LuaTEX Reference

beta 0.80.1
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 \LaTeX\ 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 \enquote{\enquote{\LaTeX}} 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{\scantexttokens} 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{\matheqnogapstep} (0.81) 26
    2.7.13 \texttt{\outputbox} (0.37) 26
2.7.14 Font syntax 26
2.7.15 File syntax (0.45) 27
2.7.16 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 close
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 (0.43)
4.1.3.3 process_jobname (0.71)
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 (0.61)
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_pdffile
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) 75
4.4.5.1.14 math table 75
4.4.5.1.15 validation_state table 77
4.4.5.1.16 horiz_base and vert_base table 77
4.4.5.1.17 altuni table 78
4.4.5.1.18 vert_variants and horiz_variants table 78
4.4.5.1.19 mathkern table 78
4.4.5.1.20 kerns table 78
4.4.5.1.21 vkerns table 79
4.4.5.1.22 texdata table 79
4.4.5.1.23 lookups table 79
4.5 The img library 80
4.5.1 img.new 80
4.5.2 img.keys 81
4.5.3 img.scan 82
4.5.4 img.copy 83
4.5.5 img.write 83
4.5.6 img.immediatewrite 83
4.5.7 img.node 83
4.5.8 img.types 84
4.5.9 img.boxes 84
4.6 The kpse library 84
4.6.1 kpse.set_program_name and kpse.new 84
4.6.2 find_file 85
4.6.3 lookup 86
4.6.4 init_prog 87
4.6.5 readable_file 87
4.6.6 expand_path 87
4.6.7 expand_var 87
4.6.8 expand_braces 88
4.6.9 show_path 88
4.6.10 var_value 88
4.6.11 version 88
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 124
4.14.9 Semantic nest levels (0.51) 125
4.14.10 Print functions 125
4.14.10.1 `tex.print` 126
4.14.10.2 `tex.sprint` 126
4.14.10.3 `tex.tprint` 127
4.14.10.4 `tex.write` 127
4.14.11 Helper functions 127
4.14.11.1 `tex.round` 127
4.14.11.2 `tex.scale` 127
4.14.11.3 `tex.sp` (0.51) 128
4.14.11.4 `tex.definefont` 128
4.14.11.5 `tex.error` (0.61) 128
4.14.11.6 `tex.hashtokens` (0.25) 128
4.14.12 Functions for dealing with primitives 129
4.14.12.1 `tex.enableprimitives` 129
4.14.12.2 `tex.extraprimitives` 130
4.14.12.3 `tex.primitives` 133
4.14.13 Core functionality interfaces 133
4.14.13.1 `tex.badness` (0.53) 133
4.14.13.2 `tex.linebreak` (0.53) 134
4.14.13.3 `tex.shipout` (0.51) 135
4.15 The `texconfig` table 135
4.16 The `texio` library 137
4.16.1 Printing functions 137
4.16.1.1 `texio.write` 137
4.16.1.2 `texio.write_nl` 137
4.17 The `token` library 137
4.17.1 `token.get_next` 137
4.17.2 `token.is_expandable` 138
4.17.3 `token.expand` 138
4.17.4 `token.is_activechar` 138
4.17.5 `token.create` 138
4.17.6 token.command_name
4.17.7 token.command_id
4.17.8 token.csname_name
4.17.9 token.csname_id
4.17.10 The newtoken library

5 Math
5.1 The current math style
5.1.1 \mathstyle
5.1.2 \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
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
9.1 Changes from \TeX{} 3.1415926
9.2 Changes from e-\TeX{} 2.2
9.3 Changes from PDF\TeX{} 1.40
9.4 Changes from ALEPH RC4
9.5 Changes from standard WEB2C

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

11 Known bugs and limitations, TODO
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:

- PDFTEX 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 ENCTEX can be used, these encoding-related functions are superseded by a LUA-based solution (reader callbacks). Also, some experimental PDFTEX features are removed. These can be implemented in LUA instead.
2 Basic \TeX\ enhancements

2.1 Introduction

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

However, starting with beta 0.39.0, most of that functionality is hidden by default. When \texttt{\LaTeX} 0.40.0 starts up in 'iniluatex' mode (\texttt{luatex -ini}), it defines only the primitive commands known by \TeX82 and the one extra command \texttt{\textbackslash directlua}.

As is fitting, a \texttt{\LaTeX} 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:

\texttt{\textbackslash directlua \{ tex.enableprimitives('',tex.extraprimitives()) \}}

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

\texttt{\textbackslash catcode '={1}
\texttt{\textbackslash catcode '{}=2}

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{\textbackslash 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 \texttt{\LaTeX}:

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

The official \texttt{\LaTeX} version is defined as follows:
• The major version is the integer result of \texttt{\luatexversion} divided by 100. The primitive is an 'internal variable', so you may need to prefix its use with \texttt{\the} depending on the context.
• The minor version is the two-digit result of \texttt{\luatexversion} modulo 100.
• The revision is the given by \texttt{\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 ($2^{20} + 2^{16} - 1 = 0x10FFFF$).

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 \texttt{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, \texttt{\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: \texttt{\char}, \texttt{\lccode}, \texttt{\uccode}, \texttt{\catcode}, \texttt{\sfcode}, \texttt{\efcode}, \texttt{\lpcode}, \texttt{\rpcode}, \texttt{\chardef}.

As far as the core engine is concerned, all input and output to text files is \texttt{UTF-8} encoded. Input files can be pre-processed using the \texttt{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 $c \geq 1,114,112$, \texttt{LuaTeX} will actually print the single byte corresponding to $c$ minus 1,114,112.

Output to the terminal uses ^^ notation for the lower control range ($c < 32$), with the exception of ^^I, ^^J and ^^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 \texttt{\epsilon-\TeX} registers can be 16-bit numbers as in \texttt{AlepH}. The affected commands are:

\begin{verbatim}
\count \toks \toksdef \unhcopy
\dimen \countdef \box \unvcopy
\skip \dimendef \unhbox \wd
\muskip \skipdef \unvbox \ht
\marks \muskipdef \copy \dp
\end{verbatim}
The glyph properties (like `\efcode`) introduced in PDF\TeX{} 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 Lua\TeX{}. 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.

\begin{verbatim}
\attribute ⟨16-bit number⟩ ⟨optional equals⟩ ⟨32-bit number⟩
\attributedef ⟨csname⟩ ⟨optional equals⟩ ⟨16-bit number⟩
\end{verbatim}

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: `−7FFFFFFF` in hexadecimal, a.k.a. `−2147483647` in decimal. It follows that the value `−7FFFFFFF` cannot be used as a legal attribute value, but you can assign `−7FFFFFFF` to 'unset' an attribute. All attributes start out in this 'unset' state in \ini\TeX{} (prior to 0.37, there could not be valid negative attribute values, and the 'unset' value was `−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 Lua\TeX{} 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 attr. An example:

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

This will set the attribute list of box 2 to \attribute2=12, and the attributes of box 0 will be \attribute2=5. As you can
see, assigning the maximum negative value causes an attribute to be ignored.

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

### 2.6 LUA related primitives

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

#### 2.6.1 \directlua

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

\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

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 \the\toks. On the LUA side, each \directlua block is treated as a separate
chunk. In such a chunk you can use the 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 \--\) 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 %), or
change the \TeX\ category codes locally. Another possibility is to say:

\begingroup
\endlinechar=10
\directlua ...
\endgroup

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

The 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 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
\begin{verbatim}
a25b
\end{verbatim}

Here is another example:

$\pi = \directlua{tex.print(math.pi)}$

will result in
\begin{verbatim}
π = 3.1415926535898
\end{verbatim}

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 \texttt{\TeX’s \scantokens}.

For a description of print functions look at 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.

\subsection{\luafn}

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{\luafunction}. This
command is used as follows:

\directlua {
    local t = lua.get_functions_table()
    t[1] = function() tex.print("!") end
    t[2] = function() tex.print("?") end
}
\luafunction1
\luafunction2

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
parsing and thereby give no gain. The function, when called in fact gets one argument, being the index,
so in:

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

the number 8 gets typeset.

\subsection{\texttt{\latelua}}

\texttt{\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{\pdfliteral} and \texttt{\write}. Within the Lua code you can print PDF statements
directly to the PDF file via \texttt{pdf.print}, or you can write to other output streams via \texttt{texio.write} or
simply using lua’s I/O routines.

\texttt{\latelua \langle general text \rangle \latelua name \langle general text \rangle \langle general text \rangle \latelua \langle 16-bit number \rangle \langle general text \rangle}

Expansion of macros etcetera in the final \texttt{<general text>} is delayed until just before the whatsit
is executed (like in \texttt{\write}). With regard to PDF output stream \texttt{\latelua} behaves as \texttt{\pdfliteral page}.
The \texttt{name \langle general text \rangle} and \texttt{\langle 16-bit number \rangle} behave in the same way as they do for \texttt{\directlua}

\subsection{\texttt{\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
is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to n and r respectively. The token sequence is fully expanded.

\luaescapestring \langle general text \rangle

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

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

## 2.7 New $\epsilon$-\TeX{} primitives

### 2.7.1 \clearmarks

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

`\clearmarks \langle 16\text{-}bit\ number \rangle`

### 2.7.2 \noligs and \nokerns

These primitives prohibit ligature and kerning insertion at the time when the initial node list is built by \textit{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.

`\noligs \langle integer \rangle`

`\nokerns \langle integer \rangle`

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

### 2.7.3 \formatname

`\formatname`'s syntax is identical to `\jobname`.

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

### 2.7.4 \scantextokens

The syntax of `\scantextokens` is identical to `\scantokens`. This primitive is a slightly adapted version of $\epsilon$-\TeX{}'s `\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 alignments commands (0.45)

LuATeX 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><code>\alignmark</code></td>
<td>Duplicates the functionality of <code>#</code> inside alignment preambles</td>
</tr>
<tr>
<td><code>\aligntab</code></td>
<td>Duplicates the functionality of <code>&amp;</code> 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 iniTeX.

2.7.6.2 \initcatcodetable

\initcatcodetable (15-bit number)

The primitive `\initcatcodetable` creates a new table with catcodes identical to those defined by iniTeX:

| 0 \ | escape            |
| 5 `\^^M` | return `car_ret` (this name may change) |
| 9 `\^^@` | null `ignore` |
| 10 `<space>` | space `spacer` |
| 11 a – z | letter |
| 11 A – Z | letter |
| 12 everything else | other |
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 \langle 15-bit number \rangle

\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 \LaTeX{} 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 \texttt{\ifx} equal to \texttt{\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 \LaTeX{} will not complain about \texttt{\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 \LaTeX{} will not complain about non-expandable commands appearing in the middle of a \texttt{\ifcsname} expansion. Instead, it will keep getting expanded tokens from the input until it encounters an \texttt{\endcsname} command. Use with care! This command is experimental: if the input expansion is unbalanced wrt. \texttt{\csname ...\endcsname} pairs, the \LaTeX{} 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{\textbackslash 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{\textbackslash outer} was simply ignored. That behavior has now reverted back to be \TeX82-compatible by default.

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

The following setting will permit tokens in a math formula:

\begin{verbatim}
\textbackslash suppressmathparerror = 1
\end{verbatim}

So, the next code is valid then:

\begin{verbatim}
$ x + 1 = a $
\end{verbatim}

### 2.7.12 \texttt{\textbackslash matheqnogapstep} (0.81)

By default \TeX will add one quad between the equation and the number. This is hardcoded. A new primitive can control this:

\begin{verbatim}
\textbackslash matheqnogapstep = 1000
\end{verbatim}

Because a math quad from the math text font is used instead of a dimension, we use a step to control the size. A value of zero will suppress the gap. The step is divided by 1000 which is the usual way to mimmick floating point factors in \TeX.

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

\begin{verbatim}
\textbackslash outputbox = 65535
\end{verbatim}

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.

### 2.7.14 Font syntax

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

\begin{verbatim}
\textbackslash font \textbackslash myfont = \{cmr10\}
\end{verbatim}

This allows for embedded spaces, without the need for double quotes. Macro expansion takes place inside the argument.
\textbf{2.7.15 File syntax (0.45)}

\LaTeX{} will accept a braced argument as a file name:

\verbatiminput{plain}
\verbatiminput{plain}

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

\textbf{2.7.16 Images and Forms}

\LaTeX{} accepts optional dimension parameters for \texttt{\pdfximage} and \texttt{\pdfxform} 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{\pdfximage} without dimensions still provides the image with dimensions defined by \texttt{\pdfximage}. These optional parameters are not implemented for \texttt{\pdfxform}.

\verbatiminput{plain}
\verbatiminput{plain}

\textbf{2.8 Debugging}

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

\textbf{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.

\textbf{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 \texttt{--luaonly} option is given on the commandline, or
- if the executable is named \texttt{texlua} (or \texttt{luatexlua}), or
- if the only non-option argument (file) on the commandline has the extension \texttt{lua} or \texttt{luc}.

In this mode, it will set LUA's \texttt{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 \texttt{--luaonly} option is given on the commandline, or
- if the executable is named \texttt{texluac}

In this mode, LUATEX is exactly like \texttt{lua} from the standalone LUA distribution, except that it does not have the \texttt{-l} switch, and that it accepts (but ignores) the \texttt{--luaonly} switch.

3.1.3 Other commandline processing

When the LUATEX executable starts, it looks for the \texttt{--lua} commandline option. If there is no \texttt{--lua} option, the commandline is interpreted in a similar fashion as in traditional PDF\TeX{} and ALEPH.

The following command-line switches are understood.

\begin{verbatim}
--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)
\end{verbatim}
lua general

--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 \write18\{SHELL COMMAND\}
--enable-write18 enable \write18\{SHELL COMMAND\}
--disable-write18 disable \write18\{SHELL COMMAND\}
--shell-restricted restrict \write18 to a list of commands given in texmf.cnf
--[no-]file-line-error disable/enable file:line:erro 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 typesetting engine, except that LuaTEX has a few extra switches.
If the \texttt{--lua} option is present, \textsc{Lua}\TeX{} will enter an alternative mode of commandline processing in comparison to the standard \textsc{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: \texttt{--safer}, \texttt{--nosocket}, \texttt{--[no-]shell-escape}, \texttt{--enable-write18}, \texttt{--disable-write18}, \texttt{--shell-restricted}, \texttt{--help}, \texttt{--version}, and \texttt{--credits}.

Now it searches for the requested \textsc{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 \texttt{LUATEXDIR}, if that variable is defined in the environment.

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

\begin{verbatim}
library functions
os   execute exec setenv rename remove tmpdir
io   popen output tmpfile
lfs  rmdir mkdir chdir lock touch
\end{verbatim}

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

\texttt{--nosocket} makes the socket library unavailable, so that \textsc{Lua} cannot use networking.

The switches \texttt{--[no-]shell-escape}, \texttt{--[enable|disable]-write18}, and \texttt{--shell-restricted} have the same effects as in \textsc{pdf}\TeX{}, and additionally make \texttt{io.popen()}, \texttt{os.execute}, \texttt{os.exec} and \texttt{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 \textsc{Lua} table \texttt{arg}, beginning with \texttt{arg[0]}, containing the name of the executable. As consequence, the warning about unrecognized option is suppressed.

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 \texttt{tex}, \texttt{token}, \texttt{node} and \texttt{pdf}, are off-limits during the execution of the startup file (they are nilled). Special care is taken that \texttt{texio.write} and \texttt{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 \texttt{\jobname} yet at this point). See chapter 4 for more information about the \textsc{Lua}\TeX{}-specific \textsc{Lua} extension tables.

Everything you do in the \textsc{Lua} initialization script will remain visible during the rest of the run, with the exception of the aforementioned \texttt{tex}, \texttt{token}, \texttt{node} and \texttt{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 \texttt{texconfig} table, and register the callbacks you need.

\textsc{Lua}\TeX{} allows some of the commandline options to be overridden by reading values from the \texttt{texconfig} table at the end of script execution (see the description of the \texttt{texconfig} table later on in this document for more details on which ones exactly).
Unless the \texttt{texconfig} table tells \LaTeX{} not to initialize \texttt{kpathsea} at all (set \texttt{texconfig.kpse_init} to \texttt{false} for that), \LaTeX{} acts on some more commandline options after the initialization script is finished: in order to initialize the built-in \texttt{kpathsea} library properly, \LaTeX{} needs to know the correct program name to use, and for that it needs to check --proname, or --ini and --fmt, if --proname 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 \texttt{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 \texttt{\dimen} or \texttt{\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 \texttt{dll} file of the module itself). We plan to fix that later by switching \LaTeX{} itself to using the 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 \texttt{kpathsea} library to find \texttt{require()}d modules. For this purpose, \texttt{package.searchers[2]} is replaced by a different loader function, that decides at runtime whether to use \texttt{kpathsea} or the built-in core lua function. It uses \texttt{kpathsea} when that is already initialized at that point in time, otherwise it reverts to using the normal \texttt{package.path} loader.

Initialization of \texttt{kpathsea} can happen either implicitly (when \LaTeX{} starts up and the startup script has not set \texttt{texconfig.kpse_init} to false), or explicitly by calling the Lua function \texttt{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, \texttt{package.searchers[3]} is replaced by a different loader function, that decides at runtime whether to use \texttt{kpathsea} or the build-in core lua function. As in the previous paragraph, it uses \texttt{kpathsea} when that is already initialized at that point in time, otherwise it reverts to using the normal \texttt{package.cpath} loader.

This functionality required an extension to \texttt{kpathsea}:

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

  \begin{verbatim}
  CLUAINPUTS=.:$SELFAUTOLOC/lib/{$proname,$engine,}/lua/
  \end{verbatim}

  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 \texttt{lib} directory parallel to \texttt{bin}) would have been nicer, but that is not doable because \TeX{}Live uses a \texttt{bin/<arch>} structure.
In keeping with the other \TeX-like programs in \TeXLive, the two \texttt{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{luatex}, 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 \texttt{LF}, \texttt{CR} or \texttt{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 \texttt{UNICODE} range)
- \texttt{string.utfcharacters(s)} (returns a string with a single \texttt{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 \texttt{UTF-16} encoded data into \texttt{UTF-8}.

Starting with \LaTeX{} 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 \texttt{UNICODE}-aware. For strings in the \texttt{UTF-8} encoding, i.e., strings containing characters above code point 127, the corresponding functions from the \texttt{slnunicode} 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 path name. 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 TEXLUa 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 TEXLUa.

- `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`, `iri`, `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 LuaTEX, we can’t do that, because it makes documents unportable. While LuaTEX is running if forces the following locale settings:

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

### 3.3 LUA modules

**NOTE**: Starting with LuaTEX 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, LuaTEX will switch to loading these modules on demand.

Some modules that are normally external to Lua are statically linked in with LuaTEX, 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 LuaTEX 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.info.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(’aä’)** results in the message *bad argument #1 to 'R' (range must have two characters)*, since to **lpeg**, å is two ‘characters’ (bytes), so aä 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 LUA\TeX\ LUA Libraries

NOTE: Starting with \LATEX\ 0.74, the implied use of the built-in Lua modules \texttt{epdf, fontloader, mplib, and pdfscanner} is deprecated. If you want to use these, please start your source file with a proper \texttt{require} line. In the near future, \LATEX\ 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 \LATEX\ 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 \LATEX\'s hooks to execute Lua code.

The following sections are in alphabetical order.

4.1 The \texttt{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 \LATEX\'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:

\begin{verbatim}
id, error = callback.register (<string> callback_name, <function> func)
id, error = callback.register (<string> callback_name, nil)
id, error = callback.register (<string> callback_name, false)
\end{verbatim}

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

\LATEX\ 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 \texttt{nil} instead of a function for clearing the callback.

For some minor speed gain, you can assign the boolean \texttt{false} to the non-file related callbacks, doing so will prevent \LATEX\ 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 \emph{exactly} what you are doing! This functionality is present since version 0.38.
Currently, callbacks are not dumped into the format file.

\begin{verbatim}
<function> info = callback.list()
\end{verbatim}

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{verbatim}
<function> f = callback.find (callback_name)
\end{verbatim}

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 \texttt{find_read_file} and \texttt{find_write_file}

Your callback function should have the following conventions:

\begin{verbatim}
<string> actual_name = function (<number> id_number, <string> asked_name)
\end{verbatim}

Arguments:

- \texttt{id_number}
  - This number is zero for the log or \texttt{\input} files. For \TeX's \texttt{\read} or \texttt{\write} the number is incremented by one, so \texttt{\read0} becomes 1.
- \texttt{asked_name}
  - This is the user-supplied filename, as found by \texttt{\input}, \texttt{\openin} or \texttt{\openout}.

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 Lua\TeX\ 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 \texttt{find_font_file}

Your callback function should have the following conventions:

\begin{verbatim}
<string> actual_name = function (<string> asked_name)
\end{verbatim}

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

### 4.1.1.3 \texttt{find_output_file}

Your callback function should have the following conventions:

\texttt{<string> actual\_name = function (<string> asked\_name)}

The \texttt{asked\_name} is the PDF or DVI file for writing.

### 4.1.1.4 \texttt{find_format_file}

Your callback function should have the following conventions:

\texttt{<string> actual\_name = function (<string> asked\_name)}

The \texttt{asked\_name} is a format file for reading (the format file for writing is always opened in the current directory).

### 4.1.1.5 \texttt{find_vf\_file}

Like \texttt{find\_font\_file}, but for virtual fonts. This applies to both ALEPH's OVF files and traditional Knuthian VF files.

### 4.1.1.6 \texttt{find_map\_file}

Like \texttt{find\_font\_file}, but for map files.

### 4.1.1.7 \texttt{find_enc\_file}

Like \texttt{find\_font\_file}, but for enc files.

### 4.1.1.8 \texttt{find_sfd\_file}

Like \texttt{find\_font\_file}, but for subfont definition files.

### 4.1.1.9 \texttt{find_pk\_file}

Like \texttt{find\_font\_file}, but for pk bitmap files. The argument \texttt{asked\_name} is a bit special in this case. Its form is

\texttt{<base\_res>dpi/<fontname>.<actual\_res>pk}

So you may be asked for \texttt{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 Lu\TeX \ 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 \LaTeX{} is done with the file. \LaTeX{} 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

\LaTeX{} will run this function whenever it needs a new input line from the file.

\begin{verbatim}
function(<table> env)
    return <string> line
end
\end{verbatim}

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

\LaTeX{} will run this optional function when it decides to close the file.

\begin{verbatim}
function(<table> env)
end
\end{verbatim}

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:

\begin{verbatim}
function(<string> name)
    return <boolean> success, <string> data, <number> data_size
end
\end{verbatim}

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.

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, \LaTeX 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 \LaTeX fetches lexical tokens.

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 \LaTeX 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 \LaTeX 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.

function(<string> extrainfo)
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>\L aT\TeX 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 Lu\TeX 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><code>\hbox</code> in horizontal mode</td>
</tr>
<tr>
<td>adjusted_hbox</td>
<td><code>\hbox</code> in vertical mode</td>
</tr>
<tr>
<td>vbox</td>
<td><code>\vbox</code></td>
</tr>
<tr>
<td>vtop</td>
<td><code>\vtop</code></td>
</tr>
<tr>
<td>align</td>
<td><code>\halign</code> or <code>\valign</code></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><code>\vcenter</code></td>
</tr>
<tr>
<td>local_box</td>
<td><code>\localleftbox</code> or <code>\localrightbox</code></td>
</tr>
<tr>
<td>split_off</td>
<td>top of a <code>\vsplit</code></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.

```lua
function(<node> head, <boolean> is_display)
    return <node> newhead
end
```

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 \texttt{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{\hboxes}.

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

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.

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

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.

```latex
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 \texttt{\output}.

```latex
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

```latex
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

```
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

```
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.

```
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()
```

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

```lua
function()
end
```

This callback replaces the code that prints LuaTeX'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

```lua
function()
end
```

This callback replaces the code that prints LuaTeX's statistics and 'output written to' messages.

### 4.1.5.4 start_page_number

```lua
function()
end
```

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

### 4.1.5.5 stop_page_number

```lua
function()
end
```

Replaces the code that prints the \] at the end of \texttt{\textbackslash shipout}.

### 4.1.5.6 show_error_hook

```lua
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 \texttt{status} table useful.
This callback does not replace any internal code.

### 4.1.5.7 show_error_message

```lua
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

```lua
function()
end
```

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

### 4.1.5.9 start_file

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

This callback replaces the code that prints \texttt{Lua\TeX}'s when a file is opened like \texttt{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 (\texttt{png}, \texttt{pdf}, etc)
4. an embedded font subset
5. a fully embedded font

### 4.1.5.10 stop_file

```lua
function(category)
end
```

This callback replaces the code that prints \texttt{Lua\TeX}'s when a file is closed like the \texttt{)} for regular files.

### 4.1.6 PDF-related callbacks

#### 4.1.6.1 finish_pdffile

```lua
function()
```
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

```lua
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

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

The string `name` is the filename part of the font specification, as given by the user.

The number `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 `id` is the internal number assigned to the font.

The internal structure of the `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 `false` is pointless as it will prevent font loading completely but will nevertheless generate errors.

### 4.2 The epdf library

The `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 \LaTeX{} (and PDF\LaTeX{}), xpdf functionality is being used since long time to embed PDF files. The `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 \LaTeX{}
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 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 `epdf.open()` with file name, e.g.:

```lua
doc = epdf.open("foo.pdf")
```

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

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

There are many different userdata types defined by the epdf library, currently these are AnnotBorderStyle, AnnotBorder, Annots, Annot, Array, Attribute, Catalog, Dict, EmbFile, GString, LinkDest, Links, Link, ObjectStream, Object, PDFDoc, PDFRectangle, Page, Ref, Stream, StructElement, StructTreeRoot, TextSpan, XRefEntry and 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 Page object for page 1:

```lua
pageref = doc:getCatalog():getPageRef(1)
pageobj = doc:getXRef():fetch(pageref.num, pageref.gen)
```

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

All library objects have a `__gc` metamethod for garbage collection. The `__tostring` metamethod gives the type name for each object.

All object constructors:

```lua
<PDFDoc> = epdf.open(<string> PDF filename)
<Annot> = epdf.Annot(<XRef>, <Dict>, <Catalog>, <Ref>)
<Annots> = epdf.Annots(<XRef>, <Catalog>, <Object>)
<Array> = epdf.Array(<XRef>)
ATTRIBUTE = epdf.Attribute(<Type>,<Object>)| epdf.Attribute(<string>, <int>, <Object>)
<Dict> = epdf.Dict(<XRef>)
<Object> = epdf.Object()
```
The functions `StructElement_Type`, `Attribute_Type` and `AttributeOwner_Type` return a hash table `{<string>,<integer>}`.

Annot methods:

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

AnnotBorderStyle methods:

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

Annots methods:

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

Array methods:

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

Attribute methods:

```lua
<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()
<boolean> = <Attribute>: setHidden(<boolean>)
<string>  = <Attribute>:setFormattedValue()
<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>:getPageTreeRoot()
<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>:getValue(<integer>)
\begin{verbatim}
<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>)
\end{verbatim}
<integer> = <Object>:getType()
<string> = <Object>:getTypeName()
<boolean> = <Object>:isBool()
<boolean> = <Object>:isInt()
<boolean> = <Object>:isReal()
<boolean> = <Object>:isNum()
<boolean> = <Object>:isString()
<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>:getError()
<integer> = <Object>:getInt()
<number> = <Object>:getReal()
<integer> = <Object>:getRefNum()
<integer> = <Object>:getRefGen()
<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>:getColorInfo()
<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)
<Links> = <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()
(Stream) = <Stream>:getUndecodedStream()
(Dict) = <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()
<string> = <StructElement>:setTitle()
<string> = <StructElement>:getExpandedAbbr()
<integer> = <StructElement>:getNumChildren()
<StructElement> = <StructElement>:getChild()
<integer> = <StructElement>:getNumAttributes()
<Attribute> = <StructElement>:getAttribute(integer)
<string> = <StructElement>:appendChild<StructElement>)
<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
<integer> = <StructTreeRoot>:getNumChildren
<StructElement> = <StructTreeRoot>:getChild
<StructTreeRoot>:appendChild
<StructElement> = <StructTreeRoot>:findParentElement

TextSpan has only one method:

<string> = <TestSpan>:getText()

XRef methods:

<boolean> = <XRef>:isOk()
<integer> = <XRef>:getErrorCode()
<boolean> = <XRef>:isEncrypted()
<boolean> = <XRef>:okToPrint()
<boolean> = <XRef>:okToPrintHighRes()
<boolean> = <XRef>:okToChange()
<boolean> = <XRef>:okToCopy()
<boolean> = <XRef>:okToAddNotes()
<boolean> = <XRef>:okToFillForm()
<boolean> = <XRef>:okToAccessibility()
<boolean> = <XRef>:okToAssemble()
<Object> = <XRef>:getCatalog()
<Object> = <XRef>:fetch(<integer> object number, <integer> object generation)
<Object> = <XRef>:getDocInfo()
<Object> = <XRef>:getDocInfoNF()
<Integer> = <XRef>:getNumObjects()
<Integer> = <XRef>:getRootNum()
<Integer> = <XRef>:getRootGen()
<Integer> = <XRef>:getSize()
<Object> = <XRef>:getTrailerDict()

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 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.

<table> fnt = font.read_tfm(<string> name, <number> s)

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 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.
\begin{verbatim}
vf_fnt = font.read_vf(<string> name, <number> s)
\end{verbatim}

The meaning of the number \texttt{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.

\begin{verbatim}
font.fonts[n] = { ... }
<table> f = font.fonts[n]
\end{verbatim}

See \texttt{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:

\begin{verbatim}
<table> f = font.getfont(<number> n)
font.setfont(<number> n, <table> f)
\end{verbatim}

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:

\begin{verbatim}
<boolean> f = font.frozen(<number> n)
\end{verbatim}

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:

\begin{verbatim}
<number> i = font.define(<table> f)
\end{verbatim}

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 \texttt{chapter 7}.  

---

60 \texttt{LuaTeX Lua Libraries}
4.3.6 Projected next font id

<number> i = font.nextid()

This returns the font id number that would be returned by a 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)

<number> i = font.id(<string> csname)

This returns the font id associated with csname string, or −1 if csname is not defined; new in 0.47.

4.3.8 Currently active font

<number> i = font.current()
font.current(<number> i)

This gets or sets the currently used font number.

4.3.9 Maximum font id

<number> i = font.max()

This is the largest used index in font.fonts.

4.3.10 Iterating over all fonts

for i,v in font.each() do
  ...
end

This is an iterator over each of the defined T\TeX fonts. The first returned value is the index in font.fonts, the second the font itself, as a 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 fontloader library (0.36)

4.4.1 Getting quick information on a font

<table> info = fontloader.info(<string> filename)


This function returns either \texttt{nil}, or a \texttt{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:

\begin{verbatim}
<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>
\end{verbatim}

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 \texttt{fontloader} library, the simplest way to get that information is thus:

\begin{verbatim}
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
myfont = load_font('/opt/tex/texmf/fonts/data/arial.ttf')
\end{verbatim}

The main function call is

\begin{verbatim}
<userdata> f, <table> w = fontloader.open(<string> filename)
<userdata> f, <table> w = fontloader.open(<string> filename, <string> font-name)
\end{verbatim}

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.

\[ \text{table} \ f = \text{fontloader.to_table(\text{userdata} \ font)} \]

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 boundingbox 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:

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

### 4.4.3 Applying a ‘feature file’

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

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

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:

```lua
errors = fontloader.apply_afmfile(<userdata> font, <string> filename)
```

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 `f.glyphmin` to `f.glyphmax`.
- 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
```
if f.glyphcnt > 0 then
  for i=f.glyphmin,f.glyphmax do
    local g = f.glyphs[i]
    if g then
      print(g.name)
    end
    i = i + 1
  end
end
fontloader.close(f)

In this case, the \texttt{\LaTeX} 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{\LaTeX} 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>\texttt{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>Variable</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>italicangle</td>
<td>float</td>
<td>slant angle</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>uniqu Eid</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>glyphs</td>
<td>array</td>
<td></td>
</tr>
<tr>
<td>glyphcnt</td>
<td>number</td>
<td>number of included glyphs</td>
</tr>
<tr>
<td>glyphmax</td>
<td>number</td>
<td>maximum used index the glyphs array</td>
</tr>
<tr>
<td>glyphmin</td>
<td>number</td>
<td>minimum 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>
</tbody>
</table>
4.4.5.1.2 Glyph items

The **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 'none', 'base', 'ligature', 'mark', 'component' (if not present, the 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>topAccent</td>
<td>number</td>
<td>only if set</td>
</tr>
</tbody>
</table>
is_extended_shape  number  only if this character is part of a math extension list
altuni  table  alternate UNICODE items
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
type  enum  position, pair, substitution, alternate, multiple, ligature, lcaret, kerning, vkerning, anchors, contextpos, contextsub, chainpos, chainsub, reversesub, max, kernback, vkernback

specification  table  extra data

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:
key  | type   | explanation
----|--------|-------------
lig  | table  | uses the same substructure as a single item in the lookups table explained above
char | string |               
components | array  | linear array of named components
ccnt | number |               

The anchor table is indexed by a string signifying the anchor type, which is one of

key  | type   | explanation
----|--------|-------------
mark | table  | placement mark
basechar | table  | mark for attaching combining items to a base char
baselig | table  | mark for attaching combining items to a ligature
basemark | table  | generic mark for attaching combining items to connect to
centry | table  | cursive entry point
cexit  | table  | cursive exit point

The content of these is a short array of defined anchors, with the entry keys being the anchor names. For all except baselig, the value is a single table with this definition:

key  | type   | explanation
----|--------|-------------
x   | number | x location
y   | number | y location
ttf_pt_index | number | truetype point index, only if given

For baselig, the value is a small array of such anchor sets sets, one for each constituent item of the ligature.

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 <code>UNICODE</code> position numbers</td>
</tr>
<tr>
<td>psnames</td>
<td>array</td>
<td>of <code>PostScript</code> 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_unicodebmp</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_strikeysize 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
unicoderages    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:
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>

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</td>
<td>associated lookup(s)</td>
</tr>
<tr>
<td>offsets</td>
<td>array of numbers</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.5.10 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 <strong>gsub_single</strong>, <strong>gsub_multiple</strong>, <strong>gsub_alternate</strong>, <strong>gsub_ligature</strong>, <strong>gsub_context</strong>, <strong>gsub_contextchain</strong>, <strong>gsub_reversecontextchain</strong></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 (new in 0.47.0)
FractionDelimiterDisplayStyleSize (new in 0.47.0)

4.4.5.1.15 validation_state table

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

4.4.5.1.16 horiz_base and vert_base table

key type explanation
tags table an array of script list tags
scripts table

The scripts subtable:

key type explanation
baseline table
default_baseline number
lang table

The lang subtable:

key type explanation
tag string a script tag
ascent number
descent number
features table
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_correction</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 \textbf{vkerns table}

Substructure is identical to the per-glyph subtable.

4.4.5.1.22 \textbf{texdata table}

\begin{verbatim}
key   type      explanation
type  string   \textit{unset, text, math, mathext}
params array   22 font numeric parameters
\end{verbatim}

4.4.5.1.23 \textbf{lookups table}

Top-level \texttt{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.

\begin{verbatim}
key   type      explanation
type  string
format enum      one of \texttt{glyphs, class, coverage, reversecoverage}
tag   string
current_class array
before_class array
after_class array
rules array     an array of rule items
\end{verbatim}

Rule items have one common item and one specialized item:

\begin{verbatim}
key   type      explanation
lookups array   a linear array of lookup names
glyphs array    only if the parent’s format is \texttt{glyphs}
class array     only if the parent’s format is \texttt{class}
coverage array  only if the parent’s format is \texttt{coverage}
reversecoverage array only if the parent’s format is \texttt{reversecoverage}
\end{verbatim}

A glyph table is:

\begin{verbatim}
key   type      explanation
names string
back  string
to  string
\end{verbatim}

A class table is:

\begin{verbatim}
key   type      explanation
current array of numbers
before array of numbers
after  array of numbers
\end{verbatim}
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**

```latex
<image> var = img.new()
<image> 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

```latex
a = img.new()
```

followed by

```latex
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:

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

The generated <image> 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 LuaTEX</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 LuaTEX (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 LuaTEX (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 LuaTEX (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:

```
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.:

```
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`

```
<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

```
<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

\begin{verbatim}<image> var = img.copy(<image> var)\end{verbatim}
\begin{verbatim}<image> var = img.copy(<table> image_spec)\end{verbatim}

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

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

4.5.5 img.write

\begin{verbatim}<image> var = img.write(<image> var)\end{verbatim}
\begin{verbatim}<image> var = img.write(<table> image_spec)\end{verbatim}

By \texttt{img.write(a)} a PDF object number is allocated, and a whatst node of subtype \texttt{pdf_refximage} is generated and put into the output list. By this the image \texttt{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

\begin{verbatim}img.write ({ filename = "foo.png" })\end{verbatim}

The \texttt{<image>} variable is returned in case you want it for later processing.

4.5.6 img.immediatewrite

\begin{verbatim}<image> var = img.immediatewrite(<image> var)\end{verbatim}
\begin{verbatim}<image> var = img.immediatewrite(<table> image_spec)\end{verbatim}

By \texttt{img.immediatewrite(a)} a PDF object number is allocated, and the image file for image \texttt{a} is written out immediately into the PDF file as an image stream object (like with \texttt{\immediate\pdfximage}). The object number of the image stream dictionary is then available by the \texttt{objnum} key. No \texttt{pdf_refximage} whatst node is generated. You will need an \texttt{img.write(a)} or \texttt{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

\begin{verbatim}a = img.immediatewrite ({ filename = "foo.png" })\end{verbatim}

The \texttt{<image>} variable is returned and you will most likely need it.

4.5.7 img.node

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

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

This example outputs an image:

```lua
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 \LaTeX{} itself, and an object oriented interface that is completely on its own. The object oriented interface and kpse.new have been added in \LaTeX{} 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 \LaTeX{} 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 TeXLua 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 -format argument of kpsewhich. The supported ftype values are the same as the ones supported by the standalone kpsewhich program:
The default type is `tex`. Note: this is different from `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 LuaTeX 0.40.2.

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

`dpi` This is used for the size argument of the formats `pk`, `gf`, and `bitmap font`.

### 4.6.3 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.

```lua
<string> f, ... = kpse.lookup(<string> filename, <table> options)
```
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>mktexpk</td>
<td>boolean</td>
<td>disable/enable mktexpk generation for this lookup</td>
</tr>
<tr>
<td>mktextex</td>
<td>boolean</td>
<td>disable/enable mktextex generation for this lookup</td>
</tr>
<tr>
<td>mktexmf</td>
<td>boolean</td>
<td>disable/enable mktexmf generation for this lookup</td>
</tr>
<tr>
<td>mktextfm</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.

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

```
<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:

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

4.6.7 expand_var

Like kpsewhich’s -expand-var:
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 \LaTeX’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> 1)
lang.hyphenation(<language> 1, <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> 1)

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> 1)
lang.patterns(<language> 1, <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> 1)

Clears the pattern dictionary for this language.

<number> n = lang.prehyphenchar(<language> 1)
lang.prehyphenchar(<language> 1, <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> 1)
lang.posthyphenchar(<language> 1, <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> 1)
lang.preexhyphenchar(<language> 1, <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> 1)
lang.postexhyphenchar(<language> 1, <number> n)

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

Inserts hyphenation points (discretionary nodes) in a node list. If tail is given as argument, processing stops on that node. Currently, success is always true if head (and 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.

4.8 The lua library

This library contains one read-only item:

<string> s = lua.version

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 nil. Likewise, the retrieved value is either a function or nil.

lua.bytecode[<number> n] = <function> f
lua.bytecode[<number> n]()

The contents of the 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 nil), but this is an artifact of the current Lua implementation and thus subject to change.

The associated function calls are

<function> f = lua.getbytecode(<number> n)
lua.setbytecode(<number> n, <function> f)

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

lua.bytecode[n] = loadfile(filename)

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

lua.bytecode[n]()
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 \textsc{initex}. 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 \textsc{initex}.

4.8.2 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 latelu}a primitives.

\begin{verbatim}
lua.name[<number> n] = <string> s 
<string> s = lua.name[<number> n]
\end{verbatim}

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

\begin{verbatim}
<mpinstance> mp = mplib.new({...})
\end{verbatim}

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 variable</td>
<td>variable</td>
</tr>
<tr>
<td>interaction</td>
<td>string</td>
<td>the interaction mode, one of batch, nonstop, scroll, errorstop</td>
<td>errorstop</td>
</tr>
<tr>
<td>job_name</td>
<td>string</td>
<td>--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:

\begin{verbatim}
<string> found = finder (<string> name, <string> mode, <string> type)
\end{verbatim}

with:

name the requested file

mode the file mode: \texttt{r} or \texttt{w}
type the kind of file, one of: mp, tfm, map, pfb, enc

Return either the full pathname of the found file, or nil if the file cannot be found.
Note that the new version of MPLIB no longer uses binary mem files, so the way to preload a set of macros is simply to start off with an input command in the first mp:execute() call.

4.9.2 mp:statistics

You can request statistics with:
<table> stats = mp:statistics()

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

main_memory number memory size
hash_size number hash size
param_size number simultaneous macro parameters
max_in_open number input file nesting levels

Note that in the new version of 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 mp:execute

You can ask the MetaPost interpreter to run a chunk of code by calling
<table> rettable = mp:execute('metapost language chunk')

for various bits of MetaPost language input. Be sure to check the rettable.status (see below) because when a fatal MetaPost error occurs the 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 mpost command, there is no implied ‘input’ at the start of the first chunk.

4.9.4 mp:finish

<table> rettable = mp:finish()
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 miter limit
- **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 miter limit
- **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 tfm 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>marking only no values</td>
</tr>
<tr>
<td>1</td>
<td>greyscale one value in the range (0, 1), 'black' is 0</td>
</tr>
<tr>
<td>3</td>
<td>RGB three values in the range (0, 1), 'black' is 0, 0, 0</td>
</tr>
<tr>
<td>4</td>
<td>CMYK four values in the range (0, 1), 'black' is 0, 0, 0, 1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>number represents x</td>
</tr>
<tr>
<td>2</td>
<td>number represents y</td>
</tr>
<tr>
<td>3</td>
<td>number represents xx</td>
</tr>
<tr>
<td>4</td>
<td>number represents yx</td>
</tr>
<tr>
<td>5</td>
<td>number represents xy</td>
</tr>
<tr>
<td>6</td>
<td>number represents yy</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dashes hash an array of on-off numbers</td>
</tr>
<tr>
<td>offset number the starting offset value</td>
</tr>
</tbody>
</table>

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`

```lua
<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 \LaTeX{} node objects, the core objects within the typesetter.

\LaTeX{} nodes are represented in \La{} as userdata with the metadata type \texttt{luatex.node}. The various parts within a node can be accessed using named fields.

Each node has at least the three fields \texttt{next}, \texttt{id}, and \texttt{subtype}:

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

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

Support for \texttt{unset} (alignment) nodes is partial: they can be queried and modified from \La{} 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 \La{} 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 \LaTeX{} itself, you have not deleted nodes that are still referenced from a \texttt{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> \texttt{t = node.is_node(<any> \texttt{item})}

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

4.10.1.2 node.types

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

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

4.10.1.4 node.id

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

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

4.10.1.5 node.subtype

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

This converts a single whatsit name to its internal numeric representation (\texttt{subtype}).

4.10.1.6 node.type

<string> \texttt{type = node.type(<any> \texttt{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 \texttt{node} if the object represents a node (this is new in 0.65), and \texttt{nil} otherwise.
4.10.1.7

**node.fields**

<table>
t = node.fields(<number> id)
<table>
t = node.fields(<number> id, <number> 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 `id` and `subtype` values as well.

4.10.1.8

**node.has_field**

<boolean>
t = node.has_field(<node> n, <string> field)

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

4.10.1.9

**node.new**

<node>
n = node.new(<number> id)
<node>
n = node.new(<number> id, <number> subtype)

Creates a new node. All of the new node's fields are initialized to either zero or `nil` except for `id` and `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 `id` and `subtype` values as well.

4.10.1.10

**node.free**

node.free(<node> n)

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

4.10.1.11

**node.flush_list**

node.flush_list(<node> n)

Removes the node list `n` and the complete node list following `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 `next` field: it is up to you to make sure that the internal data structures remain correct.
4.10.1.12 \texttt{node.copy}

\begin{verbatim}
<node> m = node.copy(<node> n)
\end{verbatim}

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

4.10.1.13 \texttt{node.copy_list}

\begin{verbatim}
<node> m = node.copy_list(<node> n)
<node> m = node.copy_list(<node> n, <node> m)
\end{verbatim}

Creates a deep copy of the node list that starts at \texttt{n}. If \texttt{m} is also given, the copy stops just before node \texttt{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 \texttt{attr} field or make changes to specific attributes, the needed copying and freeing takes place automatically.

4.10.1.14 \texttt{node.next (0.65)}

\begin{verbatim}
<node> m = node.next(<node> n)
\end{verbatim}

Returns the node following this node, or \texttt{nil} if there is no such node.

4.10.1.15 \texttt{node.prev (0.65)}

\begin{verbatim}
<node> m = node.prev(<node> n)
\end{verbatim}

Returns the node preceding this node, or \texttt{nil} if there is no such node.

4.10.1.16 \texttt{node.current_attr (0.66)}

\begin{verbatim}
<node> m = node.current_attr()
\end{verbatim}

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

The intended usage of \texttt{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

\<node> h, \<number> b = node.hpack(<node> n)
\<node> h, \<number> b = node.hpack(<node> n, \<number> w, \<string> info)
\<node> 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)

\<node> h, \<number> b = node.vpack(<node> n)
\<node> h, \<number> b = node.vpack(<node> n, \<number> w, \<string> info)
\<node> 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.

\textbf{4.10.1.19 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}
\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

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

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

4.10.1.21 node.slide

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

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

4.10.1.22 node.tail

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

Returns the last node of the node list that starts at \texttt{n}.

4.10.1.23 node.length

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

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

4.10.1.24 node.count

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

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

This function also accept string \texttt{id}'s.

4.10.1.25 node.traverse

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

This is a lua iterator that loops over the node list that starts at \texttt{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 \texttt{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 next (and 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 \texttt{n} that have a matching \texttt{id} field.

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

\texttt{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)

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

Returns the first node in the list starting at \( n \) that is a glyph node with a subtype indicating it is a glyph, or \texttt{nil}. If \( m \) 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 \LaTeX{} 0.65, and the old name is deprecated now.

4.10.1.32  node.ligaturing

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

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

4.10.1.33  node.kerning

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

Apply \TeX{}-style kerning to the specified nodelist. The tail node \( m \) is optional. The two returned nodes \( h \) and \( t \) 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

\texttt{node.unprotect_glyphs(<node> n)}

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

\texttt{node.protect_glyphs(<node> n)}

Adds 256 to all glyph node subtypes in the node list starting at \( n \), 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 node> n = node.last_node()

This function pops the last node from TeX’s ‘current list’. It returns that node, or 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 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 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 id set. If val is also supplied, also tests if the value matches val. It returns the value, or, if no match is found, nil.

4.10.2.2 node.set_attribute

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

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

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

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

If the attribute was actually deleted, returns its old value. Otherwise, returns \texttt{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)

\texttt{pdf.mapfile(\texttt{map file})}
\texttt{pdf.mapfile(\texttt{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 \texttt{h} and \texttt{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 \texttt{h} and \texttt{v} keys are not real variables but looked up using a metatable call. The \texttt{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: \texttt{sx}, \texttt{rx}, \texttt{ry}, \texttt{sy}, \texttt{tx}, and \texttt{ty}. The \texttt{hasmatrix} function returns \texttt{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{\textasciitilde latex} argument. This function is not to be used inside \texttt{\textasciitilde directlua} unless you know \textit{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 `direct` or `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 \immediate\pdfobj primitives. All function variants return the object number of the newly generated object.

\begin{verbatim}
<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)
\end{verbatim}

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 provided by the user. The second version with keyword \texttt{"file"} as 1st argument puts the contents of the file with name \texttt{filename} raw into the object. The third version with keyword \texttt{"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 \texttt{"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.

\begin{verbatim}
<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)
\end{verbatim}

4.11.9 pdf.obj

This function creates a PDF object, which is written to the PDF file only when referenced, e.g., by 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 \pdfobj primitive.

\begin{verbatim}
<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)
\end{verbatim}

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

\begin{verbatim}
<number> n = pdf.obj(<integer> n, <string> objtext)
\end{verbatim}
\begin{verbatim}
<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 \texttt{type} field can have the values \texttt{raw} and \texttt{stream}, this field is required, the others are optional
(within constraints).

Note: this mode makes \texttt{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 \texttt{string} and \texttt{file} at the same time.

4.11.10 pdf.refobj

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

\texttt{pdf.refobj(<integer> n)}

This function works in both the \texttt{\directlua} and \texttt{\latelua} environment. Inside \texttt{\directlua} a new
whatstnode '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 \texttt{\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")
\end{verbatim}
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 \late\textstyle{\texttt{\textbackslash late}}. 

\texttt{pdf.registerannot (<number> objnum)}

4.12 The \texttt{pdfscanner} library (new in 0.72.0)

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

\texttt{pdfscanner.scan (<Object> stream, <table> operatortable, <table> info)}

The first argument, \texttt{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 \texttt{epdf} library).

The second argument, \texttt{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 \texttt{scanner} object itself, and the \texttt{info} table that was passed are the third argument to \texttt{pdfscanner.scan}.

Internally, \texttt{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 \texttt{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 \texttt{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:

\begin{verbatim}
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 .. " ",

\end{verbatim}
resources = dict:lookup('Resources'):getDict() }
end
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 \texttt{epdf} functions, let's focus on the type \texttt{pdfscanner} specific code instead.

From the bottom up, the line

\begin{verbatim}
dfscanner.scan(page:getContents(), operatortable, info)
\end{verbatim}

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

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

The second argument, \texttt{operatortable} defines a processing function for a single PDF operator, \texttt{Do}.

The function \texttt{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 \texttt{info} argument with an updated \texttt{space} value so that the indentation of the output nicely nests, and with an new \texttt{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 \texttt{pdfscanner}, it is just long enough. It makes use of only one \texttt{scanner} method: \texttt{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:

\begin{verbatim}
integer <number>
real <number>
boolean <boolean>
\end{verbatim}
name <string>
operator <string>
string <string>
array <table>
dict <table>

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 \texttt{status} are fetched through a metatable at run-time whenever they are accessed, so you cannot use \texttt{pairs} on \texttt{status}, but you \textit{can} use \texttt{pairs} on \texttt{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 \texttt{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 \textsc{in}TEX 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>\textsc{in}TEX 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>
4.14 The tex library

The tex table contains a large list of virtual internal \text{T\text{E}X} 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 \text{T\text{E}X} values. As a result, most of the Lua table operators (like \texttt{pairs} and \texttt{#}) 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 \texttt{\the}\texttt{\ldots}\texttt{.}
- It is a single token.
- Some special others, see the list below

This excludes parameters that need extra arguments, like \texttt{\the\scriptfont}.

The subset comprising simple integer and dimension registers are writable as well as readable (stuff like \texttt{\tracingcommands} and \texttt{\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 \texttt{tex} table, or by using one of the functions \texttt{tex.get()} and \texttt{tex.set()}. 

\begin{tabular}{ll}
pdf\textunderscore os\textunderscore objidx & PDF object stream index  
pdf\textunderscore dest\textunderscore names\textunderscore ptr & max PDF destination pointer  
dest\textunderscore names\textunderscore size & PDF destination table size  
pdf\textunderscore mem\textunderscore ptr & max PDF memory used  
pdf\textunderscore mem\textunderscore size & PDF memory size  
largest\textunderscore used\textunderscore 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  
lastererrorstring & 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 & \texttt{true} if the \texttt{\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 & \texttt{true} if this is an ini\TeX{} run (added in 0.38) 
\end{tabular}
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.

```latex
\begin{verbatim}
tex.set (<string> n, ...)
tex.set ('global', <string> n, ...)
... = tex.get (<string> n)
\end{verbatim}
```

### 4.14.1.1 Integer parameters

The integer parameters accept and return Lua numbers.

Read-write:

```latex
\begin{verbatim}
tex.adjdemerits
tex.binoppenalty
tex.brokenpenalty
tex.catcodetable
tex.clubpenalty
tex.day
tex.defaulthyphenchar
tex.defaultskewchar
tex.delimiterfactor
tex.displaywidowpenalty
tex.doublehyphendemerits
tex.endlinechar
tex.errorcontextlines
tex.escapechar
tex.exhyphenpenalty
tex.fam
tex.finalhyphendemerits
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
\end{verbatim}
```

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

```
\begin{verbatim}
tex.savinghyphcodes
\end{verbatim}
```
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 \hspace{1em} space \hspace{1em} tex.pdfhorigin

tex.delimitershorthand \hspace{1em} tex.overfullrule \hspace{1em} tex.pdflastlinedepth

tex.displayindent \hspace{1em} tex.pagebottomoffset \hspace{1em} tex.pdflinkmargin

tex.displaywidth \hspace{1em} tex.pageheight \hspace{1em} tex.pdfpageheight

tex.emergencystretch \hspace{1em} tex.pageleftoffset \hspace{1em} tex.pdfpagewidth

tex.hangindent \hspace{1em} tex.pagerightoffset \hspace{1em} tex.pdfpxdimen

tex.hfuzz \hspace{1em} tex.pagetopoffset \hspace{1em} tex.pdfthreadmargin

tex.hoffset \hspace{1em} tex.pagewidth \hspace{1em} tex.pdfvorigin

tex.hsize \hspace{1em} tex.parindent \hspace{1em} tex.predisplaysize

tex.lineskiplimit \hspace{1em} tex.pdfeachlinedepth \hspace{1em} tex.scriptspace

tex.mathsurround \hspace{1em} tex.pdfeachlineheight \hspace{1em} tex.splitmaxdepth

tex.maxdepth \hspace{1em} tex.pdffirstline-height \hspace{1em} tex.vfuzz

tex.nulldelimiter- \hspace{1em} tex.pagenumber \hspace{1em} tex.voffset

Read-only:

tex.pagedepth \hspace{1em} tex.pagefillllstretch \hspace{1em} tex.pagefillllstretch

Read-only:

tex.deadcycles \hspace{1em} tex.parshape \hspace{1em} tex.spacefactor

tex.insertpenalties \hspace{1em} tex.prevgraf
4.14.1.3 Direction parameters

The direction parameters are read-only and return a Lua string.

\texttt{tex.bodydir} \hspace{1cm} \texttt{tex.pagedir} \hspace{1cm} \texttt{tex.textdir}

\texttt{tex.mathdir} \hspace{1cm} \texttt{tex.pardir}

4.14.1.4 Glue parameters

The glue parameters accept and return a userdata object that represents a glue\_spec node.

\texttt{tex.abovedisplayshortskip} \hspace{1cm} \texttt{tex.abovedisplayskip} \hspace{1cm} \texttt{tex.baselineskip}

\texttt{tex.belowdisplayshortskip} \hspace{1cm} \texttt{tex.belowdisplayskip} \hspace{1cm} \texttt{tex.leftskip}

\texttt{tex.baselineskip} \hspace{1cm} \texttt{tex.lineskip} \hspace{1cm} \texttt{tex.parfillskip}

\texttt{tex.abovedisplayshortskip} \hspace{1cm} \texttt{tex.parfillskip} \hspace{1cm} \texttt{tex.rightskip}

\texttt{tex.baselineskip} \hspace{1cm} \texttt{tex.parfillskip} \hspace{1cm} \texttt{tex.rightskip}

\texttt{tex.baselineskip} \hspace{1cm} \texttt{tex.parfillskip} \hspace{1cm} \texttt{tex.rightskip}

4.14.1.5 Muglue parameters

All muglue parameters are to be used read-only and return a Lua string.

\texttt{tex.medmuskip} \hspace{1cm} \texttt{tex.thickmuskip} \hspace{1cm} \texttt{tex.thinmuskip}

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.

\texttt{tex.errhelp} \hspace{1cm} \texttt{tex.everyjob} \hspace{1cm} \texttt{tex.pdfpageattr}

\texttt{tex.everycr} \hspace{1cm} \texttt{tex.everymath} \hspace{1cm} \texttt{tex.pdfpageresources}

\texttt{tex.everydisplay} \hspace{1cm} \texttt{tex.everypar} \hspace{1cm} \texttt{tex.pdfpagesattr}

\texttt{tex.everyeof} \hspace{1cm} \texttt{tex.everyvbox} \hspace{1cm} \texttt{tex.pdfpkmode}

\texttt{tex.everyhbox} \hspace{1cm} \texttt{tex.output}
4.14.2 Convert commands

All ‘convert’ commands are read-only and return a Lua string. The supported commands at this moment are:

\begin{verbatim}
\texttt{tex.e\TeX\textit{Version}} \hspace{1em} \texttt{tex.pdffontobjnum(number)}
\texttt{tex.e\TeX\textit{revision}} \hspace{1em} \texttt{tex.pdffontsize(number)}
\texttt{tex.formatname} \hspace{1em} \texttt{tex.uniformdeviate(number)}
\texttt{tex.jobname} \hspace{1em} \texttt{tex.number(number)}
\texttt{tex.luatexbanner} \hspace{1em} \texttt{tex.romannumeral(number)}
\texttt{tex.luatexrevision} \hspace{1em} \texttt{tex.pdfpageref(number)}
\texttt{tex.pdfnormaldeviate} \hspace{1em} \texttt{tex.pdfformname(number)}
\texttt{tex.fontname(number)} \hspace{1em} \texttt{tex.fontidentifier(number)}
\texttt{tex.pdffontname(number)} \hspace{1em} \texttt{tex.pdfpageref(number)}
\end{verbatim}

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 \texttt{tex.fontidentifier} returns the \textit{csname} string that matches a font id number (if there is one).

The \texttt{pdf\TeX} related primitives will be removed from versions after 0.80. All the version related details are available in the \textit{status} table at the Lua end, for instance one can say:

\begin{verbatim}
\let\pdftexbanner \luatexbanner
\let\pdftexversion \luatexversion
\let\pdfxformversion \luatexversion
\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:

\begin{verbatim}
\texttt{tex.lastpenalty} \hspace{1em} \texttt{tex.pdflastximagepages}
\texttt{tex.lastkern} \hspace{1em} \texttt{tex.pdflastannot}
\texttt{tex.lastskip} \hspace{1em} \texttt{tex.pdflastxpos}
\texttt{tex.lastnodetype} \hspace{1em} \texttt{tex.pdfxform}
\texttt{tex.inputlineno} \hspace{1em} \texttt{tex.pdfrandomseed}
\texttt{tex.pdflastobj} \hspace{1em} \texttt{tex.pdflastlink}
\texttt{tex.pdflastxform} \hspace{1em} \texttt{tex.luatexversion}
\texttt{tex.pdflastximage} \hspace{1em} \texttt{tex.e\TeX\textit{version}}
\texttt{tex.e\TeX\textit{minorversion}} \hspace{1em} \texttt{tex.currentgrouplevel}
\texttt{tex.currentgrouptype} \hspace{1em} \texttt{tex.currentiflevel}
\texttt{tex.currentiftype} \hspace{1em} \texttt{tex.currentifbranch}
\texttt{tex.currentifdepth}
\end{verbatim}
4.14.4 Attribute, count, dimension, skip and token registers

\TeX's attribute (\attribute), counters (\count), dimensions (\dimen), skips (\skip) and token (\toks) registers can be accessed and written to using two times five virtual sub-tables of the \tex table:

\begin{verbatim}
tex.attribute  tex.dimen  tex.toks
tex.count      tex.skip
\end{verbatim}

It is possible to use the names of relevant \attributedef, \countdef, \dimendef, \skipdef, or \toksdef control sequences as indices to these tables:

\begin{verbatim}
tex.count.scratchcounter = 0
enormous = tex.dimen['maxdimen']
\end{verbatim}

In this case, \LaTeX looks up the value for you on the fly. You have to use a valid \countdef (or \attributedef, or \dimendef, or \skipdef, or \toksdef), anything else will generate an error (the intent is to eventually also allow <chardef tokens> and even macros that expand into a number).

The attribute and count registers accept and return Lua numbers.

The dimension registers accept 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 Lua strings. Lua strings are converted to and from token lists using \texttt{\toks} style expansion: all category codes are either space (10) or other (12).

The skip registers accept and return 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 \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 (\lccode, \uccode, \sfcode, \catcode, \mathcode, \delcode) can be accessed and written to using six virtual subtables of the \tex table.
The function call interfaces are roughly as above, but there are a few twists. *sfcodes* are the simple ones:

```
tex.setsfcode (<number> n, <number> s)
tex.setsfcode ('global', <number> n, <number> s)
<number> s = tex.getsfcode (<number> n)
```

The function call interface for *lccode* and *uccode* additionally allows you to set the associated sibling at the same time:

```
tex.setlccode ('global', <number> n, <number> lc)
tex.setlccode ('global', <number> n, <number> lc, <number> uc)
<number> lc = tex.getlccode (<number> n)
<number> uc = tex.getuccode (<number> n)
```

The function call interface for *catcode* also allows you to specify a category table to use on assignment or on query (default in both cases is the current one):

```
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)
```

The interfaces for *delcode* and *mathcode* use small array tables to set and retrieve values:

```
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)
```

Where the table for *mathcode* is an array of 3 numbers, like this:

```
{<number> mathclass, <number> family, <number> character}
```

And the table for *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 \Udelcode usage, but the returned table can have values there (if the delimiter code was set using \delcode, for example). Unset *delcode*’s can be recognized because *dval*[1] is −1.
4.14.6 Box registers

It is possible to set and query actual boxes, using the node interface as defined in the `node` library:

```latex
\texttt{tex.box}
```

for array access, or

```latex
\texttt{tex.setbox(<number> n, <node> s)}
\texttt{tex.setbox(<string> cs, <node> s)}
\texttt{tex.setbox('global', <number> n, <node> s)}
\texttt{tex.setbox('global', <string> cs, <node> s)}
<node> n = \texttt{tex.getbox(<number> n)}
<node> n = \texttt{tex.getbox(<string> cs)}
```

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

```latex
\texttt{tex.box[0] = tex.box[2]}
```

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 `node.copy_list()` unless you are assigning to a temporary variable:

```latex
\texttt{tex.box[0] = node.copy_list(tex.box[2])}
```

4.14.7 Math parameters

It is possible to set and query the internal math parameters using:

```latex
\texttt{tex.setmath(<string> n, <string> t, <number> n)}
\texttt{tex.setmath('global', <string> n, <string> t, <number> n)}
<number> n = \texttt{tex.getmath(<string> n, <string> t)}
```

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:

- \texttt{quad}
- \texttt{axis}
- \texttt{operatorsize}
- \texttt{overbarkern}
- \texttt{overbarrule}
- \texttt{overbarvgap}
- \texttt{underbarkern}
- \texttt{underbarrule}
- \texttt{underbarvgap}
- \texttt{radicalkern}
- \texttt{radicalrule}
- \texttt{radicalvgap}
- \texttt{radicaldegreebefore}
- \texttt{radicaldegreeafter}
- \texttt{radicaldegreeraise}
- \texttt{stackvgap}
- \texttt{stacknumup}
- \texttt{stackdenomdown}
4.14.8 Special list heads

The virtual table tex.lists contains the set of internal registers that keep track of building page lists.

The values for the style parameter name are:

display crampeddisplay
text crampedtext
script crampedscript
scriptscript crampedscriptscript
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>
<tbody>
<tr>
<td>mode</td>
<td>number</td>
<td>all</td>
<td>The current mode. This is a number representing the main mode at this level:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 == no mode (this happens during \write)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 == vertical,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>127 = horizontal,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>253 = display math,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−1 == internal vertical,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−127 = restricted horizontal,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−253 = inline math.</td>
</tr>
<tr>
<td>modeline</td>
<td>number</td>
<td>all</td>
<td>source input line where this mode was entered in, negative inside the output routine.</td>
</tr>
<tr>
<td>head</td>
<td>node</td>
<td>all</td>
<td>the head of the current list</td>
</tr>
<tr>
<td>tail</td>
<td>node</td>
<td>all</td>
<td>the tail of the current list</td>
</tr>
<tr>
<td>prevgraf</td>
<td>number</td>
<td>vmode</td>
<td>number of lines in the previous paragraph</td>
</tr>
<tr>
<td>prevdepth</td>
<td>number</td>
<td>vmode</td>
<td>depth of the previous paragraph (equal to \pdfignoreddimen when it is to be ignored)</td>
</tr>
<tr>
<td>spacefactor</td>
<td>number</td>
<td>hmode</td>
<td>the current space factor</td>
</tr>
<tr>
<td>dirs</td>
<td>node</td>
<td>hmode</td>
<td>used for temporary storage by the line break algorithm</td>
</tr>
<tr>
<td>noad</td>
<td>node</td>
<td>mmode</td>
<td>used for temporary storage of a pending fraction numerator, for \over etc.</td>
</tr>
<tr>
<td>delimptr</td>
<td>node</td>
<td>mmode</td>
<td>used for temporary storage of the previous math delimiter, for \middle.</td>
</tr>
<tr>
<td>mathdir</td>
<td>boolean</td>
<td>mmode</td>
<td>true when during math processing the \mathdir is not the same as the surrounding \textdir</td>
</tr>
<tr>
<td>mathstyle</td>
<td>number</td>
<td>mmode</td>
<td>the current \mathstyle</td>
</tr>
</tbody>
</table>

4.14.10 Print functions

The \texttt{tex} table also contains the three print functions that are the major interface from 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 \directlua.

The total amount of returnable text from a \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 tex.print

```markdown
\begin{verbatim}
tex.print(<string> s, ...)
tex.print(<number> n, <string> s, ...)
tex.print(<table> t)
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 \catcodetable n. If n is −1, the currently active catcode regime is used. If n is −2, the resulting catcodes are the result of \the\toks: all category codes are 12 (other) except for the space character, that has category code 10 (space). Otherwise, if 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 tex.print() command in a \directlua will not have the \endlinechar appended, all others do.

### 4.14.10.2 tex.sprint

```markdown
\begin{verbatim}
tex.sprint(<string> s, ...)
tex.sprint(<number> n, <string> s, ...)
tex.sprint(<table> t)
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 \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

```lua
before\directlua{tex.sprint("\\relax")tex.sprint(" inbetween")}after
```

the space before inbetween will be gobbled as a result of the ‘normal’ scanning of \relax.
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`

`tex.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`

`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 `\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`

`<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`

`<number> n = tex.scale(<number> o, <number> delta)`  
`<table> n = tex.scale(table o, <number> delta)`

Multiplies the Lua numbers `o` and `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 \LaTeX{}'s internal, 100% portable scale function will be added at a later date.

### 4.14.11.3 tex.sp (0.51)

\begin{verbatim}
<number> n = tex.sp(<number> o)
<number> n = tex.sp(<string> s)
\end{verbatim}

Converts the number \texttt{o} or a string \texttt{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 \LaTeX{} 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 (\texttt{fil...}) are forbidden
3. \texttt{mu} units do not generate an error (but may not be useful either)

### 4.14.11.4 tex.definefont

\begin{verbatim}
tex.definefont(<string> csname, <number> fontid)
tex.definefont(<boolean> global, <string> csname, <number> fontid)
\end{verbatim}

Associates \texttt{csname} with the internal font number \texttt{fontid}. The definition is global if (and only if) \texttt{global} is specified and true (the setting of \texttt{globaldefs} is not taken into account).

### 4.14.11.5 tex.error (0.61)

\begin{verbatim}
tex.error(<string> s)
tex.error(<string> s, <table> help)
\end{verbatim}

This creates an error somewhat like the combination of \texttt{\textbackslash errhelp} and \texttt{\textbackslash errmessage} would. During this error, deletions are disabled.

The array part of the \texttt{help} table has to contain strings, one for each line of error help.

### 4.14.11.6 tex.hashtokens (0.25)

\begin{verbatim}
for i,v in pairs (tex.hashtokens()) do ... end
\end{verbatim}

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 \TeX\82, 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), LuaTEX in --ini mode contains only the \TeX\82 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 TeX82-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`

```table
<table>
<thead>
<tr>
<th>name</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>tex</td>
<td>□</td>
</tr>
</tbody>
</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:
Note that '\texttt{luatex}' does not contain \texttt{directlua}, as that is considered to be a core primitive, along with all the \TeX\ 82 primitives, so it is part of the list that is returned from \texttt{'core'}.\texttt{'umath'} is a subset of \texttt{'luatex'} that covers the Unicode math primitives and have been added in \LaTeX\ 0.75.0 as it might be desired to handle the prefixing of that subset differently.

Running \texttt{tex.extraprimitives()} will give you the complete list of primitives that are not defined at \LaTeX\ 0.39.0 -ini startup. It is exactly equivalent to \texttt{tex.extraprimitives('etex', 'pdf-tex', 'omega', 'aleph', 'luatex')}

\subsection*{4.14.12.3 tex.primitives}

\begin{verbatim}
t = tex.primitives()
\end{verbatim}

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.

\subsection*{4.14.13 Core functionality interfaces}

\subsection*{4.14.13.1 tex.badness (0.53)}

\begin{verbatim}
b = tex.badness(<number> t, <number> s)
\end{verbatim}

This function returns the badness of two values.
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)}

```latex
\begin{verbatim}
local \texttt{nodelist, info =}
    \texttt{tex.linebreak(\texttt{node} listhead, \texttt{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>adjdemerits</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>doublehyphenpenalty</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>finalhyphenpenalty</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{\textbackslash interlinepenalties}</td>
</tr>
<tr>
<td>clubpenalty</td>
<td>number or table</td>
<td>if a table, then it is an array like \texttt{\textbackslash clubpenalties}</td>
</tr>
<tr>
<td>widowpenalty</td>
<td>number or table</td>
<td>if a table, then it is an array like \texttt{\textbackslash 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>pdffignoreddimen</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{\textbackslash 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 pdfignoreddimen.

It is your own job to make sure that 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 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 \parfillskip after that penalty node
- make sure all the prev pointers are OK

The result is a node list, it still needs to be vpacked if you want to assign it to a \vbox.

The returned info table contains four values that are all numbers:

prevdepth  depth of the last line in the broken paragraph
prevgraf   number of lines in the broken paragraph
looseness  the actual looseness value in the broken paragraph
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 pdfadjustspacing, because the settings for that take place elsewhere. The same is true for hbadness and hfuzz etc. All these are in the hpack() routine, and that fetches its own variables via globals.

4.14.13.3 tex.shipout (0.51)

tex.shipout(<number> n)

Ships out box number n to the output file, and clears the box register.

4.15 The 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</td>
</tr>
<tr>
<td>Key</td>
<td>Type</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>shell_escape</td>
<td>string</td>
<td>'f'</td>
<td>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)</td>
</tr>
<tr>
<td>shell_escape_commands</td>
<td>string</td>
<td></td>
<td>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 (&quot;) for any needed argument or path quoting (new in 0.37)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string_vacancies</td>
<td>number</td>
<td>75000</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>pool_free</td>
<td>number</td>
<td>5000</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>max_strings</td>
<td>number</td>
<td>15000</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>strings_free</td>
<td>number</td>
<td>100</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>nest_size</td>
<td>number</td>
<td>50</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>max_in_open</td>
<td>number</td>
<td>15</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>param_size</td>
<td>number</td>
<td>60</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>save_size</td>
<td>number</td>
<td>4000</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>stack_size</td>
<td>number</td>
<td>300</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>dvi_buf_size</td>
<td>number</td>
<td>16384</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>error_line</td>
<td>number</td>
<td>79</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>half_error_line</td>
<td>number</td>
<td>50</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>max_print_line</td>
<td>number</td>
<td>79</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>hash_extra</td>
<td>number</td>
<td>0</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>pk_dpi</td>
<td>number</td>
<td>72</td>
<td>cf. web2c docs</td>
</tr>
<tr>
<td>trace_file_names</td>
<td>boolean</td>
<td>true</td>
<td>false disables \TeX's normal file open-close feedback (the assumption is that callbacks will take care of that)</td>
</tr>
<tr>
<td>file_line_error</td>
<td>boolean</td>
<td>false</td>
<td>do file:line style error messages</td>
</tr>
<tr>
<td>halt_on_error</td>
<td>boolean</td>
<td>false</td>
<td>abort run on the first encountered error</td>
</tr>
<tr>
<td>formatname</td>
<td>string</td>
<td></td>
<td>if no format name was given on the commandline, this key will be tested first instead of simply quitting</td>
</tr>
<tr>
<td>jobname</td>
<td>string</td>
<td></td>
<td>if no input file name was given on the commandline, this key will be tested first instead of simply giving up</td>
</tr>
</tbody>
</table>

**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 \texttt{texio} library

This library takes care of the low-level I/O interface.

4.16.1 Printing functions

4.16.1.1 \texttt{texio.write}

\texttt{texio.write(<string> target, <string> s, ...)}
\texttt{texio.write(<string> s, ...)}

Without the \texttt{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 \texttt{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 \texttt{target} must be specified explicitly to prevent \texttt{Lua} from interpreting the first string as the target.

4.16.1.2 \texttt{texio.write_nl}

\texttt{texio.write_nl(<string> target, <string> s, ...)}
\texttt{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 \texttt{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 \texttt{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 \texttt{token.get_next}

\texttt{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 t could be expanded.

### 4.17.3 token.expand

token.expand(<token> t)

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 `token.get_next()`.

### 4.17.4 token.is_activechar

<boolean> b = token.is_activechar(<token> t)

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.

### 4.17.5 token.create

```
token t = token.create(<string> csname)
```

```
token t = token.create(<number> charcode)
```

```
token t = token.create(<number> charcode, <number> catcode)
```

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 `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 `token.create()`!

### 4.17.6 token.command_name

```
<string> commandname = token.command_name(<token> t)
```

luatexlua libraries
This returns the name associated with the ‘command’ value of the token in LuaTEX. There is not always a direct connection between these names and primitives. For instance, all \ifxxx tests are grouped under 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:
function argument result

is_token token checks if the given argument is a token userdatum
get_next string returns the next token in the input
scan_keyword string returns true if the given keyword is gobbled
scan_int infinity, mu-units returns a number representing a dimension and or two
numbers being the filler and order
scan_dimen infinity, 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 se-
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 se-
quence 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{bar}
\def\foo{foo-\bar}

we get:

\directlua{newtoken.scan_string()}{foo} foo full expansion
\directlua{newtoken.scan_string()}{foo} foo letters and others
\directlua{newtoken.scan_string()}{\foo} foo-bar meaning

The \foo case only gives the meaning, but one can pass an already expanded definition (\edef’d). In the case of the braced variant one can of course use the \detokenize and \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)
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:

```
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\mmacro#1{%
    \directlua {
        mymacro(\number\dimexpr#1)
    }%
}\mmacro{12pt}
\mmacro{\dimen0}
```

But starting with version 0.80 you can also do this:

```latex
\directlua {
    function mymacro()
        local d = newtoken.scan_dimen()
    end
}
```
... 
end
}
\def\mymacro{%
  \directlua {
    mymacro()
  }%
}
\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 LUA code to see what gets picked up. Instead of passing from \TeX{} to LUA we let 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 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 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 TeX82 (and therefore PDFTeX) handles math. First, LuaTeX adds primitives and extends some others so that Unicode input can be used easily. Second, all of TeX82’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{\textbackslash mathstyle}. This is a ‘convert command’ like e.g. \texttt{\textbackslash roman\textbackslash numeral}: its value can only be read, not set.

5.1.1 \texttt{\textbackslash 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=\cramped\textstyle
  \message{cramped text style}
\fi \fi
\end{verbatim}

5.1.2 \texttt{\textbackslash 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{\textbackslash over}, \texttt{\textbackslash atop}, \texttt{\textbackslash overwithdelims}, \texttt{\textbackslash atopwithdelims}) would therefore normally return wrong values for \texttt{\textbackslash mathstyle}. To fix this, LuaTeX introduces a special prefix command: \texttt{\textbackslash Ustack}:

$\texttt{\textbackslash Ustack \{a \over b\}}$

The \texttt{\textbackslash 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 X\TeX.

The math primitives from TeX are kept as they are, except for the ones that convert from input to math commands: mathcode, and delcode. These two now allow for a 21-bit character argument on the left hand side of the equals sign.

Some of the new LuaTeX 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:

```
def\overbrace {\Umathaccent 0 1 "23DE }
```

Altered TeX82 primitives:

<table>
<thead>
<tr>
<th>primitive</th>
<th>value range (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\mathcode</td>
<td>0–10FFFF = 0–8000</td>
</tr>
<tr>
<td>\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>\mathchardef</td>
<td>0–8000</td>
</tr>
<tr>
<td>\mathchar</td>
<td>0–7FFF</td>
</tr>
<tr>
<td>\mathaccent</td>
<td>0–7FFF</td>
</tr>
<tr>
<td>\delimiter</td>
<td>0–7FFFFFF</td>
</tr>
<tr>
<td>\radical</td>
<td>0–7FFFFFF</td>
</tr>
</tbody>
</table>

New primitives that are compatible with X\TeX:

<table>
<thead>
<tr>
<th>primitive</th>
<th>value range (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Umathchardef</td>
<td>0+0+0–7+FF+10FFFF</td>
</tr>
<tr>
<td>\Umathcharnumdef</td>
<td>-80000000–7FFFFFF</td>
</tr>
<tr>
<td>\Umathcode</td>
<td>0–10FFFF = 0+0+0–7+FF+10FFFF</td>
</tr>
<tr>
<td>\Udelcode</td>
<td>0–10FFFF = 0+0–FF+10FFFF</td>
</tr>
<tr>
<td>\Umathchar</td>
<td>0+0+0–7+FF+10FFFF</td>
</tr>
<tr>
<td>\Umathaccent</td>
<td>0+0+0–7+FF+10FFFF</td>
</tr>
<tr>
<td>\Udelimiter</td>
<td>0+0+0–7+FF+10FFFF</td>
</tr>
<tr>
<td>\Uradical</td>
<td>0+0–FF+10FFFF</td>
</tr>
<tr>
<td>\Umathcharnum</td>
<td>-80000000–7FFFFFF</td>
</tr>
<tr>
<td>\Umathcodenum</td>
<td>0–10FFFF = -80000000–7FFFFFF</td>
</tr>
<tr>
<td>\Udelcodenum</td>
<td>0–10FFFF = -80000000–7FFFFFF</td>
</tr>
</tbody>
</table>

Note 1: \Umathchardef<csname>="8"0"0 and \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 \mathoperator\size parameter (more information a following section).

Note 3: For these three primitives, all information is packed into a single signed integer. For the first two (\mathcharnum and \mathcodenum), 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 \delcodenum 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 \textgreek{LuaTEX} 0.65, \mathaccent accepts optional keywords to control various details regarding math accents. See section 5.7 below for details.

Note 5: \mathcharnumdef was added in release 0.72.

New primitives that exist in LuaTEX 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>\root</td>
<td>0+0–FF+10FFFF²</td>
</tr>
<tr>
<td>\overdelimiter</td>
<td>0+0–FF+10FFFF²</td>
</tr>
<tr>
<td>\underdelimiter</td>
<td>0+0–FF+10FFFF²</td>
</tr>
<tr>
<td>\delimiterover</td>
<td>0+0–FF+10FFFF²</td>
</tr>
<tr>
<td>\delimiterunder</td>
<td>0+0–FF+10FFFF²</td>
</tr>
</tbody>
</table>

5.3 Cramped math styles

\textgreek{LuaTEX} 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 \textgreek{LuaTEX}, the font dimension parameters that \textgreek{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>\mathquad</td>
<td>the width of 18mu's</td>
</tr>
</tbody>
</table>
\texttt{\textbackslash Umathaxis} \hspace{1cm} \text{height of the vertical center axis of the math formula above the baseline}

\texttt{\textbackslash Umathoperatorsize} \hspace{1cm} \text{minimum size of large operators in display mode}

\texttt{\textbackslash Umathoverbarkern} \hspace{1cm} \text{vertical clearance above the rule}

\texttt{\textbackslash Umathoverbarrule} \hspace{1cm} \text{the width of the rule}

\texttt{\textbackslash Umathoverbarvgap} \hspace{1cm} \text{vertical clearance below the rule}

\texttt{\textbackslash Umathunderbarkern} \hspace{1cm} \text{vertical clearance below the rule}

\texttt{\textbackslash Umathunderbarrule} \hspace{1cm} \text{the width of the rule}

\texttt{\textbackslash Umathunderbarvgap} \hspace{1cm} \text{vertical clearance above the rule}

\texttt{\textbackslash Umathradicalkern} \hspace{1cm} \text{vertical clearance above the rule}

\texttt{\textbackslash Umathradicalrule} \hspace{1cm} \text{the width of the rule}

\texttt{\textbackslash Umathradicalvgap} \hspace{1cm} \text{vertical clearance below the rule}

\texttt{\textbackslash Umathradicaldegreebefore} \hspace{1cm} \text{the forward kern that takes place before placement of the radical degree}

\texttt{\textbackslash Umathradicaldegreeafter} \hspace{1cm} \text{the backward kern that takes place after placement of the radical degree}

\texttt{\textbackslash Umathradicaldegreeraise} \hspace{1cm} \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 \textit{percents}, so 60\% is expressed as the integer 60.}

\texttt{\textbackslash Umathstackvgap} \hspace{1cm} \text{vertical clearance between the two elements in a \texttt{\atop} stack}

\texttt{\textbackslash Umathstacknumup} \hspace{1cm} \text{numerator shift upward in \texttt{\atop} stack}

\texttt{\textbackslash Umathstackdenomdown} \hspace{1cm} \text{denominator shift downward in \texttt{\atop} stack}

\texttt{\textbackslash Umathfractionrule} \hspace{1cm} \text{the width of the rule in a \texttt{\over} stack}

\texttt{\textbackslash Umathfractionnumvgap} \hspace{1cm} \text{vertical clearance between the numerator and the rule}

\texttt{\textbackslash Umathfractionnumup} \hspace{1cm} \text{numerator shift upward in \texttt{\over} stack}

\texttt{\textbackslash Umathfractiondenomvgap} \hspace{1cm} \text{vertical clearance between the denominator and the rule}

\texttt{\textbackslash Umathfractiondenomdown} \hspace{1cm} \text{denominator shift downward in \texttt{\over} stack}

\texttt{\textbackslash Umathfractiondelsize} \hspace{1cm} \text{minimum delimiter size for \texttt{\ldots withdelims}}

\texttt{\textbackslash Umathlimitabovevgap} \hspace{1cm} \text{vertical clearance for limits above operators}

\texttt{\textbackslash Umathlimitabovebgap} \hspace{1cm} \text{vertical baseline clearance for limits above operators}

\texttt{\textbackslash Umathlimitaboverule} \hspace{1cm} \text{space reserved at the top of the limit}

\texttt{\textbackslash Umathlimitbelowkgap} \hspace{1cm} \text{vertical clearance for limits below operators}

\texttt{\textbackslash Umathlimitbelowbgap} \hspace{1cm} \text{vertical baseline clearance for limits below operators}

\texttt{\textbackslash Umathlimitbelowrule} \hspace{1cm} \text{space reserved at the bottom of the limit}

\texttt{\textbackslash Umathoverdelimitervgap} \hspace{1cm} \text{vertical clearance for limits above delimiters}

\texttt{\textbackslash Umathoverdelimiterbgap} \hspace{1cm} \text{vertical baseline clearance for limits above delimiters}

\texttt{\textbackslash Umathunderdelimitervgap} \hspace{1cm} \text{vertical clearance for limits below delimiters}

\texttt{\textbackslash Umathunderdelimiterbgap} \hspace{1cm} \text{vertical baseline clearance for limits below delimiters}

\texttt{\textbackslash Umathsubshift} \hspace{1cm} \text{subscript drop for boxes and subformulas}

\texttt{\textbackslash Umathsubshiftdown} \hspace{1cm} \text{subscript drop for characters}

\texttt{\textbackslash Umathsupshift} \hspace{1cm} \text{superscript drop (raise, actually) for boxes and subformulas}

\texttt{\textbackslash Umathsupshiftup} \hspace{1cm} \text{superscript raise for characters}

\texttt{\textbackslash Umathsubsupshift} \hspace{1cm} \text{subscript drop in the presence of a superscript}
the top of standalone subscripts cannot be higher than this above the baseline
\texttt{\textbackslash Umathsupbottommin} the bottom of standalone superscripts cannot be less than this above the baseline
\texttt{\textbackslash Umathsupsubbottommax} the bottom of the superscript of a combined super- and subscript be at least as high as this above the baseline
\texttt{\textbackslash Umathsubsupvgap} vertical clearance between super- and subscript
\texttt{\textbackslash Umathspaceafterscript} additional space added after a super- or subscript
\texttt{\textbackslash Umathconnectoroverlapmin} minimum overlap between parts in an extensible recipe

Each of the parameters in this section can be set by a command like this:

\texttt{\textbackslash Umathquad\textbackslash displaystyle=1em}

they obey grouping, and you can use \texttt{\the\textbackslash Umathquad\textbackslash 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, \LaTeX{} 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 \texttt{tfm}-based fonts like \texttt{cmsy} and \texttt{cmex}), or based on the named values in a potential \texttt{MathConstants} table when the font is loaded via Lua. If there is a \texttt{MathConstants} table, this takes precedence over font dimensions, and in that case no attention is paid to which family is being assigned to: the \texttt{MathConstants} tables in the last assigned family sets all parameters.

In the table below, the one-letter style abbreviations and symbolic \texttt{tfm} font dimension names match those using in the \TeX{}book. Assignments to \texttt{\textfont} set the values for the cramped and uncramped display and text styles. Use \texttt{\scriptfont} for the script styles, and \texttt{\scriptscriptfont} for the scriptscript styles (totalling eight parameters for three font sizes). In the \texttt{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, \LaTeX{} 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>\texttt{\textbackslash Umathaxis}</td>
<td>–</td>
<td>AxisHeight</td>
<td>axis_height</td>
</tr>
<tr>
<td>\texttt{\textbackslash Umathoperatorsize}</td>
<td>D, D'</td>
<td>DisplayOperatorMinHeight</td>
<td>delim1</td>
</tr>
<tr>
<td>\texttt{\textbackslash Umathfractiondelsize}</td>
<td>T, T', S, S', SS, SS'</td>
<td>FractionDelimiterDisplayStyleSize</td>
<td>delim2</td>
</tr>
<tr>
<td>\texttt{\textbackslash Umathfractiondenomdown}</td>
<td>D, D'</td>
<td>FractionDenominatorDisplayStyleShiftDown</td>
<td>denom1</td>
</tr>
<tr>
<td>\texttt{\textbackslash Umathfractiondenomvgap}</td>
<td>T, T', S, S', SS, SS'</td>
<td>FractionDenominatorDisplayStyleGapMin</td>
<td>denom2</td>
</tr>
<tr>
<td>\texttt{\textbackslash Umathfractionnumup}</td>
<td>D, D'</td>
<td>FractionNumeratorDisplayStyleShiftUp</td>
<td>3*default_rule_thickness</td>
</tr>
<tr>
<td>\texttt{\textbackslash Umathfractionnumvgap}</td>
<td>T, T', S, S', SS, SS'</td>
<td>FractionNumeratorDisplayStyleGapMin</td>
<td>default_rule_thickness</td>
</tr>
<tr>
<td>\texttt{\textbackslash Umathfractionrule}</td>
<td>–</td>
<td>FractionRuleThickness</td>
<td>num1</td>
</tr>
</tbody>
</table>

Math 147
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textmathlimitabovegap</td>
<td>UpperLimitBaselineRiseMin big_op_spacing3</td>
</tr>
<tr>
<td>\textmathlimitabovekern</td>
<td>0 ^{1} big_op_spacing5</td>
</tr>
<tr>
<td>\textmathlimitabovevspace</td>
<td>UpperLimitGapMin big_op_spacing1</td>
</tr>
<tr>
<td>\textmathlimitbelowgap</td>
<td>LowerLimitBaselineDropMin big_op_spacing4</td>
</tr>
<tr>
<td>\textmathlimitbelowkern</td>
<td>0 ^{1} big_op_spacing5</td>
</tr>
<tr>
<td>\textmathlimitbelowvspace</td>
<td>LowerLimitGapMin big_op_spacing2</td>
</tr>
<tr>
<td>\textmathoverdelimitervspace</td>
<td>StretchStackGapBelowMin big_op_spacing1</td>
</tr>
<tr>
<td>\textmathunderdelimitervspace</td>
<td>StretchStackGapAboveMin big_op_spacing2</td>
</tr>
<tr>
<td>\textmathunderbarkern</td>
<td>StretchStackBottomShiftDown big_op_spacing4</td>
</tr>
<tr>
<td>\textmathunderbarrule</td>
<td>OverbarExtraAscender default_rule_thickness</td>
</tr>
<tr>
<td>\textmathunderbarvspace</td>
<td>OverbarRuleThickness default_rule_thickness</td>
</tr>
<tr>
<td>\textmathunderbarvspace</td>
<td>OverbarVerticalGap 3*default_rule_thickness</td>
</tr>
<tr>
<td>\textmathquad</td>
<td>\langle font_size(f)\rangle ^{1} math_quad</td>
</tr>
<tr>
<td>\textmathradicalkern</td>
<td>RadicalExtraAscender default_rule_thickness</td>
</tr>
<tr>
<td>\textmathradicalrule</td>
<td>RadicalRuleThickness \langle not set\rangle ^{2}</td>
</tr>
<tr>
<td>\textmathradicalvgap</td>
<td>D, D' RadicalDisplayStyleVerticalGap \langle default_rule_thickness+ (abs(math_x_height)/4)\rangle ^{1}</td>
</tr>
<tr>
<td>\textmathradicaldegreebefore</td>
<td>RadicalKernBeforeDegree \langle not set\rangle ^{2}</td>
</tr>
<tr>
<td>\textmathradicaldegreeafter</td>
<td>RadicalKernAfterDegree \langle not set\rangle ^{2}</td>
</tr>
<tr>
<td>\textmathradicaldegreeafter</td>
<td>RadicalDegreeBottomRaisePercent \langle not set\rangle ^{2}</td>
</tr>
<tr>
<td>\textmathspaceafterscript</td>
<td>SpaceAfterScript script_space ^{4}</td>
</tr>
<tr>
<td>\textmathstackdenomdown</td>
<td>StackBottomDisplayStyleShiftDown denom1</td>
</tr>
<tr>
<td>\textmathstacknumup</td>
<td>StackTopDisplayStyleShiftUp num1</td>
</tr>
<tr>
<td>\textmathstackvgap</td>
<td>StackDisplayStyleGapMin 7*default_rule_thickness</td>
</tr>
<tr>
<td>\textmathsubshiftdown</td>
<td>SubscriptShiftDown sub1</td>
</tr>
<tr>
<td>\textmathsubshiftdown</td>
<td>SubscriptBaselineDropMin sub_drop</td>
</tr>
<tr>
<td>\textmathsubupshiftdown</td>
<td>SubscriptShiftDownWithSuperscript sup1</td>
</tr>
<tr>
<td>\textmathsubupshifttopmax</td>
<td>SubscriptTopMax (abs(math_x_height * 4) / 5)</td>
</tr>
<tr>
<td>\textmathsubupvtopgap</td>
<td>SubSuperscriptMaxMin (abs(math_x_height) / 4)</td>
</tr>
<tr>
<td>\textmathsubupbottommin</td>
<td>SuperscriptBottomMin sup_3</td>
</tr>
<tr>
<td>\textmathsubupbottommax</td>
<td>SuperscriptBaselineDropMax sup_3</td>
</tr>
<tr>
<td>\textmathsubupshiftup</td>
<td>SuperscriptShiftUp sup_1</td>
</tr>
<tr>
<td>\textmathsubupshifttopmax</td>
<td>SuperscriptShiftUpCramped sup_3</td>
</tr>
<tr>
<td>\textmathsubupshiftvspace</td>
<td>SuperscriptShiftUpWithSubscript (abs(math_x_height * 4) / 5)</td>
</tr>
<tr>
<td>\textmathunderbarkern</td>
<td>UnderbarExtraDescender default_rule_thickness</td>
</tr>
<tr>
<td>\textmathunderbarrule</td>
<td>UnderbarRuleThickness default_rule_thickness</td>
</tr>
<tr>
<td>\textmathunderbarvspace</td>
<td>UnderbarVerticalGap 3*default_rule_thickness</td>
</tr>
<tr>
<td>\textmathconnectoroverlapspace</td>
<td>MinConnectorOverlap 0 ^{5}</td>
</tr>
</tbody>
</table>

Note 1: OpenType fonts set \textmathlimitabovekern and \textmathlimitbelowkern to zero and set \textmathquad to the font size of the used font, because these are not supported in the MATH table.

Note 2: TFM fonts do not set \textmathradicalrule because \TeX{}82 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 \textmathradicaldegreebefore, \textmathradicaldegreeafter, and \textmathradicaldegreeraise. These are then automatically initialized to 5/18quad, –10/18quad, and 60.

Note 3: If tfm fonts are used, then the \textmathradicalvgap 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 \( \text{T\kern-.1667em E\kern-.125em X82} \), but that compatibility is only partial: once the \texttt{\textbackslash Umathradicalv-gap} is set, it will not be recalculated any more.

Note 4: (also if \texttt{\textbackslash T\kern-.1667em F\kern-.125em M} fonts are used) A similar situation arises wrt. \texttt{\textbackslash Umathspaceafterscript}: it is not set until the first time \texttt{\textbackslash L\kern-.125em U\kern-.125em A\kern-.125em T\kern-.125em X} has to typeset a formula. This provides some backward compatibility with \( \text{T\kern-.1667em E\kern-.125em X82} \). But once the \texttt{\textbackslash Umathspaceafterscript} is set, \texttt{\textbackslash scriptspace} will never be looked at again.

Note 5: \texttt{\textbackslash T\kern-.1667em F\kern-.125em M} fonts set \texttt{\textbackslash Umathconnectoroverlapmin} to zero because \( \text{T\kern-.1667em E\kern-.125em X82} \) always stacks extensibles without any overlap.

Note 6: The \texttt{\textbackslash Umathoperatorsize} is only used in \texttt{\textbackslash displaystyle}, and is only set in \texttt{O\!P\!E\!N\!T\!Y\!P\!E} fonts. In \texttt{T\kern-.1667em F\kern-.125em M} 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 \( \text{T\kern-.1667em E\kern-.125em X82} \).

Note 7: The \texttt{\textbackslash Umathradicaldegree raise} 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{\textbackslash SubscriptShiftDownWithSuperscript} 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{\textbackslash FractionDelimiterDisplayStyleSize} and \texttt{\textbackslash FractionDelimiterSize} 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 \( \text{T\kern-.1667em E\kern-.125em X} \)book). The primitive names are a simple matter of combining two math atom types, but for completeness’ sake, here is the whole list:

\begin{verbatim}
\texttt{\textbackslash Umathordordspacing} \hspace{2cm} \texttt{\textbackslash Umathbinopspacing}
\texttt{\textbackslash Umathordopspacing} \hspace{2cm} \texttt{\textbackslash Umathbinbinspacing}
\texttt{\textbackslash Umathordbinspacing} \hspace{2cm} \texttt{\textbackslash Umathbinrelspacing}
\texttt{\textbackslash Umathordrelspacing} \hspace{2cm} \texttt{\textbackslash Umathbinopenspacing}
\texttt{\textbackslash Umathordopenspacing} \hspace{2cm} \texttt{\textbackslash Umathbinclosespacing}
\texttt{\textbackslash Umathordclosespacing} \hspace{2cm} \texttt{\textbackslash Umathbinpunctspacing}
\texttt{\textbackslash Umathordpunctspacing} \hspace{2cm} \texttt{\textbackslash Umathbinnerspacing}
\texttt{\textbackslash Umathordinnerspacing} \hspace{2cm} \texttt{\textbackslash Umathrelordspacing}
\texttt{\textbackslash Umathopordspacing} \hspace{2cm} \texttt{\textbackslash Umathrelopspacing}
\texttt{\textbackslash Umathopopspacing} \hspace{2cm} \texttt{\textbackslash Umathrelbinspacing}
\texttt{\textbackslash Umathopbinspacing} \hspace{2cm} \texttt{\textbackslash Umathrelrelspacing}
\texttt{\textbackslash Umathoprelspacing} \hspace{2cm} \texttt{\textbackslash Umathrelopenspacing}
\texttt{\textbackslash Umathopopenspacing} \hspace{2cm} \texttt{\textbackslash Umathrelclosespacing}
\texttt{\textbackslash Umathopclosespacing} \hspace{2cm} \texttt{\textbackslash Umathrelpunctspacing}
\texttt{\textbackslash Umathoppunctspacing} \hspace{2cm} \texttt{\textbackslash Umathrelinnerspacing}
\texttt{\textbackslash Umathopinnerspacing} \hspace{2cm} \texttt{\textbackslash Umathopenordspacing}
\texttt{\textbackslash Umathbinordspacing} \hspace{2cm} \texttt{\textbackslash Umathopenopspacing}
\end{verbatim}
These parameters are of type \muskip, so setting a parameter can be done like this:

\Umathopordspacing\displaystyle=4\mu\text{ plus }2\mu

They are all initialized by initex to the values mentioned in the table in Chapter 18 of the \TeXbook.

Note 1: for ease of use as well as for backward compatibility, \thinmuskip, \medmuskip and \thickmuskip are treated especially. In their case a pointer to the corresponding internal parameter is saved, not the actual \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 * in the \TeXbook. 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 \Umathaccent.

The keyword bottom after \Umathaccent signals that a bottom accent is needed, and the keyword 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 `top_accent` value, then this value will be used to place the accent instead of the `\skewchar` kern used by \TeX\textgreek{82}.

The `top_accent` value represents a vertical line somewhere in the accentee. The accent will be shifted horizontally such that its own `top_accent` line coincides with the one from the accentee. If the `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 `x_height` of the font of the accentee (as explained in the \TeX\textgreek{X}book), but if value that turns out to be zero and the font had a MathConstants table, then `AccentBaseHeight` is used instead.

If a math bottom accent has to be placed, the `bot_accent` value is checked instead of `top_accent`. Because bottom accents do not exist in \TeX\textgreek{82}, the `\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 `\Uroot` allows the construction of a radical node including a degree field. Its syntax is an extension of `\Uradical`:

\begin{verbatim}
\Uradical <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 `\Umathradicaldegreebefore`, `\Umathradicaldegreeafter`, and `\Umathradicaldegreeraise`. The degree will be typeset in `\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 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, \LaTeX 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 \LaTeX 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 \Uunderdelimiter and \Uoverdelimiter (both from 0.35) allow the placement of a subscript or superscript on an automatically extensible item and \Udelimiterunder and \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 \Umathunderdelimiterbgap, \Umathunderdelimitervgap, \Umathoverdelimiterbgap, and \Umathoverdelimitervgap in a similar way as limit placements on large operators. The superscript in \Uoverdelimiter is typeset in a suitable scripted style, the subscript in \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 ^, _, $, and $$.  

<table>
<thead>
<tr>
<th>primitive</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Usuperscript</td>
<td>Duplicates the functionality of ^</td>
</tr>
<tr>
<td>\Usubscript</td>
<td>Duplicates the functionality of _</td>
</tr>
<tr>
<td>\Ustartmath</td>
<td>Duplicates the functionality of $, when used in non-math mode.</td>
</tr>
</tbody>
</table>
\texttt{\textbackslash stopmath} \quad \text{Duplicates the functionality of \$}, when used in inline math mode.
\texttt{\textbackslash startdisplaymath} \quad \text{Duplicates the functionality of $$}, when used in non-math mode.
\texttt{\textbackslash stopdisplaymath} \quad \text{Duplicates the functionality of $$}, 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{\mathchar} and \texttt{\mathchar} and control sequences that are the result of \texttt{\mathchardef} or \texttt{\mathchardef} are also acceptable in the horizontal and vertical modes. In those cases, the \texttt{\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 glue.

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 \LaTeX, it works like this: if \LaTeX 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 \LaTeX 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 \LaTeX 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 \LaTeX documents.

Fifth, \texttt{\setlanguage} no longer creates whatsis. 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 whatsis. These whatsis 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 \TeX 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 \LaTeX, 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{tracing\_commands} 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 \LaTeX is quite different from the one in \TeX, 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. \texttt{-} indicates a desired simple discretionary, cf. \texttt{-} 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. \texttt{=} 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\textbackslash discretionary{-}{}{}{}ble)</td>
</tr>
<tr>
<td>ba{\textbackslash k-}{}{\textbackslash c}ken</td>
<td>backen</td>
<td>ba\textbackslash discretionary{\textbackslash k-}{}{\textbackslash c}ken</td>
</tr>
</tbody>
</table>

The resultant patterns and exception dictionary will be stored under the language code that is the present value of \texttt{\textbackslash 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 Lua\TeX 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 Lua\TeX 0.63.0, it is possible to specify extra hyphenation points in compound words by using \{-\}{\textbackslash -}{\textbackslash -} for the explicit hyphen character (replace \texttt{-} 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{\textbackslash hyphenation\{multi{-}\}{\textbackslash -}{\textbackslash word{-}\{}\{}\{}\textbackslash -\}{\textbackslash -}\textbackslash boun-daries}

The motivation behind the \texttt{\textbackslash epsilon\-\TeX} extension \texttt{\textbackslash savinghyphcodes} was that hyphenation heavily depended on font encodings. This is no longer true in Lua\TeX, and the corresponding primitive is ignored pending complete removal. The future semantics of \texttt{\textbackslash uppercase} and \texttt{\textbackslash lowercase} are still under consideration, no changes have taken place yet.

### 6.4 Applying hyphenation

The internal structures Lua\TeX 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 Lua\TeX 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 \texttt{Lua\TeX} and \TeX82 that are a direct result of that:

- \texttt{Lua\TeX} 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\textunderscore 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.
- \texttt{Lua\TeX} 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=\textbackslash-, \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}.

\texttt{Lua\TeX} 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\textunderscore 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 \texttt{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 \texttt{office}, hyphenated \texttt{of-fice}, using a ‘normal’ font with all the \texttt{f-f} and \texttt{f-i} type ligatures:

Initial: \{o\}{f\}{f\}{i\}{c\}{e}\n
After hyphenation: \{o\}{f\}{f\}{{-},{},{}}{f\}{i\}{c\}{e}\n
First ligature stage: \{o\}{{f-},{f},{<ff}>}{i\}{c\}{e}\n
Final result: \{o\}{{f-},{<fi>},{<ffi>}}{c\}{e}\n
That’s bad enough, but let us assume that there is also a hyphenation point between the \texttt{f} and the \texttt{i}, to create \texttt{of-f-ice}. Then the final result should be:

\{o\}{{f-},{{f-},{i},{<fi>}},{{<ff>-},{i},{<ffi>}}}{c\}{e}\n
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:

\begin{center}
\begin{tabular}{c|ccc}
 & pre & post & replace \\
\hline
topdisc & \texttt{f}\textsuperscript{-1} & sub\texttt{1} & sub\texttt{2} \\
sub\texttt{1} & \texttt{f}\textsuperscript{-2} & \texttt{i}\textsuperscript{3} & \texttt{<fi}>\textsuperscript{4} \\
sub\texttt{2} & \texttt{<ff>-}\textsuperscript{5} & \texttt{i}\textsuperscript{6} & \texttt{<ffi>}\textsuperscript{7} \\
\end{tabular}
\end{center}
When line breaking is choosing its breakpoints, the following fields will eventually be selected:

- **of-f-ice**  
  - $f^{-1}$  
  - $f^{-2}$  
  - $i^3$

- **off-ice**  
  - $f^{-1}$  
  - $\langle fi \rangle^4$

- **office**  
  - $\langle ffi \rangle^7$

The current solution in LuaTeX 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 LuaTeX 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>$\langle fi \rangle^4$</td>
</tr>
<tr>
<td>disc1.replace</td>
<td>$\langle ffi \rangle^7$</td>
</tr>
<tr>
<td>disc2.pre</td>
<td>$f^{-2}$</td>
</tr>
<tr>
<td>disc2.post</td>
<td>$i^3$</td>
</tr>
<tr>
<td>disc2.replace</td>
<td>$\langle ff \rangle^{-5}$</td>
</tr>
</tbody>
</table>

What is actually generated after ligaturing has been applied is therefore:

```
{o}{{f-},
  {{\langle fi \rangle},
   {{\langle ffi \rangle}}}   
{{f-},
  {i},
  {\langle ff \rangle-}}}{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 $\langle ff \rangle$—appearing in `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 LuaTeX 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 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 `define_font` callback, or if they result from the normal TFM/VF reading routines if there is no `define_font` callback defined.

The column ‘from vf’ means that this key will be created by the `font.read_vf()` routine, ‘from tfm’ means that the key will be created by the `font.read_tfm()` routine, and ‘used’ means whether or not the LuaTeX 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>expected size (default: 655360 == 10pt)</td>
</tr>
<tr>
<td>designsize</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>number</td>
<td>expected size (default: 655360 == 10pt)</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 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, LuaTeX assumes per-glyph touncement 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>
shrink no no yes number the ‘shrink’ value from \pdffontexpand
step no no yes number the ‘step’ value from \pdffontexpand
auto_expand no no yes boolean the ‘autoexpand’ keyword from \pdffontexpand
expansion_factor no no no number the actual expansion factor of an expanded font
attributes no no yes string the \pdffontattr
cache no no yes string this key controls caching of the lua table on the tex end. yes: use a reference to the table that is passed to LuaTeX (this is the default). no: don’t store the table reference, don’t cache any lua data for this font. renew: don’t store the table reference, but save a reference to the table that is created at the first access to one of its fields in font.fonts. (new in 0.40.0, before that caching was always yes). Note: the saved reference is thread-local, so be careful when you are using coroutines: an error will be thrown if the table has been cached in one thread, but you reference it from another thread (≈ coroutine)
nomath no no yes boolean 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)
slant no no yes number This has the same semantics as the SlantFont operator in font map files. (0.47.0)
extent no no yes number This has the same semantics as the ExtendFont operator in font map files. (0.50.0)

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:
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</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>slant</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>space</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>space_stretch</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>space_shrink</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>x_height</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>quad</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>extra_space</td>
<td>7</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:

```
[102] = {
    ['width'] = 200250,
    ['height'] = 455111,
    ['depth'] = 0,
    ['italic'] = 50973,
    ['kerns'] = {
```
The following top-level keys can be present inside a character hash:

- **width**: Character's width, in sp (default 0)
- **height**: Character's height, in sp (default 0)
- **depth**: Character's depth, in sp (default 0)
- **italic**: Character's italic correction, in sp (default zero)
- **top_accent**: Character's top accent alignment place, in sp (default zero)
- **bot_accent**: Character's bottom accent alignment place, in sp (default zero)
- **left_protruding**: Character's \lpc ode
- **right_protruding**: Character's \rpc ode
- **expansion_factor**: Character's \ef code
- **tounicode**: Character's Unicode equivalent(s), in UTF-16BE hexadecimal format
- **next**: The 'next larger' character index
- **extensible**: The constituent parts of an extensible recipe
- **vert_variants**: Constituent parts of a vertical variant set
horiz_variants  no  no  yes  table  constituent parts of a horizontal variant set
kerns         no  yes  yes  table  kerning information
ligatures     no  yes  yes  table  ligaturing information
commands      yes  no  yes  array  virtual font commands
name          no  no  no  no  string  the character (PostScript) name
index         no  no  yes  yes  number  the (OpenType or TrueType) font glyph index
used          no  no  yes  yes  boolean  typeset already (default: false)?
mathkern      no  no  yes  yes  table  math cut-in specifications

The values of top_accent, bot_accent 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 in turn can be overruled by vert_variants.

The extensible table is very simple:

key  type  description
---  -----  -------------------
top  number ‘top’ character index
mid  number ‘middle’ character index
bot  number ‘bottom’ character index
rep  number ‘repeatable’ character index

The horiz_variants and vert_variants are arrays of components. Each of those components is itself a hash of up to five keys:

key  type  explanation
-----  -----  -------------------
glyph number  The character index (note that this is an encoding number, not a name).
extender number  One (1) if this part is repeatable, zero (0) otherwise.
start number  Maximum overlap at the starting side (in scaled points).
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:

key          type          description
  type number  the type of this ligature command, default 0
  char number  the character index of the resultant ligature

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'.

textual (Knuth) number string result
  l + r :=: n 0 =: | n
  l + r =:= n 1 =:= | nr
  l + r |==: n 2 |==: | ln
  l + r |=: n 3 |=: | lnr
  l + r =:=> n 5 =:=> n|r
  l + r |=:=> n 6 |=:=> |ln
  l + r |=:=> n 7 |=:=> l|nr
  l + r |=:>> n 11 |=:>> ln|r

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:
value   description
real    this is a base font
virtual this is a virtual font

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:

value   description
type1   this is a PostScript Type1 font
type3   this is a bitmapped (PK) font
trueype this is a TrueType or TrueType-based OpenType font
opentype this is a PostScript-based OpenType font

(`type3` fonts are provided for backward compatibility only, and do not support the new wide encoding options.)

Values for `embedding` are:

value   description
no      don’t embed the font at all
subset  include and attempt to subset the font
full    include this font in its entirety

It is not possible to artificially modify the transformation matrix for the font at the moment.

The other fields are used as follows: The `full`name 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 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 fullname of the font.

Typeset strings are written out in a wide format using 2 bytes per glyph, using the 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 index numbers for Type1 fonts is by loading the font via fontloader.open; use the table indices as 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 define_font as a virtual font:

- Set the top-level key type to virtual.
- Make sure there is at least one valid entry in fonts (see below).
- Give a commands array to every character (see below).

The presence of the toplevel type key with the specific value 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 base or real to type will inhibit LuaTeX from looking for a virtual font file, thereby saving you a disk search.

The 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 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 ptrmr8a loaded at 10pt, and the second is 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 fonts 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></td>
<td>do nothing</td>
</tr>
<tr>
<td>pop</td>
<td>0</td>
<td></td>
<td>pop position</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 \special command</td>
</tr>
<tr>
<td>lua</td>
<td>1</td>
<td>string</td>
<td>execute a Lua script (at \latex time)</td>
</tr>
<tr>
<td>image</td>
<td>1</td>
<td>image</td>
<td>output an image (the argument can be either an &lt;image&gt; variable or an image_spec 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 can not 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
      end
      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 \texttt{lastnodetype} primitive is \texttt{\LaTeX} 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 \texttt{\LaTeX}. 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>next</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>next</td>
<td><code>&lt;node&gt;</code></td>
<td>pointer to the next attribute</td>
</tr>
<tr>
<td>number</td>
<td><code>number</code></td>
<td>the attribute type id</td>
</tr>
<tr>
<td>value</td>
<td><code>number</code></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>action_type</td>
<td><code>number</code></td>
<td></td>
</tr>
<tr>
<td>action_id</td>
<td><code>number or string</code></td>
<td></td>
</tr>
<tr>
<td>named_id</td>
<td><code>number</code></td>
<td></td>
</tr>
<tr>
<td>file</td>
<td><code>string</code></td>
<td></td>
</tr>
<tr>
<td>new_window</td>
<td><code>number</code></td>
<td></td>
</tr>
<tr>
<td>data</td>
<td><code>string</code></td>
<td></td>
</tr>
<tr>
<td>ref_count</td>
<td><code>number</code></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

Id: 0

<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. (0.46.0), 3 = paragraph indentation box, 4 = alignment column or row, 5 = 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**

<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><code>&lt;node&gt;</code></td>
<td>unused</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. <strong>see 8.1.4.7</strong></td>
</tr>
</tbody>
</table>

---

8.1.2.4 **ins nodes**

Valid fields: **attr, cost, depth, height, spec, head**

<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><code>&lt;node&gt;</code></td>
<td>the penalty associated with this insert</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>the first node of the body of this insert</td>
</tr>
<tr>
<td>depth</td>
<td>number</td>
<td>a pointer to the <code>\splittopskip</code> 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**

<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><code>&lt;node&gt;</code></td>
<td>unused</td>
</tr>
<tr>
<td>class</td>
<td><code>&lt;node&gt;</code></td>
<td>unused</td>
</tr>
<tr>
<td>mark</td>
<td><code>&lt;node&gt;</code></td>
<td>unused</td>
</tr>
</tbody>
</table>

<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><code>&lt;node&gt;</code></td>
<td>unused</td>
</tr>
<tr>
<td>class</td>
<td><code>&lt;node&gt;</code></td>
<td>unused</td>
</tr>
<tr>
<td>mark</td>
<td><code>&lt;node&gt;</code></td>
<td>unused</td>
</tr>
</tbody>
</table>
field type explanation
subtype number unused
attr <node> the mark class
class number a table representing a token list
mark table

8.1.2.6 adjust nodes

Valid fields: attr, head
Id: 5

field type explanation
subtype number 0 = normal, 1 = ‘pre’
attr <node> adjusted material
head <node> adjusted material

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

field type explanation
subtype number 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)
attr <node> pre <node> pointer to the pre-break text
post <node> replace <node> pointer to the no-break text

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**

<table>
<thead>
<tr>
<th>Id</th>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>subtype</td>
<td>number</td>
<td>0 = \skip, 1–18 = internal glue parameters, 100-103 = ‘leader’ subtypes</td>
</tr>
<tr>
<td></td>
<td>attr</td>
<td>&lt;node&gt;</td>
<td>pointer to a glue_spec item</td>
</tr>
<tr>
<td></td>
<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 \splittopskip`
- `12 \tabskip`
- `13 \spaceskip`
- `14 \xspaceskip`
- `15 \parfillskip`
- `16 \thinspace`
- `17 \medmuskip`
- `18 \thickmuskip`
- `100 \leaders`
- `101 \cleaders`
- `102 \xleaders`
- `103 \gleaders`

### 8.1.2.10 kern nodes

Valid fields: **attr**, **kern**, **expansion_factor**

<table>
<thead>
<tr>
<th>Id</th>
<th>field</th>
<th>type</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>subtype</td>
<td>number</td>
<td>0 = \mathsurround, 1–18 = internal glue parameters, 100-103 = ‘leader’ subtypes</td>
</tr>
<tr>
<td></td>
<td>attr</td>
<td>&lt;node&gt;</td>
<td>width of the \mathsurround kern</td>
</tr>
<tr>
<td></td>
<td>surround</td>
<td>number</td>
<td>width of the \mathsurround kern</td>
</tr>
</tbody>
</table>
8.1.2.11 penalty nodes

Valid fields: attr, penalty
Id: 12

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

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:

bit   meaning
0   character
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 \LaTeX{} already replaced multiple instances of fonts in the backend by scaling but contrary to pdf\TeX{} in \LaTeX{} 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: \textit{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 \textit{<kernel>}).

The \textit{next} and \textit{prev} fields for these subnodes are unused.

#### 8.1.3.1.1 math_char and math_text_char subnodes

Valid fields: \textit{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>
char  number
fam   number

The \textit{math_char} is the simplest subnode field, it contains the character and family for a single glyph object. The \textit{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: \texttt{attr, head}

<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 \texttt{sub_box}, the \texttt{head} points to a ‘normal’ \texttt{vbox} or \texttt{hbox}. For \texttt{sub_mlist}, the \texttt{head} points to a math list that is yet to be converted.

A warning: never assign a node list to the \texttt{head} field unless you are sure its internal link structure is correct, otherwise an error may be result.

Note: the new field name \texttt{head} was introduced in 0.65 to replace the old name \texttt{list}. Use of the name \texttt{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 \texttt{next} and \texttt{prev} fields are unused.

#### 8.1.3.2.1 delim subnodes

Valid fields: \texttt{attr, small_fam, small_char, large_fam, large_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>
<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 \texttt{large_char} and \texttt{large_fam} can be zero, in that case the font that is sed for the \texttt{small_fam} is expected to provide the large version as an extension to the \texttt{small_char}. 
8.1.3.3 Math core nodes

First, there are the objects (the TeXbook calls them '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.

number node sub type
0   Ord
1   Op, \displaylimits
2   Op, \limits
3   Op, \nolimits
4   Bin
5   Rel
6   Open
7   Close
8   Punct
9   Inner
10  Under
11  Over
12  Vcent

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
nucleus  <kernel>
sub      <kernel>
sup      <kernel>
accent   <kernel>
bot_accent <kernel>

8.1.3.3 style nodes

Valid fields: attr, style
Id: 14

field   type   explanation
style   string  contains the style

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.

8.1.3.3.4 choice nodes

Valid fields: attr, display, text, script, scriptscript
Id: 15

field   type   explanation
attr    <node>
display <node>
text    <node>
script  <node>
scriptscript <node>

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.

8.1.3.3.5 radical nodes

Valid fields: attr, nucleus, sub, sup, left, degree
Id: 24

field   type   explanation
attr    <node>
nucleus <kernel>
sub     <kernel>
sup     <kernel>
left    <delim>
degree  <kernel> Only set by \Uroot
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><code>attr</code></td>
<td><code>&lt;node&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>width</code></td>
<td>number</td>
<td></td>
</tr>
<tr>
<td><code>num</code></td>
<td><code>&lt;kernel&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>denom</code></td>
<td><code>&lt;kernel&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>left</code></td>
<td><code>&lt;delim&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>right</code></td>
<td><code>&lt;delim&gt;</code></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><code>subtype</code></td>
<td>number</td>
<td>1 = \left, 2 = \middle, 3 = \right</td>
</tr>
<tr>
<td><code>attr</code></td>
<td><code>&lt;node&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>delim</code></td>
<td><code>&lt;delim&gt;</code></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 |
8.1.4.2 write nodes

Valid fields: attr, stream, data
Id: 8, 1

<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>stream</td>
<td>number</td>
<td>TeX's stream id number</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>file name</td>
</tr>
<tr>
<td>ext</td>
<td>string</td>
<td>file extension</td>
</tr>
<tr>
<td>area</td>
<td>string</td>
<td>file area (this may become obsolete)</td>
</tr>
</tbody>
</table>

8.1.4.3 close nodes

Valid fields: attr, stream
Id: 8, 2

<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>stream</td>
<td>number</td>
<td>TeX's stream id number</td>
</tr>
</tbody>
</table>

8.1.4.4 special nodes

Valid fields: attr, data
Id: 8, 3

<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>the \special information</td>
</tr>
</tbody>
</table>

8.1.4.5 language nodes

LuaTeX 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`  
Id: 8, 7

<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`  
Id: 8, 8

<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 LuaTEX.

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 \textsf{LaTeX}.

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

Field       Type          Explanation
attr       <node>          
width       number          
height      number          
depth       number          
named_id    number          is the thread_id a string value?
thread_id   number or string the thread id
thread_attr number          extra thread information

8.1.4.17 pdf_start_thread nodes
Valid fields: attr, width, depth, height, named_id, thread_id, thread_attr
Id: 8, 21

Field       Type          Explanation
attr       <node>          
width       number          
height      number          
depth       number          
named_id    number          is the thread_id a string value?
thread_id   number or string the thread id
thread_attr number          extra thread information

8.1.4.18 pdf_end_thread nodes
Valid fields: attr
Id: 8, 22

Field   Type   Explanation
attr     <node>
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 \latelu. 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></td>
</tr>
<tr>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;node&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>table</td>
<td></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 LuaTEX and LuaJITTEX with and without just in time compilation enabled, and with and without using 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 LuaTEX{} has, an internal reference will always be a simple data type (like a number or light userdata in 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 LuaTEX 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 otf 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>
</tbody>
</table>
getprev
getlist
getleader
getsubtype
has_glyph
has_attribute
has_field
hpnext
id
insert_after
insert_before
is_direct
is_node
kerning
last_node
length
ligaturing
mlist_to_hlist
new
next
prev
tostring
protect_glyphs
protrusion_skippable
remove
set_attribute
setbox
setfield
slide
subtype
tail
todirect	onode
traverse
traverse_id
type
types
unprotect_glyphs
unset_attribute
usedlist
vpack
whatsthis
write
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 \LaTeXe 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 of\{\{}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 \epsilon-\TeX 2.2

- The \epsilon-\TeX functionality is always present and enabled (but see below about \TeXXXeT), so the prepended asterisk or -etex switch for ini\TeX is not needed.
- \TeXXXeT is not present, so the primitives

\TeXXeTstate
\beginR
\beginL
\endR
\endL

are missing.
- Some of the tracing information that is output by \epsilon-\TeX's \tracingassigns and \tracingre- stores 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 \epsilon-\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 PDF\TeX). Likewise for virtual fonts (\LaTeX uses the variable OVFFONTS instead of VFFONTS).

9.3 Changes from PDF\TeX 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 `\pdfsnappycomp`.

• 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`, `\knbccode`, 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 \LaTeX\ 0.42:
  
  `\pdfoptionalwaysusepdffilebox` `\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.

• \LaTeX\ 0.45.0 introduces two extra token lists, `\pdfxformresources` and `\pdfxformattr`, as an alternative to `\pdfxform` keywords.

• As of \LaTeX\ 0.50.0 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 \LaTeX\ 0.75.0, the extended 16-bit math primitives (`\omathcode` etc.) have been removed.

• Starting with \LaTeX\ 0.63.0, OCP processing is no longer supported at all. As a consequence, the following primitives have been removed:
  
  `\ocp` `\pushocplist` 
  `\externalocep` `\popocplist` 
  `\ocplist` `\clearocplist`
\addbeforeocplist \removeafterocplist
\addafterocplist \ocptracelevel
\removebeforeocplist

- **\LaTeX** only understands 4 of the 16 direction specifiers of **\aleph**: TLT (latin), TRT (arabic), RTT (cjk), LTL (mongolian). All other direction specifiers generate an error (**\LaTeX** 0.45).
- The input translations from **\aleph** are not implemented, the related primitives are not available:
  \begin{verbatim}
  \DefaultInputMode \DefaultInputTranslation
  \noDefaultInputMode \noDefaultInputTranslation
  \noInputMode \noInputTranslation
  \InputMode \InputTranslation
  \DefaultOutputMode \DefaultOutputTranslation
  \noDefaultOutputMode \noDefaultOutputTranslation
  \noOutputMode \noOutputTranslation
  \OutputMode \OutputTranslation
  \end{verbatim}
- The `\hoffset` bug when `pagedir TRT` is fixed, removing the need for an explicit fix to `\hoffset`.
- A bug causing `\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 `\tracingcommands` output.
- The internal function `\scan_dir()` has been renamed to `\scan_direction()` to prevent a naming clash, and it now allows an optional space after the direction is completely parsed.
- The `^^` 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:
  - `\-8bit`
  - `\-translate-file=TCXNAME`
  - `\mltex`
  - `\-enc`
  - `\-etex`
- `\openout` whatsis 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: `shell-escape` is off (but that is not a problem because of *Lua*’s os.execute), and the paranoia checks on `openin` and `openout` do not happen (however, it is easy for a *Lua* script to do this itself by overloading `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{\scode}, \texttt{\lcode} 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 $\texttt{\catcode}$, $\texttt{\sfcode}$, $\texttt{\lccode}$ and $\texttt{\uccode}$ assignments do not yet show up when using the etex tracing routines $\texttt{\tracingassigns}$ and $\texttt{\tracingrestores}$ (code simply not written yet).

A side-effect of the current implementation is that $\texttt{\global}$ is now more expensive in terms of processing than non-global assignments.

See $\texttt{mathcodes.c}$ and $\texttt{textcodes.c}$ 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 $\texttt{read_xxx_file}$ 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 $\texttt{getc}$ 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.