executing this bytecode register:

\begin{verbatim}
lua.bytecode[n]()
\end{verbatim}
Note that the path of the file is stored in the Lua bytecode to be used in stack backtraces and therefore dumped into the format file if the above code is used in \texttt{\textbackslash ini\TeX}. If it contains private information, i.e. the user name, this information is then contained in the format file as well. This should be kept in mind when preloading files into a bytecode register in \texttt{\textbackslash ini\TeX}.

4.8.2 \texttt{LUA} chunk name registers

There is an array of 65536 (0–65535) potential chunk names for use with the \texttt{\textbackslash directlua} and \texttt{\textbackslash latelua} primitives.

\texttt{lua.name[<number> n] = <string> s}
\texttt{<string> s = lua.name[<number> n]}

If you want to unset a lua name, you can assign \texttt{nil} to it.

4.9 The \texttt{mplib} library

The MetaPost library interface registers itself in the table \texttt{mplib}. It is based on \texttt{MPLIB} version 1.999.

4.9.1 \texttt{mplib.new}

To create a new MetaPost instance, call

\texttt{<mpinstance> mp = mplib.new({...})}

This creates the \texttt{mp} instance object. The argument hash can have a number of different fields, as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_line</td>
<td>number</td>
<td>error line width</td>
<td>79</td>
</tr>
<tr>
<td>print_line</td>
<td>number</td>
<td>line length in ps output</td>
<td>100</td>
</tr>
<tr>
<td>random_seed</td>
<td>number</td>
<td>the initial random seed</td>
<td>variable</td>
</tr>
<tr>
<td>interaction</td>
<td>string</td>
<td>the interaction mode, one of \texttt{batch}, \texttt{errorstop}, \texttt{nonstop}, \texttt{scroll}, \texttt{errorstop}</td>
<td></td>
</tr>
<tr>
<td>job_name</td>
<td>string</td>
<td>\texttt{--jobname}</td>
<td>mpout</td>
</tr>
<tr>
<td>find_file</td>
<td>function</td>
<td>a function to find files</td>
<td>only local files</td>
</tr>
</tbody>
</table>

The \texttt{find_file} function should be of this form:

\texttt{<string> found = finder (<string> name, <string> mode, <string> type)}

with:

<table>
<thead>
<tr>
<th>name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>\texttt{r} or \texttt{w}</td>
</tr>
</tbody>
</table>
type  the kind of file, one of: \texttt{mp}, \texttt{tfm}, \texttt{map}, \texttt{pfb}, \texttt{enc}

Return either the full pathname of the found file, or \texttt{nil} if the file cannot be found.

Note that the new version of \texttt{mplib} no longer uses binary mem files, so the way to preload a set of macros is simply to start off with an \texttt{input} command in the first \texttt{mp:execute()} call.

### 4.9.2 \texttt{mp:statistics}

You can request statistics with:

\begin{verbatim}
<table>
  stats = mp:statistics()
</table>
\end{verbatim}

This function returns the vital statistics for an \texttt{mplib} instance. There are four fields, giving the maximum number of used items in each of four allocated object classes:

- \texttt{main_memory}  number  memory size
- \texttt{hash_size}    number  hash size
- \texttt{param_size}   number  simultaneous macro parameters
- \texttt{max_in_open}  number  input file nesting levels

Note that in the new version of \texttt{mplib}, this is informational only. The objects are all allocated dynamically, so there is no chance of running out of space unless the available system memory is exhausted.

### 4.9.3 \texttt{mp:execute}

You can ask the MetaPost interpreter to run a chunk of code by calling

\begin{verbatim}
<table>
  rettable = mp:execute('metapost language chunk')
</table>
\end{verbatim}

for various bits of MetaPost language input. Be sure to check the \texttt{rettable.status} (see below) because when a fatal MetaPost error occurs the \texttt{mplib} instance will become unusable thereafter.

Generally speaking, it is best to keep your chunks small, but beware that all chunks have to obey proper syntax, like each of them is a small file. For instance, you cannot split a single statement over multiple chunks.

In contrast with the normal standalone \texttt{mpost} command, there is no implied ‘input’ at the start of the first chunk.

### 4.9.4 \texttt{mp:finish}

\begin{verbatim}
<table>
  rettable = mp:finish()
</table>
\end{verbatim}
If for some reason you want to stop using an mplib instance while processing is not yet actually done, you can call `mp:finish`. Eventually, used memory will be freed and open files will be closed by the Lua garbage collector, but an explicit `mp:finish` is the only way to capture the final part of the output streams.

### 4.9.5 Result table

The return value of `mp:execute` and `mp:finish` is a table with a few possible keys (only `status` is always guaranteed to be present).

- `log` string: output to the 'log' stream
- `term` string: output to the 'term' stream
- `error` string: output to the 'error' stream (only used for 'out of memory')
- `status` number: the return value: 0=good, 1=warning, 2=errors, 3=fatal error
- `fig` table: an array of generated figures (if any)

When `status` equals 3, you should stop using this mplib instance immediately, it is no longer capable of processing input.

If it is present, each of the entries in the `fig` array is a userdata representing a figure object, and each of those has a number of object methods you can call:

- `boundingbox` function: returns the bounding box, as an array of 4 values
- `postscript` function: returns a string that is the ps output of the `fig`. This function accepts two optional integer arguments for specifying the values of `prologues` (first argument) and `procset` (second argument)
- `svg` function: returns a string that is the svg output of the `fig`. This function accepts an optional integer argument for specifying the value of `prologues`
- `objects` function: returns the actual array of graphic objects in this `fig`
- `copy_objects` function: returns a deep copy of the array of graphic objects in this `fig`
- `filename` function: the filename this `fig`'s PostScript output would have written to in standalone mode
- `width` function: the `charwd` value
- `height` function: the `charht` value
- `depth` function: the `chardp` value
- `italcorr` function: the `charit` value
- `charcode` function: the (rounded) `charcode` value

**NOTE:** you can call `fig:objects()` only once for any one `fig` object!

When the boundingbox represents a 'negated rectangle', i.e. when the first set of coordinates is larger than the second set, the picture is empty.

Graphical objects come in various types that each has a different list of accessible values. The types are: `fill`, `outline`, `text`, `start_clip`, `stop_clip`, `start_bounds`, `stop_bounds`, `special`.

There is helper function (`mplib.fields(obj)`) to get the list of accessible values for a particular object, but you can just as easily use the tables given below.
All graphical objects have a field `type` that gives the object type as a string value; it is not explicit mentioned in the following tables. In the following, `numbers` are PostScript points represented as a floating point number, unless stated otherwise. Field values that are of type `table` are explained in the next section.

### 4.9.5.1 fill

- `path`: table the list of knots
- `htap`: table the list of knots for the reversed trajectory
- `pen`: table knots of the pen
- `color`: table the object’s color
- `linejoin`: number line join style (bare number)
- `miterlimit`: number miterlimit
- `prescript`: string the prescript text
- `postscript`: string the postscript text

The entries `htap` and `pen` are optional.

There is helper function `mplib.pen_info(obj)` that returns a table containing a bunch of vital characteristics of the used pen (all values are floats):

- `width`: number width of the pen
- `sx`: number $x$ scale
- `rx`: number $xy$ multiplier
- `ry`: number $yx$ multiplier
- `sy`: number $y$ scale
- `tx`: number $x$ offset
- `ty`: number $y$ offset

### 4.9.5.2 outline

- `path`: table the list of knots
- `pen`: table knots of the pen
- `color`: table the object’s color
- `linejoin`: number line join style (bare number)
- `miterlimit`: number miterlimit
- `linecap`: number line cap style (bare number)
- `dash`: table representation of a dash list
- `prescript`: string the prescript text
- `postscript`: string the postscript text

The entry `dash` is optional.
4.9.5.3 text

- `text`: string, the text
- `font`: string, font name
- `dsize`: number, font size
- `color`: table, the object's color
- `width`: number
- `height`: number
- `depth`: number
- `transform`: table, a text transformation
- `prescript`: string, the prescript text
- `postscript`: string, the postscript text

4.9.5.4 special

- `prescript`: string, special text

4.9.5.5 start_bounds, start_clip

- `path`: table, the list of knots

4.9.5.6 stop_bounds, stop_clip

Here are no fields available.

4.9.6 Subsidiary table formats

4.9.6.1 Paths and pens

Paths and pens (that are really just a special type of paths as far as MPLib is concerned) are represented by an array where each entry is a table that represents a knot.

- `left_type`: string, when present: 'endpoint', but usually absent
- `right_type`: string, like `left_type`
- `x_coord`: number, X coordinate of this knot
- `y_coord`: number, Y coordinate of this knot
- `left_x`: number, X coordinate of the precontrol point of this knot
- `left_y`: number, Y coordinate of the precontrol point of this knot
- `right_x`: number, X coordinate of the postcontrol point of this knot
- `right_y`: number, Y coordinate of the postcontrol point of this knot

There is one special case: pens that are (possibly transformed) ellipses have an extra string-valued key `type` with value `elliptical` besides the array part containing the knot list.
4.9.6.2 Colors

A color is an integer array with 0, 1, 3 or 4 values:

- 0 marking only no values
- 1 greyscale one value in the range (0, 1), 'black' is 0
- 3 RGB three values in the range (0, 1), 'black' is 0, 0, 0
- 4 CMYK four values in the range (0, 1), 'black' is 0, 0, 0, 1

If the color model of the internal object was uninitialized, then it was initialized to the values representing 'black' in the colorspace defaultcolormodel that was in effect at the time of the shipout.

4.9.6.3 Transforms

Each transform is a six-item array.

- 1 number represents x
- 2 number represents y
- 3 number represents xx
- 4 number represents yx
- 5 number represents xy
- 6 number represents yy

Note that the translation (index 1 and 2) comes first. This differs from the ordering in PostScript, where the translation comes last.

4.9.6.4 Dashes

Each dash is two-item hash, using the same model as PostScript for the representation of the dashlist. dashes is an array of 'on' and 'off', values, and offset is the phase of the pattern.

dashes hash an array of on-off numbers
offset number the starting offset value

4.9.7 Character size information

These functions find the size of a glyph in a defined font. The fontname is the same name as the argument to infont; the char is a glyph id in the range 0 to 255; the returned w is in AFM units.

4.9.7.1 mp:char_width

<number> w = mp:char_width(<string> fontname, <number> char)
4.9.7.2 mp:char_height

<number> w = mp:char_height(<string> fontname, <number> char)

4.9.7.3 mp:char_depth

<number> w = mp:char_depth(<string> fontname, <number> char)

4.10 The node library

The node library contains functions that facilitate dealing with (lists of) nodes and their values. They allow you to create, alter, copy, delete, and insert LuaTEX node objects, the core objects within the typesetter.

LuaTEX nodes are represented in Lua as userdata with the metadata type luatex.node. The various parts within a node can be accessed using named fields.

Each node has at least the three fields next, id, and subtype:

- The next field returns the userdata object for the next node in a linked list of nodes, or nil, if there is no next node.
- The id indicates TeX's 'node type'. The field id has a numeric value for efficiency reasons, but some of the library functions also accept a string value instead of id.
- The subtype is another number. It often gives further information about a node of a particular id, but it is most important when dealing with 'whatsits', because they are differentiated solely based on their subtype.

The other available fields depend on the id (and for 'whatsits', the subtype) of the node. Further details on the various fields and their meanings are given in chapter 8.

Support for unset (alignment) nodes is partial: they can be queried and modified from Lua code, but not created.

Nodes can be compared to each other, but: you are actually comparing indices into the node memory. This means that equality tests can only be trusted under very limited conditions. It will not work correctly in any situation where one of the two nodes has been freed and/or reallocated: in that case, there will be false positives.

At the moment, memory management of nodes should still be done explicitly by the user. Nodes are not 'seen' by the Lua garbage collector, so you have to call the node freeing functions yourself when you are no longer in need of a node (list). Nodes form linked lists without reference counting, so you have to be careful that when control returns back to LuaTEX itself, you have not deleted nodes that are still referenced from a next pointer elsewhere, and that you did not create nodes that are referenced more than once.

There are statistics available with regards to the allocated node memory, which can be handy for tracing.
4.10.1 Node handling functions

4.10.1.1 node.is_node

<boolean> \( t = \text{node.is_node}(\text{<any> item}) \)

This function returns true if the argument is a userdata object of type <node>.

4.10.1.2 node.types

<table> \( t = \text{node.types}() \)

This function returns an array that maps node id numbers to node type strings, providing an overview of the possible top-level id types.

4.10.1.3 node.whatsits

<table> \( t = \text{node.whatsits}() \)

TeX’s ‘whatsits’ all have the same id. The various subtypes are defined by their subtype fields. The function is much like node.types, except that it provides an array of subtype mappings.

4.10.1.4 node.id

<number> \( id = \text{node.id}(\text{<string> type}) \)

This converts a single type name to its internal numeric representation.

4.10.1.5 node.subtype

<number> \( subtype = \text{node.subtype}(\text{<string> type}) \)

This converts a single whatsit name to its internal numeric representation (subtype).

4.10.1.6 node.type

<string> \( type = \text{node.type}(\text{<any> n}) \)

In the argument is a number, then this function converts an internal numeric representation to an external string representation. Otherwise, it will return the string node if the object represents a node (this is new in 0.65), and nil otherwise.
4.10.1.7 node.fields

\[ \text{t} = \text{node.fields}(<\text{number}> \text{id}) \]
\[ \text{t} = \text{node.fields}(<\text{number}> \text{id}, <\text{number}> \text{subtype}) \]

This function returns an array of valid field names for a particular type of node. If you want to get the valid fields for a ‘whatsit’, you have to supply the second argument also. In other cases, any given second argument will be silently ignored.

This function accepts string \text{id} and \text{subtype} values as well.

4.10.1.8 node.has_field

\[ \text{t} = \text{node.has_field}(<\text{node}> \text{n}, <\text{string}> \text{field}) \]

This function returns a boolean that is only true if \text{n} is actually a node, and it has the field.

4.10.1.9 node.new

\[ \text{n} = \text{node.new}(<\text{number}> \text{id}) \]
\[ \text{n} = \text{node.new}(<\text{number}> \text{id}, <\text{number}> \text{subtype}) \]

Creates a new node. All of the new node's fields are initialized to either zero or \text{nil} except for \text{id} and \text{subtype} (if supplied). If you want to create a new whatsit, then the second argument is required, otherwise it need not be present. As with all node functions, this function creates a node on the \TeX level.

This function accepts string \text{id} and \text{subtype} values as well.

4.10.1.10 node.free

\text{node.free(<node> \text{n})}

Removes the node \text{n} from \TeX's memory. Be careful: no checks are done on whether this node is still pointed to from a register or some \text{next} field: it is up to you to make sure that the internal data structures remain correct.

4.10.1.11 node.flush_list

\text{node.flush_list(<node> \text{n})}

Removes the node list \text{n} and the complete node list following \text{n} from \TeX's memory. Be careful: no checks are done on whether any of these nodes is still pointed to from a register or some \text{next} field: it is up to you to make sure that the internal data structures remain correct.
4.10.1.12 node.copy

`<node> m = node.copy(<node> n)`

Creates a deep copy of node `n`, including all nested lists as in the case of a hlist or vlist node. Only the `next` field is not copied.

4.10.1.13 node.copy_list

`<node> m = node.copy_list(<node> n)`
`<node> m = node.copy_list(<node> n, <node> m)`

Creates a deep copy of the node list that starts at `n`. If `m` is also given, the copy stops just before node `m`.

Note that you cannot copy attribute lists this way, specialized functions for dealing with attribute lists will be provided later but are not there yet. However, there is normally no need to copy attribute lists as when you do assignments to the `attr` field or make changes to specific attributes, the needed copying and freeing takes place automatically.

4.10.1.14 node.next (0.65)

`<node> m = node.next(<node> n)`

Returns the node following this node, or `nil` if there is no such node.

4.10.1.15 node.prev (0.65)

`<node> m = node.prev(<node> n)`

Returns the node preceding this node, or `nil` if there is no such node.

4.10.1.16 node.current_attr (0.66)

`<node> m = node.current_attr()`

Returns the currently active list of attributes, if there is one.

The intended usage of `current_attr` is as follows:

```lua
local x1 = node.new("glyph")
x1.attr = node.current_attr()
local x2 = node.new("glyph")
x2.attr = node.current_attr()
```

or:
local x1 = node.new("glyph")
local x2 = node.new("glyph")
local ca = node.current_attr()
x1.attr = ca
x2.attr = ca

The attribute lists are ref counted and the assignment takes care of incrementing the refcount. You cannot expect the value `ca` to be valid any more when you assign attributes (using `tex.setattribute`) or when control has been passed back to TeX.

Note: this function is somewhat experimental, and it returns the actual attribute list, not a copy thereof. Therefore, changing any of the attributes in the list will change these values for all nodes that have the current attribute list assigned to them.

### 4.10.1.17 node.hpack

```lua
<nodel> h, <number> b = node.hpack(<node> n)
<nodel> h, <number> b = node.hpack(<node> n, <number> w, <string> info)
<nodel> h, <number> b = node.hpack(<node> n, <number> w, <string> info, <string> dir)
```

This function creates a new hlist by packaging the list that begins at node `n` into a horizontal box. With only a single argument, this box is created using the natural width of its components. In the three argument form, `info` must be either additional or exactly, and `w` is the additional (\hbox spread) or exact (\hbox to) width to be used.

Direction support added in LuaTeX 0.45.

The second return value is the badness of the generated box, this extension was added in 0.51.

Caveat: at this moment, there can be unexpected side-effects to this function, like updating some of the \marks and \inserts. Also note that the content of `h` is the original node list `n`: if you call `node.free(h)` you will also free the node list itself, unless you explicitly set the list field to nil beforehand. And in a similar way, calling `node.free(n)` will invalidate `h` as well!

### 4.10.1.18 node.vpack (since 0.36)

```lua
<nodel> h, <number> b = node.vpack(<node> n)
<nodel> h, <number> b = node.vpack(<node> n, <number> w, <string> info)
<nodel> h, <number> b = node.vpack(<node> n, <number> w, <string> info, <string> dir)
```

This function creates a new vlist by packaging the list that begins at node `n` into a vertical box. With only a single argument, this box is created using the natural height of its components. In the three argument form, `info` must be either additional or exactly, and `w` is the additional (\vbox spread) or exact (\vbox to) height to be used.

Direction support added in LuaTeX 0.45.
The second return value is the badness of the generated box, this extension was added in 0.51.
See the description of \texttt{node.hpack()} for a few memory allocation caveats.

4.10.1.19 \texttt{node.dimensions (0.43)}

\begin{verbatim}
<number> w, <number> h, <number> d = node.dimensions(<node> n)
<number> w, <number> h, <number> d = node.dimensions(<node> n, <string> dir)
<number> w, <number> h, <number> d = node.dimensions(<node> n, <node> t)
<number> w, <number> h, <number> d = node.dimensions(<node> n, <node> t, <string> dir)
\end{verbatim}

This function calculates the natural in-line dimensions of the node list starting at node \texttt{n} and terminating just before node \texttt{t} (or the end of the list, if there is no second argument). The return values are scaled points. An alternative format that starts with glue parameters as the first three arguments is also possible:

\begin{verbatim}
<number> w, <number> h, <number> d =
    node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order, <node> n)
<number> w, <number> h, <number> d =
    node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order, <node> n, <string> dir)
<number> w, <number> h, <number> d =
    node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order, <node> n, <node> t)
<number> w, <number> h, <number> d =
    node.dimensions(<number> glue_set, <number> glue_sign, <number> glue_order, <node> n, <node> t, <string> dir)
\end{verbatim}

This calling method takes glue settings into account and is especially useful for finding the actual width of a sublist of nodes that are already boxed, for example in code like this, which prints the width of the space inbetween the \texttt{a} and \texttt{b} as it would be if \texttt{\box0} was used as-is:

\begin{verbatim}
\setbox0 = \hbox to 20pt {a b}
\end{verbatim}

\begin{verbatim}
\directlua{print (node.dimensions(tex.box[0].glue_set, tex.box[0].glue_sign, tex.box[0].glue_order, tex.box[0].head.next, node.tail(tex.box[0].head)))}
\end{verbatim}

Direction support added in \LaTeX{} 0.45.
4.10.1.20 node.mlist_to_hlist

`<node> h = node.mlist_to_hlist(<node> n,`<br>`  <string> display_type, <boolean> penalties)`

This runs the internal mlist to hlist conversion, converting the math list in `n` into the horizontal list `h`. The interface is exactly the same as for the callback `mlist_to_hlist`.

4.10.1.21 node.slide

`<node> m = node.slide(<node> n)`

Returns the last node of the node list that starts at `n`. As a side-effect, it also creates a reverse chain of `prev` pointers between nodes.

4.10.1.22 node.tail

`<node> m = node.tail(<node> n)`

Returns the last node of the node list that starts at `n`.

4.10.1.23 node.length

`<number> i = node.length(<node> n)`<br>`<number> i = node.length(<node> n, <node> m)`

Returns the number of nodes contained in the node list that starts at `n`. If `m` is also supplied it stops at `m` instead of at the end of the list. The node `m` is not counted.

4.10.1.24 node.count

`<number> i = node.count(<number> id, <node> n)`<br>`<number> i = node.count(<number> id, <node> n, <node> m)`

Returns the number of nodes contained in the node list that starts at `n` that have a matching `id` field. If `m` is also supplied, counting stops at `m` instead of at the end of the list. The node `m` is not counted.

This function also accept string `id`'s.

4.10.1.25 node.traverse

`<node> t = node.traverse(<node> n)`

This is a lua iterator that loops over the node list that starts at `n`. Typical input code like this
for n in node.traverse(head) do
...
end

is functionally equivalent to:

do
  local n
  local function f (head, var)
    local t
    if var == nil then
      t = head
    else
      t = var.next
    end
    return t
  end
  while true do
    n = f (head, n)
    if n == nil then break end
    ...
  end
end

It should be clear from the definition of the function \( f \) that even though it is possible to add or remove nodes from the node list while traversing, you have to take great care to make sure all the \texttt{next} (and \texttt{prev}) pointers remain valid.

If the above is unclear to you, see the section 'For Statement' in the Lua Reference Manual.

\textbf{4.10.1.26 node.traverse\_id}

\texttt{<node> t = node.traverse\_id(<number> id, <node> n)}

This is an iterator that loops over all the nodes in the list that starts at \( n \) that have a matching \texttt{id} field.

See the previous section for details. The change is in the local function \( f \), which now does an extra while loop checking against the upvalue \texttt{id}:

local function f (head, var)
  local t
  if var == nil then
    t = head
  else
    t = var.next
  end
while not t.id == id do
    t = t.next
end
return t
end

4.10.1.27 node.end_of_math (0.76)

<node> t = node.end_of_math(<node> start)

Looks for and returns the next math_node following the start. If the given node is a math endnode this helper return that node, else it follows the list and return the next math endnote. If no such node is found nil is returned.

4.10.1.28 node.remove

<node> head, current = node.remove(<node> head, <node> current)

This function removes the node current from the list following head. It is your responsibility to make sure it is really part of that list. The return values are the new head and current nodes. The returned current is the node following the current in the calling argument, and is only passed back as a convenience (or nil, if there is no such node). The returned head is more important, because if the function is called with current equal to head, it will be changed.

4.10.1.29 node.insert_before

<node> head, new = node.insert_before(<node> head, <node> current, <node> new)

This function inserts the node new before current into the list following head. It is your responsibility to make sure that current is really part of that list. The return values are the (potentially mutated) head and the node new, set up to be part of the list (with correct next field). If head is initially nil, it will become new.

4.10.1.30 node.insert_after

<node> head, new = node.insert_after(<node> head, <node> current, <node> new)

This function inserts the node new after current into the list following head. It is your responsibility to make sure that current is really part of that list. The return values are the head and the node new, set up to be part of the list (with correct next field). If head is initially nil, it will become new.
4.10.1.31  node.first_glyph (0.65)

<nodelist> n = node.first_glyph(<nodelist> n)
<nodelist> n = node.first_glyph(<nodelist> n, <nodelist> m)

Returns the first node in the list starting at \texttt{n} that is a glyph node with a subtype indicating it is a glyph, or \texttt{nil}. If \texttt{m} is given, processing stops at (but including) that node, otherwise processing stops at the end of the list.

Note: this function used to be called \texttt{first_character}. It has been renamed in \texttt{Lua\TeX} 0.65, and the old name is deprecated now.

4.10.1.32  node.ligaturing

<nodelist> h, <nodelist> t, <boolean> success = node.ligaturing(<nodelist> n)
<nodelist> h, <nodelist> t, <boolean> success = node.ligaturing(<nodelist> n, <nodelist> m)

Apply \TeX-style ligaturing to the specified nodelist. The tail node \texttt{m} is optional. The two returned nodes \texttt{h} and \texttt{t} are the new head and tail (both \texttt{n} and \texttt{m} can change into a new ligature).

4.10.1.33  node.kerning

<nodelist> h, <nodelist> t, <boolean> success = node.kerning(<nodelist> n)
<nodelist> h, <nodelist> t, <boolean> success = node.kerning(<nodelist> n, <nodelist> m)

Apply \TeX-style kerning to the specified nodelist. The tail node \texttt{m} is optional. The two returned nodes \texttt{h} and \texttt{t} are the head and tail (either one of these can be an inserted kern node, because special kernings with word boundaries are possible).

4.10.1.34  node.unprotect_glyphs

\begin{verbatim}
nodetext.unprotect_glyphs(<nodelist> n)
\end{verbatim}

Subtracts 256 from all glyph node subtypes. This and the next function are helpers to convert from \texttt{characters} to \texttt{glyphs} during node processing.

4.10.1.35  node.protect_glyphs

\begin{verbatim}
nodetext.protect_glyphs(<nodelist> n)
\end{verbatim}

Adds 256 to all glyph node subtypes in the node list starting at \texttt{n}, except that if the value is 1, it adds only 255. The special handling of 1 means that \texttt{characters} will become \texttt{glyphs} after subtraction of 256.
4.10.1.36 node.last_node

<n> n = node.last_node()

This function pops the last node from \TeX's 'current list'. It returns that node, or \texttt{nil} if the current list is empty.

4.10.1.37 node.write

node.write(<node> n)

This is an experimental function that will append a node list to \TeX's 'current list' (the node list is not deep-copied any more since version 0.38). There is no error checking yet!

4.10.1.38 node.protrusion_skippable (0.60.1)

<boolean> skippable = node.protrusion_skippable(<node> n)

Returns \texttt{true} if, for the purpose of line boundary discovery when character protrusion is active, this node can be skipped.

4.10.2 Attribute handling

Attributes appear as linked list of userdata objects in the \texttt{attr} field of individual nodes. They can be handled individually, but it is much safer and more efficient to use the dedicated functions associated with them.

4.10.2.1 node.has_attribute

<number> v = node.has_attribute(<node> n, <number> id)
<number> v = node.has_attribute(<node> n, <number> id, <number> val)

Tests if a node has the attribute with number \texttt{id} set. If \texttt{val} is also supplied, also tests if the value matches \texttt{val}. It returns the value, or, if no match is found, \texttt{nil}.

4.10.2.2 node.set_attribute

node.set_attribute(<node> n, <number> id, <number> val)

Sets the attribute with number \texttt{id} to the value \texttt{val}. Duplicate assignments are ignored. \textit{[needs explanation]}
4.10.2.3 node.unset_attribute

\[<number> \ v = \text{node.unset_attribute}(<node> \ n, \ <number> \ id)\]
\[<number> \ v = \text{node.unset_attribute}(<node> \ n, \ <number> \ id, \ <number> \ val)\]

Unsets the attribute with number \(id\). If \(val\) is also supplied, it will only perform this operation if the value matches \(val\). Missing attributes or attribute-value pairs are ignored.

If the attribute was actually deleted, returns its old value. Otherwise, returns \text{nil}.

4.11 The pdf library

This contains variables and functions that are related to the PDF backend.

4.11.1 pdf.mapfile, pdf.mapline (new in 0.53.0)

\[\text{pdf.mapfile}(<string> \ \text{map file})\]
\[\text{pdf.mapfile}(<string> \ \text{map line})\]

These two functions can be used to replace primitives \texttt{\pdfmapfile} and \texttt{\pdfmapline} from PDF\TeX. They expect a string as only parameter and have no return value.

The also functions replace the former variables \texttt{pdf.pdfmapfile} and \texttt{pdf.pdfmapline}.

4.11.2 pdf.catalog, pdf.info, pdf.names, pdf.trailer (new in 0.53.0)

These variables offer a read-write interface to the corresponding PDF\TeX token lists. The value types are strings and they are written out to the PDF file directly after the PDF\TeX token registers.

The preferred interface is now \texttt{pdf.setcatalog}, \texttt{pdf.setinfo} \texttt{pdf.setnames} and \texttt{pdf.settrailer} for setting these properties and \texttt{pdf.getcatalog}, \texttt{pdf.getinfo} \texttt{pdf.getnames} and \texttt{pdf.gettrailer} for querying them.

The corresponding 'pdf' parameter names \texttt{pdf.pdfcatalog}, \texttt{pdf.pdfinfo}, \texttt{pdf.pdfnames}, and \texttt{pdf.pdftrailer} are removed in 0.79.0.

4.11.3 pdf.<set/get>pageattributes, pdf.<set/get>pageresources, pdf.<set/get>pagesattributes

These variables offer a read-write interface to related token lists. The value types are strings. The variables have no interaction with the corresponding PDF\TeX token registers \texttt{\pdfpageattr}, \texttt{\pdfpageresources}, and \texttt{\pdfpagesattr}. They are written out to the PDF file directly after the PDF\TeX token registers.
The preferred interface is now `pdf.setpageattributes`, `pdf.setpagesattributes` and `pdf.setpageresources` for setting these properties and `pdf.getpageattributes`, `pdf.getpagesattributes` and `pdf.getpageresources` for querying them.

### 4.11.4 pdf.h, pdf.v

These are the `h` and `v` values that define the current location on the output page, measured from its lower left corner. The values can be queried using scaled points as units.

```lua
local h = pdf.h
local v = pdf.v
```

### 4.11.5 pdf.getpos, pdf.gethpos, pdf.getvpos

These are the function variants of `pdf.h` and `pdf.v`. Sometimes using a function is preferred over a key so this saves wrapping. Also, these functions are faster then the key based access, as `h` and `v` keys are not real variables but looked up using a metatable call. The `getpos` function returns two values, the other return one.

```lua
local h, v = pdf.getpos()
```

### 4.11.6 pdf.hasmatrix, pdf.getmatrix

The current matrix transformation is available via the `getmatrix` command, which returns 6 values: `sx`, `rx`, `ry`, `sy`, `tx`, and `ty`. The `hasmatrix` function returns `true` when a matrix is applied.

```lua
if pdf.hasmatrix() then
    local sx, rx, ry, sy, tx, ty = pdf.getmatrix()
    -- do something useful or not
end
```

### 4.11.7 pdf.print

A print function to write stuff to the PDF document that can be used from within a \texttt{\LaTeX} argument. This function is not to be used inside \texttt{\directlua} unless you know exactly what you are doing.

```lua
pdf.print(<string> s)
pdf.print(<string> type, <string> s)
```

The optional parameter can be used to mimic the behavior of \texttt{\pdfliteral}: the \texttt{type} is \texttt{direct} or \texttt{page}. 
4.11.8 pdf.immediateobj

This function creates a PDF object and immediately writes it to the PDF file. It is modelled after PDF\TeX's \texttt{immediate\pdfobj} primitives. All function variants return the object number of the newly generated object.

<number> n = pdf.immediateobj(<string> objtext)
<number> n = pdf.immediateobj("file", <string> filename)
<number> n = pdf.immediateobj("stream", <string> streamtext, <string> attrtext)
<number> n = pdf.immediateobj("streamfile", <string> filename, <string> attrtext)

The first version puts the \texttt{objtext} raw into an object. Only the object wrapper is automatically generated, but any internal structure (like \texttt{<< >>} dictionary markers) needs to be provided by the user. The second version with keyword "file" as 1st argument puts the contents of the file with name \texttt{filename} raw into the object. The third version with keyword "stream" creates a stream object and puts the \texttt{streamtext} raw into the stream. The stream length is automatically calculated. The optional \texttt{attrtext} goes into the dictionary of that object. The fourth version with keyword "streamfile" does the same as the 3rd one, it just reads the stream data raw from a file.

An optional first argument can be given to make the function use a previously reserved PDF object.

<number> n = pdf.immediateobj(<integer> n, <string> objtext)
<number> n = pdf.immediateobj(<integer> n, "file", <string> filename)
<number> n = pdf.immediateobj(<integer> n, "stream", <string> streamtext, <string> attrtext)
<number> n = pdf.immediateobj(<integer> n, "streamfile", <string> filename, <string> attrtext)

4.11.9 pdf.obj

This function creates a PDF object, which is written to the PDF file only when referenced, e.g., by \texttt{pdf.refobj()}. All function variants return the object number of the newly generated object, and there are two separate calling modes.

The first mode is modelled after PDF\TeX's \texttt{pdfobj} primitive.

<number> n = pdf.obj(<string> objtext)
<number> n = pdf.obj("file", <string> filename)
<number> n = pdf.obj("stream", <string> streamtext, <string> attrtext)
<number> n = pdf.obj("streamfile", <string> filename, <string> attrtext)

An optional first argument can be given to make the function use a previously reserved PDF object.

<number> n = pdf.obj(<integer> n, <string> objtext)
<number> n = pdf.obj(<integer> n, "file", <string> filename)
<number> n = pdf.obj(<integer> n, "stream", <string> streamtext, <string> attrtext)
<number> n = pdf.obj(<integer> n, "streamfile", <string> filename, <string> attrtext)

The second mode accepts a single argument table with key–value pairs.

<number> n = pdf.obj{ type = <string>,
    immediate = <boolean>,
    objnum = <number>,
    attr = <string>,
    compresslevel = <number>,
    objcompression = <boolean>,
    file = <string>,
    string = <string>}

The type field can have the values raw and stream, this field is required, the others are optional (within constraints).

Note: this mode makes pdf.obj look more flexible than it actually is: the constraints from the separate parameter version still apply, so for example you can’t have both string and file at the same time.

4.11.10 pdf.refobj

This function, the Lua version of the \pdfrefobj primitive, references an object by its object number, so that the object will be written out.

df.refobj(<integer> n)

This function works in both the \directlua and \latelua environment. Inside \directlua a new whatsit node 'pdf_refobj' is created, which will be marked for flushing during page output and the object is then written directly after the page, when also the resources objects are written out. Inside \latelua the object will be marked for flushing.
This function has no return values.

4.11.11 pdf.reserveobj

This function creates an empty PDF object and returns its number.

<number> n = pdf.reserveobj()
<number> n = pdf.reserveobj("annot")
4.11.12 pdf.registerannot (new in 0.47.0)

This function adds an object number to the /Annots array for the current page without doing anything else. This function can only be used from within \LaTeX.

pdf.registerannot (<number> objnum)

4.12 The pdfscanner library (new in 0.72.0)

The pdfscanner library allows interpretation of PDF content streams and /ToUnicode (cmap) streams. You can get those streams from the epdf library, as explained in an earlier section. There is only a single top-level function in this library:

pdfscanner.scan (<Object> stream, <table> operatortable, <table> info)

The first argument, stream, should be either a PDF stream object, or a PDF array of PDF stream objects (those options comprise the possible return values of <Page>:getContents() and <Object>:getStream() in the epdf library).

The second argument, operatortable, should be a Lua table where the keys are PDF operator name strings and the values are Lua functions (defined by you) that are used to process those operators. The functions are called whenever the scanner finds one of these PDF operators in the content stream(s). The functions are called with two arguments: the scanner object itself, and the info table that was passed are the third argument to pdfscanner.scan.

Internally, pdfscanner.scan loops over the PDF operators in the stream(s), collecting operands on an internal stack until it finds a PDF operator. If that PDF operator's name exists in operatortable, then the associated function is executed. After the function has run (or when there is no function to execute) the internal operand stack is cleared in preparation for the next operator, and processing continues.

The scanner argument to the processing functions is needed because it offers various methods to get the actual operands from the internal operand stack. The most important of those functions is

A simple example of processing a PDF's document stream could look like this:

```lua
function Do (scanner, info)
    local val = scanner:pop()
    print (info.space ..'Use XObject '.. name)
    local resources = info.resources
    local xobject = resources:lookup("XObject"):getDict():lookup(name)
    if (xobject and xobject:isStream()) then
        local dict = xobject:getStream():getDict()
        if dict then
            local name = dict:lookup('Subtype')
            if name:getName() == 'Form' then
                local newinfo = { space = info.space .. " " };
```
resources = dict:lookup('Resources'):getDict() }

pdfscanner.scan(xobject, operatortable, newinfo)
end
end
end
operatortable = {Do = Do}

doc = epdf.open(arg[1])
pagenum = 1
while pagenum <= doc:getNumPages() do
  local page = doc:getCatalog():getPage(pagenum)
  local info = { space = " ", resources = page:getResourceDict()}
  print ('Page ' .. pagenum)
  pdfscanner.scan(page:getContents(), operatortable, info)
  pagenum = pagenum + 1
end

This example iterates over all the actual content in the PDF, and prints out the found XObject names. While the code demonstrates quite some of the epdf functions, let's focus on the type pdfscanner specific code instead.

From the bottom up, the line

pdfscanner.scan(page:getContents(), operatortable, info)

runs the scanner with the PDF page's top-level content.

The third argument, info, contains two entries: space is used to indent the printed output, and resources is needed so that embedded XForms can find their own content.

The second argument, operatortable defines a processing function for a single PDF operator, Do.

The function Do prints the name of the current XObject, and then starts a new scanner for that object's content stream, under the condition that the XObject is in fact a /Form. That nested scanner is called with new info argument with an updated space value so that the indentation of the output nicely nests, and with an new resources field to help the next iteration down to properly process any other, embedded XObjects.

Of course, this is not a very useful example in practise, but for the purpose of demonstrating pdfscanner, it is just long enough. It makes use of only one scanner method: scanner:pop(). That function pops the top operand of the internal stack, and returns a lua table where the object at index one is a string representing the type of the operand, and object two is its value.

The list of possible operand types and associated lua value types is:

integer  <number>
real     <number>
boolean  <boolean>
In case of **integer** or **real**, the value is always a Lua (floating point) number.

In case of **name**, the leading slash is always stripped.

In case of **string**, please bear in mind that PDF actually supports different types of strings (with different encodings) in different parts of the PDF document, so may need to reencode some of the results; **pdfscanner** always outputs the byte stream without reencoding anything. **pdfscanner** does not differentiate between literal strings and hexadecimal strings (the hexadecimal values are decoded), and it treats the stream data for inline images as a string that is the single operand for `EI`.

In case of **array**, the table content is a list of **pop** return values.

In case of **dict**, the table keys are PDF name strings and the values are **pop** return values.

There are few more methods defined that you can ask **scanner**:

- **pop** as explained above
- **popNumber** return only the value of a **real** or **integer**
- **popName** return only the value of a **name**
- **popString** return only the value of a **string**
- **popArray** return only the value of a **array**
- **popDict** return only the value of a **dict**
- **popBool** return only the value of a **boolean**
- **done** abort further processing of this **scan()** call

The **popXXX** are convenience functions, and come in handy when you know the type of the operands beforehand (which you usually do, in PDF). For example, the **Do** function could have used `local name = scanner:popName()` instead, because the single operand to the **Do** operator is always a PDF name object.

The **done** function allows you to abort processing of a stream once you have learned everything you want to learn. This comes in handy while parsing `/ToUnicode`, because there usually is trailing garbage that you are not interested in. Without **done**, processing only end at the end of the stream, possibly wasting CPU cycles.

### 4.13 The status library

This contains a number of run-time configuration items that you may find useful in message reporting, as well as an iterator function that gets all of the names and values as a table.

```<table> info = status.list()```
The keys in the table are the known items, the value is the current value. Almost all of the values in `status` are fetched through a metatable at run-time whenever they are accessed, so you cannot use `pairs` on `status`, but you can use `pairs` on `info`, of course. If you do not need the full list, you can also ask for a single item by using its name as an index into `status`.

The current list is:

<table>
<thead>
<tr>
<th>key</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdf_gone</td>
<td>written PDF bytes</td>
</tr>
<tr>
<td>pdf_ptr</td>
<td>not yet written PDF bytes</td>
</tr>
<tr>
<td>dvi_gone</td>
<td>written DVI bytes</td>
</tr>
<tr>
<td>dvi_ptr</td>
<td>not yet written DVI bytes</td>
</tr>
<tr>
<td>total_pages</td>
<td>number of written pages</td>
</tr>
<tr>
<td>output_file_name</td>
<td>name of the PDF or DVI file</td>
</tr>
<tr>
<td>log_name</td>
<td>name of the log file</td>
</tr>
<tr>
<td>banner</td>
<td>terminal display banner</td>
</tr>
<tr>
<td>var_used</td>
<td>variable (one-word) memory in use</td>
</tr>
<tr>
<td>dyn_used</td>
<td>token (multi-word) memory in use</td>
</tr>
<tr>
<td>str_ptr</td>
<td>number of strings</td>
</tr>
<tr>
<td>init_str_ptr</td>
<td>number of INITEX strings</td>
</tr>
<tr>
<td>max_strings</td>
<td>maximum allowed strings</td>
</tr>
<tr>
<td>pool_ptr</td>
<td>string pool index</td>
</tr>
<tr>
<td>init_pool_ptr</td>
<td>INITEX string pool index</td>
</tr>
<tr>
<td>pool_size</td>
<td>current size allocated for string characters</td>
</tr>
<tr>
<td>node_mem_usage</td>
<td>a string giving insight into currently used nodes</td>
</tr>
<tr>
<td>var_mem_max</td>
<td>number of allocated words for nodes</td>
</tr>
<tr>
<td>fix_mem_max</td>
<td>number of allocated words for tokens</td>
</tr>
<tr>
<td>fix_mem_end</td>
<td>maximum number of used tokens</td>
</tr>
<tr>
<td>cs_count</td>
<td>number of control sequences</td>
</tr>
<tr>
<td>hash_size</td>
<td>size of hash</td>
</tr>
<tr>
<td>hash_extra</td>
<td>extra allowed hash</td>
</tr>
<tr>
<td>font_ptr</td>
<td>number of active fonts</td>
</tr>
<tr>
<td>max_in_stack</td>
<td>max used input stack entries</td>
</tr>
<tr>
<td>max_nest_stack</td>
<td>max used nesting stack entries</td>
</tr>
<tr>
<td>max_param_stack</td>
<td>max used parameter stack entries</td>
</tr>
<tr>
<td>max_buf_stack</td>
<td>max used buffer position</td>
</tr>
<tr>
<td>max_save_stack</td>
<td>max used save stack entries</td>
</tr>
<tr>
<td>stack_size</td>
<td>input stack size</td>
</tr>
<tr>
<td>nest_size</td>
<td>nesting stack size</td>
</tr>
<tr>
<td>param_size</td>
<td>parameter stack size</td>
</tr>
<tr>
<td>buf_size</td>
<td>current allocated size of the line buffer</td>
</tr>
<tr>
<td>save_size</td>
<td>save stack size</td>
</tr>
<tr>
<td>obj_ptr</td>
<td>max PDF object pointer</td>
</tr>
<tr>
<td>obj_tab_size</td>
<td>PDF object table size</td>
</tr>
<tr>
<td>pdf_os_cntr</td>
<td>max PDF object stream pointer</td>
</tr>
</tbody>
</table>
pdf_os_objidx     PDF object stream index
pdf_dest_names_ptr max PDF destination pointer
dest_names_size   PDF destination table size
pdf_mem_ptr       max PDF memory used
pdf_mem_size      PDF memory size
largest_used_mark max referenced marks class
filename          name of the current input file
inputid           numeric id of the current input
linenumber        location in the current input file
lasterrorstring   last error string
luabytecodes      number of active LUA bytecode registers
luabytecode_bytes number of bytes in LUA bytecode registers
luastate_bytes    number of bytes in use by LUA interpreters
output_active     true if the \output routine is active
callbacks         total number of executed callbacks so far
indirect_callbacks number of those that were themselves a result of other callbacks (e.g. file readers)
luatex_svn         the luatex repository id (added in 0.51)
luatex_version     the luatex version number (added in 0.38)
luatex_revision    the luatex revision string (added in 0.38)
ini_version        true if this is an \ini\TeX run (added in 0.38)

4.14 The tex library

The tex table contains a large list of virtual internal \TeX parameters that are partially writable.

The designation ‘virtual’ means that these items are not properly defined in Lua, but are only frontends
that are handled by a metatable that operates on the actual \TeX values. As a result, most of the LUA
table operators (like \pairs and \#) do not work on such items.

At the moment, it is possible to access almost every parameter that has these characteristics:

- You can use it after \the
- It is a single token.
- Some special others, see the list below

This excludes parameters that need extra arguments, like \the\scriptfont.

The subset comprising simple integer and dimension registers are writable as well as readable (stuff like
\tracingcommands and \parindent).

4.14.1 Internal parameter values

For all the parameters in this section, it is possible to access them directly using their names as index
in the tex table, or by using one of the functions tex.get() and tex.set().
The exact parameters and return values differ depending on the actual parameter, and so does whether `tex.set` has any effect. For the parameters that can be set, it is possible to use `global` as the first argument to `tex.set`; this makes the assignment global instead of local.

```lua
tex.set (<string> n, ...)  
tex.set ('global', <string> n, ...)  
... = tex.get (<string> n)
```

### 4.14.1.1 Integer parameters

The integer parameters accept and return Lua numbers.

Read-write:

- `tex.adjdemerits`
- `tex.binoppenalty`
- `tex.brokenpenalty`
- `tex.catcodetable`
- `tex.clubpenalty`
- `tex.day`
- `tex.defaulthyphenchar`
- `tex.defaultskewchar`
- `tex.delimiterfactor`
- `tex.displaywidowpenalty`
- `tex.doublehyphendememerits`
- `tex.endlinechar`
- `tex.errorcontextlines`
- `tex.escapechar`
- `tex.exhyphenpenalty`
- `tex.fam`
- `tex.finalhyphendememerits`
- `tex.floatingpenalty`
- `tex.globaldefs`
- `tex.hangafter`
- `tex.hbadness`
- `tex.holdinginserts`
- `tex.hyphenpenalty`
- `tex.interlinepenalty`
- `tex.language`
- `tex.lastlinefit`
- `tex.lefthyphenmin`
- `tex.linepenalty`
- `tex.localbrokenpenalty`
- `tex.localinterlinepenalty`
- `tex.looseness`
- `tex.mag`
- `tex.maxdeadcycles`
- `tex.month`
- `tex.newlinechar`
- `tex.outputpenalty`
- `tex.pausing`
- `tex.pdfadjustspacing`
- `tex.pdfcompresslevel`
- `tex.pdfdecimaldigits`
- `tex.pdfgamma`
- `tex.pdfgentounicode`
- `tex.pdfimageapplygamma`
- `tex.pdfimagegamma`
- `tex.pdfimageresolution`
- `tex.pdfinclusionerrorlevel`
- `tex.pdfminorversion`
- `tex.pdfobjcompresslevel`
- `tex.pdfoutput`
- `tex.pdfpagebox`
- `tex.pdfpkresolution`
- `tex.pdfprotrudechars`
- `tex.pdftracingfonts`
- `tex.pdfuniqueresname`
- `tex.postdisplaypenalty`
- `tex.predisplaydirection`
- `tex.predisplaypenalty`
- `tex.pretolerance`
- `tex.relpenalty`
- `tex.righthyphenmin`
- `tex.savinghyphcodes`
4.14.1.2 Dimension parameters

The dimension parameters accept Lua numbers (signifying scaled points) or strings (with included dimension). The result is always a number in scaled points.

Read-write:

tex.boxmaxdepth  tex.displayindent  tex.displaywidth  tex.emergencystretch  tex.hangindent  tex.hsize  tex.linestukiplot  tex.mathsurround  tex.maxdepth  tex.nulldelimiter-

Read-only:

tex.pagedepth  tex.pagefillstrch  tex.pdfhorigin  tex.pagebottomoffset  tex.pageheight  tex.pagerightoffset  tex.pagetopoffset  tex.pagenumber  tex.parindent  tex.pdfdestmargin  tex.pdfpageheight  tex.pdfpagewidth  tex.pdfpxdimen  tex.pdfthreadmargin  tex.predisplaysize  tex.pdforign  tex.pdforigin  tex.pdfvoffset  tex.vfuzz  tex.vsize

4.14.1.2 Dimension parameters

The dimension parameters accept Lua numbers (signifying scaled points) or strings (with included dimension). The result is always a number in scaled points.

Read-write:

tex.boxmaxdepth  tex.displayindent  tex.displaywidth  tex.emergencystretch  tex.hangindent  tex.hsize  tex.linestukiplot  tex.mathsurround  tex.maxdepth  tex.nulldelimiter-

Read-only:

tex.pagedepth  tex.pagefillstrch  tex.pdfhorigin  tex.pagebottomoffset  tex.pageheight  tex.pagerightoffset  tex.pagetopoffset  tex.pagenumber  tex.parindent  tex.pdfdestmargin  tex.pdfpageheight  tex.pdfpagewidth  tex.pdfpxdimen  tex.pdfthreadmargin  tex.predisplaysize  tex.pdforign  tex.pdforigin  tex.pdfvoffset  tex.vfuzz  tex.vsize
4.14.1.3 Direction parameters

The direction parameters are read-only and return a Lua string.

\begin{verbatim}
tex.bodydir tex.pagedir tex.textdir
tex.mathdir tex.pardir
\end{verbatim}

4.14.1.4 Glue parameters

The glue parameters accept and return a userdata object that represents a glue_spec node.

\begin{verbatim}
tex.abovedisplayshortskip tex.abovedisplayskip tex.baselineskip
tex.belowdisplayshortskip tex.belowdisplayskip tex.leftskip
tex.baselineskip tex.lineskip tex.parfillskip
tex.belowdisplayshortskip tex.parskip tex.rightskip
\end{verbatim}

4.14.1.5 Muglue parameters

All muglue parameters are to be used read-only and return a Lua string.

\begin{verbatim}
tex.medmuskip tex.thickmuskip tex.thinmuskip
\end{verbatim}

4.14.1.6 Tokenlist parameters

The tokenlist parameters accept and return Lua strings. Lua strings are converted to and from token lists using \texttt{\the\toks} style expansion: all category codes are either space (10) or other (12). It follows that assigning to some of these, like ‘tex.output’, is actually useless, but it feels bad to make exceptions in view of a coming extension that will accept full-blown token strings.

\begin{verbatim}
tex.errhelp tex.everyjob tex.pdfpageattr
tex.everycr tex.everymath tex.pdfpageresources
tex.everydisplay tex.everypar tex.pdfpagesattr
tex.everyeof tex.everyvbox tex.pdfpkmode
tex.everyhbox tex.output
\end{verbatim}
4.14.2 Convert commands

All ‘convert’ commands are read-only and return a Lua string. The supported commands at this moment are:

- `tex.eTeXVersion`
- `tex.eTeXrevision`
- `tex.formatname`
- `tex.jobname`
- `tex.luatemxbanner`
- `tex.luatexrevision`
- `tex.pdfnormaldeviate`
- `tex.fontname(number)`
- `tex.pdffontname(number)`
- `tex.pdffontobjnum(number)`
- `tex.pdffontsize(number)`
- `tex.uniformdeviate(number)`
- `tex.number(number)`
- `tex.romannumeral(number)`
- `tex.pdfpageref(number)`
- `tex.pdfxformname(number)`
- `tex.fontidentifier(number)`

If you are wondering why this list looks haphazard; these are all the cases of the ‘convert’ internal command that do not require an argument, as well as the ones that require only a simple numeric value.

The special (lua-only) case of `tex.fontidentifier` returns the `csname` string that matches a font id number (if there is one).

The pdFTeX related primitives will be removed from versions after 0.80. All the version related details are available in the `status` table at the Lua end, for instance one can say:

\begin{verbatim}
\let\pdftexbanner \luatemxbanner
\let\pdftexversion \luatemxversion
\let\pdftexrevision\luatemxrevision
\end{verbatim}

if these are really needed in a macro package.

4.14.3 Last item commands

All ‘last item’ commands are read-only and return a number.

The supported commands at this moment are:

- `tex.lastpenalty`
- `tex.lastkern`
- `tex.lastskip`
- `tex.lastnodetype`
- `tex.inputlineno`
- `tex.pdflastobj`
- `tex.pdflastxform`
- `tex.pdflastimage`
- `tex.pdflastimagepages`
- `tex.pdflastannot`
- `tex.pdflastxpos`
- `tex.pdflastypostscript`
- `tex.pdfrandomseed`
- `tex.pdflastlink`
- `tex.luatemxversion`
- `tex.eTeXversion`
- `tex.eTeXminorversion`
- `tex.currentgrouplevel`
- `tex.currentgrouptype`
- `tex.currentiflevel`
- `tex.currentiftype`
- `tex.currentifbranch`
- `tex.pdflastximagecolordepth`
4.14.4 Attribute, count, dimension, skip and token registers

\TeX{}'s attributes (\texttt{\\textbackslash attribute}), counters (\texttt{\\textbackslash count}), dimensions (\texttt{\\textbackslash dimen}), skips (\texttt{\\textbackslash skip}) and token (\texttt{\\textbackslash toks}) registers can be accessed and written to using two times five virtual sub-tables of the \texttt{tex} table:

\texttt{tex.attribute} \hspace{1cm} \texttt{tex.dimen} \hspace{1cm} \texttt{tex.toks}
\texttt{tex.count} \hspace{1cm} \texttt{tex.skip}

It is possible to use the names of relevant \texttt{\\textbackslash attributedef}, \texttt{\\textbackslash countdef}, \texttt{\\textbackslash dimendef}, \texttt{\\textbackslash skipdef}, or \texttt{\\textbackslash toksdef} control sequences as indices to these tables:

\begin{verbatim}
  tex.count.scratchcounter = 0
  enormous = tex.dimen['maxdimen']
\end{verbatim}

In this case, \texttt{Lua\TeX{}} looks up the value for you on the fly. You have to use a valid \texttt{\\textbackslash countdef} (or \texttt{\\textbackslash attributedef}, or \texttt{\\textbackslash dimendef}, or \texttt{\\textbackslash skipdef}, or \texttt{\\textbackslash toksdef}), anything else will generate an error (the intent is to eventually also allow \texttt{\\textbackslash chardef tokens} and even macros that expand into a number).

The attribute and count registers accept and return \texttt{Lua} numbers.

The dimension registers accept \texttt{Lua} numbers (in scaled points) or strings (with an included absolute dimension; \texttt{em} and \texttt{ex} and \texttt{px} are forbidden). The result is always a number in scaled points.

The token registers accept and return \texttt{Lua} strings. \texttt{Lua} strings are converted to and from token lists using \texttt{\\textbackslash the\\textbackslash toks} style expansion: all category codes are either space (10) or other (12).

The skip registers accept and return \texttt{glue_spec} userdata node objects (see the description of the node interface elsewhere in this manual).

As an alternative to array addressing, there are also accessor functions defined for all cases, for example, here is the set of possibilities for \texttt{\\textbackslash skip} registers:

\begin{verbatim}
  tex.setskip (<number> n, <node> s)
  tex.setskip (<string> s, <node> s)
  tex.setskip ('global',<number> n, <node> s)
  tex.setskip ('global',<string> s, <node> s)
  <node> s = tex.getskip (<number> n)
  <node> s = tex.getskip (<string> s)
\end{verbatim}

In the function-based interface, it is possible to define values globally by using the string 'global' as the first function argument.

4.14.5 Character code registers (0.63)

\TeX{}'s character code tables (\texttt{\\textbackslash lccode}, \texttt{\\textbackslash uccode}, \texttt{\\textbackslash sfcode}, \texttt{\\textbackslash catcode}, \texttt{\\textbackslash mathcode}, \texttt{\\textbackslash delcode}) can be accessed and written to using six virtual sub-tables of the \texttt{tex} table.

\texttt{Lua\TeX{}} Lua Libraries 121
The function call interfaces are roughly as above, but there are a few twists. \texttt{sfcodes} are the simple ones:

\begin{verbatim}
tex.setsfcode (<number> n, <number> s)
tex.setsfcode ('global', <number> n, <number> s)
<number> s = tex.getsfcode (<number> n)
\end{verbatim}

The function call interface for \texttt{lccode} and \texttt{uccode} additionally allows you to set the associated sibling at the same time:

\begin{verbatim}
tex.setlccode (["global"], <number> n, <number> lc)
tex.setlccode (["global"], <number> n, <number> lc, <number> uc)
<number> lc = tex.getlccode (<number> n)
tex.setuccode (["global"], <number> n, <number> uc)
tex.setuccode (["global"], <number> n, <number> uc, <number> lc)
<number> uc = tex.getuccode (<number> n)
\end{verbatim}

The function call interface for \texttt{catcode} also allows you to specify a category table to use on assignment or on query (default in both cases is the current one):

\begin{verbatim}
tex.setcatcode (["global"], <number> n, <number> c)
tex.setcatcode (["global"], <number> cattable, <number> n, <number> c)
<number> lc = tex.getcatcode (<number> n)
<number> lc = tex.getcatcode (<number> cattable, <number> n)
\end{verbatim}

The interfaces for \texttt{delcode} and \texttt{mathcode} use small array tables to set and retrieve values:

\begin{verbatim}
tex.setmathcode (["global"], <number> n, <table> mval)
<table> mval = tex.getmathcode (<number> n)
tex.setdelcode (["global"], <number> n, <table> dval)
<table> dval = tex.getdelcode (<number> n)
\end{verbatim}

Where the table for \texttt{mathcode} is an array of 3 numbers, like this:

\{<number> mathclass, <number> family, <number> character\}

And the table for \texttt{delcode} is an array with 4 numbers, like this:

\{<number> small_fam, <number> small_char, <number> large_fam, <number> large_char\}

Normally, the third and fourth values in a delimiter code assignment will be zero according to \texttt{\Udelcode} usage, but the returned table can have values there (if the delimiter code was set using \texttt{\delcode}, for example). Unset \texttt{delcode}‘s can be recognized because \texttt{dval[1]} is $-1$. 

\textbf{LuaTEX Lua Libraries}

122
4.14.6 Box registers

It is possible to set and query actual boxes, using the node interface as defined in the node library:

\texttt{tex.box}

for array access, or

\begin{verbatim}
tex.setbox(<number> n, <node> s)
tex.setbox(<string> cs, <node> s)
tex.setbox('global', <number> n, <node> s)
tex.setbox('global', <string> cs, <node> s)
<node> n = tex.getbox(<number> n)
<node> n = tex.getbox(<string> cs)
\end{verbatim}

for function-based access. In the function-based interface, it is possible to define values globally by using the string 'global' as the first function argument.

Be warned that an assignment like

\begin{verbatim}
tex.box[0] = tex.box[2]
\end{verbatim}

does not copy the node list, it just duplicates a node pointer. If \texttt{\box2} will be cleared by \TeX{} commands later on, the contents of \texttt{\box0} becomes invalid as well. To prevent this from happening, always use \texttt{node.copy_list()} unless you are assigning to a temporary variable:

\begin{verbatim}
tex.box[0] = node.copy_list(tex.box[2])
\end{verbatim}

4.14.7 Math parameters

It is possible to set and query the internal math parameters using:

\begin{verbatim}
tex.setmath(<string> n, <string> t, <number> n)
tex.setmath('global', <string> n, <string> t, <number> n)
<number> n = tex.getmath(<string> n, <string> t)
\end{verbatim}

As before an optional first parameter 'global' indicates a global assignment.

The first string is the parameter name minus the leading 'Umath', and the second string is the style name minus the trailing 'style'.

Just to be complete, the values for the math parameter name are:

\begin{verbatim}
quad       axis          operatorsize
overbarkern overbarrule overbarvgap
underbarkern underbarrule underbarvgap
radicalkern radicalrule radicalvgap
radicaldegreebefore radicaldegreeafter radicaldegreeraise
stackvgap stacknumup stackdenomdown
\end{verbatim}
The values for the style parameter name are:

<table>
<thead>
<tr>
<th>field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display</td>
<td>crampeddisplay</td>
</tr>
<tr>
<td>text</td>
<td>crampedtext</td>
</tr>
<tr>
<td>script</td>
<td>crampedscript</td>
</tr>
<tr>
<td>scriptscript</td>
<td>crampedscriptscript</td>
</tr>
</tbody>
</table>

### 4.14.8 Special list heads

The virtual table `tex.lists` contains the set of internal registers that keep track of building page lists.

<table>
<thead>
<tr>
<th>field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>page_ins_head</td>
<td>circular list of pending insertions</td>
</tr>
<tr>
<td>contrib_head</td>
<td>the recent contributions</td>
</tr>
<tr>
<td>page_head</td>
<td>the current page content</td>
</tr>
<tr>
<td>hold_head</td>
<td>used for held-over items for next page</td>
</tr>
<tr>
<td>adjust_head</td>
<td>head of the current \adjust list</td>
</tr>
<tr>
<td>pre_adjust_head</td>
<td>head of the current \adjust pre list</td>
</tr>
</tbody>
</table>
4.14.9 Semantic nest levels (0.51)

The virtual table \texttt{tex.nest} contains the currently active semantic nesting state. It has two main parts: a zero-based array of userdata for the semantic nest itself, and the numerical value \texttt{tex.nest.ptr}, which gives the highest available index. Neither the array items in \texttt{tex.nest[]} nor \texttt{tex.nest.ptr} can be assigned to (as this would confuse the typesetting engine beyond repair), but you can assign to the individual values inside the array items, e.g. \texttt{tex.nest[tex.nest.ptr].prevdepth}.

\texttt{tex.nest[tex.nest.ptr]} is the current nest state, \texttt{tex.nest[0]} the outermost (main vertical list) level.

The known fields are:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>modes</th>
<th>explanation</th>
</tr>
</thead>
</table>
| mode      | number   | all   | The current mode. This is a number representing the main mode at this level:  
|           |          |       | 0 == no mode (this happens during \texttt{\write})  
|           |          |       | 1 == vertical,  
|           |          |       | 127 = horizontal,  
|           |          |       | 253 = display math.  
|           |          |       | −1 == internal vertical,  
|           |          |       | −127 = restricted horizontal,  
|           |          |       | −253 = inline math.  
| modeline  | number   | all   | source input line where this mode was entered in, negative inside the output routine.  
| head      | node     | all   | the head of the current list  
| tail      | node     | all   | the tail of the current list  
| prevgraf  | number   | vmode | number of lines in the previous paragraph  
| prevdepth | number   | vmode | depth of the previous paragraph (equal to \texttt{\pdfignoreddimen} when it is to be ignored)  
| spacefactor | number | hmode | the current space factor  
| dirs      | node     | hmode | used for temporary storage by the line break algorithm  
| noad      | node     | mmode | used for temporary storage of a pending fraction numerator, for \texttt{\over} etc.  
| delimptr  | node     | mmode | used for temporary storage of the previous math delimiter, for \texttt{\middle}.  
| mathdir   | boolean  | mmode | true when during math processing the \texttt{\mathdir} is not the same as the surrounding \texttt{\textdir}  
| mathstyle | number   | mmode | the current \texttt{\mathstyle}  

4.14.10 Print functions

The \texttt{tex} table also contains the three print functions that are the major interface from \texttt{LUA} scripting to \TeX.
The arguments to these three functions are all stored in an in-memory virtual file that is fed to the \TeX scanner as the result of the expansion of \texttt{\directlua}.

The total amount of returnable text from a \texttt{\directlua} command is only limited by available system RAM. However, each separate printed string has to fit completely in \TeX's input buffer.

The result of using these functions from inside callbacks is undefined at the moment.

### 4.14.10.1 \texttt{tex.print}

\begin{verbatim}
\texttt{tex.print(<string> s, ...)}
\texttt{tex.print(<number> n, <string> s, ...)}
\texttt{tex.print(<table> t)}
\texttt{tex.print(<number> n, <table> t)}
\end{verbatim}

Each string argument is treated by \TeX as a separate input line. If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process). This syntax was added in 0.36.

The optional parameter can be used to print the strings using the catcode regime defined by \texttt{\catcodetable n}. If \texttt{n} is $-1$, the currently active catcode regime is used. If \texttt{n} is $-2$, the resulting catcodes are the result of \texttt{\the\toks}: all category codes are 12 (other) except for the space character, that has category code 10 (space). Otherwise, if \texttt{n} is not a valid catcode table, then it is ignored, and the currently active catcode regime is used instead.

The very last string of the very last \texttt{tex.print()} command in a \texttt{\directlua} will not have the \texttt{\endlinechar} appended, all others do.

### 4.14.10.2 \texttt{tex.sprint}

\begin{verbatim}
\texttt{tex.sprint(<string> s, ...)}
\texttt{tex.sprint(<number> n, <string> s, ...)}
\texttt{tex.sprint(<table> t)}
\texttt{tex.sprint(<number> n, <table> t)}
\end{verbatim}

Each string argument is treated by \TeX as a special kind of input line that makes it suitable for use as a partial line input mechanism:

- \TeX does not switch to the 'new line' state, so that leading spaces are not ignored.
- No \texttt{\endlinechar} is inserted.
- Trailing spaces are not removed.

Note that this does not prevent \TeX itself from eating spaces as result of interpreting the line. For example, in

\begin{verbatim}
before\directlua{tex.sprint("\relax")tex.sprint(" inbetween")}after
\end{verbatim}

the space before \texttt{inbetween} will be gobbled as a result of the 'normal' scanning of \texttt{\relax}.  

126 \LaTeX\luatex\ lua\ libraries
If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process). This syntax was added in 0.36.

The optional argument sets the catcode regime, as with `tex.print()`.

### 4.14.10.3 tex.tprint

```lua
text.tprint({<number> n, <string> s, ...}, {...})
```

This function is basically a shortcut for repeated calls to `tex.sprint(<number> n, <string> s, ...), once for each of the supplied argument tables.

### 4.14.10.4 tex.write

```lua
tex.write(<string> s, ...)
tex.write(<table> t)
```

Each string argument is treated by \TeX as a special kind of input line that makes it suitable for use as a quick way to dump information:

- All catcodes on that line are either ‘space’ (for ‘ ’) or ‘character’ (for all others).
- There is no \texttt{\endlinechar} appended.

If there is a table argument instead of a list of strings, this has to be a consecutive array of strings to print (the first non-string value will stop the printing process). This syntax was added in 0.36.

### 4.14.11 Helper functions

#### 4.14.11.1 tex.round

```lua
<number> n = tex.round(<number> o)
```

Rounds Lua number \( o \), and returns a number that is in the range of a valid \TeX register value. If the number starts out of range, it generates a ‘number to big’ error as well.

#### 4.14.11.2 tex.scale

```lua
<number> n = tex.scale(<number> o, <number> delta)
<table> n = tex.scale(table o, <number> delta)
```

Multiplies the Lua numbers \( o \) and \( \text{delta} \), and returns a rounded number that is in the range of a valid \TeX register value. In the table version, it creates a copy of the table with all numeric top–level values scaled in that manner. If the multiplied number(s) are of range, it generates ‘number to big’ error(s) as well.
Note: the precision of the output of this function will depend on your computer’s architecture and operating system, so use with care! An interface to LuaTeX’s internal, 100% portable scale function will be added at a later date.

4.14.11.3 tex.sp (0.51)

<number> n = tex.sp(<number> o)
<number> n = tex.sp(<string> s)

Converts the number o or a string s that represents an explicit dimension into an integer number of scaled points.

For parsing the string, the same scanning and conversion rules are used that LuaTeX would use if it was scanning a dimension specifier in its TeX-like input language (this includes generating errors for bad values), expect for the following:

1. only explicit values are allowed, control sequences are not handled
2. infinite dimension units (fil...) are forbidden
3. mu units do not generate an error (but may not be useful either)

4.14.11.4 tex.definefont

tex.definefont(<string> csname, <number> fontid)
<function> tex.definefont(booleand global, <string> csname, <number> fontid)

Associates csname with the internal font number fontid. The definition is global if (and only if) global is specified and true (the setting of globaldefs is not taken into account).

4.14.11.5 tex.error (0.61)

tex.error(<string> s)
tex.error(<string> s, <table> help)

This creates an error somewhat like the combination of \errhelp and \errmessage would. During this error, deletions are disabled.

The array part of the help table has to contain strings, one for each line of error help.

4.14.11.6 tex.hashtokens (0.25)

for i,v in pairs (tex.hashtokens()) do ... end

Returns a name and token table pair (see section 4.17 about token tables) iterator for every non-zero entry in the hash table. This can be useful for debugging, but note that this also reports control sequences that may be unreachable at this moment due to local redefinitions: it is strictly a dump of the hash table.
4.14.12 Functions for dealing with primitives

4.14.12.1 tex.enableprimitives

tex.enableprimitives(<string> prefix, <table> primitive names)

This function accepts a prefix string and an array of primitive names.

For each combination of 'prefix' and 'name', the tex.enableprimitives first verifies that 'name' is an actual primitive (it must be returned by one of the tex.extraprimitives() calls explained below, or part of TeX82, or \directlua). If it is not, tex.enableprimitives does nothing and skips to the next pair.

But if it is, then it will construct a csname variable by concatenating the 'prefix' and 'name', unless the 'prefix' is already the actual prefix of 'name'. In the latter case, it will discard the 'prefix', and just use 'name'.

Then it will check for the existence of the constructed csname. If the csname is currently undefined (note: that is not the same as \relax), it will globally define the csname to have the meaning: run code belonging to the primitive 'name'. If for some reason the csname is already defined, it does nothing and tries the next pair.

An example:

tex.enableprimitives('LuaTeX', {'formatname'})

will define \LuaTeXformatname with the same intrinsic meaning as the documented primitive \formatname, provided that the control sequences \LuaTeXformatname is currently undefined.

Second example:

tex.enableprimitives('Omega',tex.extraprimitives ('omega'))

will define a whole series of csnames like \Omegatextdir, \Omegapardir, etc., but it will stick with \OmegaVersion instead of creating the doubly-prefixed \OmegaOmegaVersion.

Starting with version 0.39.0 (and this is why the above two functions are needed), Lua\TeX{} in --ini mode contains only the TeX82 primitives and \directlua, no extra primitives at all.

So, if you want to have all the new functionality available using their default names, as it is now, you will have to add

\ifx\directlua\undefined \else
  \directlua {tex.enableprimitives('',tex.extraprimitives ())}
\fi

near the beginning of your format generation file. Or you can choose different prefixes for different subsets, as you see fit.
Calling some form of `tex.enableprimitives()` is highly important though, because if you do not, you will end up with a \TeX\-lookalike that can run Lua code but not do much else. The defined csnames are (of course) saved in the format and will be available at runtime.

### 4.14.12.2 `tex.extraprimitives`

```latex
\begin{table}
t = tex.extraprimitives(<string> s, ...)
\end{table}
```

This function returns a list of the primitives that originate from the engine(s) given by the requested string value(s). The possible values and their (current) return values are:

<table>
<thead>
<tr>
<th>name</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tex</code></td>
<td>- / above abovedisplayshortskip abovedisplayskip abovewithdelims accent adjdemerits advance afterassignment aftergroup at otpwithdelims badness baselineskip batchmode begingroup belowskipbelowdisplayshortskip belowdisplayskip binoppenalty botmark boxboxmaxdepth brokenpenalty catcode char chardef closein closeout clubpenalty copy count countdef cr crr csname day deadcycles def defaulthyphenchar defaulthyphenchar delcode delimeter delimitershortfall dimen dimendef discretionary displayindent displaylimits displaystyle displaywidowpenalty displaywidth divide doublehyphenemergencies dp dump edef else emergencystretch end endcsname endgroup endinput endline char eqno errhelp ermessage errorcontextlines errorstopmode escapechar everycr everydisplay everygroup everyjob everymath everypar everybox exhyphenexhyphenpenalty exhyphenpenalty expandafter fam fi finalhyphenemergencies firstmark floatingpenalty font fontdimen fontname futurelet gdef global globaldefs halig hangafter hangindent hbadness hbox hfil hfill hfilneg h fuzz hookedinserts hrule hsize hskip hss ht hyphenation hyphenchar hyphenpenalty if ifcase icat ifdim ifdo cfalse ifhbox ifinner ifmathmode innum ifnum iftrue ifnum ifvbox ifvmode ifvoid ifx ignorespaces immediate indent input inputlinenew insert insertpenalties interlinepenalty jobname kern language lastbox lastkern lastpenalty lastskip lcode leaders leftleftlyphenmin leftskip leqno let limits linepenalty lineskip lineskiplimit long looseness lower lowercase margin mathaccent mathbin mathchar mathchardef mathchoice mathclose mathcode mathinner mathop mathopen mathord mathpunct mathrel mathsurround maxdepth meaning medmuskip message middle mkern month move left movextra mskip multiply muskip muskipdef newlinechar noalign noboundary exexpanding noindent nolimits nonscript nonstopmode nulldelimiterspace nullfont number omit openin openout or outer outputpenalty over overfullrule overline overwithdelims pagedepth pagefillstretch pagefillstretch pagefillstretch pagegoal pageshrink pagestretch pagetotal par parfillskip parindent parshape parskip patterns pausing penalty postdisplaypenalty predisplaypenalty predisplaysize pretolerance prevdepth prevgraf radical raise read relax rpenalty right righthyphenmin rightskip romannumeral scriptfont scriptscriptfont scriptscriptstyle scriptspace scriptstyle scrollmode setbox setlanguage sfcode shipout show showbox showboxbreadth showboxdepth showlists showthe skewchar skip skipdef spacefactor spaceskip span special splitbotmark splitfirstmark splitmaxdepth splittopmark string tabskip textfont textstyle the thickmuskip thinmuskip time toks toksdef tolerance topmark topskip tracingcommands</td>
</tr>
</tbody>
</table>
Umathopenpunctspacing Umathopenrelspsacing Umathoperatorsize Umathopinnerspacing
Umathopopenspacing Umathopopspacing Umathopordspacing Umathopppunctspacing
Umathoprelspsacing Umathordbinspsacing Umathordclosespacing Umathordinnnerspacing
Umathordopenspacing Umathordopspacing Umathordordspacing Umathordingpunctspacing
Umathordrelspsacing Umathoverbarkern Umathoverbarrule Umathoverbarvgap
Umathoverdelimiterbgap Umathoverdelimitervgap Umathpunctbinspsacing
Umathpunctclosespacing Umathpunctinnnerspacing Umathpunctopenspacing
Umathpuncttopspacing Umathpunctordspacing Umathpunctpunctspacing Umathpunctrelopspsacing
Umathquad Umathradicaldegreeafter Umathradicaldegreebefore Umathradicaldegreeraise
Umathradicalkern Umathradicalrule Umathradicalvgap Umathrelbinspsacing
Umathrelclosespacing Umathrelinnnerspacing Umathrelopenspacing Umathrelrelopspsacing
Umathrelordspsacing Umathrelospunctspacing Umathrelrelopspsacing Umathspaceafterscript
Umathstackdenomdown Umathstacknumup Umathstackvgap Umathsubshiftdown
Umathsusbspspsacing Umathsusbsuperbottommin Umathsusbsuperbottommax
Umathsubshiftdrop Umathsusbsupshiftdown Umathsusbsupvgap Umathsubtopmax
Umathsubsupbottommin Umathsubsupshiftdown Umathsubsupvgap Umathsubtopmax
Umathunderbarkern Umathunderbarrule Umathunderbarvgap Umathunderdelimiterbgap
Umathunderdelimitervgap Uoverdelimiter Uradical Uroot Ustack Ustartdisplaymath
Ustopmath Ustopequation Ustopmath Usubscript Usuperscript Uunderdelimiter

Note that 'luatex' does not contain directlua, as that is considered to be a core primitive, along
with all the \TeX82 primitives, so it is part of the list that is returned from 'core'.

'umath' is a subset of 'luatex' that covers the Unicode math primitives and have been added in
Lua\TeX\ 0.75.0 as it might be desired to handle the prefixing of that subset differently.

Running tex.extraprimitives() will give you the complete list of primitives that are not defined
at Lua\TeX\ 0.39.0 -ini startup. It is exactly equivalent to tex.extraprimitives('etex', 'pdf-
tex', 'omega', 'aleph', 'luatex')

4.14.12.3 tex.primitives

<table> t = tex.primitives()

This function returns a hash table listing all primitives that \LaTeX\ knows about. The keys in the hash
are primitives names, the values are tables representing tokens (see section 4.17). The third value is
always zero.

4.14.13 Core functionality interfaces

4.14.13.1 tex.badness (0.53)

<number> b = tex.badness(<number> t, <number> s)
This helper function is useful during linebreak calculations. \( t \) and \( s \) are scaled values; the function returns the badness for when total \( t \) is supposed to be made from amounts that sum to \( s \). The returned number is a reasonable approximation of \( 100(t/s)^3 \);

4.14.13.2 \texttt{tex.linebreak} (0.53)

\begin{verbatim}
local <node> nodelist, <table> info =
    tex.linebreak(<node> listhead, <table> parameters)
\end{verbatim}

The understood parameters are as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pardir</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>pretolerance</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>tracingparagraphs</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>tolerance</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>looseness</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hyphenpenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>exhyphenpenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>pdfadjustspacing</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>adjsmerits</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>pdfprotrudechars</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>linepenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lastlinefit</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>doublehyphenpenalties</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>finalhyphenpenalties</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>hangafter</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>interlinepenalty</td>
<td>number or table</td>
<td>if a table, then it is an array like \texttt{\interlinepenalties}</td>
</tr>
<tr>
<td>clubpenalty</td>
<td>number or table</td>
<td>if a table, then it is an array like \texttt{\clubpenalties}</td>
</tr>
<tr>
<td>widowpenalty</td>
<td>number or table</td>
<td>if a table, then it is an array like \texttt{\widowpenalties}</td>
</tr>
<tr>
<td>brokenpenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>emergencystretch</td>
<td>number</td>
<td>in scaled points</td>
</tr>
<tr>
<td>hangindent</td>
<td>number</td>
<td>in scaled points</td>
</tr>
<tr>
<td>hsize</td>
<td>number</td>
<td>in scaled points</td>
</tr>
<tr>
<td>leftskip</td>
<td>glue_spec node</td>
<td></td>
</tr>
<tr>
<td>rightskip</td>
<td>glue_spec node</td>
<td></td>
</tr>
<tr>
<td>pdfeachlineheight</td>
<td>number</td>
<td>in scaled points</td>
</tr>
<tr>
<td>pdfeachlinedepth</td>
<td>number</td>
<td>in scaled points</td>
</tr>
<tr>
<td>pdffirstlineheight</td>
<td>number</td>
<td>in scaled points</td>
</tr>
<tr>
<td>pdflastlinedepth</td>
<td>number</td>
<td>in scaled points</td>
</tr>
<tr>
<td>pdffignoredimensions</td>
<td>number</td>
<td>in scaled points</td>
</tr>
<tr>
<td>parshape</td>
<td>table</td>
<td></td>
</tr>
</tbody>
</table>

Note that there is no interface for \texttt{\displaywidowpenalties}, you have to pass the right choice for \texttt{\widowpenalties} yourself.
The meaning of the various keys should be fairly obvious from the table (the names match the \TeX and pdf\TeX primitives) except for the last 5 entries. The four pdf...line... keys are ignored if their value equals \texttt{pdfforeigndimen}.

It is your own job to make sure that \texttt{listhead} is a proper paragraph list: this function does not add any nodes to it. To be exact, if you want to replace the core line breaking, you may have to do the following (when you are not actually working in the pre\_linebreak\_filter or linebreak\_filter callbacks, or when the original list starting at listhead was generated in horizontal mode):

- add an ‘indent box’ and perhaps a \texttt{local\_par} node at the start (only if you need them)
- replace any found final glue by an infinite penalty (or add such a penalty, if the last node is not a glue)
- add a glue node for the \texttt{\parfillskip} after that penalty node
- make sure all the \texttt{prev} pointers are OK

The result is a node list, it still needs to be vpacked if you want to assign it to a \texttt{\vbox}.

The returned \texttt{info} table contains four values that are all numbers:

- \texttt{prevdepth} depth of the last line in the broken paragraph
- \texttt{prevgraf} number of lines in the broken paragraph
- \texttt{looseness} the actual looseness value in the broken paragraph
- \texttt{demerits} the total demerits of the chosen solution

Note there are a few things you cannot interface using this function: You cannot influence font expansion other than via \texttt{pdfadjustspacing}, because the settings for that take place elsewhere. The same is true for hbadness and hfuzz etc. All these are in the \texttt{hpack()} routine, and that fetches its own variables via globals.

### 4.14.13.3 \texttt{tex\_shipout (0.51)}

\texttt{tex\_shipout(<number> n)}

Ships out box number \texttt{n} to the output file, and clears the box register.

#### 4.15 The \texttt{texconfig} table

This is a table that is created empty. A startup Lua script could fill this table with a number of settings that are read out by the executable after loading and executing the startup file.

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>default</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>kpse_init</td>
<td>boolean</td>
<td>true</td>
<td>false, totally disables kpse_init, and enables interpretation of the following numeric key-value pairs (only ever unset this if you implement all file find callbacks!)</td>
</tr>
</tbody>
</table>
shell_escape string 'f' Use 'y' or 't' or '1' to enable \write 18 unconditionally, 'p' to enable the commands that are listed in shell_escape_commands (new in 0.37)

shell_escape_commands string Comma-separated list of command names that may be executed by \write 18 even if shell_escape is set to 'p'. Do not use spaces around commas, separate any required command arguments by using a space, and use the ASCII double quote (") for any needed argument or path quoting (new in 0.37)

string_vacancies number 75000 cf. web2c docs
pool_free number 5000 cf. web2c docs
max_strings number 15000 cf. web2c docs
strings_free number 100 cf. web2c docs
nest_size number 50 cf. web2c docs
max_in_open number 15 cf. web2c docs
param_size number 60 cf. web2c docs
save_size number 4000 cf. web2c docs
stack_size number 300 cf. web2c docs
dvi_buf_size number 16384 cf. web2c docs
error_line number 79 cf. web2c docs
half_error_line number 50 cf. web2c docs
max_print_line number 79 cf. web2c docs
hash_extra number 0 cf. web2c docs
pk_dpi number 72 cf. web2c docs
trace_file_names boolean true false disables \TeX’s normal file open-close feedback (the assumption is that callbacks will take care of that)

file_line_error boolean false do file:line style error messages
halt_on_error boolean false abort run on the first encountered error
formatname string if no format name was given on the commandline, this key will be tested first instead of simply quitting
jobname string if no input file name was given on the commandline, this key will be tested first instead of simply giving up

Note: the numeric values that match web2c parameters are only used if kpse_init is explicitly set to false. In all other cases, the normal values from texmf.cnf are used.

4.16 The texio library
This library takes care of the low-level I/O interface.

4.16.1 Printing functions
4.16.1.1 texio.write

texio.write(<string> target, <string> s, ...)
texio.write(<string> s, ...)

Without the target argument, writes all given strings to the same location(s) \TeX writes messages to at this moment. If \texttt{\textbackslash batchmode} is in effect, it writes only to the log, otherwise it writes to the log and the terminal. The optional target can be one of three possibilities: \texttt{term}, \texttt{log} or \texttt{term and log}.

Note: If several strings are given, and if the first of these strings is or might be one of the targets above, the target must be specified explicitly to prevent Lua from interpreting the first string as the target.

4.16.1.2 texio.write_nl

texio.write_nl(<string> target, <string> s, ...)
texio.write_nl(<string> s, ...)

This function behaves like \texttt{texio.write}, but make sure that the given strings will appear at the beginning of a new line. You can pass a single empty string if you only want to move to the next line.

4.17 The token library

The \texttt{token} table contains interface functions to \TeX's handling of tokens. These functions are most useful when combined with the \texttt{token_filter} callback, but they could be used standalone as well.

A token is represented in Lua as a small table. For the moment, this table consists of three numeric entries:

<table>
<thead>
<tr>
<th>index</th>
<th>meaning</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>command code</td>
<td>this is a value between 0 and 130 (approximately)</td>
</tr>
<tr>
<td>2</td>
<td>command modifier</td>
<td>this is a value between 0 and 2^{21}</td>
</tr>
<tr>
<td>3</td>
<td>control sequence id</td>
<td>for commands that are not the result of control sequences, like letters and characters, it is zero, otherwise, it is a number pointing into the 'equivalence table'</td>
</tr>
</tbody>
</table>

4.17.1 token.get_next

token t = token.get_next()

This fetches the next input token from the current input source, without expansion.

4.17.2 token.is_expandable

<boolean> b = token.is_expandable(<token> t)
This tests if the token \texttt{t} could be expanded.

\subsection{token.expand}
\begin{verbatim}
token.expand(<token> t)
\end{verbatim}

If a token is expandable, this will expand one level of it, so that the first token of the expansion will now be the next token to be read by \texttt{token.get_next()}. 

\subsection{token.is_activechar}
\begin{verbatim}
<boolean> b = token.is_activechar(<token> t)
\end{verbatim}

This is a special test that is sometimes handy. Discovering whether some control sequence is the result of an active character turned out to be very hard otherwise.

\subsection{token.create}
\begin{verbatim}
token t = token.create(<string> csname)
token t = token.create(<number> charcode)
token t = token.create(<number> charcode, <number> catcode)
\end{verbatim}

This is the token factory. If you feed it a string, then it is the name of a control sequence (without leading backslash), and it will be looked up in the equivalence table. 

If you feed it number, then this is assumed to be an input character, and an optional second number gives its category code. This means it is possible to overrule a character’s category code, with a few exceptions: the category codes 0 (escape), 9 (ignored), 13 (active), 14 (comment), and 15 (invalid) cannot occur inside a token. The values 0, 9, 14 and 15 are therefore illegal as input to \texttt{token.create()}, and active characters will be resolved immediately.

Note: unknown string sequences and never defined active characters will result in a token representing an ‘undefined control sequence’ with a near-random name. It is not possible to define brand new control sequences using \texttt{token.create}!

\subsection{token.command_name}
\begin{verbatim}
<string> commandname = token.command_name(<token> t)
\end{verbatim}

This returns the name associated with the ‘command’ value of the token in \LaTeX{}. There is not always a direct connection between these names and primitives. For instance, all \texttt{\textbackslash ifxxx} tests are grouped under \texttt{if_test}, and the ‘command modifier’ defines which test is to be run.
4.17.7 token.command_id

<number> i = token.command_id(<string> commandname)

This returns a number that is the inverse operation of the previous command, to be used as the first item in a token table.

4.17.8 token.csname_name

<string> csname = token.csname_name(<token> t)

This returns the name associated with the 'equivalence table' value of the token in LuaTeX. It returns the string value of the command used to create the current token, or an empty string if there is no associated control sequence.

Keep in mind that there are potentially two control sequences that return the same csname string: single character control sequences and active characters have the same 'name'.

4.17.9 token.csname_id

<number> i = token.csname_id(<string> csname)

This returns a number that is the inverse operation of the previous command, to be used as the third item in a token table.

4.17.10 The newtoken library

The current token library will be replaced by a new one that is more flexible and powerful. The transition takes place in steps. In version 0.80 we have newtoken and in version 0.85 the old lib will be replaced completely. So if you use this new mechanism in production code you need to be aware of incompatible updates between 0.80 and 0.90. Because the related in- and output code will also be cleaned up and rewritten you should be aware of incompatible logging and error reporting too.

The old library presents tokens as triplets or numbers, the new library presents a userdata object. The old library used a callback to intercept tokens in the input but the new library provides a basic scanner infrastructure that can be used to write macros that accept a wide range of arguments. This interface is on purpose kept general and as performance is quite ok one can build additional parsers without too much overhead. It’s up to macro package writers to see how they can benefit from this as the main principle behind LuaTeX is to provide a minimal set of tools and no solutions.

The current functions in the newtoken namespace are given in the next table:

<table>
<thead>
<tr>
<th>function</th>
<th>argument</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_token</td>
<td>token</td>
<td>checks if the given argument is a token userdata</td>
</tr>
<tr>
<td>get_next</td>
<td></td>
<td>returns the next token in the input</td>
</tr>
</tbody>
</table>
scan_keyword string returns true if the given keyword is gobbled
scan_int returns a number
scan_dimen infinity, mu-units returns a number representing a dimension and or two numbers being the filler and order
scan_glue mu-units returns a glue spec node
scan_toks definer, expand returns a table of tokens token list (this can become a linked list in later releases)
scan_code bitset returns a character if its category is in the given bitset (representing catcodes)
scan_string returns a string given between {}, as \macro or as sequence of characters with catcode 11 or 12
scan_word returns a sequence of characters with catcode 11 or 12 as string
create returns a userdata token object of the given control sequence name (or character); this interface can change

The scanners can be considered stable apart from the one scanning for a token. This is because futures releases can return a linked list instead of a table (as with nodes). The scan_code function takes an optional number, the keyword function a normal Lua string. The infinity boolean signals that we also permit fill as dimension and the mu-units flags the scanner that we expect math units. When scanning tokens we can indicate that we are defining a macro, in which case the result will also provide information about what arguments are expected and in the result this is separated from the meaning by a separator token. The expand flag determines if the list will be expanded.

The string scanner scans for something between curly braces and expands on the way, or when it sees a control sequence it will return its meaning. Otherwise it will scan characters with catcode letter or other. So, given the following definition:

\def\bar{\texttt{bar}}
\def\foo{\texttt{foo-\bar}}

we get:

\directlua{newtoken.scan_string()}{\texttt{foo}} \texttt{foo} full expansion
\directlua{newtoken.scan_string()}{\texttt{foo} \texttt{foo}} letters and others
\directlua{newtoken.scan_string()}{\texttt{foo}} \texttt{foo-bar} meaning

The \texttt{\foo} case only gives the meaning, but one can pass an already expanded definition (\edef\texttt{d}). In the case of the braced variant one can of course use the \texttt{detokenize} and \texttt{unexpanded} primitives as there we do expand.

The scan_word scanner can be used to implement for instance a number scanner:

function newtokens.scan_number(base)
    return tonumber(newtoken.scan_word(),base)
end

This scanner accepts any valid Lua number so it is a way to pick up floats in the input.
The creator function can be used as follows:

```lua
local t = newtoken("relax")
```

This gives back a token object that has the properties of the `\relax` primitive. The possible properties of tokens are:

- `command`: a number representing the internal command number
- `cmdname`: the type of the command (for instance the catcode in case of a character or the classifier that determines the internal treatment)
- `csname`: the associated control sequence (if applicable)
- `id`: the unique id of the token
- `active`: a boolean indicating the active state of the token
- `expandable`: a boolean indicating if the token (macro) is expandable
- `protected`: a boolean indicating if the token (macro) is protected

The numbers that represent a catcode are the same as in TeX itself, so using this information assumes that you know a bit about TeX's internals. The other numbers and names are used consistently but are not frozen. So, when you use them for comparing you can best query a known primitive or character first to see the values.

More interesting are the scanners. You can use the Lua interface as follows:

```latex
\directlua{
    function mymacro(n)
        ...
    end
}
```

```latex
\def\mymacro#1{%
    \directlua{
        mymacro(\number\dimexpr#1)
    }%
}
```

```latex
\mymacro{12pt}
\mymacro{\dimen0}
```

But starting with version 0.80 you can also do this:

```latex
\directlua{
    function mymacro()
        local d = newtoken.scan_dimen()
        ...
    end
}
```
\def\mymacro{%
  \directlua {
    mymacro()
  }
%
}

\mymacro \dimen0
\mymacro 12pt
\mymacro \dimen0

It is quite clear from looking at the code what the first method needs as argument(s). For the second method you need to look at the \texttt{Lua} code to see what gets picked up. Instead of passing from \TeX{} to \texttt{Lua} we let \texttt{Lua} fetch from the input stream.

In the first case the input is tokenized and then turned into a string when it’s passed to \texttt{Lua} where it gets interpreted. In the second case only a function call gets interpreted but then the input is picked up by explicitly calling the scanner functions. These return proper \texttt{Lua} variables so no further conversion has to be done. This is more efficient but in practice (given what \TeX{} has to do) this effect should not be overestimated. For numbers and dimensions it saves a bit but for passing strings conversion to and from tokens has to be done anyway (although we can probably speed up the process in later versions if needed).

When the interface is stable and has replaced the old one completely we will add some more information here. By that time the internals have been cleaned up a bit more so we know then what will stay and go. A positive side effect of this transition is that we can simplify the input part because we no longer need to intercept using callbacks.
5 Math

The handling of mathematics in LuaTEX differs quite a bit from how \TeX\82 (and therefore pdf\TeX\) handles math. First, LuaTEX adds primitives and extends some others so that Unicode input can be used easily. Second, all of \TeX\82’s internal special values (for example for operator spacing) have been made accessible and changeable via control sequences. Third, there are extensions that make it easier to use OpenType math fonts. And finally, there are some extensions that have been proposed in the past that are now added to the engine.

5.1 The current math style

Starting with LuaTEX 0.39.0, it is possible to discover the math style that will be used for a formula in an expandable fashion (while the math list is still being read). To make this possible, LuaTEX adds the new primitive: \texttt{\mathstyle}. This is a ‘convert command’ like e.g. \texttt{\romannumeral}: its value can only be read, not set.

5.1.1 \texttt{\mathstyle}

The returned value is between 0 and 7 (in math mode), or \texttt{-1} (all other modes). For easy testing, the eight math style commands have been altered so that the can be used as numeric values, so you can write code like this:

\begin{verbatim}
\ifnum\mathstyle=\textstyle
  \message{normal text style}
\else \ifnum\mathstyle=\crampedtextstyle
  \message{cramped text style}
\fi \fi
\end{verbatim}

5.1.2 \texttt{\Ustack}

There are a few math commands in \TeX\ where the style that will be used is not known straight from the start. These commands (\texttt{\over}, \texttt{\atop}, \texttt{\overwithdelims}, \texttt{\atopwithdelims}) would therefore normally return wrong values for \texttt{\mathstyle}. To fix this, LuaTEX introduces a special prefix command: \texttt{\Ustack}:

$\Ustack \{a \over b\}$

The \texttt{\Ustack} command will scan the next brace and start a new math group with the correct (numerator) math style.
5.2 Unicode math characters

Character handling is now extended up to the full Unicode range (the \U prefix), which is compatible with Xe\TeX.

The math primitives from \TeX are kept as they are, except for the ones that convert from input to math commands: \texttt{mathcode}, and \texttt{delcode}. These two now allow for a 21-bit character argument on the left hand side of the equals sign.

Some of the new Lua\TeX primitives read more than one separate value. This is shown in the tables below by a plus sign in the second column.

The input for such primitives would look like this:

\begin{verbatim}
def\overbrace {\Umathaccent 0 1 "23DE }
\end{verbatim}

Altered \TeX82 primitives:

<table>
<thead>
<tr>
<th>primitive</th>
<th>value range (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mathcode}</td>
<td>0–10FFFF = 0–8000</td>
</tr>
<tr>
<td>\texttt{delcode}</td>
<td>0–10FFFF = 0–FFFFFF</td>
</tr>
</tbody>
</table>

Unaltered:

<table>
<thead>
<tr>
<th>primitive</th>
<th>value range (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mathchardef}</td>
<td>0–8000</td>
</tr>
<tr>
<td>\texttt{mathchar}</td>
<td>0–7FFF</td>
</tr>
<tr>
<td>\texttt{mathaccent}</td>
<td>0–7FFF</td>
</tr>
<tr>
<td>\texttt{delimiter}</td>
<td>0–7FFFFFFFF</td>
</tr>
<tr>
<td>\texttt{radical}</td>
<td>0–7FFFFFFFF</td>
</tr>
</tbody>
</table>

New primitives that are compatible with Xe\TeX:

<table>
<thead>
<tr>
<th>primitive</th>
<th>value range (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Umathchardef}</td>
<td>0+0+0–7+FF+10FFFF</td>
</tr>
<tr>
<td>\texttt{Umathcharnumdef}</td>
<td>-80000000–7FFFFFF 3</td>
</tr>
<tr>
<td>\texttt{Umathcode}</td>
<td>0–10FFFF = 0+0+0–7+FF+10FFFF 1</td>
</tr>
<tr>
<td>\texttt{Udelcode}</td>
<td>0–10FFFF = 0+0–FF+10FFFF 2</td>
</tr>
<tr>
<td>\texttt{Umathchar}</td>
<td>0+0+0–7+FF+10FFFF</td>
</tr>
<tr>
<td>\texttt{Umathaccent}</td>
<td>0+0+0–7+FF+10FFFF 2.4</td>
</tr>
<tr>
<td>\texttt{Udelimiter}</td>
<td>0+0+0–7+FF+10FFFF 2</td>
</tr>
<tr>
<td>\texttt{Urational}</td>
<td>0+0–FF+10FFFF 2</td>
</tr>
<tr>
<td>\texttt{Umathcharnum}</td>
<td>-80000000–7FFFFFF 3</td>
</tr>
<tr>
<td>\texttt{Umathcodenum}</td>
<td>0–10FFFF = -80000000–7FFFFFF 3</td>
</tr>
<tr>
<td>\texttt{Udelcodenum}</td>
<td>0–10FFFF = -80000000–7FFFFFF 3</td>
</tr>
</tbody>
</table>

Note 1: \texttt{Umathchardef<csname>="8"0"0} and \texttt{Umathchardef<number>="8"0"0} are also accepted.
Note 2: The new primitives that deal with delimiter-style objects do not set up a 'large family'. Selecting a suitable size for display purposes is expected to be dealt with by the \Umathoperatorsize parameter (more information in a following section).

Note 3: For these three primitives, all information is packed into a single signed integer. For the first two (\Umathcharnum and \Umathcodenum), the lowest 21 bits are the character code, the 3 bits above that represent the math class, and the family data is kept in the topmost bits (This means that the values for math families 128–255 are actually negative). For \Udelcodenum there is no math class; the math family information is stored in the bits directly on top of the character code. Using these three commands is not as natural as using the two- and three-value commands, so unless you know exactly what you are doing and absolutely require the speedup resulting from the faster input scanning, it is better to use the verbose commands instead.

Note 4: As of \LUAETE X 0.65, \Umathaccent accepts optional keywords to control various details regarding math accents. See section 5.7 below for details.

Note 5: \Umathcharnumdef was added in release 0.72.

New primitives that exist in \LUAETE X only (all of these will be explained in following sections):

<table>
<thead>
<tr>
<th>primitive</th>
<th>value range (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Uroot</td>
<td>0+0–FF+10FFFF^2</td>
</tr>
<tr>
<td>\Uoverdelimiter</td>
<td>0+0–FF+10FFFF^2</td>
</tr>
<tr>
<td>\Uunderdelimiter</td>
<td>0+0–FF+10FFFF^2</td>
</tr>
<tr>
<td>\Udelimiterover</td>
<td>0+0–FF+10FFFF^2</td>
</tr>
<tr>
<td>\Udelimiterunder</td>
<td>0+0–FF+10FFFF^2</td>
</tr>
</tbody>
</table>

5.3 Cramped math styles

\LUAETE X has four new primitives to set the cramped math styles directly:

\crampeddisplaystyle
\crampedtextstyle
\crampedscriptstyle
\crampedscriptscriptstyle

These additional commands are not all that valuable on their own, but they come in handy as arguments to the math parameter settings that will be added shortly.

5.4 Math parameter settings

In \LUAETE X, the font dimension parameters that TEX used in math typesetting are now accessible via primitive commands. In fact, refactoring of the math engine has resulted in many more parameters than were accessible before.

<table>
<thead>
<tr>
<th>primitive name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Umathquad</td>
<td>the width of 18mu's</td>
</tr>
</tbody>
</table>
\mathaxis \quad \text{height of the vertical center axis of the math formula above the baseline}

\mathoperatorsize \quad \text{minimum size of large operators in display mode}

\mathoverbarkern \quad \text{vertical clearance above the rule}

\mathoverbarrule \quad \text{the width of the rule}

\mathoverbarvgap \quad \text{vertical clearance below the rule}

\mathunderbarkern \quad \text{vertical clearance below the rule}

\mathunderbarrule \quad \text{the width of the rule}

\mathunderbarvgap \quad \text{vertical clearance above the rule}

\mathradicalkern \quad \text{vertical clearance above the rule}

\mathradicalrule \quad \text{the width of the rule}

\mathradicalvgap \quad \text{vertical clearance below the rule}

\mathradicaldegreebefore \quad \text{the forward kern that takes place before placement of the radical degree}

\mathradicaldegreeafter \quad \text{the backward kern that takes place after placement of the radical degree}

\mathradicaldegreeraise \quad \text{this is the percentage of the total height and depth of the radical sign that the degree is raised by. It is expressed in percents, so 60\% is expressed as the integer 60.}

\mathstackvgap \quad \text{vertical clearance between the two elements in a \texttt{\atop} stack}

\mathstacknumup \quad \text{numerator shift upward in \texttt{\atop} stack}

\mathstackdenomdown \quad \text{denominator shift downward in \texttt{\atop} stack}

\mathfractionrule \quad \text{the width of the rule in a \texttt{\over}}

\mathfractionnumvgap \quad \text{vertical clearance between the numerator and the rule}

\mathfractionnumup \quad \text{numerator shift upward in \texttt{\over}}

\mathfractiondenomgap \quad \text{vertical clearance between the denominator and the rule}

\mathfractiondenomdown \quad \text{denominator shift downward in \texttt{\over}}

\mathfractiondelsize \quad \text{minimum delimiter size for \texttt{\ldots withdelims}}

\mathlimitabovevgap \quad \text{vertical clearance for limits above operators}

\mathlimitabovebgap \quad \text{vertical baseline clearance for limits above operators}

\mathlimitabovekern \quad \text{space reserved at the top of the limit}

\mathlimitbelowvgap \quad \text{vertical clearance for limits below operators}

\mathlimitbelowbgap \quad \text{vertical baseline clearance for limits below operators}

\mathlimitbelowellern \quad \text{space reserved at the bottom of the limit}

\mathoverdelimitervgap \quad \text{vertical clearance for limits above delimiters}

\mathoverdelimiterbgap \quad \text{vertical baseline clearance for limits above delimiters}

\mathunderdelimitervgap \quad \text{vertical clearance for limits below delimiters}

\mathunderdelimiterbgap \quad \text{vertical baseline clearance for limits below delimiters}

\mathsubshiftdrop \quad \text{subscript drop for boxes and subformulas}

\mathsubshiftdown \quad \text{subscript drop for characters}

\mathsupshiftdrop \quad \text{superscript drop (raise, actually) for boxes and subformulas}

\mathsupshiftup \quad \text{superscript raise for characters}

\mathsubsupshiftdown \quad \text{subscript drop in the presence of a superscript}
Each of the parameters in this section can be set by a command like this:

\Umathquad\displaystyle=1em

ey they obey grouping, and you can use \the\Umathquad\displaystyle if needed.

5.5 Font-based Math Parameters

While it is nice to have these math parameters available for tweaking, it would be tedious to have to set each of them by hand. For this reason, LuaTeX initializes a bunch of these parameters whenever you assign a font identifier to a math family based on either the traditional math font dimensions in the font (for assignments to math family 2 and 3 using TFM-based fonts like cmey and cmex), or based on the named values in a potential MathConstants table when the font is loaded via Lua. If there is a MathConstants table, this takes precedence over font dimensions, and in that case no attention is paid to which family is being assigned to: the MathConstants tables in the last assigned family sets all parameters.

In the table below, the one-letter style abbreviations and symbolic tfm font dimension names match those using in the Texbook. Assignments to \textfont set the values for the cramped and uncramped display and text styles. Use \scriptfont for the script styles, and \scriptscriptfont for the scriptscript styles (totalling eight parameters for three font sizes). In the TFM case, assignments only happen in family 2 and family 3 (and of course only for the parameters for which there are font dimensions).

Besides the parameters below, LuaTeX also looks at the ‘space’ font dimension parameter. For math fonts, this should be set to zero.

<table>
<thead>
<tr>
<th>variable</th>
<th>style</th>
<th>default value opentype</th>
<th>default value tfm</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Umathaxis</td>
<td>–</td>
<td>AxisHeight</td>
<td>axis_height</td>
</tr>
<tr>
<td>\Umathoperatorsize</td>
<td>D, D'</td>
<td>DisplayOperatorMinHeight</td>
<td>delim1, delim2</td>
</tr>
<tr>
<td>\Umathfractiondelsize</td>
<td>T, T', S, SS, SS'</td>
<td>FractionDelimiterDisplayStyleSize, FractionDelimiterSize</td>
<td>delim1, delim2</td>
</tr>
<tr>
<td>\Umathfractiondenomdown D, D'</td>
<td>FractionDenominatorDisplayStyleShiftDown, FractionDenominatorShiftDown</td>
<td>denom1, denom2</td>
<td></td>
</tr>
<tr>
<td>\Umathfractiondenomvgap D, D'</td>
<td>FractionDenominatorDisplayStyleGapMin</td>
<td>3*default_rule_thickness, default_rule_thickness</td>
<td></td>
</tr>
<tr>
<td>\Umathfractionnumup T, T', S, SS, SS'</td>
<td>FractionNumeratorDisplayStyleShiftUp, FractionNumeratorShiftUp</td>
<td>num1, num2</td>
<td></td>
</tr>
<tr>
<td>\Umathfractionnumvgap D, D'</td>
<td>FractionNumeratorDisplayStyleGapMin</td>
<td>3*default_rule_thickness, default_rule_thickness</td>
<td></td>
</tr>
<tr>
<td>\Umathfractionrule –</td>
<td>FractionRuleThickness</td>
<td>default_rule_thickness, default_rule_thickness</td>
<td></td>
</tr>
</tbody>
</table>
\[ \text{\textbackslash mathlimitabovebgap} \quad \text{\textbackslash mathlimitbelowbgap} \quad \text{\textbackslash mathoverdelimitervgap} \quad \text{\textbackslash mathunderdelimitervgap} \quad \text{\textbackslash mathoverbarkern} \quad \text{\textbackslash mathoverbarrule} \quad \text{\textbackslash mathoverbarvargap} \quad \text{\textbackslash mathradicalkern} \quad \text{\textbackslash mathradicalrule} \quad \text{\textbackslash mathradicalvgap} \quad \text{\textbackslash mathradicaldegreebefore} \quad \text{\textbackslash mathradicaldegreeafter} \quad \text{\textbackslash mathradicaldegreeraise} \quad \text{\textbackslash mathspaceafterscript} \quad \text{\textbackslash mathstackdenomdown} \quad \text{\textbackslash mathstacknumup} \quad \text{\textbackslash mathsubshiftdown} \quad \text{\textbackslash mathsubsupshiftdown} \quad \text{\textbackslash mathunderbarkern} \quad \text{\textbackslash mathunderbarrule} \quad \text{\textbackslash mathunderbarvargap} \quad \text{\textbackslash mathconnectoroverlapmin} \]

Note 1: OpenType fonts set \textbackslash mathlimitabovekern and \textbackslash mathlimitbelowkern to zero and set \textbackslash mathquad to the font size of the used font, because these are not supported in the MATH table.

Note 2: tfm fonts do not set \textbackslash mathradicalrule because TeX82 uses the height of the radical instead. When this parameter is indeed not set when LATEX has to typeset a radical, a backward compatibility mode will kick in that assumes that an oldstyle TeX font is used. Also, they do not set \textbackslash mathradicaldegreebefore, \textbackslash mathradicaldegreeafter, and \textbackslash mathradicaldegreeraise. These are then automatically initialized to 5/18 quad, −10/18 quad, and 60.

Note 3: If tfm fonts are used, then the \textbackslash mathradicalvgap is not set until the first time LATEX has to typeset a formula because this needs parameters from both family2 and family3. This provides a partial
backward compatibility with \TeX82, but that compatibility is only partial: once the \texttt{\mathradicalv-gap} is set, it will not be recalculated any more.

Note 4: (also if \texttt{tfm} fonts are used) A similar situation arises wrt. \texttt{\mathspaceafterscript}: it is not set until the first time \LaTeX has to typeset a formula. This provides some backward compatibility with \TeX82. But once the \texttt{\mathspaceafterscript} is set, \texttt{\scriptspace} will never be looked at again.

Note 5: \texttt{\textit{Tfm}} fonts set \texttt{\mathconnectoroverlapmin} to zero because \TeX82 always stacks extensibles without any overlap.

Note 6: The \texttt{\mathoperatorsize} is only used in \texttt{\displaystyle}, and is only set in \textsf{OpenType} fonts. In \texttt{Tfm} font mode, it is artificially set to one scaled point more than the initial attempt's size, so that always the 'first next' will be tried, just like in \TeX82.

Note 7: The \texttt{\mathradicaldegreeraise} is a special case because it is the only parameter that is expressed in a percentage instead of as a number of scaled points.

Note 8: \texttt{\mathsubscriptshiftdownwithsuperscript} does not actually exist in the 'standard' Opentype Math font Cambria, but it is useful enough to be added. New in version 0.38.

Note 9: \texttt{\mathfractiondelimiterdisplaystyle} and \texttt{\mathfractiondelimiter} do not actually exist in the 'standard' Opentype Math font Cambria, but were useful enough to be added. New in version 0.47.

5.6 Math spacing setting

Besides the parameters mentioned in the previous sections, there are also 64 new primitives to control the math spacing table (as explained in Chapter 18 of the \TeXbook). The primitive names are a simple matter of combining two math atom types, but for completeness' sake, here is the whole list:

\texttt{\mathordordspacing} \texttt{\mathbinordspacing}
\texttt{\mathordopspacing} \texttt{\mathbinopspacing}
\texttt{\mathordbinspacing} \texttt{\mathbinbinspacing}
\texttt{\mathordrelspacing} \texttt{\mathbinrelspacing}
\texttt{\mathordclosespacing} \texttt{\mathbinclosespacing}
\texttt{\mathordpunctspacing} \texttt{\mathbinpunctspacing}
\texttt{\mathordinnerspacing} \texttt{\mathbininnerspacing}
\texttt{\mathopordspacing} \texttt{\mathopordspacing}
\texttt{\mathopopspacing} \texttt{\mathopopspacing}
\texttt{\mathopbinspacing} \texttt{\mathopbinspacing}
\texttt{\mathoprelspacing} \texttt{\mathoprelspacing}
\texttt{\mathopclosespacing} \texttt{\mathopclosespacing}
\texttt{\mathoppunctspacing} \texttt{\mathoppunctspacing}
\texttt{\mathopenordspacing} \texttt{\mathopenordspacing}
\texttt{\mathopenopspacing} \texttt{\mathopenopspacing}
These parameters are of type $\texttt{muskip}$, so setting a parameter can be done like this:

\begin{verbatim}
\texttt{\textbackslash{Umathpordspacing}displaystyle=4mu plus 2mu}
\end{verbatim}

They are all initialized by \texttt{initex} to the values mentioned in the table in Chapter 18 of the \TeX{}book.

Note 1: for ease of use as well as for backward compatibility, $\texttt{\textbackslash{thinmuskip}}$, $\texttt{\textbackslash{medmuskip}}$ and $\texttt{\textbackslash{thickmuskip}}$ are treated especially. In their case a pointer to the corresponding internal parameter is saved, not the actual $\texttt{\textbackslash{muskip}}$ value. This means that any later changes to one of these three parameters will be taken into account.

Note 2: Careful readers will realise that there are also primitives for the items marked $\ast$ in the \TeX{}book. These will not actually be used as those combinations of atoms cannot actually happen, but it seemed better not to break orthogonality. They are initialized to zero.

## 5.7 Math accent handling

\LaTeX{} supports both top accents and bottom accents in math mode, and math accents stretch automatically (if this is supported by the font the accent comes from, of course). Bottom and combined accents as well as fixed-width math accents are controlled by optional keywords following $\texttt{\textbackslash{Umathaccent}}$.

The keyword $\texttt{bottom}$ after $\texttt{\textbackslash{Umathaccent}}$ signals that a bottom accent is needed, and the keyword $\texttt{both}$ signals that both a top and a bottom accent are needed (in this case two accents need to be specified, of course).

Then the set of three integers defining the accent is read. This set of integers can be prefixed by the fixed keyword to indicate that a non-stretching variant is requested (in case of both accents, this step is repeated).

A simple example:
If a math top accent has to be placed and the accentee is a character and has a non-zero \texttt{top\_accent} value, then this value will be used to place the accent instead of the \texttt{skewchar} kern used by \TeX82. The \texttt{top\_accent} value represents a vertical line somewhere in the accentee. The accent will be shifted horizontally such that its own \texttt{top\_accent} line coincides with the one from the accentee. If the \texttt{top\_accent} value of the accent is zero, then half the width of the accent followed by its italic correction is used instead.

The vertical placement of a top accent depends on the \texttt{x\_height} of the font of the accentee (as explained in the \TeXbook), but if value that turns out to be zero and the font had a MathConstants table, then \texttt{AccentBaseHeight} is used instead.

If a math bottom accent has to be placed, the \texttt{bot\_accent} value is checked instead of \texttt{top\_accent}. Because bottom accents do not exist in \TeX82, the \texttt{skewchar} kern is ignored.

The vertical placement of a bottom accent is straight below the accentee, no correction takes place.

### 5.8 Math root extension

The new primitive \texttt{\Uroot} allows the construction of a radical node including a degree field. Its syntax is an extension of \texttt{\Urational}:

\begin{verbatim}
\Urational <fam integer> <char integer> <radicand>
\Uroot <fam integer> <char integer> <degree> <radicand>
\end{verbatim}

The placement of the degree is controlled by the math parameters \texttt{\Umathradicaldegreebefore}, \texttt{\Umathradicaldegreeafter}, and \texttt{\Umathradicaldegreeraise}. The degree will be typeset in \texttt{\scriptscriptstyle}.

### 5.9 Math kerning in super- and subscripts

The character fields in a lua-loaded OpenType math font can have a ‘mathkern’ table. The format of this table is the same as the ‘mathkern’ table that is returned by the \texttt{fontloader} library, except that all height and kern values have to be specified in actual scaled points.

When a super- or subscript has to be placed next to a math item, \texttt{\LuaTeX} checks whether the super- or subscript and the nucleus are both simple character items. If they are, and if the fonts of both character items are OpenType fonts (as opposed to legacy \TeX fonts), then \texttt{\LuaTeX} will use the OpenType MATH algorithm for deciding on the horizontal placement of the super- or subscript.

This works as follows:

- The vertical position of the script is calculated.
- The default horizontal position is flat next to the base character.
- For superscripts, the italic correction of the base character is added.
For a superscript, two vertical values are calculated: the bottom of the script (after shifting up), and the top of the base. For a subscript, the two values are the top of the (shifted down) script, and the bottom of the base.

For each of these two locations:
- find the mathkern value at this height for the base (for a subscript placement, this is the bottom_right corner, for a superscript placement the top_right corner)
- find the mathkern value at this height for the script (for a subscript placement, this is the top_left corner, for a superscript placement the bottom_left corner)
- add the found values together to get a preliminary result.

The horizontal kern to be applied is the smallest of the two results from previous step.

The mathkern value at a specific height is the kern value that is specified by the next higher height and kern pair, or the highest one in the character (if there is no value high enough in the character), or simply zero (if the character has no mathkern pairs at all).

### 5.10 Scripts on horizontally extensible items like arrows

The new primitives \texttt{\textbackslash Uunderdelimiter} and \texttt{\textbackslash Uoverdelimiter} (both from 0.35) allow the placement of a subscript or superscript on an automatically extensible item and \texttt{\textbackslash Udelimiterunder} and \texttt{\textbackslash Udelimiterover} (both from 0.37) allow the placement of an automatically extensible item as a subscript or superscript on a nucleus.

The vertical placements are controlled by \texttt{\textbackslash Umathunderdelimiterbgap}, \texttt{\textbackslash Umathunderdelimitervgap}, \texttt{\textbackslash Umathoverdelimiterbgap}, and \texttt{\textbackslash Umathoverdelimitervgap} in a similar way as limit placements on large operators. The superscript in \texttt{\textbackslash Uoverdelimiter} is typeset in a suitable scripted style, the subscript in \texttt{\textbackslash Uunderdelimiter} is cramped as well.

### 5.11 Extensible delimiters

\LaTeX{} internally uses a structure that supports OpenType 'MathVariants' as well as TFM 'extensible recipes'.

### 5.12 Other Math changes

#### 5.12.1 Verbose versions of single-character math commands

\LaTeX{} defines six new primitives that have the same function as \textbackslash ^, \_\_\_, $, and $\$\$

<table>
<thead>
<tr>
<th>primitive</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\textbackslash Usuperscript}</td>
<td>Duplicates the functionality of \textbackslash ^</td>
</tr>
<tr>
<td>\texttt{\textbackslash Usubscript}</td>
<td>Duplicates the functionality of ___</td>
</tr>
<tr>
<td>\texttt{\textbackslash Ustartmath}</td>
<td>Duplicates the functionality of $, when used in non-math mode.</td>
</tr>
</tbody>
</table>
\texttt{\textbackslash stopmath} \hspace{1em} Duplicates the functionality of $\$, when used in inline math mode.
\texttt{\textbackslash stopdisplaymath} \hspace{1em} Duplicates the functionality of $$, when used in non-math mode.
\texttt{\textbackslash stopdisplaymath} \hspace{1em} Duplicates the functionality of $$, when used in display math mode.

All are new in version 0.38. The \texttt{\textbackslash stopmath} and \texttt{\textbackslash stopdisplaymath} primitives check if the current math mode is the correct one (inline vs. displayed), but you can freely intermix the four mathon/mathoff commands with explicit dollar sign(s).

### 5.12.2 Allowed math commands in non-math modes

The commands \texttt{\textbackslash mathchar}, and \texttt{\textbackslash umathchar} and control sequences that are the result of \texttt{\textbackslash mathchardef} or \texttt{\textbackslash umathchardef} are also acceptable in the horizontal and vertical modes. In those cases, the \texttt{\textbackslash textfont} from the requested math family is used.

### 5.13 Math todo

The following items are still todo.

- Pre-scripts.
- Multi-story stacks.
- Flattened accents for high characters (?).
- Better control over the spacing around displays and handling of equation numbers.
- Support for multi-line displays using MathML style alignment points.
6 Languages and characters, fonts and glyphs

LuaTeX’s internal handling of the characters and glyphs that eventually become typeset is quite different from the way TeX82 handles those same objects. The easiest way to explain the difference is to focus on unrestricted horizontal mode (i.e. paragraphs) and hyphenation first. Later on, it will be easy to deal with the differences that occur in horizontal and math modes.

In TeX82, the characters you type are converted into char_node records when they are encountered by the main control loop. TeX attaches and processes the font information while creating those records, so that the resulting ‘horizontal list’ contains the final forms of ligatures and implicit kerning. This packaging is needed because we may want to get the effective width of for instance a horizontal box.

When it becomes necessary to hyphenate words in a paragraph, TeX converts (one word at time) the char_node records into a string array by replacing ligatures with their components and ignoring the kerning. Then it runs the hyphenation algorithm on this string, and converts the hyphenated result back into a ‘horizontal list’ that is consecutively spliced back into the paragraph stream. Keep in mind that the paragraph may contain unboxed horizontal material, which then already contains ligatures and kerns and the words therein are part of the hyphenation process.

The char_node records are somewhat misnamed, as they are glyph positions in specific fonts, and therefore not really ‘characters’ in the linguistic sense. There is no language information inside the char_node records. Instead, language information is passed along using language whatsit records inside the horizontal list.

In LuaTeX, the situation is quite different. The characters you type are always converted into glyph_node records with a special subtype to identify them as being intended as linguistic characters. LuaTeX stores the needed language information in those records, but does not do any font-related processing at the time of node creation. It only stores the index of the font.

When it becomes necessary to typeset a paragraph, LuaTeX first inserts all hyphenation points right into the whole node list. Next, it processes all the font information in the whole list (creating ligatures and adjusting kerning), and finally it adjusts all the subtype identifiers so that the records are ‘glyph nodes’ from now on.

That was the broad overview. The rest of this chapter will deal with the minutiae of the new process.

6.1 Characters and glyphs

TeX82 (including pdftex) differentiated between char_nodes and lig_nodes. The former are simple items that contained nothing but a ‘character’ and a ‘font’ field, and they lived in the same memory as tokens did. The latter also contained a list of components, and a subtype indicating whether this ligature was the result of a word boundary, and it was stored in the same place as other nodes like boxes and kerns and glues.

In LuaTeX, these two types are merged into one, somewhat larger structure called a glyph_node. Besides having the old character, font, and component fields, and the new special fields like ‘attr’ (see section 8.1.2.12), these nodes also contain:
• A subtype, split into four main types:
  – **character**, for characters to be hyphenated: the lowest bit (bit 0) is set to 1.
  – **glyph**, for specific font glyphs: the lowest bit (bit 0) is not set.
  – **ligature**, for ligatures (bit 1 is set)
  – **ghost**, for ‘ghost objects’ (bit 2 is set)

The latter two make further use of two extra fields (bits 3 and 4):
  – **left**, for ligatures created from a left word boundary and for ghosts created from `\leftghost`
  – **right**, for ligatures created from a right word boundary and for ghosts created from `\rightghost`

For ligatures, both bits can be set at the same time (in case of a single-glyph word).

• **glyph_nodes** of type ‘character’ also contain language data, split into four items that were current when the node was created: the `\setlanguage` (15 bits), `\lefthyphenmin` (8 bits), `\righthyphenmin` (8 bits), and `\uchyph` (1 bit).

Incidentally, LuaTeX allows 16383 separate languages, and words can be 256 characters long.

Because the `\uchyph` value is saved in the actual nodes, its handling is subtly different from \TeX82: changes to `\uchyph` become effective immediately, not at the end of the current partial paragraph.

Typeset boxes now always have their language information embedded in the nodes themselves, so there is no longer a possible dependency on the surrounding language settings. In \TeX82, a mid-paragraph statement like `\unhbox0` would process the box using the current paragraph language unless there was a `\setlanguage` issued inside the box. In LuaTeX, all language variables are already frozen.

### 6.2 The main control loop

In LuaTeX’s main loop, almost all input characters that are to be typeset are converted into **glyph_node** records with subtype ‘character’, but there are a few small exceptions.

First, the `\accent` primitives creates nodes with subtype ‘glyph’ instead of ‘character’: one for the actual accent and one for the accentee. The primary reason for this is that `\accent` in \TeX82 is explicitly dependent on the current font encoding, so it would not make much sense to attach a new meaning to the primitive’s name, as that would invalidate many old documents and macro packages. A secondary reason is that in \TeX82, `\accent` prohibits hyphenation of the current word. Since in LuaTeX hyphenation only takes place on ‘character’ nodes, it is possible to achieve the same effect.

This change of meaning did happen with `\char`, that now generates ‘character’ nodes, consistent with its changed meaning in \XeTeX. The changed status of `\char` is not yet finalized, but if it stays as it is now, a new primitive `\glyph` should be added to directly insert a font glyph id.

Second, all the results of processing in math mode eventually become nodes with ‘glyph’ subtypes.

Third, the \aleph-derived commands `\leftghost` and `\rightghost` create nodes of a third subtype: ‘ghost’. These nodes are ignored completely by all further processing until the stage where inter-glyph kerning is added.

Fourth, automatic discretionaries are handled differently. \TeX82 inserts an empty discretionary after sensing an input character that matches the `\hyphenchar` in the current font. This test is wrong, in our
opinion: whether or not hyphenation takes place should not depend on the current font, it is a language property.

In LuaTeX, it works like this: if LuaTeX senses a string of input characters that matches the value of the new integer parameter \texttt{\exhyphenchar}, it will insert an explicit discretionary after that series of nodes. Initex sets the \texttt{\exhyphenchar=`\-}. Incidentally, this is a global parameter instead of a language-specific one because it may be useful to change the value depending on the document structure instead of the text language.

Note: as of LuaTeX 0.63.0, the insertion of discretionaries after a sequence of explicit hyphens happens at the same time as the other hyphenation processing, not inside the main control loop.

The only use LuaTeX has for \texttt{\hyphenchar} is at the check whether a word should be considered for hyphenation at all. If the \texttt{\hyphenchar} of the font attached to the first character node in a word is negative, then hyphenation of that word is abandoned immediately. This behavior is added for backward compatibility only, and the use of \texttt{\hyphenchar=-1} as a means of preventing hyphenation should not be used in new LuaTeX documents.

Fifth, \texttt{\setlanguage} no longer creates whatsits. The meaning of \texttt{\setlanguage} is changed so that it is now an integer parameter like all others. That integer parameter is used in \texttt{\glyph_node} creation to add language information to the glyph nodes. In conjunction, the \texttt{\language} primitive is extended so that it always also updates the value of \texttt{\setlanguage}.

Sixth, the \texttt{\noboundary} command (this command prohibits word boundary processing where that would normally take place) now does create whatsits. These whatsits are needed because the exact place of the \texttt{\noboundary} command in the input stream has to be retained until after the ligature and font processing stages.

Finally, there is no longer a \texttt{main_loop} label in the code. Remember that \TeX82 did quite a lot of processing while adding \texttt{char_nodes} to the horizontal list? For speed reasons, it handled that processing code outside of the ‘main control’ loop, and only the first character of any ‘word’ was handled by that ‘main control’ loop. In LuaTeX, there is no longer a need for that (all hard work is done later), and the (now very small) bits of character-handling code have been moved back inline. When \texttt{\tracingcommands} is on, this is visible because the full word is reported, instead of just the initial character.

### 6.3 Loading patterns and exceptions

The hyphenation algorithm in LuaTeX is quite different from the one in \TeX82, although it uses essentially the same user input.

After expansion, the argument for \texttt{\patterns} has to be proper UTF-8 with individual patterns separated by spaces, no \texttt{\char} or \texttt{\chardef-ed} commands are allowed. (The current implementation is even more strict, and will reject all non-\texttt{UNICODE} characters, but that will be changed in the future. For now, the generated errors are a valuable tool in discovering font-encoding specific pattern files)

Likewise, the expanded argument for \texttt{\hyphenation} also has to be proper UTF-8, but here a tiny little bit of extra syntax is provided:
1. three sets of arguments in curly braces \{\{}\{\} indicates a desired complex discretionary, with arguments as in \texttt{\textbackslash discretionary}'s command in normal document input.
2. \- indicates a desired simple discretionary, cf. \textbackslash - and \texttt{\textbackslash discretionary\{-\}{\}} in normal document input.
3. Internal command names are ignored. This rule is provided especially for \texttt{\textbackslash discretionary}, but it also helps to deal with \texttt{\textbackslash relax} commands that may sneak in.
4. \= indicates a (non-discretionary) hyphen in the document input.

The expanded argument is first converted back to a space-separated string while dropping the internal command names. This string is then converted into a dictionary by a routine that creates key–value pairs by converting the other listed items. It is important to note that the keys in an exception dictionary can always be generated from the values. Here are a few examples:

<table>
<thead>
<tr>
<th>value</th>
<th>implied key (input)</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ta-ble</td>
<td>table</td>
<td>ta-ble (= ta\texttt{\textbackslash discretionary{-}}{}{}ble)</td>
</tr>
<tr>
<td>ba{-}{}cken</td>
<td>backen</td>
<td>ba\texttt{\textbackslash discretionary{-}}{}cken</td>
</tr>
</tbody>
</table>

The resultant patterns and exception dictionary will be stored under the language code that is the present value of \texttt{\language}.

In the last line of the table, you see there is no \texttt{\textbackslash discretionary} command in the value: the command is optional in the \TeX-based input syntax. The underlying reason for that is that it is conceivable that a whole dictionary of words is stored as a plain text file and loaded into \LaTeX using one of the functions in the \texttt{\lua lang} library. This loading method is quite a bit faster than going through the \TeX language primitives, but some (most?) of that speed gain would be lost if it had to interpret command sequences while doing so.

Starting with \LaTeX 0.63.0, it is possible to specify extra hyphenation points in compound words by using \{-\}{}\{-\} for the explicit hyphen character (replace \- by the actual explicit hyphen character if needed). For example, this matches the word ‘multi-word-boundaries’ and allows an extra break inbetween ‘boun’ and ‘daries’:

\texttt{\hyphenation\{-\}}\{-\}\texttt{\word\{-\}}\{-\}boun-daries

The motivation behind the \texttt{\epsilon -\TeX} extension \texttt{\savinghyphcodes} was that hyphenation heavily depended on font encodings. This is no longer true in \LaTeX, and the corresponding primitive is ignored pending complete removal. The future semantics of \texttt{\uppercase} and \texttt{\lowercase} are still under consideration, no changes have taken place yet.

### 6.4 Applying hyphenation

The internal structures \LaTeX uses for the insertion of discretionaries in words is very different from the ones in \TeX82, and that means there are some noticeable differences in handling as well.

First and foremost, there is no ‘compressed trie’ involved in hyphenation. The algorithm still reads \texttt{\patgen}-generated pattern files, but \LaTeX uses a finite state hash to match the patterns against the word to be hyphenated. This algorithm is based on the ‘\libhnj’ library used by OpenOffice, which in turn
is inspired by \TeX. The memory allocation for this new implementation is completely dynamic, so the \texttt{web2c} setting for \texttt{trie\_size} is ignored.

Differences between \LaTeX\ and \TeX82 that are a direct result of that:

- \LaTeX\ happily hyphenates the full \texttt{UNICODE} character range.
- Pattern and exception dictionary size is limited by the available memory only, all allocations are done dynamically. The trie-related settings in \texttt{texmf.cnf} are ignored.
- Because there is no ‘trie preparation’ stage, language patterns never become frozen. This means that the primitive \texttt{\patterns} (and its \texttt{lua} counterpart \texttt{lang.patterns}) can be used at any time, not only in \texttt{initex}.
- Only the string representation of \texttt{\patterns} and \texttt{\hyphenation} is stored in the format file. At format load time, they are simply re-evaluated. It follows that there is no real reason to preload languages in the format file. In fact, it is usually not a good idea to do so. It is much smarter to load patterns no sooner than the first time they are actually needed.
- \LaTeX\ uses the language-specific variables \texttt{\prehyphenchar} and \texttt{\posthyphenchar} in the creation of implicit discretionaries, instead of \TeX82’s \texttt{\hyphenchar}, and the values of the language-specific variables \texttt{\preexhyphenchar} and \texttt{\postexhyphenchar} for explicit discretionaries (instead of \TeX82’s empty discretionary).

Inserted characters and ligatures inherit their attributes from the nearest glyph node item (usually the preceding one, but the following one for the items inserted at the left-hand side of a word).

Word boundaries are no longer implied by font switches, but by language switches. One word can have two separate fonts and still be hyphenated correctly (but it can not have two different languages, the \texttt{\setlanguage} command forces a word boundary).

All languages start out with \texttt{\prehyphenchar=`\-, \posthyphenchar=0, \preexhyphenchar=0} and \texttt{\postexhyphenchar=0}. When you assign the values of one of these four parameters, you are actually changing the settings for the current \texttt{\language}, this behavior is compatible with \texttt{\patterns} and \texttt{\hyphenation}.

\LaTeX\ also hyphenates the first word in a paragraph.

Words can be up to 256 characters long (up from 64 in \TeX82). Longer words generate an error right now, but eventually either the limitation will be removed or perhaps it will become possible to silently ignore the excess characters (this is what happens in \TeX82, but there the behavior cannot be controlled).

If you are using the \texttt{lua} function \texttt{lang.hyphenate}, you should be aware that this function expects to receive a list of ‘character’ nodes. It will operate properly in the presence of ‘glyph’, ‘ligature’, or ‘ghost’ nodes, nor does it know how to deal with kerning. In the near future, it will be able to skip over ‘ghost’ nodes, and we may add a less fuzzy function you can call as well.

The hyphenation exception dictionary is maintained as key-value hash, and that is also dynamic, so the \texttt{hyph\_size} setting is not used either.

A technical paper detailing the new algorithm will be released as a separate document.
6.5 Applying ligatures and kerning

After all possible hyphenation points have been inserted in the list, \LaTeX will process the list to convert the ‘character’ nodes into ‘glyph’ and ‘ligature’ nodes. This is actually done in two stages: first all ligatures are processed, then all kerning information is applied to the result list. But those two stages are somewhat dependent on each other: If the used font makes it possible to do so, the ligaturing stage adds virtual ‘character’ nodes to the word boundaries in the list. While doing so, it removes and interprets noboundary nodes. The kerning stage deletes those word boundary items after it is done with them, and it does the same for ‘ghost’ nodes. Finally, at the end of the kerning stage, all remaining ‘character’ nodes are converted to ‘glyph’ nodes.

This work separation is worth mentioning because, if you overrule from Lua only one of the two callbacks related to font handling, then you have to make sure you perform the tasks normally done by \LaTeX itself in order to make sure that the other, non-overruled, routine continues to function properly.

Work in this area is not yet complete, but most of the possible cases are handled by our rewritten ligaturing engine. We are working hard to make sure all of the possible inputs will become supported soon.

For example, take the word office, hyphenated of-fice, using a ‘normal’ font with all the f-f and f-i type ligatures:

Initial: \{o\}{f}\{f\}{i}\{c\}{e}\{\}
After hyphenation: \{o\}{f}\{\{-\},\{\}\}\{f\}\{i\}\{c\}\{e\}
First ligature stage: \{o\}\{\{-\},\{f\},\{<ff>\}\}\{i\}\{c\}\{e\}
Final result: \{o\}\{\{-\},\{<fi>\},\{<ffi>\}\}\{c\}\{e\}

That’s bad enough, but let us assume that there is also a hyphenation point between the f and the i, to create of-f-ice. Then the final result should be:

\{o\}\{\{-\},
  \{\{-\},
    \{<fi>\},
      \{<ffi>\},
        \{<ffi>\}\}\{c\}\{e\}

with discretionaries in the post-break text as well as in the replacement text of the top-level discretionary that resulted from the first hyphenation point.

Here is that nested solution again, in a different representation:

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>topdisc</td>
<td>f^{-1}</td>
<td>sub1</td>
</tr>
<tr>
<td>sub1</td>
<td>f^{-2}</td>
<td>i^{3}</td>
</tr>
<tr>
<td>sub2</td>
<td>&lt;ff&gt;^{-5}</td>
<td>i^{6}</td>
</tr>
</tbody>
</table>
When line breaking is choosing its breakpoints, the following fields will eventually be selected:

- **of-f-ice**
  - \text{disc1.pre} \ f^{-1}
  - \text{disc1.post} \ <fi>^4
  - \text{disc1.replace} \ <ffi>^7
- **off-ice**
  - \text{disc2.pre} \ f^{-2}
  - \text{disc2.post} \ i^3.6
  - \text{disc2.replace} \ <ff>^-5

The current solution in \LaTeX{} is not able to handle nested discretionaries, but it is in fact smart enough to handle this fictional *of-f-ice* example. It does so by combining two sequential discretionary nodes as if they were a single object (where the second discretionary node is treated as an extension of the first node).

One can observe that the *of-f-ice* and *off-ice* cases both end with the same actual post replacement list \(i\), and that this would be the case even if that \(i\) was the first item of a potential following ligature like \(ic\). This allows \LaTeX{} to do away with one of the fields, and thus make the whole stuff fit into just two discretionary nodes.

The mapping of the seven list fields to the six fields in this discretionary node pair is as follows:

<table>
<thead>
<tr>
<th>field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disc1.pre</td>
<td>\ f^{-1}</td>
</tr>
<tr>
<td>disc1.post</td>
<td>\ &lt;fi&gt;^4</td>
</tr>
<tr>
<td>disc1.replace</td>
<td>&lt;ffi&gt;^7</td>
</tr>
<tr>
<td>disc2.pre</td>
<td>\ f^{-2}</td>
</tr>
<tr>
<td>disc2.post</td>
<td>\ i^3.6</td>
</tr>
<tr>
<td>disc2.replace</td>
<td>&lt;ff&gt;^-5</td>
</tr>
</tbody>
</table>

What is actually generated after ligaturing has been applied is therefore:

\[
\{o\}\{\{f-\},
  \{<fi>\},
  \{<ffi>\}\}\{\{f-\},
  \{i\},
  \{<ff>\}\}\{c\}{e}
\]

The two discretionaries have different subtypes from a discretionary appearing on its own: the first has subtype 4, and the second has subtype 5. The need for these special subtypes stems from the fact that not all of the fields appear in their ‘normal’ location. The second discretionary especially looks odd, with things like the \(<ff>-\) appearing in \text{disc2.replace}. The fact that some of the fields have different meanings (and different processing code internally) is what makes it necessary to have different subtypes: this enables \LaTeX{} to distinguish this sequence of two joined discretionary nodes from the case of two standalone discretionaries appearing in a row.
6.6 Breaking paragraphs into lines

This code is still almost unchanged, but because of the above-mentioned changes with respect to discretionaries and ligatures, line breaking will potentially be different from traditional \TeX. The actual line breaking code is still based on the \TeX82 algorithms, and it does not expect there to be discretionaries inside of discretionaries.

But that situation is now fairly common in \LaTeX, due to the changes to the ligaturing mechanism. And also, the \LaTeX discretionary nodes are implemented slightly different from the \TeX82 nodes: the \texttt{no_break} text is now embedded inside the disc node, where previously these nodes kept their place in the horizontal list (the discretionary node contained a counter indicating how many nodes to skip).

The combined effect of these two differences is that \LaTeX does not always use all of the potential breakpoints in a paragraph, especially when fonts with many ligatures are used.
# 7 Font structure

All \TeX\ fonts are represented to \texttt{Lua} code as tables, and internally as C structures. All keys in the table below are saved in the internal font structure if they are present in the table returned by the \texttt{define_font} callback, or if they result from the normal \texttt{TFM/VF} reading routines if there is no \texttt{define_font} callback defined.

The column ‘from vf’ means that this key will be created by the \texttt{font.read_vf()} routine, ‘from tfm’ means that the key will be created by the \texttt{font.read_tfm()} routine, and ‘used’ means whether or not the \texttt{Lua\TeX} engine itself will do something with the key.

The top-level keys in the table are as follows:

<table>
<thead>
<tr>
<th>key</th>
<th>from vf</th>
<th>from tfm</th>
<th>used</th>
<th>value type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>string</td>
<td>metric (file) name</td>
</tr>
<tr>
<td>area</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>string</td>
<td>(directory) location, typically empty</td>
</tr>
<tr>
<td>used</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>boolean</td>
<td>used already? (initial: false)</td>
</tr>
<tr>
<td>characters</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>table</td>
<td>the defined glyphs of this font</td>
</tr>
<tr>
<td>checksum</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>number</td>
<td>default: 0</td>
</tr>
<tr>
<td>designsize</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>expected size (default: 655360 (= 10\text{pt}))</td>
</tr>
<tr>
<td>direction</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>default: 0 (TLT)</td>
</tr>
<tr>
<td>encodingbytes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>default: depends on \texttt{format}</td>
</tr>
<tr>
<td>encodingname</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>encoding name</td>
</tr>
<tr>
<td>fonts</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>table</td>
<td>locally used fonts</td>
</tr>
<tr>
<td>psname</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>actual (PostScript) name (this is the PS fontname in the incoming font source, also used as fontname identifier in the PDF output, new in 0.43)</td>
</tr>
<tr>
<td>fullname</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>output font name, used as a fallback in the PDF output if the psname is not set</td>
</tr>
<tr>
<td>header</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>string</td>
<td>header comments, if any</td>
</tr>
<tr>
<td>hyphenchar</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>default: TeX’s <code>\hyphenchar</code></td>
</tr>
<tr>
<td>parameters</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>hash</td>
<td>default: 7 parameters, all zero</td>
</tr>
<tr>
<td>size</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>loaded (at) size. (default: same as designsize)</td>
</tr>
<tr>
<td>skewchar</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>default: TeX’s <code>\skewchar</code></td>
</tr>
<tr>
<td>type</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>basic type of this font</td>
</tr>
<tr>
<td>format</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>disk format type</td>
</tr>
<tr>
<td>embedding</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>PDF inclusion</td>
</tr>
<tr>
<td>filename</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>disk file name</td>
</tr>
<tr>
<td>tounicode</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>if 1, \texttt{Lua\TeX} assumes per-glyph touni-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>code entries are present in the font</td>
</tr>
<tr>
<td>stretch</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>the ‘stretch’ value from <code>\pdffontexpand</code></td>
</tr>
</tbody>
</table>

Font structure 163
<table>
<thead>
<tr>
<th>Key</th>
<th>Required</th>
<th>Used</th>
<th>Target</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shrink</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>the ‘shrink’ value from \pdffontexpand</td>
</tr>
<tr>
<td>step</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>the ‘step’ value from \pdffontexpand</td>
</tr>
<tr>
<td>auto_expand</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>boolean</td>
<td>the ‘autoexpand’ keyword from \pdffontexpand</td>
</tr>
<tr>
<td>expansion_factor</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>number</td>
<td>the actual expansion factor of an expanded font</td>
</tr>
<tr>
<td>attributes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>the \pdffontattr</td>
</tr>
<tr>
<td>cache</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>string</td>
<td>this key controls caching of the lua table on the tex end.</td>
</tr>
<tr>
<td>nomath</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>boolean</td>
<td>this key allows a minor speedup for text fonts. If it is present and true, then LuaTeX will not check the character entries for math-specific keys. (0.42.0)</td>
</tr>
<tr>
<td>slant</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>This has the same semantics as the SlantFont operator in font map files.</td>
</tr>
<tr>
<td>extent</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>This has the same semantics as the ExtendFont operator in font map files.</td>
</tr>
</tbody>
</table>

The key name is always required. The keys stretch, shrink, step and optionally auto_expand only have meaning when used together: they can be used to replace a post-loading \pdffontexpand command. The expansion_factor is value that can be present inside a font in font fonts. It is the actual expansion factor (a value between -shrink and stretch, with step step) of a font that was automatically generated by the font expansion algorithm. The key attributes can be used to replace \pdffontattr. The key used is set by the engine when a font is actively in use, this makes sure that the font’s definition is written to the output file (dvi or pdf). The tfm reader sets it to false. The direction is a number signalling the ‘normal’ direction for this font. There are sixteen possibilities:
<table>
<thead>
<tr>
<th>number</th>
<th>meaning</th>
<th>number</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>8</td>
<td>TT</td>
</tr>
<tr>
<td>1</td>
<td>LL</td>
<td>9</td>
<td>TL</td>
</tr>
<tr>
<td>2</td>
<td>LB</td>
<td>10</td>
<td>TB</td>
</tr>
<tr>
<td>3</td>
<td>LR</td>
<td>11</td>
<td>TR</td>
</tr>
<tr>
<td>4</td>
<td>RT</td>
<td>12</td>
<td>BT</td>
</tr>
<tr>
<td>5</td>
<td>RL</td>
<td>13</td>
<td>BL</td>
</tr>
<tr>
<td>6</td>
<td>RB</td>
<td>14</td>
<td>BB</td>
</tr>
<tr>
<td>7</td>
<td>RR</td>
<td>15</td>
<td>BR</td>
</tr>
</tbody>
</table>

These are Omega-style direction abbreviations: the first character indicates the 'first' edge of the character glyphs (the edge that is seen first in the writing direction), the second the 'top' side.

The **parameters** is a hash with mixed key types. There are seven possible string keys, as well as a number of integer indices (these start from 8 up). The seven strings are actually used instead of the bottom seven indices, because that gives a nicer user interface.

The names and their internal remapping are:

<table>
<thead>
<tr>
<th>name</th>
<th>internal remapped number</th>
</tr>
</thead>
<tbody>
<tr>
<td>slant</td>
<td>1</td>
</tr>
<tr>
<td>space</td>
<td>2</td>
</tr>
<tr>
<td>space_stretch</td>
<td>3</td>
</tr>
<tr>
<td>space_shrink</td>
<td>4</td>
</tr>
<tr>
<td>x_height</td>
<td>5</td>
</tr>
<tr>
<td>quad</td>
<td>6</td>
</tr>
<tr>
<td>extra_space</td>
<td>7</td>
</tr>
</tbody>
</table>

The keys **type**, **format**, **embedding**, **fullname** and **filename** are used to embed OpenType fonts in the result PDF.

The **characters** table is a list of character hashes indexed by an integer number. The number is the 'internal code' TeX knows this character by.

Two very special string indexes can be used also: **left_boundary** is a virtual character whose ligatures and kerns are used to handle word boundary processing. **right_boundary** is similar but not actually used for anything (yet!).

Other index keys are ignored.

Each character hash itself is a hash. For example, here is the character 'f' (decimal 102) in the font cmr10 at 10 points:

```plaintext
[102] = {
    ['width'] = 200250,
    ['height'] = 455111,
    ['depth'] = 0,
    ['italic'] = 50973,
    ['kerns'] = {
```
The following top-level keys can be present inside a character hash:

<table>
<thead>
<tr>
<th>key</th>
<th>from vf</th>
<th>from tfm</th>
<th>used</th>
<th>value type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>character’s width, in sp (default 0)</td>
</tr>
<tr>
<td>height</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>character’s height, in sp (default 0)</td>
</tr>
<tr>
<td>depth</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>character’s depth, in sp (default 0)</td>
</tr>
<tr>
<td>italic</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>number</td>
<td>character’s italic correction, in sp (default zero)</td>
</tr>
<tr>
<td>top_accent</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s top accent alignment place, in sp (default zero)</td>
</tr>
<tr>
<td>bot_accent</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s bottom accent alignment place, in sp (default zero)</td>
</tr>
<tr>
<td>left_protruding</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s \texttt{lpcode}</td>
</tr>
<tr>
<td>right_protruding</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s \texttt{rpcode}</td>
</tr>
<tr>
<td>expansion_factor</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>number</td>
<td>character’s \texttt{efcode}</td>
</tr>
<tr>
<td>tounicode</td>
<td>no</td>
<td>no</td>
<td>maybe</td>
<td>string</td>
<td>character’s Unicode equivalent(s), in UTF-16BE hexadecimal format</td>
</tr>
<tr>
<td>next</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>the ‘next larger’ character index</td>
</tr>
<tr>
<td>extensible</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>table</td>
<td>the constituent parts of an extensible recipe</td>
</tr>
<tr>
<td>vert_variants</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>table</td>
<td>constituent parts of a vertical variant set</td>
</tr>
</tbody>
</table>
The values of `top_accen`, `bot_accen` and `mathkern` are used only for math accent and superscript placement, see the math chapter 143 in this manual for details.

The values of `left_protruding` and `right_protruding` are used only when `\pdfprotrudechars` is non-zero.

Whether or not `expansion_factor` is used depends on the font’s global expansion settings, as well as on the value of `\pdfadjustspacing`.

The usage of `tounicode` is this: if this font specifies a `tounicode=1` at the top level, then LuaTeX will construct a `/ToUnicode` entry for the PDF font (or font subset) based on the character-level `tounicode` strings, where they are available. If a character does not have a sensible Unicode equivalent, do not provide a string either (no empty strings).

If the font-level `tounicode` is not set, then LuaTeX will build up `/ToUnicode` based on the TeX code points you used, and any character-level `tounicodes` will be ignored. At the moment, the string format is exactly the format that is expected by Adobe CMap files (UTF-16BE in hexadecimal encoding), minus the enclosing angle brackets. This may change in the future. Small example: the `tounicode` for a `fi` ligature would be 00660069.

The presence of `extensible` will overrule `next`, if that is also present. It in turn can be overruled by `vert_variants`.

The `extensible` table is very simple:

<table>
<thead>
<tr>
<th>key type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>‘top’ character index</td>
</tr>
<tr>
<td>mid</td>
<td>‘middle’ character index</td>
</tr>
<tr>
<td>bot</td>
<td>‘bottom’ character index</td>
</tr>
<tr>
<td>rep</td>
<td>‘repeatable’ character index</td>
</tr>
</tbody>
</table>

The `horiz_variants` and `vert_variants` are arrays of components. Each of those components is itself a hash of up to five keys:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>glyph</td>
<td>number</td>
<td>The character index (note that this is an encoding number, not a name).</td>
</tr>
<tr>
<td>extender</td>
<td>number</td>
<td>One (1) if this part is repeatable, zero (0) otherwise.</td>
</tr>
<tr>
<td>start</td>
<td>number</td>
<td>Maximum overlap at the starting side (in scaled points).</td>
</tr>
</tbody>
</table>
end number Maximum overlap at the ending side (in scaled points).
advance number Total advance width of this item (can be zero or missing, then the natural size of
the glyph for character component is used).

The kerns table is a hash indexed by character index (and ‘character index’ is defined as either a non-
negative integer or the string value right_boundary), with the values the kerning to be applied, in
scaled points.

The ligatures table is a hash indexed by character index (and ‘character index’ is defined as either
a non-negative integer or the string value right_boundary), with the values being yet another small
hash, with two fields:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>number</td>
<td>the type of this ligature command, default 0</td>
</tr>
<tr>
<td>char</td>
<td>number</td>
<td>the character index of the resultant ligature</td>
</tr>
</tbody>
</table>

The char field in a ligature is required.

The type field inside a ligature is the numerical or string value of one of the eight possible ligature
types supported by TeX. When TeX inserts a new ligature, it puts the new glyph in the middle of the
left and right glyphs. The original left and right glyphs can optionally be retained, and when at least
one of them is kept, it is also possible to move the new ‘insertion point’ forward one or two places. The
glyph that ends up to the right of the insertion point will become the next ‘left’.

<table>
<thead>
<tr>
<th>textual (Knuth)</th>
<th>number</th>
<th>string</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>l + r :=: n</td>
<td>0</td>
<td>=:</td>
<td>n</td>
</tr>
<tr>
<td>l + r =:= n</td>
<td>1</td>
<td>=:=</td>
<td>nr</td>
</tr>
<tr>
<td>l + r</td>
<td>:=: n</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>l + r =:=</td>
<td>n</td>
<td>3</td>
<td>=:=</td>
</tr>
<tr>
<td>l + r =:=</td>
<td>&gt; n</td>
<td>5</td>
<td>=:=</td>
</tr>
<tr>
<td>l + r</td>
<td>:=:</td>
<td>&gt; n</td>
<td>6</td>
</tr>
<tr>
<td>l + r</td>
<td>:=:</td>
<td>&gt; n</td>
<td>7</td>
</tr>
<tr>
<td>l + r</td>
<td>:=:</td>
<td>&gt;&gt; n</td>
<td>11</td>
</tr>
</tbody>
</table>

The default value is 0, and can be left out. That signifies a ‘normal’ ligature where the ligature replaces
both original glyphs. In this table the | indicates the final insertion point.

The commands array is explained below.

## 7.1 Real fonts

Whether or not a TeX font is a ‘real’ font that should be written to the PDF document is decided by
the type value in the top-level font structure. If the value is real, then this is a proper font, and the
inclusion mechanism will attempt to add the needed font object definitions to the PDF.

Values for type:
The actions to be taken depend on a number of different variables:

- Whether the used font fits in an 8-bit encoding scheme or not
- The type of the disk font file
- The level of embedding requested

A font that uses anything other than an 8-bit encoding vector has to be written to the PDF in a different way.

The rule is: if the font table has `encodingbytes` set to 2, then this is a wide font, in all other cases it isn’t. The value 2 is the default for OpenType and TrueType fonts loaded via Lua. For Type1 fonts, you have to set `encodingbytes` to 2 explicitly. For PK bitmap fonts, wide font encoding is not supported at all.

If no special care is needed, LuaTEX currently falls back to the mapfile-based solution used by PDFTEX and dvips. This behavior will be removed in the future, when the existing code becomes integrated in the new subsystem.

But if this is a ‘wide’ font, then the new subsystem kicks in, and some extra fields have to be present in the font structure. In this case, LuaTEX does not use a map file at all.

The extra fields are: `format`, `embedding`, `fullname`, `cidinfo` (as explained above), `filename`, and the `index` key in the separate characters.

Values for `format` are:

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type1</td>
<td>this is a PostScript Type1 font</td>
</tr>
<tr>
<td>type3</td>
<td>this is a bitmapped (PK) font</td>
</tr>
<tr>
<td>truetype</td>
<td>this is a TrueType or TrueType-based OpenType font</td>
</tr>
<tr>
<td>opentype</td>
<td>this is a PostScript-based OpenType font</td>
</tr>
</tbody>
</table>

(type3 fonts are provided for backward compatibility only, and do not support the new wide encoding options.)

Values for `embedding` are:

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>don’t embed the font at all</td>
</tr>
<tr>
<td>subset</td>
<td>include and attempt to subset the font</td>
</tr>
<tr>
<td>full</td>
<td>include this font in its entirety</td>
</tr>
</tbody>
</table>

It is not possible to artificially modify the transformation matrix for the font at the moment.

The other fields are used as follows: The `fullname` will be the PostScript/PDF font name. The `cidinfo` will be used as the character set (the CID /Ordering and /Registry keys). The `filename` points to the actual font file. If you include the full path in the `filename` or if the file is in the local directory,
LuaTeX will run a little bit more efficient because it will not have to re-run the \texttt{find_xxx_file} callback in that case.

Be careful: when mixing old and new fonts in one document, it is possible to create PostScript name clashes that can result in printing errors. When this happens, you have to change the \texttt{fullname} of the font.

Typeset strings are written out in a wide format using 2 bytes per glyph, using the \texttt{index} key in the character information as value. The overall effect is like having an encoding based on numbers instead of traditional (PostScript) name-based reencoding. The way to get the correct \texttt{index} numbers for Type1 fonts is by loading the font via \texttt{fontloader.open}; use the table indices as \texttt{index} fields.

This type of reencoding means that there is no longer a clear connection between the text in your input file and the strings in the output PDF file. Dealing with this is high on the agenda.

### 7.2 Virtual fonts

You have to take the following steps if you want LuaTeX to treat the returned table from \texttt{define_font} as a virtual font:

- Set the top-level key \texttt{type} to \texttt{virtual}.
- Make sure there is at least one valid entry in \texttt{fonts} (see below).
- Give a \texttt{commands} array to every character (see below).

The presence of the toplevel \texttt{type} key with the specific value \texttt{virtual} will trigger handling of the rest of the special virtual font fields in the table, but the mere existence of 'type' is enough to prevent LuaTeX from looking for a virtual font on its own.

Therefore, this also works 'in reverse': if you are absolutely certain that a font is not a virtual font, assigning the value \texttt{base} or \texttt{real} to \texttt{type} will inhibit LuaTeX from looking for a virtual font file, thereby saving you a disk search.

The \texttt{fonts} is another Lua array. The values are one- or two-key hashes themselves, each entry indicating one of the base fonts in a virtual font. In case your font is referring to itself, you can use the \texttt{font.nextid()} function which returns the index of the next to be defined font which is probably the currently defined one.

An example makes this easy to understand

```lua
fonts = {
    { name = 'ptmr8a', size = 655360 },
    { name = 'psyr', size = 600000 },
    { id = 38 }
}
```

says that the first referenced font (index 1) in this virtual font is \texttt{ptmr8a} loaded at 10pt, and the second is \texttt{psyr} loaded at a little over 9pt. The third one is previously defined font that is known to LuaTeX as fontid ‘38’.
The array index numbers are used by the character command definitions that are part of each character.

The `commands` array is a hash where each item is another small array, with the first entry representing a command and the extra items being the parameters to that command. The allowed commands and their arguments are:

<table>
<thead>
<tr>
<th>command name</th>
<th>arguments</th>
<th>arg type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>font</td>
<td>1</td>
<td>number</td>
<td>select a new font from the local <code>fonts</code> table</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>number</td>
<td>typeset this character number from the current font, and move right by the character's width</td>
</tr>
<tr>
<td>node</td>
<td>1</td>
<td>node</td>
<td>output this node (list), and move right by the width of this list</td>
</tr>
<tr>
<td>slot</td>
<td>2</td>
<td>number</td>
<td>a shortcut for the combination of a font and char command</td>
</tr>
<tr>
<td>push</td>
<td>0</td>
<td>number</td>
<td>save current position</td>
</tr>
<tr>
<td>nop</td>
<td>0</td>
<td>do nothing</td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td>0</td>
<td>pop position</td>
<td></td>
</tr>
<tr>
<td>rule</td>
<td>2</td>
<td>2 numbers</td>
<td>output a rule $ht \ast wd$, and move right.</td>
</tr>
<tr>
<td>down</td>
<td>1</td>
<td>number</td>
<td>move down on the page</td>
</tr>
<tr>
<td>right</td>
<td>1</td>
<td>number</td>
<td>move right on the page</td>
</tr>
<tr>
<td>special</td>
<td>1</td>
<td>string</td>
<td>output a <code>\special</code> command</td>
</tr>
<tr>
<td>lua</td>
<td>1</td>
<td>string</td>
<td>execute a Lua script (at <code>\latelua</code> time)</td>
</tr>
<tr>
<td>image</td>
<td>1</td>
<td>image</td>
<td>output an image (the argument can be either an <code>&lt;image&gt;</code> variable or an <code>image_spec</code> table)</td>
</tr>
<tr>
<td>comment</td>
<td>any</td>
<td>any</td>
<td>the arguments of this command are ignored</td>
</tr>
</tbody>
</table>

Here is a rather elaborate glyph commands example:

```plaintext
...
commands = {
    {'push'}, -- remember where we are
    {'right', 5000}, -- move right about 0.08pt
    {'font', 3}, -- select the fonts[3] entry
    {'char', 97}, -- place character 97 (ASCII 'a')
    {'pop'}, -- go all the way back
    {'down', -200000}, -- move upwards by about 3pt
    {'special', 'pdf: 1 0 0 rg'} -- switch to red color
    {'rule', 500000, 20000} -- draw a bar
    {'special', 'pdf: 0 g'} -- back to black
}
...
```

The default value for `font` is always 1 at the start of the `commands` array. Therefore, if the virtual font is essentially only a re-encoding, then you do usually not have create an explicit 'font' command in the array.

Rules inside of `commands` arrays are built up using only two dimensions: they do not have depth. For correct vertical placement, an extra `down` command may be needed.
Regardless of the amount of movement you create within the commands, the output pointer will always move by exactly the width that was given in the width key of the character hash. Any movements that take place inside the commands array are ignored on the upper level.

### 7.2.1 Artificial fonts

Even in a ‘real’ font, there can be virtual characters. When LuaTeX encounters a commands field inside a character when it becomes time to typeset the character, it will interpret the commands, just like for a true virtual character. In this case, if you have created no ‘fonts’ array, then the default (and only) ‘base’ font is taken to be the current font itself. In practice, this means that you can create virtual duplicates of existing characters which is useful if you want to create composite characters.

Note: this feature does not work the other way around. There cannot be ‘real’ characters in a virtual font! You cannot use this technique for font re-encoding either; you need a truly virtual font for that (because characters that are already present cannot be altered).

### 7.2.2 Example virtual font

Finally, here is a plain TeX input file with a virtual font demonstration:

```
\directlua {  
  callback.register('define_font',  
    function (name,size)  
      if name == 'cmr10-red' then  
        f = font.read_tfm('cmr10',size)  
        f.name = 'cmr10-red'  
        f.type = 'virtual'  
        f.fonts = {{ name = 'cmr10', size = size }}  
        for i,v in pairs(f.characters) do  
          if (string.char(i)):find('[tacohanshartmut]') then  
            v.commands = {  
              {'special','pdf: 1 0 0 rg'},  
              {'char',i},  
              {'special','pdf: 0 g'},  
            }  
          else  
            v.commands = {{'char',i}}  
          end  
        end  
      else  
        f = font.read_tfm(name,size)  
      end  
      return f  
    end
```
\font\myfont = cmr10-red at 10pt \myfont This is a line of text \par
\font\myfontx= cmr10 at 10pt \myfontx Here is another line of text \par
8 Nodes

8.1 LUA node representation

\TeX's nodes are represented in Lua as userdata object with a variable set of fields. In the following syntax tables, such the type of such a userdata object is represented as \langle node \rangle.

The current return value of node.types() is: hlist (0), vlist (1), rule (2), ins (3), mark (4), adjust (5), disc (7), whatsit (8), math (9), glue (10), kern (11), penalty (12), unset (13), style (14), choice (15), noad (16), op (17), bin (18), rel (19), open (20), close (21), punct (22), inner (23), radical (24), fraction (25), under (26), over (27), accent (28), vcenter (29), fence (30), math_char (31), sub_box (32), sub_mlist (33), math_text_char (34), delim (35), margin_kern (36), glyph (37), align_record (38), pseudo_file (39), pseudo_line (40), page_insert (41), split_insert (42), expr_stack (43), nested_list (44), span (45), attribute (46), glue_spec (47), attribute_list (48), action (49), temp (50), align_stack (51), movement_stack (52), if_stack (53), unhyphenated (54), hyphenated (55), delta (56), passive (57), shape (58), fake (100).

NOTE: The \lastnodetype primitive is \e-\TeX compliant. The valid range is still -1 .. 15 and glyph nodes have number 0 (used to be char node) and ligature nodes are mapped to 7. That way macro packages can use the same symbolic names as in traditional \e-\TeX. Keep in mind that the internal node numbers are different and that there are more node types than 15.

8.1.1 Auxiliary items

A few node-typed userdata objects do not occur in the ‘normal’ list of nodes, but can be pointed to from within that list. They are not quite the same as regular nodes, but it is easier for the library routines to treat them as if they were.

8.1.1.1 glue_spec items

Skips are about the only type of data objects in traditional \TeX that are not a simple value. The structure that represents the glue components of a skip is called a glue_spec, and it has the following accessible fields:

<table>
<thead>
<tr>
<th>key</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>stretch</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>stretch_order</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>shrink</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>shrink_order</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>writable</td>
<td>boolean</td>
<td>If this is true, you can’t assign to this glue_spec because it is one of the preallocated special cases. New in 0.52</td>
</tr>
</tbody>
</table>
These objects are reference counted, so there is actually an extra read-only field named `ref_count` as well. This item type will likely disappear in the future, and the glue fields themselves will become part of the nodes referencing glue items.

### 8.1.1.2 `attribute_list` and `attribute` items

The newly introduced attribute registers are non-trivial, because the value that is attached to a node is essentially a sparse array of key-value pairs.

It is generally easiest to deal with attribute lists and attributes by using the dedicated functions in the `node` library, but for completeness, here is the low-level interface.

An `attribute_list` item is used as a head pointer for a list of `attribute` items. It has only one user-visible field:

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>next</code></td>
<td><code>&lt;node&gt;</code></td>
<td>pointer to the first attribute</td>
</tr>
</tbody>
</table>

A normal node’s attribute field will point to an item of type `attribute_list`, and the `next` field in that item will point to the first defined ‘attribute’ item, whose `next` will point to the second ‘attribute’ item, etc.

Valid fields in `attribute` items:

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>next</code></td>
<td><code>&lt;node&gt;</code></td>
<td>pointer to the next attribute</td>
</tr>
<tr>
<td><code>number</code></td>
<td>number</td>
<td>the attribute type id</td>
</tr>
<tr>
<td><code>value</code></td>
<td>number</td>
<td>the attribute value</td>
</tr>
</tbody>
</table>

As mentioned it’s better to use the official helpers rather than edit these fields directly. For instance the `prev` field is used for other purposes and there is no double linked list.

### 8.1.1.3 `action` item

Valid fields: `action_type`, `named_id`, `action_id`, `file`, `new_window`, `data`, `ref_count`

Id: 49

These are a special kind of item that only appears inside pdf start link objects.

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>action_type</code></td>
<td>number</td>
<td></td>
</tr>
<tr>
<td><code>action_id</code></td>
<td>number or string</td>
<td></td>
</tr>
<tr>
<td><code>named_id</code></td>
<td>number</td>
<td></td>
</tr>
<tr>
<td><code>file</code></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td><code>new_window</code></td>
<td>number</td>
<td></td>
</tr>
<tr>
<td><code>data</code></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td><code>ref_count</code></td>
<td>number</td>
<td>(read-only)</td>
</tr>
</tbody>
</table>
8.1.2 Main text nodes

These are the nodes that comprise actual typesetting commands.

A few fields are present in all nodes regardless of their type, these are:

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>&lt;node&gt;</td>
<td>The next node in a list, or nil</td>
</tr>
<tr>
<td>id</td>
<td>number</td>
<td>The node's type (id) number</td>
</tr>
<tr>
<td>subtype</td>
<td>number</td>
<td>The node subtype identifier</td>
</tr>
</tbody>
</table>

The subtype is sometimes just a stub entry. Not all nodes actually use the subtype, but this way you can be sure that all nodes accept it as a valid field name, and that is often handy in node list traversal. In the following tables next and id are not explicitly mentioned.

Besides these three fields, almost all nodes also have an attr field, and there is also a field called prev. That last field is always present, but only initialized on explicit request: when the function node.slide() is called, it will set up the prev fields to be a backwards pointer in the argument node list.

8.1.2.1 hlist nodes

Valid fields: attr, width, depth, height, dir, shift, glue_order, glue_sign, glue_set, head

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = unknown origin, 1 = created by linebreaking, 2 = explicit box command.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.46.0), 3 = paragraph indentation box, 4 = alignment column or row, 5 =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>alignment cell (0.62.0)</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td>The head of the associated attribute list</td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>shift</td>
<td>number</td>
<td>a displacement perpendicular to the character progression direction</td>
</tr>
<tr>
<td>glue_order</td>
<td>number</td>
<td>a number in the range 0–4, indicating the glue order</td>
</tr>
<tr>
<td>glue_set</td>
<td>number</td>
<td>the calculated glue ratio</td>
</tr>
<tr>
<td>glue_sign</td>
<td>number</td>
<td>0 = normal, 1 = stretching, 2 = shrinking</td>
</tr>
<tr>
<td>head</td>
<td>&lt;node&gt;</td>
<td>the first node of the body of this list</td>
</tr>
<tr>
<td>dir</td>
<td>string</td>
<td>the direction of this box. see 8.1.4.7</td>
</tr>
</tbody>
</table>

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may result.

Note: the new field name head was introduced in 0.65 to replace the old name list. Use of the name list is now deprecated, but it will stay available until at least version 0.80.
8.1.2.2 vlist nodes

Valid fields: As for hlist, except that ‘shift’ is a displacement perpendicular to the line progression direction, and ‘subtype’ only has subtypes 0, 4, and 5.

8.1.2.3 rule nodes

Valid fields: attr, width, depth, height, dir
Id: 2

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>unused</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>the width of the rule; the special value −1073741824 is used for ‘running’ glue dimensions</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>the height of the rule (can be negative)</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>the depth of the rule (can be negative)</td>
</tr>
<tr>
<td>dir</td>
<td>string</td>
<td>the direction of this rule. see 8.1.4.7</td>
</tr>
</tbody>
</table>

8.1.2.4 ins nodes

Valid fields: attr, cost, depth, height, spec, head
Id: 3

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>the insertion class</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>cost</td>
<td>number</td>
<td>the penalty associated with this insert</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>head</td>
<td>&lt;node&gt;</td>
<td>the first node of the body of this insert</td>
</tr>
<tr>
<td>spec</td>
<td>&lt;node&gt;</td>
<td>a pointer to the \Splittopskip glue spec</td>
</tr>
</tbody>
</table>

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may be result.

Note: the new field name head was introduced in 0.65 to replace the old name list. Use of the name list is now deprecated, but it will stay available until at least version 0.80.

8.1.2.5 mark nodes

Valid fields: attr, class, mark
Id: 4
<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>unused</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>class</td>
<td>number</td>
<td>the mark class</td>
</tr>
<tr>
<td>mark</td>
<td>table</td>
<td>a table representing a token list</td>
</tr>
</tbody>
</table>

8.1.2.6 adjust nodes

Valid fields: attr, head

Id: 5

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = normal, 1 = ‘pre’</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td>adjusted material</td>
</tr>
<tr>
<td>head</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

A warning: never assign a node list to the head field unless you are sure its internal link structure is correct, otherwise an error may be result.

Note: the new field name head was introduced in 0.65 to replace the old name list. Use of the name list is now deprecated, but it will stay available until at least version 0.80.

8.1.2.7 disc nodes

Valid fields: attr, pre, post, replace

Id: 7

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>indicates the source of a discretionary. 0 = the \discretionary command, 1 = the - command, 2 = added automatically following a -, 3 = added by the hyphenation algorithm (simple), 4 = added by the hyphenation algorithm (hard, first item), 5 = added by the hyphenation algorithm (hard, second item)</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td>pointer to the pre-break text</td>
</tr>
<tr>
<td>pre</td>
<td>&lt;node&gt;</td>
<td>pointer to the post-break text</td>
</tr>
<tr>
<td>post</td>
<td>&lt;node&gt;</td>
<td>pointer to the no-break text</td>
</tr>
</tbody>
</table>

The subtype numbers 4 and 5 belong to the ‘of-f-ice’ explanation given elsewhere.

A warning: never assign a node list to the pre, post or replace field unless you are sure its internal link structure is correct, otherwise an error may be result.

8.1.2.8 math nodes

Valid fields: attr, surround

Id: 9
8.1.2.9 glue nodes

Valid fields: attr, spec, leader

Id: 10

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = \skip, 1–18 = internal glue parameters, 100-103 = ‘leader’ subtypes</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>spec</td>
<td>&lt;node&gt;</td>
<td>pointer to a glue_spec item</td>
</tr>
<tr>
<td>leader</td>
<td>&lt;node&gt;</td>
<td>pointer to a box or rule for leaders</td>
</tr>
</tbody>
</table>

The exact meanings of the subtypes are as follows:

1 \lineskip
2 \baselineskip
3 \parskip
4 \abovedisplayskip
5 \belowdisplayskip
6 \abovedisplayshortskip
7 \belowdisplayshortskip
8 \leftskip
9 \rightskip
10 \topskip
11 \split topskip
12 \tab skip
13 \spaceskip
14 \xspaceskip
15 \parfillskip
16 \thinsk uskip
17 \medmuskip
18 \thickmuskip
100 \leaders
101 \cleaders
102 \xleaders
103 \gleaders

8.1.2.10 kern nodes

Valid fields: attr, kern, expansion_factor

Id: 11
### 8.1.2.11 penalty nodes

Valid fields: **attr, penalty**

Id: 12

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>not used</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>penalty</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

### 8.1.2.12 glyph nodes

Valid fields: **attr, char, font, lang, left, right, uchyp, components, xoffset, yoffset, width, height, depth, expansion_factor**

Id: 37

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>bitfield</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>font</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>lang</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>right</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>uchyp</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>components</td>
<td>&lt;node&gt;</td>
<td>pointer to ligature components</td>
</tr>
<tr>
<td>xoffset</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>yoffset</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td>(new in 0.53)</td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td>(new in 0.53)</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>(new in 0.53)</td>
</tr>
<tr>
<td>expansion_factor</td>
<td>number</td>
<td>(new in 0.78)</td>
</tr>
</tbody>
</table>

A warning: never assign a node list to the components field unless you are sure its internal link structure is correct, otherwise an error may be result.

Valid bits for the **subtype** field are:

<table>
<thead>
<tr>
<th>bit</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>character</td>
</tr>
</tbody>
</table>
See section 6.1 for a detailed description of the *subtype* field.

The *expansion_factor* is relatively new and the result of extensive experiments with a more efficient implementation of expansion. Early versions of LuaTeX already replaced multiple instances of fonts in the backend by scaling but contrary to PDFTeX in LuaTeX we now also got rid of font copies in the frontend and replaced them by expansion factors that travel with glyph nodes. Apart from a cleaner approach this is also a step towards a better separation between front- and backend.

### 8.1.2.13 margin_kern nodes

Valid fields: *attr*, *width*, *glyph*

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>0 = left side, 1 = right side</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>glyph</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

### 8.1.3 Math nodes

These are the so-called ‘noad’s and the nodes that are specifically associated with math processing. Most of these nodes contain sub-nodes so that the list of possible fields is actually quite small. First, the subnodes:

#### 8.1.3.1 Math kernel subnodes

Many object fields in math mode are either simple characters in a specific family or math lists or node lists. There are four associated subnodes that represent these cases (in the following node descriptions these are indicated by the word `<kernel>`).

The *next* and *prev* fields for these subnodes are unused.

##### 8.1.3.1.1 math_char and math_text_char subnodes

Valid fields: *attr*, *fam*, *char*

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>
The **math_char** is the simplest subnode field, it contains the character and family for a single glyph object. The **math_text_char** is a special case that you will not normally encounter, it arises temporarily during math list conversion (its sole function is to suppress a following italic correction).

### 8.1.3.1.2 sub_box and sub_mlist subnodes

Valid fields: **attr**, **head**

Id: 32

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>head</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

These two subnode types are used for subsidiary list items. For **sub_box**, the **head** points to a ‘normal’ vbox or hbox. For **sub_mlist**, the **head** points to a math list that is yet to be converted.

A warning: never assign a node list to the **head** field unless you are sure its internal link structure is correct, otherwise an error may be result.

Note: the new field name **head** was introduced in 0.65 to replace the old name **list**. Use of the name **list** is now deprecated, but it will stay available until at least version 0.80.

### 8.1.3.2 Math delimiter subnode

There is a fifth subnode type that is used exclusively for delimiter fields. As before, the **next** and **prev** fields are unused.

#### 8.1.3.2.1 delim subnodes

Valid fields: **attr**, **small_fam**, **small_char**, **large_fam**, **large_char**

Id: 35

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>small_char</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>small_fam</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>large_char</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>large_fam</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

The fields **large_char** and **large_fam** can be zero, in that case the font that is sed for the **small_fam** is expected to provide the large version as an extension to the **small_char**.
8.1.3.3 Math core nodes

First, there are the objects (the \TeXbook calls then 'atoms') that are associated with the simple math objects: Ord, Op, Bin, Rel, Open, Close, Punct, Inner, Over, Under, Vcent. These all have the same fields, and they are combined into a single node type with separate subtypes for differentiation.

8.1.3.3.1 simple nodes

Valid fields: attr, nucleus, sub, sup

Id: 16

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>see below</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>nucleus</td>
<td>&lt;kernel&gt;</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>&lt;kernel&gt;</td>
<td></td>
</tr>
<tr>
<td>sup</td>
<td>&lt;kernel&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Operators are a bit special because they occupy three subtypes. subtype.

<table>
<thead>
<tr>
<th>number</th>
<th>node sub type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ord</td>
</tr>
<tr>
<td>1</td>
<td>Op, \displaylimits</td>
</tr>
<tr>
<td>2</td>
<td>Op, \limits</td>
</tr>
<tr>
<td>3</td>
<td>Op, \nolimits</td>
</tr>
<tr>
<td>4</td>
<td>Bin</td>
</tr>
<tr>
<td>5</td>
<td>Rel</td>
</tr>
<tr>
<td>6</td>
<td>Open</td>
</tr>
<tr>
<td>7</td>
<td>Close</td>
</tr>
<tr>
<td>8</td>
<td>Punct</td>
</tr>
<tr>
<td>9</td>
<td>Inner</td>
</tr>
<tr>
<td>10</td>
<td>Under</td>
</tr>
<tr>
<td>11</td>
<td>Over</td>
</tr>
<tr>
<td>12</td>
<td>Vcent</td>
</tr>
</tbody>
</table>

8.1.3.3.2 accent nodes

Valid fields: attr, nucleus, sub, sup, accent, bot_accent

Id: 28

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

The first bit is used for a fixed top accent flag (if the accent field is present), the second bit for a fixed bottom accent flag (if the bot_accent field is present). Example: the actual value 3 means: do not stretch either accent.
8.1.3.3 style nodes

Valid fields: attr, style
Id: 14

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>style</td>
<td>string</td>
<td>contains the style</td>
</tr>
</tbody>
</table>

There are eight possibilities for the string value: one of 'display', 'text', 'script', or 'scriptscript'. Each of these can have a trailing ' ' to signify 'cramped' styles.

8.1.3.4 choice nodes

Valid fields: attr, display, text, script, scriptscript
Id: 15

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>display</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>text</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>script</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>scriptscript</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

A warning: never assign a node list to the display, text, script, or scriptscript field unless you are sure its internal link structure is correct, otherwise an error may be result.

8.1.3.5 radical nodes

Valid fields: attr, nucleus, sub, sup, left, degree
Id: 24

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>nucleus</td>
<td>&lt;kernel&gt;</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>&lt;kernel&gt;</td>
<td></td>
</tr>
<tr>
<td>sup</td>
<td>&lt;kernel&gt;</td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>&lt;delim&gt;</td>
<td></td>
</tr>
<tr>
<td>degree</td>
<td>&lt;kernel&gt;</td>
<td>Only set by \Uroot</td>
</tr>
</tbody>
</table>

nucleus  <kernel>
sub      <kernel>
sup      <kernel>
accent   <kernel>
bot_accent <kernel>
A warning: never assign a node list to the nucleus, sub, sup, left, or degree field unless you are sure its internal link structure is correct, otherwise an error may be result.

### 8.1.3.3.6 fraction nodes

Valid fields: attr, width, num, denom, left, right

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>num</td>
<td>&lt;kernel&gt;</td>
<td></td>
</tr>
<tr>
<td>denom</td>
<td>&lt;kernel&gt;</td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>&lt;delim&gt;</td>
<td></td>
</tr>
<tr>
<td>right</td>
<td>&lt;delim&gt;</td>
<td></td>
</tr>
</tbody>
</table>

A warning: never assign a node list to the num, or denom field unless you are sure its internal link structure is correct, otherwise an error may be result.

### 8.1.3.3.7 fence nodes

Valid fields: attr, delim

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtype</td>
<td>number</td>
<td>1 = \left, 2 = \middle, 3 = \right</td>
</tr>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>delim</td>
<td>&lt;delim&gt;</td>
<td></td>
</tr>
</tbody>
</table>

### 8.1.4 whatsit nodes

Whatsit nodes come in many subtypes that you can ask for by running node.whatsits(): open (0), write (1), close (2), special (3), local_par (6), dir (7), pdf_literal (8), pdf_refobj (10), pdf_refxform (12), pdf_refximage (14), pdf_annot (15), pdf_start_link (16), pdf_end_link (17), pdf_dest (19), pdf_thread (20), pdf_start_thread (21), pdf_end_thread (22), pdf_save_pos (23), pdf_thread_data (24), pdf_link_data (25), late_lua (35), close_lua (36), pdf_colorstack (39), pdf_setmatrix (40), pdf_save (41), pdf_restore (42), cancel_boundary (43), user_defined (44)

### 8.1.4.1 open nodes

Valid fields: attr, stream, name, area, ext

Id: 8, 0
field  type  explanation
attr  <node>
stream number  T\TeX{}’s stream id number
name    string  file name
ext     string  file extension
area    string  file area (this may become obsolete)

8.1.4.2 write nodes
Valid fields: attr, stream, data
Id: 8, 1

field  type  explanation
attr  <node>
stream number  T\TeX{}’s stream id number
data    table   a table representing the token list to be written

8.1.4.3 close nodes
Valid fields: attr, stream
Id: 8, 2

field  type  explanation
attr  <node>
stream number  T\TeX{}’s stream id number

8.1.4.4 special nodes
Valid fields: attr, data
Id: 8, 3

field  type  explanation
attr  <node>
data    string   the \texttt{\special} information

8.1.4.5 language nodes
Lua\TeX{} does not have language whatsits any more. All language information is already present inside the glyph nodes themselves. This whatsit subtype will be removed in the next release.

8.1.4.6 local_par nodes
Valid fields: attr, pen_inter, pen_broken, dir, box_left, box_left_width, box_right, box_right_width
Id: 8, 6
### 8.1.4.7 dir nodes

Valid fields: attr, dir, level, dvi_ptr, dvi_h

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>dir</td>
<td>string</td>
<td>the direction (but see below)</td>
</tr>
<tr>
<td>level</td>
<td>number</td>
<td>nesting level of this direction whatsit</td>
</tr>
<tr>
<td>dvi_ptr</td>
<td>number</td>
<td>a saved dvi_buffer byte offset</td>
</tr>
<tr>
<td>dir_h</td>
<td>number</td>
<td>a saved dvi_position</td>
</tr>
</tbody>
</table>

A note on dir strings. Direction specifiers are three-letter combinations of T, B, R, and L.

These are built up out of three separate items:

- the first is the direction of the 'top' of paragraphs.
- the second is the direction of the 'start' of lines.
- the third is the direction of the 'top' of glyphs.

However, only four combinations are accepted: TLT, TRT, RTT, and LTL.

Inside actual dir whatsit nodes, the representation of dir is not a three-letter but a four-letter combination. The first character in this case is always either + or −, indicating whether the value is pushed or popped from the direction stack.

### 8.1.4.8 pdf_literal nodes

Valid fields: attr, mode, data

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>
mode number the 'mode' setting of this literal
data string the \pdfliteral information

Mode values:

value corresponding \pdftex keyword
0 setorigin
1 page
2 direct

8.1.4.9 pdf_refobj nodes

Valid fields: attr, objnum
Id: 8, 10

field type explanation
attr <node>
objnum number the referenced PDF object number

8.1.4.10 pdf_refxform nodes

Valid fields: attr, width, depth, height, objnum
Id: 8, 12.

field type explanation
attr <node>
width number
height number
depth number
objnum number the referenced PDF object number

Be aware that pdf_refxform nodes have dimensions that are used by \LaTeX.

8.1.4.11 pdf_refximage nodes

Valid fields: attr, width, depth, height, transform, index
Id: 8, 14

field type explanation
attr <node>
width number
height number
depth number
objnum number the referenced PDF object number
Be aware that `pdf_refximage` nodes have dimensions that are used by LuaTEX.

### 8.1.4.12 pdf_annot nodes

Valid fields: `attr`, `width`, `depth`, `height`, `objnum`, `data`

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>objnum</td>
<td>number</td>
<td>the referenced PDF object number</td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>the annotation data</td>
</tr>
</tbody>
</table>

### 8.1.4.13 pdf_start_link nodes

Valid fields: `attr`, `width`, `depth`, `height`, `objnum`, `link_attr`, `action`

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>objnum</td>
<td>number</td>
<td>the referenced PDF object number</td>
</tr>
<tr>
<td>link_attr</td>
<td>table</td>
<td>the link attribute token list</td>
</tr>
<tr>
<td>action</td>
<td>&lt;node&gt;</td>
<td>the action to perform</td>
</tr>
</tbody>
</table>

### 8.1.4.14 pdf_end_link nodes

Valid fields: `attr`

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

### 8.1.4.15 pdf_dest nodes

Valid fields: `attr`, `width`, `depth`, `height`, `named_id`, `dest_id`, `dest_type`, `xyz_zoom`, `objnum`

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>
width         number
height        number
depth         number
named_id      number       is the dest_id a string value?
dest_id       number or string  the destination id
dest_type     number       type of destination
xyz_zoom      number
objnum        number       the PDF object number

8.1.4.16 pdf_thread nodes

Valid fields: attr, width, depth, height, named_id, thread_id, thread_attr
Id: 8, 20

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>named_id</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>tread_id</td>
<td>number or string</td>
<td></td>
</tr>
<tr>
<td>thread_attr</td>
<td>number</td>
<td>extra thread information</td>
</tr>
</tbody>
</table>

8.1.4.17 pdf_start_thread nodes

Valid fields: attr, width, depth, height, named_id, thread_id, thread_attr
Id: 8, 21

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>named_id</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>tread_id</td>
<td>number or string</td>
<td></td>
</tr>
<tr>
<td>thread_attr</td>
<td>number</td>
<td>extra thread information</td>
</tr>
</tbody>
</table>

8.1.4.18 pdf_end_thread nodes

Valid fields: attr
Id: 8, 22

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>
8.1.4.19 pdf_save_pos nodes

Valid fields: attr
Id: 8, 23

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

8.1.4.20 late_lua nodes

Valid fields: attr, reg, data, name, string
Id: 8, 35

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>data to execute</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>data to execute (0.63)</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>the name to use for lua error reporting</td>
</tr>
</tbody>
</table>

The difference between data and string is that on assignment, the data field is converted to a token list, cf. use as \latelua. The string version is treated as a literal string.

8.1.4.21 pdf_colorstack nodes

Valid fields: attr, stack, cmd, data
Id: 8, 39

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>number</td>
<td>colorstack id number</td>
</tr>
<tr>
<td>cmd</td>
<td>number</td>
<td>command to execute</td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>data</td>
</tr>
</tbody>
</table>

8.1.4.22 pdf_setmatrix nodes

Valid fields: attr, data
Id: 8, 40

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>string</td>
<td>data</td>
</tr>
</tbody>
</table>
8.1.4.23 pdf_save nodes

Valid fields: attr
Id: 8, 41

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

8.1.4.24 pdf_restore nodes

Valid fields: attr
Id: 8, 42

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
</tbody>
</table>

8.1.4.25 user_defined nodes

User-defined whatsit nodes can only be created and handled from Lua code. In effect, they are an extension to the extension mechanism. The LuaTeX engine will simply step over such whatsit nodes without ever looking at the contents.

Valid fields: attr, user_id, type, value
Id: 8, 44

<table>
<thead>
<tr>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>attr</td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td>user_id</td>
<td>number</td>
<td>id number</td>
</tr>
<tr>
<td>type</td>
<td>number</td>
<td>type of the value</td>
</tr>
<tr>
<td>value</td>
<td>number</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;node&gt; table</td>
</tr>
</tbody>
</table>

The type can have one of five distinct values:

<table>
<thead>
<tr>
<th>value</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
<td>the value is an attribute node list</td>
</tr>
<tr>
<td>100</td>
<td>the value is a number</td>
</tr>
<tr>
<td>110</td>
<td>the value is a node list</td>
</tr>
<tr>
<td>115</td>
<td>the value is a string</td>
</tr>
<tr>
<td>116</td>
<td>the value is a token list in Lua table form</td>
</tr>
</tbody>
</table>
8.2 Two access models

After doing lots of tests with \texttt{Lua\TeX} and \texttt{LuaApply\TeX} with and without just in time compilation enabled, and with and without using \texttt{ffi}, we came to the conclusion that userdata prevents a speedup. We also found that the checking of metatables as well as assignment comes with overhead that can’t be neglected. This is normally not really a problem but when processing fonts for more complex scripts it could have quite some overhead.

Because the userdata approach has some benefits, this remains the recommended way to access nodes. We did several experiments with faster access using this model, but eventually settled for the ‘direct’ approach. For code that is proven to be okay, one can use this access model that operates on nodes more directly.

Deep down in \TeX a node has a number which is an entry in a memory table. In fact, this model, where \TeX manages memory is real fast and one of the reasons why plugging in callbacks that operate on nodes is quite fast. No matter what future memory model \texttt{Lua\TeX} has, an internal reference will always be a simple data type (like a number or light userdata in \texttt{Lua} speak). So, if you use the direct model, even if you know that you currently deal with numbers, you should not depend on that property but treat it an abstraction just like traditional nodes. In fact, the fact that we use a simple basic datatype has the penalty that less checking can be done, but less checking is also the reason why it’s somewhat faster. An important aspect is that one cannot mix both methods, but you can cast both models.

So our advice is: use the indexed approach when possible and investigate the direct one when speed might be an issue. For that reason we also provide the \texttt{get*} and \texttt{set*} functions in the top level node namespace. There is a limited set of getters. When implementing this direct approach the regular index by key variant was also optimized, so direct access only makes sense when we’re accessing nodes millions of times (which happens in some font processing for instance).

We’re talking mostly of getters because setters are less important. Documents have not that many content related nodes and setting many thousands of properties is hardly a burden contrary to millions of consultations.

Normally you will access nodes like this:

```lua
local next = current.next
if next then
    -- do something
end
```

Here \texttt{next} is not a real field, but a virtual one. Accessing it results in a metatable method being called. In practice it boils down to looking up the node type and based on the node type checking for the field name. In a worst case you have a node type that sits at the end of the lookup list and a field that is last in the lookup chain. However, in successive versions of \texttt{Lua\TeX} these lookups have been optimized and the most frequently accessed nodes and fields have a higher priority.

Because in practice the \texttt{next} accessor results in a function call, there is some overhead involved. The next code does the same and performs a tiny bit faster (but not that much because it is still a function call but one that knows what to look up).
local next = node.next(current)
if next then
    -- do something
end

There are several such function based accessors now:

getnext  parsing nodelist always involves this one
getprev  used less but is logical companion to getnext
getid    consulted a lot
getsubtype consulted less but also a topper
getfont  used a lot in of handling (glyph nodes are consulted a lot)
getchar  idem and also in other places
getlist  we often parse nested lists so this is a convenient one too (only works for hlist and vlist!)
getleader comparable to list, seldom used in TeX (but needs frequent consulting like lists; leaders could have been made a dedicated node type)
getfield generic getter, sufficient for the rest (other field names are often shared so a specific getter makes no sense then)

It doesn’t make sense to add more. Profiling demonstrated that these fields can get accesses way more times than other fields. Even in complex documents, many node and fields types never get seen, or seen only a few times. Most functions in the node namespace have a companion in node.direct, but of course not the ones that don’t deal with nodes themselves. The following table summarized this:

<table>
<thead>
<tr>
<th>function</th>
<th>node</th>
<th>direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>copy_list</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>count</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>current_attr</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>dimensions</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>do_ligature_n</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>end_of_math</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>family_font</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>fields</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>first_character</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>first_glyph</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>flush_list</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>flush_node</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>free</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getbox</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>getchar</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getfield</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getfont</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getid</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>getnext</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Function</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>-------------------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>getprev</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getlist</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getleader</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getsubtype</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_glyph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_attribute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>has_field</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hpuck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>insert_after</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert_before</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_direct</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>is_node</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kerning</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>last_node</td>
<td></td>
<td></td>
</tr>
<tr>
<td>length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ligature</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mlist_to_hlist</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>new</td>
<td></td>
<td></td>
</tr>
<tr>
<td>next</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>prev</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>tostring</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>protect_glyphs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>protrusion_skippable</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set_attribute</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>setbox</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>setfield</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>slide</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>subtype</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>tail</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>todirective</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>tonode</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>traverse</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>traverse_id</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>type</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>types</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>unprotect_glyphs</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>unset_attribute</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>usedlist</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>vpack</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>whatsits</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>write</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
The `node.next` and `node.prev` functions will stay but for consistency there are variants called `get-next` and `getprev`. We had to use `get` because `node.id` and `node.subtype` are already taken for providing meta information about nodes. Note: The getters do only basic checking for valid keys. You should just stick to the keys mentioned in the sections that describe node properties.
9 Modifications

Besides the expected changes caused by new functionality, there are a number of not-so-expected changes. These are sometimes a side-effect of a new (conflicting) feature, or, more often than not, a change necessary to clean up the internal interfaces.

9.1 Changes from TEX 3.1415926

- The current code base is written in C, not Pascal web (as of LuaTEX 0.42.0).
- See chapter 6 for many small changes related to paragraph building, language handling, and hyphenation. Most important change: adding a brace group in the middle of a word (like in office) does not prevent ligature creation.
- There is no pool file, all strings are embedded during compilation.
- plus 1 fillll does not generate an error. The extra ‘l’ is simply typeset.
- The upper limit to \endlinechar and \newlinechar is 127.

9.2 Changes from $\varepsilon$-TEX 2.2

- The $\varepsilon$-TEX functionality is always present and enabled (but see below about $\TeXXeT$), so the prepended asterisk or -etex switch for initEX is not needed.
- $\TeXXeT$ is not present, so the primitives
  \begin{verbatim}
  \TeXXeTstate
  \beginR
  \beginL
  \endR
  \endL
  \end{verbatim}

  are missing.
- Some of the tracing information that is output by $\varepsilon$-TEX’s \tracingassigns and \tracingrestores is not there.
- Register management in \LaTeX uses the ALEPH model, so the maximum value is 65535 and the implementation uses a flat array instead of the mixed flat&sparse model from $\varepsilon$-TEX.
- savinghyphcodes is a no-op. See chapter 6 for details.
- When kpathsea is used to find files, \LaTeX uses the ofm file format to search for font metrics. In turn, this means that \LaTeX looks at the OFMFONTS configuration variable (like OMEGA and ALEPH) instead of TFMFONTS (like TEX and PDFTEX). Likewise for virtual fonts (\LaTeX uses the variable OVFFONTS instead of VFFONTS).

9.3 Changes from PDFTEX 1.40
• The (experimental) support for snap nodes has been removed, because it is much more natural to build this functionality on top of node processing and attributes. The associated primitives that are now gone are: \pdfsnaprefpoint, \pdfsnapy, and \pdfsnapycomp.

• The (experimental) support for specialized spacing around nodes has also been removed. The associated primitives that are now gone are: \pdfadjustinterwordglue, \pdfprependkern, and \pdfappendkern, as well as the five supporting primitives \knbscode, \stbscode, \shbscode, \knbrcode, and \knaccode.

• A number of ‘utility functions’ is removed:

\pdfelapsedtime \pdffilemoddate \pdfresettimer
\pdfescapehex \pdffilesize \pdfshellescape
\pdfescapename \pdflastmatch \pdfstrcmp
\pdfescapestring \pdfmatch \pdfunescapehex
\pdffiledump \pdfmdfivesum

• The four primitives that were already marked obsolete in PDF\TeX{} 1.40 have been removed since Lua\TeX{} 0.42:

\pdfoptionalwaysusepdfpagebox \pdfforcepagebox
\pdfoptionpdfinclusionerrorlevel \pdfmovechars

• A few other experimental primitives are also provided without the extra pdf prefix, so they can also be called like this:

\primitive \ifabsnum
\ifprimitive \ifabsdim

• The \pdftexversion is set to 200.

• The PNG transparency fix from 1.40.6 is not applied (high-level support is pending)

• LFS (PDF Files larger than 2GiB) support is not working yet.

• Lua\TeX{} 0.45.0 introduces two extra token lists, \pdfxformresources and \pdfxformattr, as an alternative to \pdfxform keywords.

• As of Lua\TeX{} 0.50.0 it is no longer possible for fonts from embedded pdf files to be replaced by merged with the document fonts of the enveloping pdf document. This regression may be temporary, depending on how the rewritten font backend will look after beta 0.60.

9.4 Changes from ALEPH RC4

• Starting with Lua\TeX{} 0.75.0, the extended 16-bit math primitives (\omathcode etc.) have been removed.

• Starting with Lua\TeX{} 0.63.0, OCP processing is no longer supported at all. As a consequence, the following primitives have been removed:

\ocp \pushocplist
\externalocp \popocplist
\ocplist \clearocplists
\addbeforeocplist \removeafterocplist
\addafterocplist \ocptracelevel
\removebeforeocplist

- **\LaTeX only understands 4 of the 16 direction specifiers of \texttt{ALEPH}: TLT (latin), TRT (arabic), RTT (cjk), LTL (mongolian). All other direction specifiers generate an error (\LaTeX 0.45).**
- **The input translations from \texttt{ALEPH} are not implemented, the related primitives are not available:**

  \begin{verbatim}
  \DefaultInputMode \DefaultInputTranslation
  \noDefaultInputMode \noDefaultInputTranslation
  \InputMode \InputTranslation
  \DefaultOutputMode \DefaultOutputTranslation
  \noDefaultOutputMode \noDefaultOutputTranslation
  \OutputMode \OutputTranslation
  \end{verbatim}

- **The \texttt{\hoffset} bug when \texttt{\pagedir TRT} is fixed, removing the need for an explicit fix to \texttt{\hoffset}**
- **A bug causing \texttt{\fam} to fail for family numbers above 15 is fixed.**
- **A fair amount of other minor bugs are fixed as well, most of these related to \texttt{\tracingcommands} output.**
- **The internal function \texttt{scan\_dir()} has been renamed to \texttt{scan\_direction()} to prevent a naming clash, and it now allows an optional space after the direction is completely parsed.**
- **The \texttt{^^} notation can come in five and six item repetitions also, to insert characters that do not fit in the BMP.**
- **Glues immediately after direction change commands are not legal breakpoints.**

### 9.5 Changes from standard WEB2C

- **There is no mltex**
- **There is no enctex**
- **The following commandline switches are silently ignored, even in non-LUA mode:**

\begin{verbatim}
  -8bit
  -translate-file=TCXNAME
  -mltex
  -enc
  -etex
\end{verbatim}

- **\texttt{\openout} whatsits are not written to the log file.**
- **Some of the so-called web2c extensions are hard to set up in non-KPSE mode because texmf.cnf is not read: \texttt{shell-escape} is off (but that is not a problem because of LUA's \texttt{os.execute}), and the paranoia checks on \texttt{openin} and \texttt{openout} do not happen (however, it is easy for a LUA script to do this itself by overloading \texttt{io.open}).**
• The ‘E’ option does not do anything useful.
10 Implementation notes

10.1 Primitives overlap

The primitives

\pdfpagewidth \pagewidth
\pdfpageheight \pageheight
\fontcharwd \charwd
\fontcharht \charht
\fontchardp \chardp
\fontcharic \charit

are all aliases of each other.

10.2 Memory allocation

The single internal memory heap that traditional \TeX{} used for tokens and nodes is split into two separate arrays. Each of these will grow dynamically when needed.

The \texttt{texmf.cnf} settings related to main memory are no longer used (these are: \texttt{main_memory}, \texttt{mem_bot}, \texttt{extra_mem_top} and \texttt{extra_mem_bot}). ‘Out of main memory’ errors can still occur, but the limiting factor is now the amount of RAM in your system, not a predefined limit.

Also, the memory (de)allocation routines for nodes are completely rewritten. The relevant code now lives in the C file \texttt{texnode.c}, and basically uses a dozen or so ‘avail’ lists instead of a doubly-linked model. An extra function layer is added so that the code can ask for nodes by type instead of directly requisitioning a certain amount of memory words.

Because of the split into two arrays and the resulting differences in the data structures, some of the macros have been duplicated. For instance, there are now \texttt{vlink} and \texttt{vinfo} as well as \texttt{token_link} and \texttt{token_info}. All access to the variable memory array is now hidden behind a macro called \texttt{vmem}.

The implementation of the growth of two arrays (via reallocation) introduces a potential pitfall: the memory arrays should never be used as the left hand side of a statement that can modify the array in question.

The input line buffer and pool size are now also reallocated when needed, and the \texttt{texmf.cnf} settings \texttt{buf_size} and \texttt{pool_size} are silently ignored.

10.3 Sparse arrays

The \texttt{\mathcode}, \texttt{\delcode}, \texttt{\catcode}, \texttt{\sfcode}, \texttt{\lccode} and \texttt{\uccode} tables are now sparse arrays that are implemented in C. They are no longer part of the \TeX{} ‘equivalence table’ and because
each had 1.1 million entries with a few memory words each, this makes a major difference in memory usage.

The \catcode, \scode, \lrcode and \uccode assignments do not yet show up when using the etex tracing routines \tracingassigns and \tracingrestores (code simply not written yet).

A side-effect of the current implementation is that \global is now more expensive in terms of processing than non-global assignments.

See mathcodes.c and textcodes.c if you are interested in the details.

Also, the glyph ids within a font are now managed by means of a sparse array and glyph ids can go up to index $2^{21} - 1$.

10.4 Simple single-character csnames

Single-character commands are no longer treated specially in the internals, they are stored in the hash just like the multiletter csnames.

The code that displays control sequences explicitly checks if the length is one when it has to decide whether or not to add a trailing space.

Active characters are internally implemented as a special type of multi-letter control sequences that uses a prefix that is otherwise impossible to obtain.

10.5 Compressed format

The format is passed through zlib, allowing it to shrink to roughly half of the size it would have had in uncompressed form. This takes a bit more CPU cycles but much less disk I/O, so it should still be faster.

10.6 Binary file reading

All of the internal code is changed in such a way that if one of the read_xxx_file callbacks is not set, then the file is read by a C function using basically the same convention as the callback: a single read into a buffer big enough to hold the entire file contents. While this uses more memory than the previous code (that mostly used getc calls), it can be quite a bit faster (depending on your I/O subsystem).


11 Known bugs and limitations, TODO

There used to be a lists of bugs and planned features below here, but that did not work out too well. There are lists of open bugs and feature requests in the tracker at http://tracker.luatex.org.