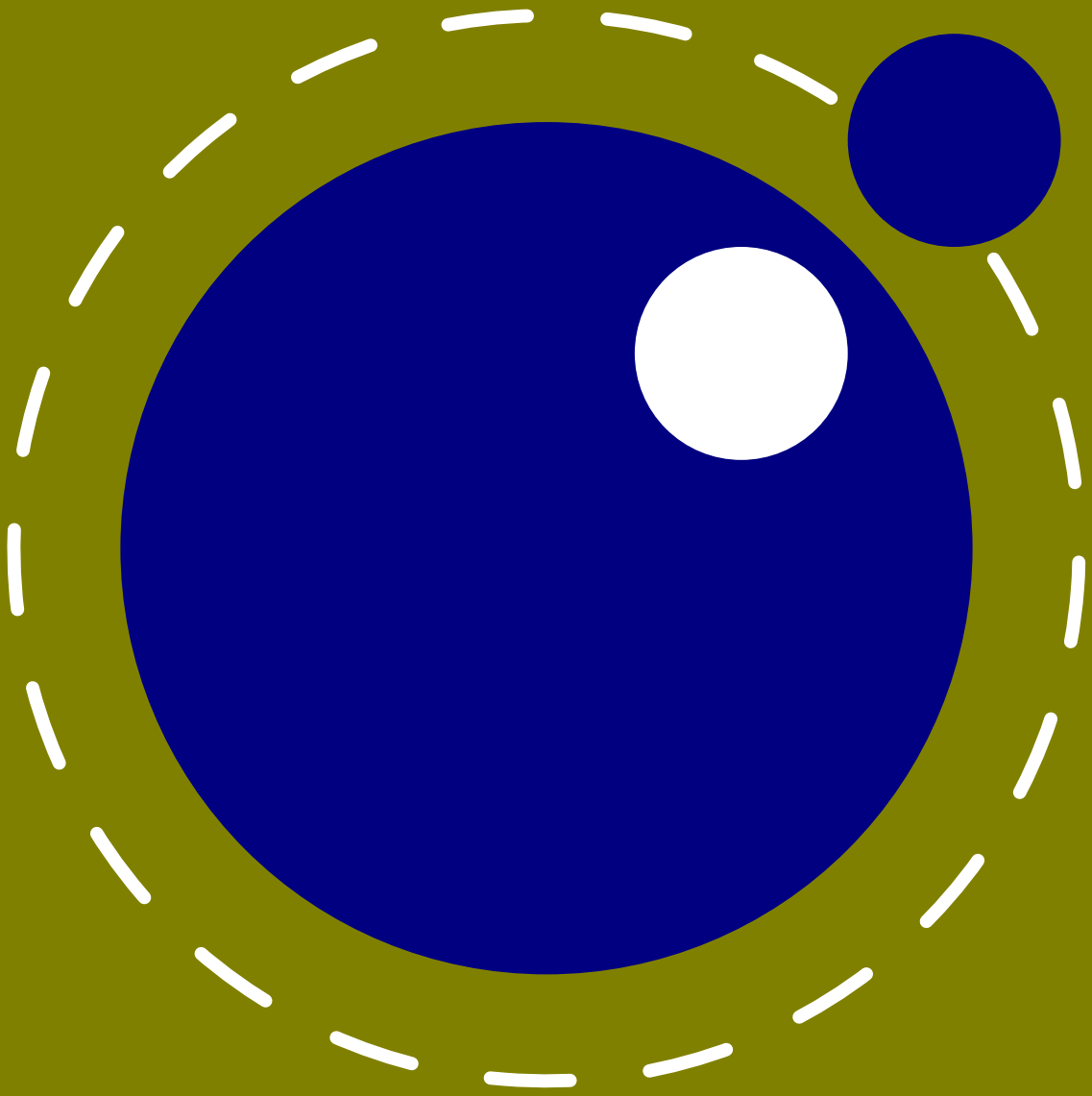
LuaTEX Reference Manual



experimental
June 2018
Version 1.09

LuaTEX Reference Manual

more info : www.luatex.org
version : June 24, 2018

Contents

Introduction

This is the reference manual of LuaTEX. We don't claim it is complete and we assume that the reader knows about TEX as described in "The TEX Book", the "ε-TEX manual", the "pdfTEX manual", etc. Additional reference material is published in journals of user groups and ConTEXt related documentation.

It took about a decade to reach stable version 1.0, but for good reason. Successive versions brought new functionality, more control, some cleanup of internals. Experimental features evolved into stable ones or were dropped. Already quite early LuaTEX could be used for production and it was used on a daily basis by the authors. Successive versions sometimes demanded an adaption to the Lua interfacing, but the concepts were unchanged. The current version can be considered stable in functionality and there will be no fundamental changes. Of course we then can decide to move towards version 2.00 with different properties.

Don't expect LuaTEX to behave the same as pdfTEX! Although the core functionality of that 8 bit engine was starting point, it has been combined with the directional support of Omega (Aleph). But, LuaTEX can behave different due to its wide (32 bit) characters, many registers and large memory support. The pdf code produced differs from pdfTEX but users will normally not notice that. There is native utf input, support for large (more than 8 bit) fonts, and the math machinery is tuned for OpenType math. There is support for directional typesetting too. The log output can differ from other engines and will likely differ more as we move forward. When you run plain TEX for sure LuaTEX runs slower than pdfTEX but when you run for instance ConTEXt MkIV in many cases it runs faster, especially when you have a bit more complex documents or input. Anyway, 32 bit all–over combined with more features has a price, but on a modern machine this is no real problem.

Testing is done with ConTEXt, but LuaTEX should work fine with other macro packages too. For that purpose we provide generic font handlers that are mostly the same as used in ConTEXt. Discussing specific implementations is beyond this manual. Even when we keep LuaTEX lean and mean, we already have enough to discuss here.

LuaTEX consists of a number of interrelated but (still) distinguishable parts. The organization of the source code is adapted so that it can glue all these components together. We continue cleaning up side effects of the accumulated code in TEX engines (especially code that is not needed any longer).

- We started out with most of pdfTEX version 1.40.9. The code base was converted to C and split in modules. Experimental features were removed and utility macros are not inherited because their functionality can be programmed in Lua. The number of backend interface commands has been reduced to a few. The so called extensions are separated from the core (which we try to keep close to the original TEX core). Some mechanisms like expansion and protrusion can behave different from the original due to some cleanup and optimization. Some whatsit based functionality (image support and reusable content) is now core functionality. We don't stay in sync with pdfTEX development.
- The direction model from Aleph RC4 (which is derived from Omega) is included. The related primitives are part of core LuaTEX but at the node level directional support is no longer based

on so called whatsits but on real nodes with relevant properties. The number of directions is limited to the useful set and the backend has been made direction aware.

- Neither Aleph's I/O translation processes, nor tcx files, nor encTEX are available. These encoding-related functions are superseded by a Lua-based solution (reader callbacks). In a similar fashion all file io can be intercepted.
- We currently use Lua 5.3.*. There are few Lua libraries that we consider part of the core Lua machinery, for instance `lpeg`. There are additional Lua libraries that interface to the internals of TEX. We also keep the Lua 5.2 `bit32` library around.
- There are various TEX extensions but only those that cannot be done using the Lua interfaces. The math machinery often has two code paths: one traditional and the other more suitable for wide OpenType fonts. Here we follow the Microsoft specifications as much as possible. Some math functionality has been opened up a bit so that users have more control.
- The fontloader uses parts of FontForge 2008.11.17 combined with additional code specific for usage in a TEX engine. We try to minimize specific font support to what TEX needs: character references and dimensions and delegate everything else to Lua. That way we keep TEX open for extensions without touching the core. In order to minimize dependencies at some point we may decide to make this an optional library.
- The MetaPost library is integral part of LuaTEX. This gives TEX some graphical capabilities using a relative high speed graphical subsystem. Again Lua is used as glue between the frontend and backend. Further development of MetaPost is closely related to LuaTEX.
- The virtual font technology that comes with TEX has been integrated into the font machinery in a way that permits creating virtual fonts at runtime. Because LuaTEX can also act as a Lua interpreter this means that a complete TEX workflow can be built without the need for additional programs.

We try to keep upcoming versions compatible but intermediate releases can contain experimental features. A general rule is that versions that end up on TEXLive and/or are released around ConTEXt meetings are stable. Future versions will probably become a bit leaner and meaner. Some libraries might become external as we don't want to bloat the binary and also don't want to add more hard coded solutions. After all, with Lua you can extend the core functionality. The less dependencies, the better.

You might find Lua helpers that are not yet documented. These are considered experimental, although when you encounter them in a ConTEXt version that has been around for a while you can assume that they will stay. Of course it can just be that we forgot to document them yet.

The TEXLive version is to be considered the current stable version. Any version between the yearly TEXLive releases are to be considered beta and in the repository end up as trunk releases. We have an experimental branch that we use for development but there is no support for any of its experimental features. Intermediate releases (from trunk) are normally available via the ConTEXt distribution channels (the garden and so called minimals).

Hans Hagen
Harmut Henkel
Taco Hoekwater
Luigi Scarso

Version : June 24, 2018
LuaTEX : luatex 1.09 / 6825
ConTEXt : MkIV 2018.06.21 12:08

1 Preamble

This is a reference manual, not a tutorial. This means that we discuss changes relative to traditonal TEX and also present new functionality. As a consequence we will refer to concepts that we assume to be known or that might be explained later.

The average user doesn't need to know much about what is in this manual. For instance fonts and languages are normally dealt with in the macro package that you use. Messing around with node lists is also often not really needed at the user level. If you do mess around, you'd better know what you're dealing with. Reading "The TEX Book" by Donald Knuth is a good investment of time then also because it's good to know where it all started. A more summarizing overview is given by "TEX by Topic" by Victor Eijkhout. You might want to peek in "The ε-TEX manual" and documentation about pdfTEX.

But ... if you're here because of Lua, then all you need to know is that you can call it from within a run. The macro package that you use probably will provide a few wrapper mechanisms but the basic `\directlua` command that does the job is:

```
\directlua{tex.print("Hi there")}
```

You can put code between curly braces but if it's a lot you can also put it in a file and load that file with the usual Lua commands.

If you still decide to read on, then it's good to know what nodes are, so we do a quick introduction here. If you input this text:

```
Hi There
```

eventually we will get a linked lists of nodes, which in ascii art looks like:

```
H <=> i <=> [glue] <=> T <=> h <=> e <=> r <=> e
```

When we have a paragraph, we actually get something:

```
[localpar] <=> H <=> i <=> [glue] <=> T <=> h <=> e <=> r <=> e <=> [glue]
```

Each character becomes a so called glyph node, a record with properties like the current font, the character code and the current language. Spaces become glue nodes. There are many node types that we will discuss later. Each node points back to a previous node or next node, given that these exist.

It's also good to know beforehand that TEX is basically centered around creating paragraphs and pages. The par builder takes a list and breaks it into lines. We turn horizontal material into vertical. Lines are so called boxes and can be separated by glue, penalties and more. The page builder accumulates lines and when feasible triggers an output routine that will take the list so far. Constructing the actual page is not part of TEX but done using primitives that permit manipulation of boxes. The result is handled back to TEX and flushed to a (often pdf) file.

The LuaTEX engine provides hooks for Lua code at nearly every reasonable point in the process: collecting content, hyphenating, applying font features, breaking into lines, etc. This means

that you can overload TEX's natural behaviour, which still is the benchmark. When we refer to 'callbacks' we means these hooks.

Where plain TEX is basically a basic framework for writing a specific style, macro packages like ConTEXt and LATEX provide the user a whole lot of additional tools to make documents look good. They hide the dirty details of font management, language demands, turning structure into typeset results, wrapping pages, including images, and so on. You should be aware of the fact that when you hook in your own code to manipulate lists, this can interfere with the macro package that you use.

When you read about nodes in the following chapters it's good to keep in mind their commands that relate to then. Here are a few:

COMMAND	NODE	EXPLANATION
\hbox	hlist	horizontal box
\vbox	vlist	vertical box with the baseline at the bottom
\vtop	vlist	vertical box with the baseline at the top
\hskip	glue	horizontal skip with optional stretch and shrink
\vskip	glue	vertical skip with optional stretch and shrink
\kern	kern	horizontal or vertical fixed skip
\discretionary	disc	hyphenation point (pre, post, replace)
\char	glyph	a character
\hrule	rule	a horizontal rule
\vrule	rule	a vertical rule
\textdir	dir	a change in text direction

For now this should be enough to enable you to understand the next chapters.

2 Basic TEX enhancements

2.1 Introduction

From day one, LuaTEX has offered extra features compared to the superset of pdfTEX and Aleph. This has not been limited to the possibility to execute Lua code via `\directlua`, but LuaTEX also adds functionality via new TEX-side primitives or extensions to existing ones.

When LuaTEX starts up in 'iniluatex' mode (`luatex -ini`), it defines only the primitive commands known by TEX82 and the one extra command `\directlua`. As is fitting, a Lua function has to be called to add the extra primitives to the user environment. The simplest method to get access to all of the new primitive commands is by adding this line to the format generation file:

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

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

```
\catcode `\{=1
\catcode `\}=2
```

More fine-grained primitives control is possible and you can look up the details in section 10.3.12. For simplicity's sake, this manual assumes that you have executed the `\directlua` command as given above.

The startup behaviour documented above is considered stable in the sense that there will not be backward-incompatible changes any more. We have promoted some rather generic pdfTEX primitives to core LuaTEX ones, and the ones inherited from Aleph (Omega) are also promoted. Effectively this means that we now only have the `tex`, `etex` and `luatex` sets left.

In Chapter 3 we discuss several primitives that are derived from pdfTEX and Aleph (Omega). Here we stick to real new ones. In the chapters on fonts and math we discuss a few more new ones.

2.2 Version information

2.2.1 `\luatexbanner`, `\luatexversion` and `\luatexrevision`

There are three new primitives to test the version of LuaTEX:

PRIMITIVE	VALUE	EXPLANATION
`\luatexbanner`	This is LuaTeX, Version 1.09.0	the banner reported on the command line
`\luatexversion`	109	a combination of major and minor number
`\luatexrevision`	0	the revision number, the current value is

The official LuaTEX version is defined as follows:

- The major version is the integer result of `\luatexversion` divided by 100. The primitive is an 'internal variable', so you may need to prefix its use with `\the` depending on the context.
- The minor version is the two-digit result of `\luatexversion` modulo 100.
- The revision is reported by `\luatexrevision`. This primitive expands to a positive integer.
- The full version number consists of the major version, minor version and revision, separated by dots.

2.2.2 `\formatname`

The `\formatname` syntax is identical to `\jobname`. In iniTEX, the expansion is empty. Otherwise, the expansion is the value that `\jobname` had during the iniTEX run that dumped the currently loaded format. You can use this token list to provide your own version info.

2.3 UNICODE text support

2.3.1 Extended ranges

Text input and output is now considered to be Unicode text, so input characters can use the full range of Unicode ($2^{20} + 2^{16} - 1 = 0$x10FFFF). Later chapters will talk of characters and glyphs. Although these are not interchangeable, they are closely related. During typesetting, a character is always converted to a suitable graphic representation of that character in a specific font. However, while processing a list of to-be-typeset nodes, its contents may still be seen as a character. Inside LuaTEX there is no clear separation between the two concepts. Because the subtype of a glyph node can be changed in Lua it is up to the user. Subtypes larger than 255 indicate that font processing has happened.

A few primitives are affected by this, all in a similar fashion: each of them has to accommodate for a larger range of acceptable numbers. For instance, `\char` now accepts values between 0 and 1,114,111. This should not be a problem for well-behaved input files, but it could create incompatibilities for input that would have generated an error when processed by older TEX-based engines. The affected commands with an altered initial (left of the equal sign) or secondary (right of the equal sign) value are: `\char`, `\lccode`, `\uccode`, `\hjcode`, `\catcode`, `\sfcode`, `\efcode`, `\lpcode`, `\rpcode`, `\chardef`.

As far as the core engine is concerned, all input and output to text files is utf-8 encoded. Input files can be pre-processed using the `reader` callback. This will be explained in section 9.2.13. Normalization of the Unicode input is on purpose not built-in and can be handled by a macro package during callback processing.

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

Output to the terminal uses ^^ 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. You can disable escaping with `texio.setescape(false)` in which case you get the normal characters on the console.

2.3.2 \Uchar

The expandable command \Uchar reads a number between 0 and 1,114,111 and expands to the associated Unicode character.

2.4 Extended tables

All traditional TEX and ε-TEX registers can be 16-bit numbers. The affected commands are:

\count	\countdef	\box	\wd
\dimen	\dimendef	\unhbox	\ht
\skip	\skipdef	\unvbox	\dp
\muskip	\muskipdef	\copy	\setbox
\marks	\toksdef	\unhcopy	\vsplit
\toks	\insert	\unvcopy	

Because font memory management has been rewritten, character properties in fonts are no longer shared among font instances that originate from the same metric file.

2.5 Attributes

2.5.1 Attribute registers

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

\attribute ⟨16-bit number⟩ ⟨optional equals⟩ ⟨32-bit number⟩
\attributedef ⟨csname⟩ ⟨optional equals⟩ ⟨16-bit number⟩

Conceptually, an attribute is either 'set' or 'unset'. Unset attributes have a special negative value to indicate that they are unset, that value is the lowest legal value: -"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 iniTEX.

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.

Attributes are stored in a sorted (sparse) linked list that are shared when possible. This permits efficient testing and updating.

2.5.2 Nodes

When TEX reads input it will interpret the stream according to the properties of the characters. Some signal a macro name and trigger expansion, others open and close groups, trigger math

mode, etc. What's left over becomes the typeset text. Internally we get linked list of nodes. Characters become `glyph` nodes that have for instance a `font` and `char` property and `\kern 10pt` becomes a `kern` node with a `width` property. Spaces are alien to TEX as they are turned into `glue` nodes. So, a simple paragraph is mostly a mix of sequences of `glyph` nodes (words) and `glue` nodes (spaces).

The sequences of characters at some point are extended with `disc` nodes that relate to hyphenation. After that font logic can be applied and we get a list where some characters can be replaced, for instance multiple characters can become one ligature, and font kerns can be injected. This is driven by the font properties.

Boxes (like `\hbox` and `\vbox`) become `hlist` or `vlist` nodes with `width`, `height`, `depth` and `shift` properties and a pointer `list` to its actual content. Boxes can be constructed explicitly or can be the result of subprocesses. For instance, when lines are broken into paragraphs, the lines are a linked list of `hlist` nodes.

We will see more of these nodes later on but for now that should be enough to be able to follow the rest of this chapter.

2.5.3 Box attributes

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

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

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

It is possible to fine-tune the list of attributes that are applied to a `hbox`, `vbox` or `vtop` by the use of the keyword `attr`. The `attr` keyword(s) should come before a `to` or `spread`, if that is also specified. An example is:

```
\attribute997=123
\attribute998=456
\setbox0=\hbox {Hello}
\setbox2=\hbox attr 999 = 789 attr 998 = -"7FFFFFFF{Hello}
```

Box 0 now has attributes 997 and 998 set while box 2 has attributes 997 and 999 set while the nodes inside that box will all have attributes 997 and 998 set. Assigning the maximum negative value causes an attribute to be ignored.

To give you an idea of what this means at the Lua end, take the following code:

```
for b=0,2,2 do
  for a=997, 999 do
    tex.sprint("box ", b, " : attr ",a," : ",tostring(tex.box[b]     [a]))
    tex.sprint("\\quad\\quad")
    tex.sprint("list ",b, " : attr ",a," : ",tostring(tex.box[b].list[a]))
    tex.sprint("\\par")
  end
end
```

Later we will see that you can access properties of a node. The boxes here are so called `hlist` nodes that have a field `list` that points to the content. Because the attributes are a list themselves you can access them by indexing the node (here we do that with `[a]`. Running this snippet gives:

```
box 0 : attr 997 : 123   list 0 : attr 997 : 123
box 0 : attr 998 : 456   list 0 : attr 998 : 456
box 0 : attr 999 : nil   list 0 : attr 999 : nil
box 2 : attr 997 : 123   list 2 : attr 997 : 123
box 2 : attr 998 : nil   list 2 : attr 998 : 456
box 2 : attr 999 : 789   list 2 : attr 999 : nil
```

Because some values are not set we need to apply the `tostring` function here so that we get the word `nil`.

2.6 LUA related primitives

2.6.1 \directlua

In order to merge Lua code with TEX input, a few new primitives are needed. The primitive `\directlua` is used to execute Lua code immediately. The syntax is

`\directlua` ⟨general text⟩
`\directlua` ⟨16-bit number⟩ ⟨general text⟩

The ⟨general text⟩ is expanded fully, and then fed into the Lua interpreter. After reading and expansion has been applied to the ⟨general text⟩, 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. Of course such an approach depends on the macro package that you use.

The ⟨16-bit number⟩ designates a name of a Lua chunk and is taken from the `lua.name` array (see the documentation of the `lua` table further in this manual). When a chunk name starts with a `@` it will be displayed as a file name. This is a side effect of the way Lua implements error handling.

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 `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:

```
\count10=20
a\directlua{tex.print(tex.count[10]+5)}b
```

expands to

a25b

Here is another example:

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

will result in

$\pi = 3.1415926535898$

Note that the expansion of `\directlua` is a sequence of characters, not of tokens, contrary to all TEX commands. So formally speaking its expansion is null, but it places material on a pseudo-file to be immediately read by TEX, as ε-TEX's `\scantokens`. For a description of print functions look at section 10.3.10.

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 LuaTEX 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.

2.6.2 `\latelua`

Contrary to `\directlua`, `\latelua` stores Lua code in a whatsit that will be processed at the time of shipping out. Its intended use is a cross between pdf literals (often available as `\pdfliteral`) and the traditional TEX extension `\write`. Within the Lua code you can print pdf statements

directly to the pdf file via `pdf.print`, or you can write to other output streams via `texio.write` or simply using Lua io routines.

`\latelua` ⟨general text⟩
`\latelua` ⟨16-bit number⟩ ⟨general text⟩

Expansion of macros in the final `<general text>` is delayed until just before the whatsit is executed (like in `\write`). With regard to pdf output stream `\latelua` behaves as pdf page literals. The `name` ⟨general text⟩ and ⟨16-bit number⟩ behave in the same way as they do for `\directlua`

2.6.3 \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` ⟨general text⟩

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

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

2.6.4 \luafunction, \luafunctioncall and \luadef

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 have millions of calls it can have some impact. For this reason there is a variant call available: `\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 the following example the number 8 gets typeset.

```
\directlua {
```


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

The \luafunctioncall primitive does the same but is unexpandable, for instance in an \edef. In addition LuaTEX provides a definer:

```
                \luadef\MyFunctionA 1
          \global\luadef\MyFunctionB 2
\protected\global\luadef\MyFunctionC 3
```

2.6.5 \luabytecode and \luabytecodecall

Analogue to the function callers discussed in the previous section we have byte code callers. Again the call variant is unexpandable.

```
\directlua {
    lua.bytecode[9998] = function(s)
        tex.sprint(s*token.scan_int())
    end
    lua.bytecode[5555] = function(s)
        tex.sprint(s*token.scan_dimen())
    end
}
```

This works with:

```
\luabytecode    9998 5  \luabytecode    5555 5sp
\luabytecodecall9998 5  \luabytecodecall5555 5sp
```

The variable s in the code is the number of the byte code register that can be used for diagnostic purposes. The advantage of bytecode registers over function calls is that they are stored in the format (but without upvalues).

2.7 Alignments

2.7.1 \alignmark and \aligntab

The primitive \alignmark duplicates the functionality of # inside alignment preambles, while \aligntab duplicates the functionality of &.

2.8 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. This subsystem is backward compatible: if you never use the following commands, your document will

not notice any difference in behaviour compared to traditional TEX. The contents of each catcode table is independent from any other catcode table, and its contents is stored and retrieved from the format file.

2.8.1 \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.

2.8.2 \initcatcodetable

\initcatcodetable ⟨15-bit number⟩

The primitive \initcatcodetable creates a new table with catcodes identical to those defined by iniTEX. 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. The initial values are:

CATCODE	CHARACTER	EQUIVALENT	CATEGORY
0	\		escape
5	^^M	return	car_ret
9	^^@	null	ignore
10	<space>	space	spacer
11	a – z		letter
11	A – Z		letter
12	everything else		other
14	%		comment
15	^^?	delete	invalid_char

2.8.3 \savecatcodetable

\savecatcodetable ⟨15-bit number⟩

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

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

2.9 Suppressing errors

2.9.1 \suppressfontnotfounderror

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

```
\suppressfontnotfounderror = 1
```

2.9.2 \suppresslongerror

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

```
\suppresslongerror = 1
```

2.9.3 \suppressifcsnameerror

If this integer parameter is non-zero, then LuaTEX will not complain about non-expandable commands appearing in the middle of a \ifcsname expansion. Instead, it will keep getting expanded tokens from the input until it encounters an \endcsname command. If the input expansion is unbalanced with respect to \csname ...\endcsname pairs, the LuaTEX process may hang indefinitely.

```
\suppressifcsnameerror = 1
```

2.9.4 \suppressoutererror

If this new integer parameter is non-zero, then LuaTEX will not complain about \outer commands encountered in contexts where that is normally prohibited.

```
\suppressoutererror = 1
```

2.9.5 \suppressmathparerror

The following setting will permit \par tokens in a math formula:

```
\suppressmathparerror = 1
```

So, the next code is valid then:

```
$ x + 1 =

a $
```

2.9.6 \suppressprimitiveerror

When set to a non-zero value the following command will not issue an error:

```
\suppressprimitiveerror = 1
\primitive\notaprimitive
```

2.10 Math

We will cover math extensions in its own chapter because not only the font subsystem and spacing model have been enhanced (thereby introducing many new primitives) but also because

some more control has been added to existing functionality. Much of this relates to the different approaches of traditional TEX fonts and OpenType math.

2.11 Fonts

2.11.1 Font syntax

LuaTEX will accept a braced argument as a font name:

```
\font\myfont = {cmr10}
```

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

2.11.2 \fontid and \setfontid

```
\fontid\font
```

This primitive expands into a number. It is not a register so there is no need to prefix with `\number` (and using `\the` gives an error). The currently used font id is 29. Here are some more:

STYLE	COMMAND	FONT ID
normal	`\tf`	**38**
bold	`\bf`	**38**
italic	`\it`	*49*
bold italic	`\bi`	***50***

These numbers depend on the macro package used because each one has its own way of dealing with fonts. They can also differ per run, as they can depend on the order of loading fonts. For instance, when in ConTEXt virtual math Unicode fonts are used, we can easily get over a hundred ids in use. Not all ids have to be bound to a real font, after all it's just a number.

The primitive `\setfontid` can be used to enable a font with the given id, which of course needs to be a valid one.

2.11.3 \noligs and \nokerns

These primitives prohibit ligature and kerning insertion at the time when the initial node list is built by LuaTEX's main control loop. You can enable these primitives when you want to do node list processing of 'characters', where TEX's normal processing would get in the way.

```
\noligs ⟨integer⟩
\nokerns ⟨integer⟩
```

These primitives can also be implemented by overloading the ligature building and kerning functions, i.e. by assigning dummy functions to their associated callbacks. Keep in mind that when you define a font (using Lua) you can also omit the kern and ligature tables, which has the same effect as the above.

2.11.4 \nospaces

This new primitive can be used to overrule the usual \spaceskip related heuristics when a space character is seen in a text flow. The value 1 triggers no injection while 2 results in injection of a zero skip. In figure 2.1 we see the results for four characters separated by a space.

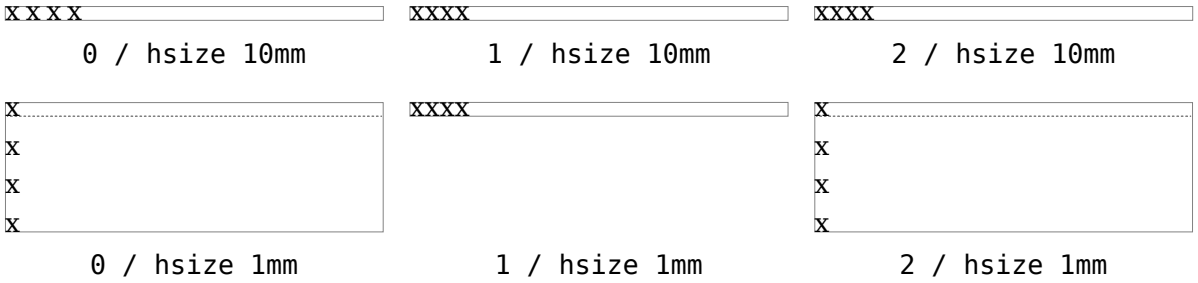


Figure 2.1 The \nospaces options.

2.12 Tokens, commands and strings

2.12.1 \scantextokens

The syntax of \scantextokens is identical to \scantokens. This primitive is a slightly adapted version of ε-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.
- There are no '... while end of file ...' error tests executed. This allows the expansion to end on a different grouping level or while a conditional is still incomplete.

2.12.2 \toksapp, \tokspre, \etoksapp and \etokspre

Instead of:

```
\toks0\expandafter{\the\toks0 foo}
```

you can use:

```
\etoksapp0{foo}
```

The pre variants prepend instead of append, and the e variants expand the passed general text.

2.12.3 \csstring, \begincsname and \lastnamedcs

These are somewhat special. The \csstring primitive is like \string but it omits the leading escape character. This can be somewhat more efficient than stripping it afterwards.

The \begincsname primitive is like \csname but doesn't create a relaxed equivalent when there is no such name. It is equivalent to

```
\ifcsname foo\endcsname
  \csname foo\endcsname
\fi
```

The advantage is that it saves a lookup (don't expect much speedup) but more important is that it avoids using the `\if` test. The `\lastnamedcs` is one that should be used with care. The above example could be written as:

```
\ifcsname foo\endcsname
  \lastnamedcs
\fi
```

This is slightly more efficient than constructing the string twice (deep down in LuaTEX this also involves some utf8 juggling), but probably more relevant is that it saves a few tokens and can make code a bit more readable.

2.12.4 `\clearmarks`

This primitive complements the ε-TEX mark primitives and clears a mark class completely, resetting all three connected mark texts to empty. It is an immediate command.

`\clearmarks` ⟨16-bit number⟩

2.12.5 `\letcharcode`

This primitive can be used to assign a meaning to an active character, as in:

```
\def\foo{bar} \letcharcode123=\foo
```

This can be a bit nicer than using the uppercase tricks (using the property of `\uppercase` that it treats active characters special).

2.12.6 `\glet`

This primitive is similar to:

```
\protected\def\glet{\global\let}
```

but faster (only measurable with millions of calls) and probably more convenient (after all we also have `\gdef`).

2.13 Boxes, rules and leaders

2.13.1 `\outputbox`

This 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.

```
\outputbox = 12345
```

2.13.2 \vpack, \hpack and \tpack

These three primitives are like \vbox, \hbox and \vtop but don't apply the related callbacks.

2.13.3 \vsplit

The \vsplit primitive has to be followed by a specification of the required height. As alternative for the to keyword you can use upto to get a split of the given size but result has the natural dimensions then.

2.13.4 Images and reused box objects

These two concepts are now core concepts and no longer whatsits. They are in fact now implemented as rules with special properties. Normal rules have subtype 0, saved boxes have subtype 1 and images have subtype 2. This has the positive side effect that whenever we need to take content with dimensions into account, when we look at rule nodes, we automatically also deal with these two types.

The syntax of the \save...resource is the same as in pdfTEX but you should consider them to be backend specific. This means that a macro package should treat them as such and check for the current output mode if applicable.

COMMAND	EXPLANATION
\saveboxresource	save the box as an object to be included later
\saveimageresource	save the image as an object to be included later
\useboxresource	include the saved box object here (by index)
\useimageresource	include the saved image object here (by index)
\lastsavedboxresourceindex	the index of the last saved box object
\lastsavedimageresourceindex	the index of the last saved image object
\lastsavedimageresourcepages	the number of pages in the last saved image object

LuaTEX accepts optional dimension parameters for \use...resource in the same format as for rules. With images, these dimensions are then used instead of the ones given to \useimageresource but the original dimensions are not overwritten, so that a \useimageresource without dimensions still provides the image with dimensions defined by \saveimageresource. These optional parameters are not implemented for \saveboxresource.

```
\useimageresource width 20mm height 10mm depth 5mm \lastsavedimageresourceindex
\useboxresource   width 20mm height 10mm depth 5mm \lastsavedboxresourceindex
```

The box resources are of course implemented in the backend and therefore we do support the attr and resources keys that accept a token list. New is the type key. When set to non-zero the /Type entry is omitted. A value of 1 or 3 still writes a /BBox, while 2 or 3 will write a /Matrix.

2.13.5 \nohrule and \novrule

Because introducing a new keyword can cause incompatibilities, two new primitives were introduced: \nohrule and \novrule. These can be used to reserve space. This is often more efficient than creating an empty box with fake dimensions.

2.13.6 \gleaders

This type of leaders is anchored to the origin of the box to be shipped out. So they are like normal \leaders in that they align nicely, except that the alignment is based on the *largest* enclosing box instead of the *smallest*. The g stresses this global nature.

2.14 Languages

2.14.1 \hyphenationmin

This primitive can be used to set the minimal word length, so setting it to a value of 5 means that only words of 6 characters and more will be hyphenated, of course within the constraints of the \lefthyphenmin and \righthyphenmin values (as stored in the glyph node). This primitive accepts a number and stores the value with the language.

2.14.2 \boundary, \noboundary, \protrusionboundary and \wordboundary

The \noboundary command is used to inject a whatsit node but now injects a normal node with type boundary and subtype 0. In addition you can say:

```
x\boundary 123\relax y
```

This has the same effect but the subtype is now 1 and the value 123 is stored. The traditional ligature builder still sees this as a cancel boundary directive but at the Lua end you can implement different behaviour. The added benefit of passing this value is a side effect of the generalization. The subtypes 2 and 3 are used to control protrusion and word boundaries in hyphenation and have related primitives.

2.15 Control and debugging

2.15.1 Tracing

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

2.15.2 \outputmode

The \outputmode variable tells LuaTEX what it has to produce:

VALUE	OUTPUT
0	dvi code
1	pdf code

2.15.3 \draftmode

The value of the \draftmode counter signals the backend if it should output less. The pdf backend accepts a value of 1, while the dvi backend ignores the value.

2.16 Files

2.16.1 File syntax

LuaTEX will accept a braced argument as a file name:

```
\input {plain}
\openin 0 {plain}
```

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

The \tracingfonts primitive that has been inherited from pdfTEX has been adapted to support variants in reporting the font. The reason for this extension is that a csname not always makes sense. The zero case is the default.

VALUE	REPORTED
0	\foo xyz
1	\foo (bar)
2	<bar> xyz
3	<bar @ ..pt> xyz
4	<id>
5	<id: bar>
6	<id: bar @ ..pt> xyz

2.16.2 Writing to file

You can now open upto 127 files with \openout. When no file is open writes will go to the console and log. As a consequence a system command is no longer possible but one can use os.execute to do the same.

2.16.3 \expanded, \immediateassignment, \immediateassigned

The \expanded primitive takes a token list and expands it content which can come in handy: it avoids a tricky mix of \expandafter and \noexpand. You can compare it with what happens

inside the body of an \edef. But this kind of expansion it still doesn't expand some primitive operations.

```
\newcount\NumberOfCalls

\def\TestMe{\advance\NumberOfCalls1 }

\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\edef\Tested{\TestMe foo:\the\NumberOfCalls}

\meaning\Tested
```

The result is a macro that has the not expanded code in its body:

macro:->\advance \NumberOfCalls 1 foo:0

Instead we can define \TestMe in a way that expands the assignment immediately. You need of course to be aware of preventing look ahead interference by using a space or \relax (often an expression works better as it doesn't leave an \relax).

```
\def\TestMe{\immediateassignment\advance\NumberOfCalls1 }

\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\edef\Tested{\TestMe foo:\the\NumberOfCalls}
\edef\Tested{\TestMe foo:\the\NumberOfCalls}

\meaning\Tested
```

This time the counter gets updates and we don't see interference in the resulting \Tested macro:

macro:->foo:3

Here is a somewhat silly example of expanded comparison:

```
\def\expandeddoifelse#1#2#3#4%
  {\immediateassignment\edef\tempa{#1}%
   \immediateassignment\edef\tempb{#2}%
   \ifx\tempa\tempb
     \immediateassignment\def\next{#3}%
   \else
     \immediateassignment\def\next{#4}%
   \fi
   \next}

\edef\Tested
  {(\expandeddoifelse{abc}{def}{yes}{nop}/%
    \expandeddoifelse{abc}{abc}{yes}{nop})}

\meaning\Tested
```


It gives:

macro:->(nop/yes)

A variant is:

```
\def\expandeddoifelse#1#2#3#4%
  {\immediateassigned{
     \edef\tempa{#1}%
     \edef\tempb{#2}%
   }%
   \ifx\tempa\tempb
     \immediateassignment\def\next{#3}%
   \else
     \immediateassignment\def\next{#4}%
   \fi
   \next}
```

The possible error messages are the same as using assignments in preambles of alignments and after the `\accent` command. The supported assignments are the so called prefixed commands (except box assignments).

2.16.4 \ifcondition

This is a somewhat special one. When you write macros conditions need to be properly balanced in order to let TEX's fast branch skipping work well. This new primitive is basically a no–op flagged as a condition so that the scanner can recognize it as an if-test. However, when a real test takes place the work is done by what follows, in the next example `\something`.

```
\unexpanded\def\something#1#2%
  {\edef\tempa{#1}%
   \edef\tempb{#2}
   \ifx\tempa\tempb}

\ifcondition\something{a}{b}%
    \ifcondition\something{a}{a}%
        true 1
    \else
        false 1
    \fi
\else
    \ifcondition\something{a}{a}%
        true 2
    \else
        false 2
    \fi
\fi
```

If you are familiar with MetaPost, this is a bit like `vardef` where the macro has a return value. Here the return value is a test.

4

3 Modifications

3.1 The merged engines

3.1.1 The need for change

The first version of LuaTEX only had a few extra primitives and it was largely the same as pdfTEX. Then we merged substantial parts of Aleph into the code and got more primitives. When we got more stable the decision was made to clean up the rather hybrid nature of the program. This means that some primitives have been promoted to core primitives, often with a different name, and that others were removed. This made it possible to start cleaning up the code base. In chapter 2 we discussed some new primitives, here we will cover most of the adapted ones.

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. These will also be mentioned.

3.1.2 Changes from TEX 3.1415926

Of course it all starts with traditional TEX. Even if we started with pdfTEX, most still comes from the original. But we divert a bit.

- The current code base is written in C, not Pascal. We use cweb when possible. As a consequence instead of one large file plus change files, we now have multiple files organized in categories like `tex`, `pdf`, `lang`, `font`, `lua`, etc. There are some artifacts of the conversion to C, but in due time we will clean up the source code and make sure that the documentation is done right. Many files are in the cweb format, but others, like those interfacing to Lua, are C files. Of course we want to stay as close as possible to the original so that the documentation of the fundamentals behind TEX by Don Knuth still applies.
- See chapter 5 for many small changes related to paragraph building, language handling and hyphenation. The most important change is that adding a brace group in the middle of a word (like in `of{}fice`) does not prevent ligature creation.
- There is no pool file, all strings are embedded during compilation.
- The specifier `plus 1 fillll` does not generate an error. The extra 'l' is simply typeset.
- The upper limit to `\endlinechar` and `\newlinechar` is 127.
- Magnification (`\mag`) is only supported in dvi output mode. You can set this parameter and it even works with `true` units till you switch to pdf output mode. When you use pdf output you can best not touch the `\mag` variable. This fuzzy behaviour is not much different from using pdf backend related functionality while eventually dvi output is required.
  After the output mode has been frozen (normally that happens when the first page is shipped out) or when pdf output is enabled, the `true` specification is ignored. When you preload a plain format adapted to LuaTEX it can be that the `\mag` parameter already has been set.

3.1.3 Changes from ε-TEX 2.2

Being the de factor standard extension of course we provide the ε-TEX functionality, but with a few small adaptations.

- The ε-TEX functionality is always present and enabled so the prepended asterisk or `-etex` switch for iniTEX is not needed.
- The TEXXeT extension is not present, so the primitives `\TeXXeTstate`, `\beginR`, `\beginL`, `\endR` and `\endL` are missing. Instead we used the Omega/Aleph approach to directionality as starting point.
- Some of the tracing information that is output by ε-TEX's `\tracingassigns` and `\tracingrestores` is not there.
- Register management in LuaTEX uses the Omega/Aleph model, so the maximum value is 65535 and the implementation uses a flat array instead of the mixed flat & sparse model from ε-TEX.
- When kpathsea is used to find files, LuaTEX uses the `ofm` file format to search for font metrics. In turn, this means that LuaTEX looks at the `OFMFONTS` configuration variable (like Omega and Aleph) instead of `TFMFONTS` (like TEX and pdfTEX). Likewise for virtual fonts (LuaTEX uses the variable `OVFFONTS` instead of `VFFONTS`).

3.1.4 Changes from PDFTEX 1.40

Because we want to produce pdf the most natural starting point was the popular pdfTEX program. We inherit the stable features, dropped most of the experimental code and promoted some functionality to core LuaTEX functionality which in turn triggered renaming primitives.

For compatibility reasons we still refer to `\pdf...` commands but LuaTEX has a different backend interface. Instead of these primitives there are three interfacing primitives: `\pdfextension`, `\pdfvariable` and `\pdffeedback` that take keywords and optional further arguments (below we will still use the `\pdf` prefix names as reference). This way we can extend the features when needed but don't need to adapt the core engine. The front- and backend are decoupled as much as possible.

- 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 gone are: `\pdfsnaprefpoint`, `\pdfsnapy`, and `\pdfsnapycomp`.
- The (experimental) support for specialized spacing around nodes has also been removed. The associated primitives that are gone are: `\pdfadjustinterwordglue`, `\pdfprependkern`, and `\pdfappendkern`, as well as the five supporting primitives `\knbscode`, `\stbscode`, `\shbscode`, `\knbccode`, and `\knaccode`.
- A number of 'pdfTEX primitives' have been removed as they can be implemented using Lua: `\pdfelapsedtime`, `\pdfescapehex`, `\pdfescapename`, `\pdfescapestring`, `\pdffiledump`, `\pdffilemoddate`, `\pdffilesize`, `\pdfforcepagebox`, `\pdflastmatch`, `\pdfmatch`, `\pdfmdfivesum`, `\pdfmovechars`, `\pdfoptionalwaysusepdfpagebox`, `\pdfoptionpdfinclusionerrorlevel`, `\pdfresettimer`, `\pdfshellescape`, `\pdfstrcmp` and `\pdfunescapehex`.
- The version related primitives `\pdftexbanner`, `\pdftexversion` and `\pdftexrevision` are no longer present as there is no longer a relationship with pdfTEX development.

- The experimental snapper mechanism has been removed and therefore also the primitives `\pdfignoreddimen`, `\pdffirstlineheight`, `\pdfeachlineheight`, `\pdfeachlinedepth` and `\pdflastlinedepth`.
- The experimental primitives `\primitive`, `\ifprimitive`, `\ifabsnum` and `\ifabsdim` are promoted to core primitives. The `\pdf*` prefixed originals are not available.
- Because LuaTEX has a different subsystem for managing images, more diversion from its ancestor happened in the meantime. We don't adapt to changes in pdfTEX.
- Two extra token lists are provided, `\pdfxformresources` and `\pdfxformattr`, as an alternative to `\pdfxform` keywords.
- Image specifications also support `visiblefilename`, `userpassword` and `ownerpassword`. The password options are only relevant for encrypted pdf files.
- The current version of LuaTEX no longer replaces and/or merges fonts in embedded pdf files with fonts of the enveloping pdf document. This regression may be temporary, depending on how the rewritten font backend will look like.
- The primitives `\pdfpagewidth` and `\pdfpageheight` have been removed because `\pagewidth` and `\pageheight` have that purpose.
- The primitives `\pdfnormaldeviate`, `\pdfuniformdeviate`, `\pdfsetrandomseed` and `\pdfrandomseed` have been promoted to core primitives without `pdf` prefix so the original commands are no longer recognized.
- The primitives `\ifincsname`, `\expanded` and `\quitvmode` are now core primitives.
- As the hz and protrusion mechanism are part of the core the related primitives `\lpcode`, `\rpcode`, `\efcode`, `\leftmarginkern`, `\rightmarginkern` are promoted to core primitives. The two commands `\protrudechars` and `\adjustspacing` replace their prefixed with `\pdf` originals.
- The hz optimization code has been partially redone so that we no longer need to create extra font instances. The front- and backend have been decoupled and more efficient (pdf) code is generated.
- When `\adjustspacing` has value 2, hz optimization will be applied to glyphs and kerns. When the value is 3, only glyphs will be treated. A value smaller than 2 disables this feature.
- The `\tagcode` primitive is promoted to core primitive.
- The `\letterspacefont` feature is now part of the core but will not be changed (improved). We just provide it for legacy use.
- The `\pdfnoligatures` primitive is now `\ignoreligaturesinfont`.
- The `\pdfcopyfont` primitive is now `\copyfont`.
- The `\pdffontexpand` primitive is now `\expandglyphsinfont`.
- Because position tracking is also available in dvi mode the `\savepos`, `\lastxpos` and `\lastypos` commands now replace their `pdf` prefixed originals.
- The introspective primitives `\pdflastximagecolordepth` and `\pdfximagebbox` have been removed. One can use external applications to determine these properties or use the built-in `img` library.
- The initializers `\pdfoutput` has been replaced by `\outputmode` and `\pdfdraftmode` is now `\draftmode`.
- The pixel multiplier dimension `\pdfpxdimen` lost its prefix and is now called `\pxdimen`.
- An extra `\pdfimageaddfilename` option has been added that can be used to block writing the filename to the pdf file.
- The primitive `\pdftracingfonts` is now `\tracingfonts` as it doesn't relate to the backend.

- The experimental primitive `\pdfinsertht` is kept as `\insertht`.
- There is some more control over what metadata goes into the pdf file.
- The promotion of primitives to core primitives as well as the separation of font- and backend means that the initialization namespace `pdftex` is gone.

One change involves the so called xforms and ximages. In pdfTEX these are implemented as so called whatsits. But contrary to other whatsits they have dimensions that need to be taken into account when for instance calculating optimal line breaks. In LuaTEX these are now promoted to a special type of rule nodes, which simplifies code that needs those dimensions.

Another reason for promotion is that these are useful concepts. Backends can provide the ability to use content that has been rendered in several places, and images are also common. As already mentioned in section 2.13.4, we now have:

LUATEX	PDFTEX
`\saveboxresource`	`\pdfxform`
`\saveimageresource`	`\pdfximage`
`\useboxresource`	`\pdfrefxform`
`\useimageresource`	`\pdfrefximage`
`\lastsavedboxresourceindex`	`\pdflastxform`
`\lastsavedimageresourceindex`	`\pdflastximage`
`\lastsavedimageresourcepages`	`\pdflastximagepages`

There are a few `\pdffeedback` features that relate to this but these are typical backend specific ones. The index that gets returned is to be considered as 'just a number' and although it still has the same meaning (object related) as before, you should not depend on that.

The protrusion detection mechanism is enhanced a bit to enable a bit more complex situations. When protrusion characters are identified some nodes are skipped:

- zero glue
- penalties
- empty discretionaries
- normal zero kerns
- rules with zero dimensions
- math nodes with a surround of zero
- dir nodes
- empty horizontal lists
- local par nodes
- inserts, marks and adjusts
- boundaries
- whatsits

Because this can not be enough, you can also use a protrusion boundary node to make the next node being ignored. When the value is 1 or 3, the next node will be ignored in the test when locating a left boundary condition. When the value is 2 or 3, the previous node will be ignored when locating a right boundary condition (the search goes from right to left). This permits protrusion combined with for instance content moved into the margin:

```
\protrusionboundary1\llap{!\quad}«Who needs protrusion?»
```

3.1.5 Changes from ALEPH RC4

Because we wanted proper directional typesetting the Aleph mechanisms looked most attractive. These are rather close to the ones provided by Omega, so what we say next applies to both these

programs.

- The extended 16-bit math primitives (`\omathcode` etc.) have been removed.
- The OCP processing has been removed completely and as a consequence, the following primitives have been removed: `\ocp`, `\externalocp`, `\ocplist`, `\pushocplist`, `\popocplist`, `\clearocplists`, `\addbeforeocplist`, `\addafterocplist`, `\removebeforeocplist`, `\removeafterocplist` and `\ocptracelevel`.
- LuaTEX 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. In addition to a keyword driven model we also provide an integer driven one.
- The input translations from Aleph are not implemented, the related primitives are not available: `\DefaultInputMode`, `\noDefaultInputMode`, `\noInputMode`, `\InputMode`, `\DefaultOutputMode`, `\noDefaultOutputMode`, `\noOutputMode`, `\OutputMode`, `\DefaultInputTranslation`, `\noDefaultInputTranslation`, `\noInputTranslation`, `\InputTranslation`, `\DefaultOutputTranslation`, `\noDefaultOutputTranslation`, `\noOutputTranslation` and `\OutputTranslation`.
- Several bugs have been fixed and confusing implementation details have been sorted out.
- The scanner for direction specifications now allows an optional space after the direction is completely parsed.
- The ^^ notation has been extended: after ^^^^ four hexadecimal characters are expected and after ^^^^^^ six hexadecimal characters have to be given. The original TEX interpretation is still valid for the ^^ case but the four and six variants do no backtracking, i.e. when they are not followed by the right number of hexadecimal digits they issue an error message. Because ^^^ is a normal TEX case, we don't support the odd number of ^^^^^ either.
- Glues *immediately after* direction change commands are not legal breakpoints.
- Several mechanisms that need to be right-to-left aware have been improved. For instance placement of formula numbers.
- The page dimension related primitives `\pagewidth` and `\pageheight` have been promoted to core primitives. The `\hoffset` and `\voffset` primitives have been fixed.
- The primitives `\charwd`, `\charht`, `\chardp` and `\charit` have been removed as we have the ε-TEX variants `\fontchar*`.
- The two dimension registers `\pagerightoffset` and `\pagebottomoffset` are now core primitives.
- The direction related primitives `\pagedir`, `\bodydir`, `\pardir`, `\textdir`, `\mathdir` and `\boxdir` are now core primitives.
- The promotion of primitives to core primitives as well as removing of all others means that the initialization namespace `aleph` that early versions of LuaTEX provided is gone.

The above let's itself summarize as: we took the 32 bit aspects and much of the directional mechanisms and merged it into the pdfTEX code base as starting point for further development. Then we simplified directionality, fixed it and opened it up.

3.1.6 Changes from standard WEB2C

The compilation framework is web2c and we keep using that but without the Pascal to C step. This framework also provides some common features that deal with reading bytes from files and locating files in tds. This is what we do different:

- There is no mltex support.
- There is no enctex support.
- The following encoding related command line switches are silently ignored, even in non-Lua mode: `-8bit`, `-translate-file`, `-mltex`, `-enc` and `-etex`.
- The `\openout` whatsits are not written to the log file.
- Some of the so-called web2c extensions are hard to set up in non-kpse mode because `texmf.cnf` is not read: `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` and alike.
- The 'E' option does not do anything useful.

3.2 The backend primitives \pdf *

In a previous section we mentioned that some pdfTEX primitives were removed and others promoted to core LuaTEX primitives. That is only part of the story. In order to separate the backend specific primitives in de code these commands are now replaced by only a few. In traditional TEX we only had the dvi backend but now we have two: dvi and pdf. Additional functionality is implemented as 'extensions' in TEX speak. By separating more strickly we are able to keep the core (frontend) clean and stable and isolate these extensions. If for some reason an extra backend option is needed, it can be implemented without touching the core. The three pdf backend related primitives are:

```
\pdfextension command [specification]
\pdfvariable  name
\pdffeedback  name
```

An extension triggers further parsing, depending on the command given. A variable is a (kind of) register and can be read and written, while a feedback is reporting something (as it comes from the backend it's normally a sequence of tokens).

In order for LuaTEX to be more than just TEX you need to enable primitives. That has already been the case right from the start. If you want the traditional pdfTEX primitives (for as far their functionality is still around) you now can do this:

```
\protected\def\pdfliteral            {\pdfextension literal}
\protected\def\pdfcolorstack         {\pdfextension colorstack}
\protected\def\pdfsetmatrix          {\pdfextension setmatrix}
\protected\def\pdfsave               {\pdfextension save\relax}
\protected\def\pdfrestore            {\pdfextension restore\relax}
\protected\def\pdfobj                {\pdfextension obj }
\protected\def\pdfrefobj             {\pdfextension refobj }
\protected\def\pdfannot              {\pdfextension annot }
\protected\def\pdfstartlink          {\pdfextension startlink }
\protected\def\pdfendlink            {\pdfextension endlink\relax}
\protected\def\pdfoutline            {\pdfextension outline }
\protected\def\pdfdest               {\pdfextension dest }
\protected\def\pdfthread             {\pdfextension thread }
```

```
\protected\def\pdfstartthread        {\pdfextension startthread }
\protected\def\pdfendthread          {\pdfextension endthread\relax}
\protected\def\pdfinfo               {\pdfextension info }
\protected\def\pdfcatalog            {\pdfextension catalog }
\protected\def\pdfnames              {\pdfextension names }
\protected\def\pdfincludechars       {\pdfextension includechars }
\protected\def\pdffontattr           {\pdfextension fontattr }
\protected\def\pdfmapfile            {\pdfextension mapfile }
\protected\def\pdfmapline            {\pdfextension mapline }
\protected\def\pdftrailer            {\pdfextension trailer }
\protected\def\pdfglyphtounicode     {\pdfextension glyphtounicode }
```

The introspective primitives can be defined as:

```
\def\pdftexversion    {\numexpr\pdffeedback version\relax}
\def\pdftexrevision           {\pdffeedback revision}
\def\pdflastlink      {\numexpr\pdffeedback lastlink\relax}
\def\pdfretval        {\numexpr\pdffeedback retval\relax}
\def\pdflastobj       {\numexpr\pdffeedback lastobj\relax}
\def\pdflastannot     {\numexpr\pdffeedback lastannot\relax}
\def\pdfxformname     {\numexpr\pdffeedback xformname\relax}
\def\pdfcreationdate          {\pdffeedback creationdate}
\def\pdffontname      {\numexpr\pdffeedback fontname\relax}
\def\pdffontobjnum    {\numexpr\pdffeedback fontobjnum\relax}
\def\pdffontsize      {\dimexpr\pdffeedback fontsize\relax}
\def\pdfpageref       {\numexpr\pdffeedback pageref\relax}
\def\pdfcolorstackinit        {\pdffeedback colorstackinit}
```

The configuration related registers have become:

```
\edef\pdfcompresslevel            {\pdfvariable compresslevel}
\edef\pdfobjcompresslevel         {\pdfvariable objcompresslevel}
\edef\pdfrecompress               {\pdfvariable recompress}
\edef\pdfdecimaldigits            {\pdfvariable decimaldigits}
\edef\pdfgamma                    {\pdfvariable gamma}
\edef\pdfimageresolution          {\pdfvariable imageresolution}
\edef\pdfimageapplygamma          {\pdfvariable imageapplygamma}
\edef\pdfimagegamma               {\pdfvariable imagegamma}
\edef\pdfimagehicolor             {\pdfvariable imagehicolor}
\edef\pdfimageaddfilename         {\pdfvariable imageaddfilename}
\edef\pdfpkresolution             {\pdfvariable pkresolution}
\edef\pdfpkfixeddpi               {\pdfvariable pkfixeddpi}
\edef\pdfinclusioncopyfonts       {\pdfvariable inclusioncopyfonts}
\edef\pdfinclusionerrorlevel      {\pdfvariable inclusionerrorlevel}
\edef\pdfignoreunknownimages      {\pdfvariable ignoreunknownimages}
\edef\pdfgentounicode             {\pdfvariable gentounicode}
\edef\pdfomitcidset               {\pdfvariable omitcidset}
\edef\pdfpagebox                  {\pdfvariable pagebox}
```


```
\edef\pdfminorversion             {\pdfvariable minorversion}
\edef\pdfuniqueresname            {\pdfvariable uniqueresname}

\edef\pdfhorigin                  {\pdfvariable horigin}
\edef\pdfvorigin                  {\pdfvariable vorigin}
\edef\pdflinkmargin               {\pdfvariable linkmargin}
\edef\pdfdestmargin               {\pdfvariable destmargin}
\edef\pdfthreadmargin             {\pdfvariable threadmargin}
\edef\pdfxformmargin              {\pdfvariable xformmargin}

\edef\pdfpagesattr                {\pdfvariable pagesattr}
\edef\pdfpageattr                 {\pdfvariable pageattr}
\edef\pdfpageresources            {\pdfvariable pageresources}
\edef\pdfxformattr                {\pdfvariable xformattr}
\edef\pdfxformresources           {\pdfvariable xformresources}
\edef\pdfpkmode                   {\pdfvariable pkmode}

\edef\pdfsuppressoptionalinfo     {\pdfvariable suppressoptionalinfo }
\edef\pdftrailerid                {\pdfvariable trailerid }
```

The variables are internal ones, so they are anonymous. When you ask for the meaning of a few previously defined ones:

```
\meaning\pdfhorigin
\meaning\pdfcompresslevel
\meaning\pdfpageattr
```

you will get:

```
macro:->[internal backend dimension]
macro:->[internal backend integer]
macro:->[internal backend tokenlist]
```

The `\edef` can also be a `\def` but it's a bit more efficient to expand the lookup related register beforehand. After that you can adapt the defaults; these are:

```
\pdfcompresslevel         9
\pdfobjcompresslevel      1 % used: (0,9)
\pdfrecompress            0 % mostly for debugging
\pdfdecimaldigits         4 % used: (3,6)
\pdfgamma              1000
\pdfimageresolution      71
\pdfimageapplygamma       0
\pdfimagegamma         2200
\pdfimagehicolor          1
\pdfimageaddfilename      1
\pdfpkresolution         72
\pdfpkfixeddpi            0
```

```
\pdfinclusioncopyfonts    0
\pdfinclusionerrorlevel   0
\pdfignoreunknownimages   0
\pdfgentounicode          0
\pdfomitcidset            0
\pdfpagebox               0
\pdfminorversion          4
\pdfuniqueresname         0

\pdfhorigin              1in
\pdfvorigin              1in
\pdflinkmargin           0pt
\pdfdestmargin           0pt
\pdfthreadmargin         0pt
\pdfxformmargin          0pt
```

If you also want some backward compatibility, you can add:

```
\let\pdfpagewidth      \pagewidth
\let\pdfpageheight     \pageheight

\let\pdfadjustspacing  \adjustspacing
\let\pdfprotrudechars  \protrudechars
\let\pdfnoligatures    \ignoreligaturesinfont
\let\pdffontexpand     \expandglyphsinfont
\let\pdfcopyfont       \copyfont

\let\pdfxform          \saveboxresource
\let\pdflastxform      \lastsavedboxresourceindex
\let\pdfrefxform       \useboxresource

\let\pdfximage         \saveimageresource
\let\pdflastximage     \lastsavedimageresourceindex
\let\pdflastximagepages\lastsavedimageresourcepages
\let\pdfrefximage      \useimageresource

\let\pdfsavepos        \savepos
\let\pdflastxpos       \lastxpos
\let\pdflastypos       \lastypos

\let\pdfoutput         \outputmode
\let\pdfdraftmode      \draftmode

\let\pdfpxdimen        \pxdimen

\let\pdfinsertht       \insertht

\let\pdfnormaldeviate  \normaldeviate
```


```
\let\pdfuniformdeviate \uniformdeviate
\let\pdfsetrandomseed  \setrandomseed
\let\pdfrandomseed     \randomseed

\let\pdfprimitive      \primitive
\let\ifpdfprimitive    \ifprimitive

\let\ifpdfabsnum       \ifabsnum
\let\ifpdfabsdim       \ifabsdim
```

And even:

```
\newdimen\pdfeachlineheight
\newdimen\pdfeachlinedepth
\newdimen\pdflastlinedepth
\newdimen\pdffirstlineheight
\newdimen\pdfignoreddimen
```

The backend is derived from pdfTEX so the same syntax applies. However, the `outline` command accepts a `objnum` followed by a number. No checking takes place so when this is used it had better be a valid (flushed) object.

In order to be (more or less) compatible with pdfTEX we also support the option to suppress some info:

```
\pdfvariable suppressoptionalinfo \numexpr
        0
    +   1   % PTEX.FullBanner
    +   2   % PTEX.FileName
    +   4   % PTEX.PageNumber
    +   8   % PTEX.InfoDict
    +  16   % Creator
    +  32   % CreationDate
    +  64   % ModDate
    + 128   % Producer
    + 256   % Trapped
    + 512   % ID
\relax
```

In addition you can overload the trailer id, but we don't do any checking on validity, so you have to pass a valid array. The following is like the ones normally generated by the engine:

```
\pdfvariable trailerid {[
    <FA052949448907805BA83C1E78896398>
    <FA052949448907805BA83C1E78896398>
]}
```

So, you even need to include the brackets!

Although we started from a merge of pdfTEX and Aleph, by now the code base as well as functionality has diverted from those parents. Here we show the options that can be passed to the extensions.

```
\pdfextension literal
    [ direct | page | raw ] { tokens }

\pdfextension dest
    num integer | name { tokens }!crlf
    [ fitbh | fitbv | fitb | fith| fitv | fit |
      fitr <rule spec> | xyz [ zoom <integer> ]

\pdfextension annot
    reserveobjnum | useobjnum <integer>
    { tokens }

\pdfextension save

\pdfextension restore

\pdfextension setmatrix
    { tokens }

[ \immediate ] \pdfextension obj
    reserveobjnum

[ \immediate ] \pdfextension obj
    [ useobjnum <integer> ]
    [ uncompressed ]
    [ stream  [ attr { tokens } ] ]
    [ file ]
    { tokens }

\pdfextension refobj
    <integer>

\pdfextension colorstack
    <integer>
    set { tokens } | push { tokens } | pop | current

\pdfextension startlink
    [ attr { tokens } ]
    user { tokens } | goto | thread
    [ file { tokens } ]
    [ page <integer> { tokens } | name { tokens } | num  integer ]
    [ newwindow  | nonewwindow ]

\pdfextension endlink

\pdfextension startthread
```


```
    num <integer> | name { tokens }

\pdfextension endthread

\pdfextension thread
    num <integer> | name { tokens }

\pdfextension outline
    [ attr { tokens } ]
    [ useobjnum <integer> ]
    [ count <integer> ]
    { tokens }

\pdfextension glyphtounicode
    { tokens }
    { tokens }

\pdfextension catalog
    { tokens }
    [ openaction
      user { tokens } | goto | thread
      [ file { tokens } ]
      [ page <integer> { tokens } | name { tokens } | num <integer> ]
      [ newwindow  | nonewwindow ] ]

\pdfextension fontattr
    <integer>
    {tokens}

\pdfextension mapfile
    {tokens}

\pdfextension mapline
    {tokens}

\pdfextension includechars
    {tokens}

\pdfextension info
    {tokens}

\pdfextension names
    {tokens}

\pdfextension trailer
    {tokens}
```

3.3 Directions

The directional model in LuaTEX is inherited from Omega/Aleph but we tried to improve it a bit. At some point we played with recovery of modes but that was disabled later on when we found

that it interfered with nested directions. That itself had as side effect that the node list was no longer balanced with respect to directional nodes which in turn can give side effects when a series of dir changes happens without grouping.

The current (0.97 onward) approach is that we again make the list balanced but try to avoid some side effects. What happens is quite intuitive if we forget about spaces (turned into glue) but even there what happens makes sense if you look at it in detail. However that logic makes in-group switching kind of useless when no proper nested grouping is used: switching from right to left several times nested, results in spacing ending up after each other due to nested mirroring. Of course a sane macro package will manage this for the user but here we are discussing the low level dir injection.

This is what happens:

```
\textdir TRT nur {\textdir TLT run \textdir TRT NUR} nur
```

This becomes stepwise:

```
injected: [+TRT]nur {[+TLT]run [+TRT]NUR} nur
balanced: [+TRT]nur {[+TLT]run [-TLT][+TRT]NUR[-TRT]} nur[-TRT]
result  : run {RUNrun } run
```

And this:

```
\textdir TRT nur {nur \textdir TLT run \textdir TRT NUR} nur
```

becomes:

```
injected: [+TRT]nur {nur [+TLT]run [+TRT]NUR} nur
balanced: [+TRT]nur {nur [+TLT]run [-TLT][+TRT]NUR[-TRT]} nur[-TRT]
result  : run {run RUNrun } run
```

Now, in the following examples watch where we put the braces:

```
\textdir TRT nur {{\textdir TLT run} {\textdir TRT NUR}} nur
```

This becomes:

run RUN run run

Compare this to:

```
\textdir TRT nur {{\textdir TLT run }{\textdir TRT NUR}} nur
```

Which renders as:

run RUNrun run

So how do we deal with the next?

```
\def\ltr{\textdir TLT\relax}
\def\rtl{\textdir TRT\relax}

run {\rtl nur {\ltr run \rtl NUR \ltr run \rtl NUR} nur}
```


```
run {\ltr run {\rtl nur \ltr RUN \rtl nur \ltr RUN} run}
```

It gets typeset as:

run run RUNrun RUNrun run
run run runRUN runRUN run

We could define the two helpers to look back, pick up a skip, remove it and inject it after the dir node. But that way we loose the subtype information that for some applications can be handy to be kept as-is. This is why we now have a variant of `\textdir` which injects the balanced node before the skip. Instead of the previous definition we can use:

```
\def\ltr{\linedir TLT\relax}
\def\rtl{\linedir TRT\relax}
```

and this time:

```
run {\rtl nur {\ltr run \rtl NUR \ltr run \rtl NUR} nur}
run {\ltr run {\rtl nur \ltr RUN \rtl nur \ltr RUN} run}
```

comes out as a properly spaced:

run run RUN run RUN run run
run run run RUN run RUN run

Anything more complex that this, like combination of skips and penalties, or kerns, should be handled in the input or macro package because there is no way we can predict the expected behaviour. In fact, the `\linedir` is just a convenience extra which could also have been implemented using node list parsing.

Glue after a dir node is ignored in the linebreak decision but you can bypass that by setting `\breakafterdirmode` to 1. The following table shows the difference. Watch your spaces.

	0	1
pre {\textdir TLT xxx} post	pre xxx post	pre xxx post
pre {\textdir TLT xxx }post	pre xxx post	pre xxx post
pre{ \textdir TLT xxx} post	pre xxx post	pre xxx post
pre{ \textdir TLT xxx }post	pre xxx post	pre xxx post
pre { \textdir TLT xxx } post	pre xxx post	pre xxx post

```
pre {\textdir TLT\relax \space xxx} post  pre         pre
                                           xxx post
                                                    xxx
                                                    post
```

Another adaptation to the Aleph directional model is control over shapes driven by `\hangindent` and `\parshape`. This is controlled by a new parameter `\shapemode`:

VALUE	\HANGINDENT	\PARSHAPE
0	normal	normal
1	mirrored	normal
2	normal	mirrored
3	mirrored	mirrored

The value is reset to zero (like `\hangindent` and `\parshape`) after the paragraph is done with. You can use negative values to prevent this. In figure 3.1 a few examples are given.

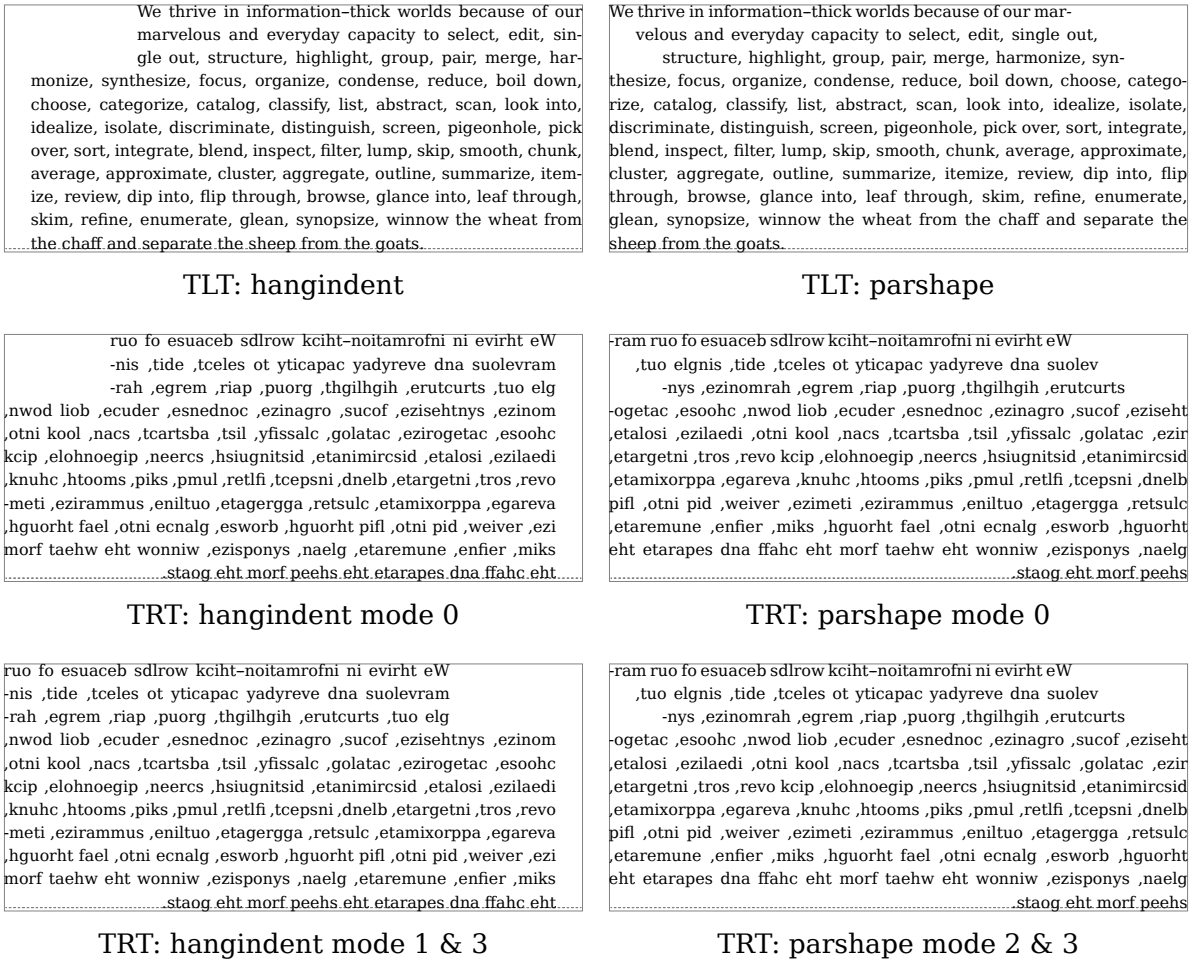


Figure 3.1 The effect of `shapemode`.

3.4 Implementation notes

3.4.1 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 `texmf.cnf` settings related to main memory are no longer used (these are: `main_memory`, `mem_bot`, `extra_mem_top` and `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 `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 `vlink` and `vinfo` as well as `token_link` and `token_info`. All access to the variable memory array is now hidden behind a macro called `vmem`. We mention this because using the TEXbook as reference is still quite valid but not for memory related details. Another significant detail is that we have double linked node lists and that most nodes carry more data.

The input line buffer and pool size are now also reallocated when needed, and the `texmf.cnf` settings `buf_size` and `pool_size` are silently ignored.

3.4.2 Sparse arrays

The `\mathcode`, `\delcode`, `\catcode`, `\sfcode`, `\lccode` and `\uccode` (and the new `\hjcode`) 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. Performance is not really hurt by this.

The `\catcode`, `\sfcode`, `\lccode`, `\uccode` and `\hjcode` assignments don't show up when using the ε-TEX tracing routines `\tracingassigns` and `\tracingrestores` but we don't see that as a real limitation.

A side-effect of the current implementation is that `\global` is now more expensive in terms of processing than non-global assignments but not many users will notice that.

The glyph ids within a font are also managed by means of a sparse array as glyph ids can go up to index $2^{21} - 1$ but these are never accessed directly so again users will not notice this.

3.4.3 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.

3.4.4 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 io, so it should still be faster.

3.4.5 Binary file reading

All of the internal code is changed in such a way that if one of the `read_xxx_file` callbacks is not set, then the file is read by a C function using basically the same convention as the callback: a single read into a buffer big enough to hold the entire file contents. While this uses more memory than the previous code (that mostly used `getc` calls), it can be quite a bit faster (depending on your io subsystem).

3.4.6 Tabs and spaces

We conform to the way other TEX engines handle trailing tabs and spaces. For decades trailing tabs and spaces (before a newline) were removed from the input but this behaviour was changed in September 2017 to only handle spaces. We are aware that this can introduce compatibility issues in existing workflows but because we don't want too many differences with upstream TEXLive we just follow up on that patch (which is a functional one and not really a fix). It is up to macro packages maintainers to deal with possible compatibility issues and in LuaTEX they can do so via the callbacks that deal with reading from files.

The previous behaviour was a known side effect and (as that kind of input normally comes from generated sources) it was normally dealt with by adding a comment token to the line in case the spaces and/or tabs were intentional and to be kept. We are aware of the fact that this contradicts some of our other choices but consistency with other engines and the fact that in kpse mode a common file io layer is used can have a side effect of breaking compatibility. We still stick to our view that at the log level we can (and might be) more incompatible. We already expose some more details.

4 LUA general

4.1 Initialization

4.1.1 LUATEX as a LUA interpreter

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

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

In this mode, it will set Lua's `arg[0]` to the found script name, pushing preceding options in negative values and the rest of the command line 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 stand alone Lua interpreter with a bunch of extra preloaded libraries.

4.1.2 LUATEX as a LUA byte compiler

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

- if a `--luaconly` option is given on the command line, or
- if the executable is named `texluac`

In this mode, LuaTEX is exactly like `luac` from the stand alone Lua distribution, except that it does not have the `-l` switch, and that it accepts (but ignores) the `--luaconly` switch. The current version of Lua can dump bytecode using `string.dump` so we might decide to drop this version of LuaTEX.

4.1.3 Other commandline processing

When the LuaTEX executable starts, it looks for the `--lua` command line option. If there is no `--lua` option, the command line is interpreted in a similar fashion as the other TEX engines. Some options are accepted but have no consequence. The following command-line options are understood:

COMMANDLINE ARGUMENT	EXPLANATION
`--credits`	display credits and exit
`--debug-format`	enable format debugging
`--draftmode`	switch on draft mode i.e. generate no output in pdf mode
`--[no-]file-line-error`	disable/enable `file:line:error` style messages
`--[no-]file-line-error-style`	aliases of `--[no-]file-line-error`
`--fmt=FORMAT`	load the format file `FORMAT`

`--halt-on-error`	stop processing at the first error
`--help`	display help and exit
`--ini`	be `iniluatex`, for dumping formats
`--interaction=STRING`	set interaction mode: `batchmode`, `nonstopmode`, `scrollmode` or `errorstopmode`
`--jobname=STRING`	set the job name to `STRING`
`--kpathsea-debug=NUMBER`	set path searching debugging flags according to the bits of `NUMBER`
`--lua=FILE`	load and execute a Lua initialization script
`--[no-]mktex=FMT`	disable/enable `mktexFMT` generation with `FMT` is `tex` or `tfm`
`--nosocket`	disable the Lua socket library
`--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
`--output-format=FORMAT`	use `FORMAT` for job output; `FORMAT` is `dvi` or `pdf`
`--progname=STRING`	set the program name to `STRING`
`--recorder`	enable filename recorder
`--safer`	disable easily exploitable Lua commands
`--[no-]shell-escape`	disable/enable system calls
`--shell-restricted`	restrict system calls to a list of commands given in `texmf.cnf`
`--synctex=NUMBER`	enable `synctex`
`--utc`	use utc times when applicable
`--version`	display version and exit

We don't support `\write 18` because `os.execute` can do the same. It simplifies the code and makes more write targets possible.

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.
- There is 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 and lacks some others. Also, if the `--lua` option is present, LuaTEX will enter an alternative mode of command line processing in comparison to the standard web2c programs. In this mode, a small series of actions is taken in the following order:

1. First, it will parse the command line as usual, but it will only interpret a small subset of the options immediately: `--safer`, `--nosocket`, `--[no-]shell-escape`, `--enable-write18`, `--disable-write18`, `--shell-restricted`, `--help`, `--version`, and `--credits`.
2. Next LuaTEX searches for the requested Lua initialization script. If it cannot be found using the actual name given on the command line, a second attempt is made by prepending the value of the environment variable `LUATEXDIR`, if that variable is defined in the environment.
3. Then it checks the various safety switches. You can use those to disable some Lua commands that can easily be abused by a malicious document. At the moment, `--safer` `nil`s the following functions:

LIBRARY	FUNCTIONS
os	execute exec spawn setenv rename remove tmpdir
io	popen output tmpfile
lfs	rmdir mkdir chdir lock touch

   Furthermore, it disables loading of compiled Lua libraries and it makes `io.open()` fail on files that are opened for anything besides reading.
4. When LuaTEX starts it sets the `locale` to a neutral value. If for some reason you use `os.locale`, you need to make sure you `nil` it afterwards because otherwise it can interfere with code that for instance generates dates. You can ignore the `locale` with:

   ```
   os.setlocale(nil,nil)
   ```

   The `--nosocket` option makes the socket library unavailable, so that Lua cannot use networking.
   The switches `--[no-]shell-escape`, `--[enable|disable]-write18`, and `--shell-restricted` have the same effects as in pdfTEX, and additionally make `io.popen()`, `os.execute`, `os.exec` and `os.spawn` adhere to the requested option.
5. Next the initialization script is loaded and executed. From within the script, the entire command line is available in the Lua table `arg`, beginning with `arg[0]`, containing the name of the executable. As consequence warnings about unrecognized options are suppressed.

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

Everything you do in the Lua initialization script will remain visible during the rest of the run, with the exception of the TEX specific libraries like `tex`, `token`, `node` and `pdf` tables. These 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 command line, set values in the `texconfig` table, and register the callbacks you need.

LuaTEX allows some of the command line options to be overridden by reading values from the `texconfig` table at the end of script execution (see the description of the `texconfig` table later on in this document for more details on which ones exactly).

Unless the `texconfig` table tells LuaTEX not to initialize kpathsea at all (set `texconfig.kpse_init` to `false` for that), LuaTEX acts on some more command line options after the initialization script is finished: in order to initialize the built-in kpathsea library properly, LuaTEX needs to know the correct program name to use, and for that it needs to check `--progname`, or `--ini` and `--fmt`, if `--progname` is missing.

4.2 LUA behaviour

Luas `tostring` function (and `string.format` may return values in scientific notation, thereby confusing the TEX end of things when it is used as the right-hand side of an assignment to a `\dimen` or `\count`. The output of these serializers also depend on the Lua version, so in Lua 5.3 you can get different output than from 5.2.

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

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

LuaTEX is able to use dynamically loadable Lua libraries, unless `--safer` was given as an option on the command line. For this purpose, `package.searchers[3]` is replaced by a different loader function, that decides at runtime whether to use kpathsea or the built-in core Lua function. It uses kpathsea when that is already initialized at that point in time, otherwise it reverts to using the normal `package.cpath` loader.

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

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

This path is imperfect (it requires a tds subtree below the binaries directory), but the architecture has to be in the path somewhere, and the currently simplest way to do that is to search below the binaries directory only. Of course it no big deal to write an alternative loader and use that in a macro package. One level up (a `lib` directory parallel to `bin`) would have been nicer, but that is not doable because TEXLive uses a `bin/<arch>` structure.

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 5.3 inside LuaTEX, and another will likely be linked to the dll file of the module itself).

In keeping with the other TEX-like programs in TEXLive, the two Lua functions `os.execute` and `io.popen`, as well as the two new functions `os.exec` and `os.spawn` that are explained below, take the value of `shell_escape` and/or `shell_escape_commands` in account. Whenever LuaTEX

is run with the assumed intention to typeset a document (and by that we mean that it is called as `luatex`, as opposed to `texlua`, and that the command line option `--luaonly` was not given), it will only run the four functions above if the matching `texmf.cnf` variable(s) or their `texconfig` (see section 10.4) counterparts allow execution of the requested system command. In 'script interpreter' runs of LuaTEX, these settings have no effect, and all four functions have their original meaning.

Some libraries have a few more functions, either coded in C or in Lua. For instance, when we started with LuaTEX we added some helpers to the `luafilesystem` namespace `lfs`. The two boolean functions `lfs.isdir` and `lfs.isfile` were speedy and better variants of what could be done with `lfs.attributes`. The additional function `lfs.shortname` takes a file name and returns its short name on `win32` platforms. Finally, for non-`win32` platforms only, we provided `lfs.readlink` that takes an existing symbolic link as argument and returns its name. However, the `ibrary evoved` so we have dropped these in favour of pure Lua variants. The `shortname` helper is obsolete and now just returns the name.

The `string` library has a few extra functions like `string.explode(s[,m])`. This function returns an array containing the string argument `s` split into sub-strings based on the value of the string argument `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 + (this special version does not create empty sub-strings). The default value for `m` is ' +' (multiple spaces). Note: `m` is not hidden by surrounding braces as it would be if this function was written in TEX macros.

The `string` library also has six extra iterators that return strings piecemeal:

- `string.utfvalues(s)`: an integer value in the Unicode range
- `string.utfcharacters(s)`: a string with a single utf-8 token in it
- `string.characters(s)`: a string containing one byte
- `string.characterpairs(s)`: two strings each containing one byte or an empty second string if the string length was odd
- `string.bytes(s)`: a single byte value
- `string.bytepairs(s)`: two byte values or nil instead of a number as its second return value if the string length was odd

The `string.characterpairs()` and `string.bytepairs()` iterators are useful especially in the conversion of utf16 encoded data into utf8.

There is also a two-argument form of `string.dump()`. The second argument is a boolean which, if true, strips the symbols from the dumped data. This matches an extension made in `luajit`. This is typically a function that gets adapted as Lua itself progresses.

The `string` library functions `len`, `lower`, `sub` etc. are not Unicode-aware. For strings in the utf8 encoding, i.e., strings containing characters above code point 127, the corresponding functions from the `slnunicode` library can be used, e.g., `unicode.utf8.len`, `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 `()` but 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.

Version 5.3 of Lua provides some native utf8 support but we have added a few similar helpers too:

- `string.utfvalue(s)`: returns the codepoints of the characters in the given string
- `string.utfcharacter(c,...)`: returns a string with the characters of the given code points
- `string.utflength(s)`: returns the length oif the given string

These three functions are relative fast and don't do much checking. They can be used as building blocks for other helpers. So, eventually we can decide to drop the `sln` library, just that you know.

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: `\directlua{tex.sprint(os.selfdir)}`.
- `os.exec(commandline)` is a variation on `os.execute`. Here `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 unescaped. 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 MS 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)` sets a variable in the environment. Passing `nil` instead of a value string will remove the variable.
- `os.env` 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()` creates 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` 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` 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`, `cygwin`, `openbsd`, `solaris`, `sunos` (pre-solaris), `hpux`, `irix`, `macosx`, `gnu` (hurd), `bsd` (unknown, but bsd-like), `sysv` (unknown, but sysv-like), `generic` (unknown).
- `os.uname()` returns a table with specific operating system information acquired at runtime. The keys in the returned table are all string values, 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
```

4.3 LUA modules

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

- `lpeg`, by Roberto Ierusalimschy, http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html. This library is not Unicode-aware, but interprets strings on a byte-per-byte basis. This mainly means that `lpeg.S` cannot be used with utf8 characters encoded in more than 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 multibyte characters. Therefore `lpeg.R('aä')` results in the message `bad argument #1 to 'R' (range must have two characters)`, since to `lpeg`, ä is two 'characters' (bytes), so aä totals three. In practice this is no real issue and with some care you can deal with Unicode just fine.
- `slnunicode`, from the `selene` libraries, http://luaforge.net/projects/sln. 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. We have no plans to provide more like this because you can basically do all that you want in Lua.
- `luazip`, from the kepler project, http://www.keplerproject.org/luazip/.

- `luafilesystem`, also from the kepler project, http://www.keplerproject.org/luafilesystem/.
- `lzlib`, by Tiago Dionizio, http://luaforge.net/projects/lzlib/.
- `md5`, by Roberto Ierusalimschy http://www.inf.puc-rio.br/~roberto/md5/md5-5/md5.html.
- `luasocket`, by Diego Nehab http://w3.impa.br/~diego/software/luasocket/. The `.lua` support modules from `luasocket` are also preloaded inside the executable, there are no external file dependencies.

4.4 Testing

For development reasons you can influence the used startup date and time. This can be done in two ways.

1. By setting the environmment variable `SOURCE_DATE_EPOCH`. This will influence the TEX parameters `time` and `date`, the random seed, the pdf timestamp and the pdf id that is derived from the time as well. This variable is consulted when the kpse library is enabled. Resolving is delegated to this library.
2. By setting the `start_time` variable in the `texconfig` table; as with other variables we use the internal name there. For compatibility reasons we also honour a `SOURCE_DATE_EPOCH` entry. It should be noted that there are no such variables in other engines and this method is only relevant in case the while setup happens in Lua.

When Universal Time is needed, you can pass the flag `utc` to the engine. This property also works when the date and time are set by LuaTEX itself. It has a complementary entry `use_utc_time` in the `texconfig` table.

There is some control possible, for instance prevent filename to be written to the pdf file. This is discussed elsewhere. In ConTEXt we provide the command line argument `--nodates` that does a bit more disabling of dates.

5 Languages, 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 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.

Those `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 at all. Instead, language information is passed along using `language whatsit` nodes 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 current font and a reference to a character in that 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.

5.1 Characters and glyphs

TEX82 (including pdfTEX) differentiates between `char` nodes and `lig` nodes. The former are simple items that contained nothing but a 'character' and a 'font' field, and they lived in the same memory as tokens did. The latter also contained a list of components, and a subtype indicating whether this ligature was the result of a word boundary, and it was stored in the same place as other nodes like boxes and kerns and glues.

In LuaTEX, these two types are merged into one, somewhat larger structure called a `glyph` node. Besides having the old character, font, and component fields there are a few more, like 'attr' that

we will see in section 8.1.2.12, these nodes also contain a subtype, that codes four main types and two additional ghost types. For ligatures, multiple bits can be set at the same time (in case of a single-glyph word).

- `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 constructed ligatures bit 1 is set.
- `ghost`, for so called 'ghost objects' bit 2 is set.
- `left`, for ligatures created from a left word boundary and for ghosts created from `\leftghost` bit 3 gets set.
- `right`, for ligatures created from a right word boundary and for ghosts created from `\rightghost` bit 4 is set.

The `glyph` nodes 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. The language is stored with each character. You can set `\firstvalidlanguage` to for instance 1 and make thereby language 0 an ignored hyphenation language.

The new primitive `\hyphenationmin` can be used to signal the minimal length of a word. This value is stored with the (current) language.

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.

In traditional TEX the process of hyphenation is driven by `lccodes`. In LuaTEX we made this dependency less strong. There are several strategies possible. When you do nothing, the currently used `lccodes` are used, when loading patterns, setting exceptions or hyphenating a list.

When you set `\savinghyphcodes` to a value greater than zero the current set of `lccodes` will be saved with the language. In that case changing a `lccode` afterwards has no effect. However, you can adapt the set with:

```
\hjcode`a=`a
```

This change is global which makes sense if you keep in mind that the moment that hyphenation happens is (normally) when the paragraph or a horizontal box is constructed. When `\savinghyphcodes` was zero when the language got initialized you start out with nothing, otherwise you already have a set.

When a `\hjcode` is greater than 0 but less than 32 is indicates the to be used length. In the following example we map a character (x) onto another one in the patterns and tell the engine that œ counts as one character. Because traditionally zero itself is reserved for inhibiting hyphenation, a value of 32 counts as zero.

Here are some examples (we assume that French patterns are used):

```
                              foobar  foo-bar
\hjcode `x=`o                 fxxbar  fxx-bar
\lefthyphenmin 3              œdipus  œdi-pus
\lefthyphenmin 4              œdipus  œdipus
\hjcode `œ=2                  œdipus  œdi-pus
\hjcode `i=32 \hjcode `d=32   œdipus  œdipus
```

Carrying all this information with each glyph would give too much overhead and also make the process of setting up these codes more complex. A solution with `hjcode` sets was considered but rejected because in practice the current approach is sufficient and it would not be compatible anyway.

Beware: the values are always saved in the format, independent of the setting of `\savinghyphcodes` at the moment the format is dumped.

A boundary node normally would mark the end of a word which interferes with for instance discretionary injection. For this you can use the `\wordboundary` as a trigger. Here are a few examples of usage:

```
discrete---discrete
```

discrete—discrete

```
discrete\discretionary{}{}{---}discrete
```

discrete
discrete

```
discrete\wordboundary\discretionary{}{}{---}discrete
```

dis-
crete
discrete

```
discrete\wordboundary\discretionary{}{}{---}\wordboundary discrete
```

dis-
crete
dis-
crete

```
discrete\wordboundary\discretionary{---}{}{}\wordboundary discrete
```

dis-
crete—
dis-
crete

We only accept an explicit hyphen when there is a preceding glyph and we skip a sequence of explicit hyphens since that normally indicates a `--` or `---` ligature in which case we can in a

worse case usage get bad node lists later on due to messed up ligature building as these dashes are ligatures in base fonts. This is a side effect of separating the hyphenation, ligaturing and kerning steps.

The start and end of a sequence of characters is signalled by a `glue`, `penalty`, `kern` or `boundary` node. But by default also a `hlist`, `vlist`, `rule`, `dir`, `whatsit`, `ins`, and `adjust` node indicate a start or end. You can omit the last set from the test by setting `\hyphenationbounds` to a non-zero value:

VALUE	BEHAVIOUR
0	not strict
1	strict start
2	strict end
3	strict start and strict end

The word start is determined as follows:

NODE	BEHAVIOUR
boundary	yes when wordboundary
hlist	when hyphenationbounds 1 or 3
vlist	when hyphenationbounds 1 or 3
rule	when hyphenationbounds 1 or 3
dir	when hyphenationbounds 1 or 3
whatsit	when hyphenationbounds 1 or 3
glue	yes
math	skipped
glyph	exhyphenchar (one only) : yes (so no – —)
otherwise	yes

The word end is determined as follows:

NODE	BEHAVIOUR
boundary	yes
glyph	yes when different language
glue	yes
penalty	yes
kern	yes when not italic (for some historic reason)
hlist	when hyphenationbounds 2 or 3
vlist	when hyphenationbounds 2 or 3
rule	when hyphenationbounds 2 or 3
dir	when hyphenationbounds 2 or 3
whatsit	when hyphenationbounds 2 or 3
ins	when hyphenationbounds 2 or 3
adjust	when hyphenationbounds 2 or 3

Figures 5.1 upto 5.5 show some examples. In all cases we set the min values to 1 and make sure that the words hyphenate at each character.

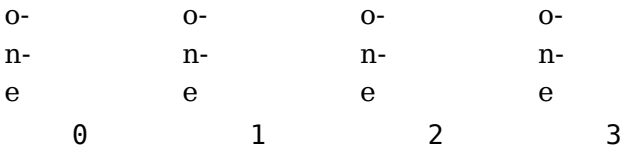


Figure 5.1 one

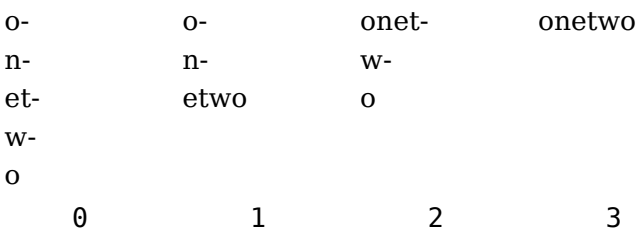


Figure 5.2 one\null two

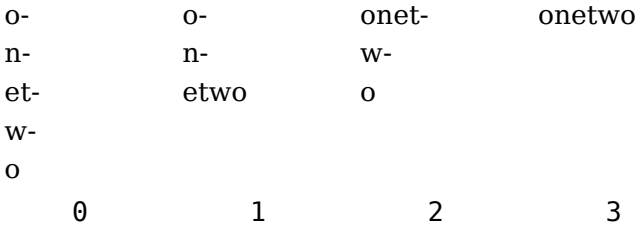


Figure 5.3 \null one\null two

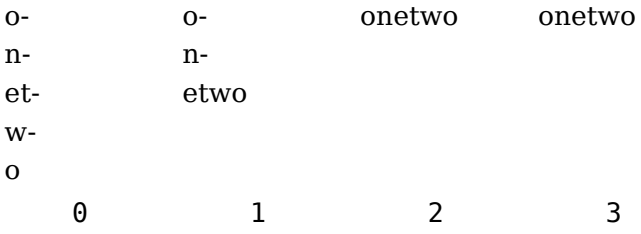


Figure 5.4 one\null two\null

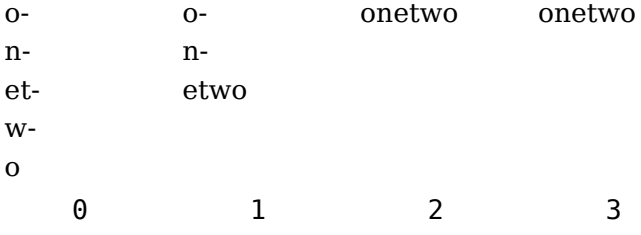


Figure 5.5 \null one\null two\null

In traditional TEX ligature building and hyphenation are interwoven with the line break mechanism. In LuaTEX these phases are isolated. As a consequence we deal differently with (a sequence of) explicit hyphens. We already have added some control over aspects of the hyphenation and yet another one concerns automatic hyphens (e.g. - characters in the input).

When \automatichyphenmode has a value of 0, a hyphen will be turned into an automatic discretionary. The snippets before and after it will not be hyphenated. A side effect is that a leading hyphen can lead to a split but one will seldom run into that situation. Setting a pre and post

character makes this more prominent. A value of 1 will prevent this side effect and a value of 2 will not turn the hyphen into a discretionary. Experiments with other options, like permitting hyphenation of the words on both sides were discarded.

In figure 5.6 and 5.7 we show what happens with three samples:

Input A:

```
before-after \par
before--after \par
before---after \par
```

Input B:

```
-before \par
after- \par
--before \par
after-- \par
---before \par
after--- \par
```

Input C:

```
before-after \par
before--after \par
before---after \par
```

As with primitive companions of other single character commands, the `\-` command has a more verbose primitive version in `\explicitdiscretionary` and the normally intercepted in the hyphenator character `-` (or whatever is configured) is available as `\automaticdiscretionary`.

5.2 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 exceptions.

1. The `\accent` primitive 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. Of course, modern Unicode aware macro packages will not use the `\accent` primitive at all but try to map directly on composed characters.
   This change of meaning did happen with `\char`, that now generates 'glyph' nodes with a character subtype. In traditional TEX there was a strong relationship between the 8-bit input encoding, hyphenation and glyphs taken from a font. In LuaTEX we have utf input, and in most cases this maps directly to a character in a font, apart from glyph replacement in the

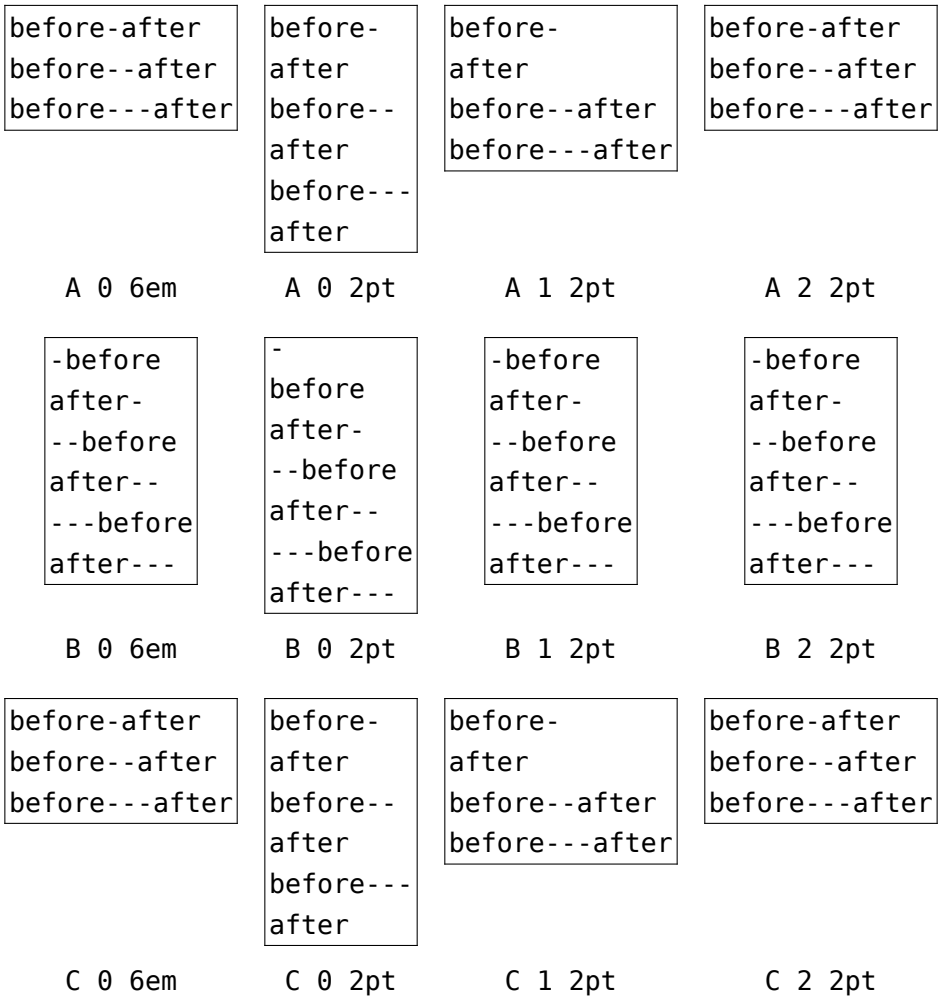


Figure 5.6 The automatic modes `0` (default), `1` and `2`, with a `\hsize` of 6em and 2pt (which triggers a linebreak).

font engine. If you want to access arbitrary glyphs in a font directly you can always use Lua to do so, because fonts are available as Lua table.

2. All the results of processing in math mode eventually become nodes with 'glyph' subtypes. In fact, the result of processing math is just a regular list of glyphs, kerns, glue, penalties, boxes etc.
3. 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.
4. 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.[1]

   In LuaTEX, it works like this: if LuaTEX senses a string of input characters that matches the value of the new integer parameter `\exhyphenchar`, it will insert an explicit discretionary

[1] When TEX showed up we didn't have Unicode yet and being limited to eight bits meant that one sometimes had to compromise between supporting character input, glyph rendering, hyphenation.

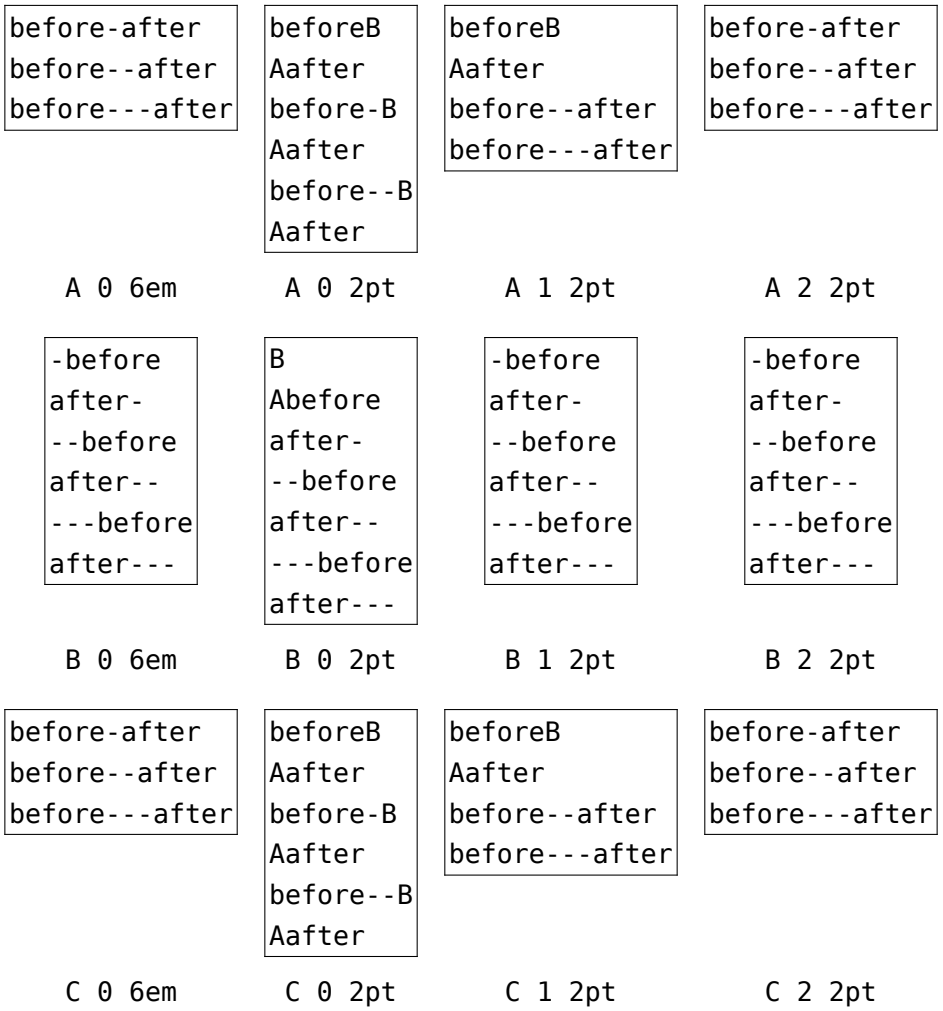


Figure 5.7 The automatic modes `0` (default), `1` and `2`, with `\preexhyphenchar` and `\postexhyphenchar` set to characters `A` and `B`.

after that series of nodes. Initially TEX sets the `\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.

The insertion of discretionaries after a sequence of explicit hyphens happens at the same time as the other hyphenation processing, *not* inside the main control loop.

The only use LuaTEX has for `\hyphenchar` is at the check whether a word should be considered for hyphenation at all. If the `\hyphenchar` of the font attached to the first character node in a word is negative, then hyphenation of that word is abandoned immediately. This behaviour is added for backward compatibility only, and the use of `\hyphenchar=-1` as a means of preventing hyphenation should not be used in new LuaTEX documents.

5. The `\setlanguage` command no longer creates whatsits. The meaning of `\setlanguage` is changed so that it is now an integer parameter like all others. That integer parameter is used in `\glyph_node` creation to add language information to the glyph nodes. In conjunction, the `\language` primitive is extended so that it always also updates the value of `\setlanguage`.
6. The `\noboundary` command (that prohibits word boundary processing where that would normally take place) now does create nodes. These nodes are needed because the exact place of the `\noboundary` command in the input stream has to be retained until after the ligature

and font processing stages.

7. There is no longer a `main_loop` label in the code. Remember that TEX82 did quite a lot of processing while adding `char_nodes` to the horizontal list? For speed reasons, it handled that processing code outside of the 'main control' loop, and only the first character of any 'word' was handled by that 'main control' loop. In LuaTEX, there is no longer a need for that (all hard work is done later), and the (now very small) bits of character-handling code have been moved back inline. When `\tracingcommands` is on, this is visible because the full word is reported, instead of just the initial character.

Because we tend to make hard coded behaviour configurable a few new primitives have been added:

```
\hyphenpenaltymode
\automatichyphenpenalty
\explicithyphenpenalty
```

The first parameter has the following consequences for automatic discs (the ones resulting from an `\exhyphenchar`:

MODE	AUTOMATIC DISC -	EXPLICIT DISC \-
0	\exhyphenpenalty	\exhyphenpenalty
1	\hyphenpenalty	\hyphenpenalty
2	\exhyphenpenalty	\hyphenpenalty
3	\hyphenpenalty	\exhyphenpenalty
4	\automatichyphenpenalty	\explicithyphenpenalty
5	\exhyphenpenalty	\explicithyphenpenalty
6	\hyphenpenalty	\explicithyphenpenalty
7	\automatichyphenpenalty	\exhyphenpenalty
8	\automatichyphenpenalty	\hyphenpenalty

other values do what we always did in LuaTEX: insert `\exhyphenpenalty`.

5.3 Loading patterns and exceptions

Although we keep the traditional approach towards hyphenation (which is still superior) the implementation of the hyphenation algorithm in LuaTEX is quite different from the one in TEX82.

After expansion, the argument for `\patterns` has to be proper utf8 with individual patterns separated by spaces, no `\char` or `\chardefd` commands are allowed. The current implementation is quite strict and will reject all non-Unicode characters. Likewise, the expanded argument for `\hyphenation` also has to be proper utf8, but here a bit of extra syntax is provided:

1. Three sets of arguments in curly braces (`{}{}{}`) indicate a desired complex discretionary, with arguments as in `\discretionary`'s command in normal document input.
2. A `-` indicates a desired simple discretionary, cf. `\-` and `\discretionary{-}{}{}` in normal document input.
3. Internal command names are ignored. This rule is provided especially for `\discretionary`, but it also helps to deal with `\relax` commands that may sneak in.
4. An = 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:

VALUE	IMPLIED KEY (INPUT)	EFFECT
`ta-ble`	table	`ta\-ble` (= `ta\discretionary{-}{}{}ble`)
`ba{k-}{}{c}ken`	backen	`ba\discretionary{k-}{}{c}ken`

The resultant patterns and exception dictionary will be stored under the language code that is the present value of `\language`.

In the last line of the table, you see there is no `\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 LuaTEX using one of the functions in the 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.

It is possible to specify extra hyphenation points in compound words by using `{-}{}{-}` for the explicit hyphen character (replace `-` by the actual explicit hyphen character if needed). For example, this matches the word 'multi-word-boundaries' and allows an extra break inbetween 'boun' and 'daries':

```
\hyphenation{multi{-}{}{-}word{-}{}{-}boun-daries}
```

The motivation behind the ε-TEX extension `\savinghyphcodes` was that hyphenation heavily depended on font encodings. This is no longer true in LuaTEX, and the corresponding primitive is basically ignored. Because we now have `\hjcode`, the case relate codes can be used exclusively for `\uppercase` and `\lowercase`.

The three curly brace pair pattern in an exception can be somewhat unexpected so we will try to explain it by example. The pattern `foo{}{}{x}bar` pattern creates a lookup `fooxbar` and the pattern `foo{}{}{}bar` creates `foobar`. Then, when a hit happens there is a replacement text (`x`) or none. Because we introduced penalties in discretionary nodes, the exception syntax now also can take a penalty specification. The value between square brackets is a multiplier for `\exceptionpenalty`. Here we have set it to 10000 so effectively we get 30000 in the example.

`x{a-}{-b}{}x{a-}{-b}{}x{a-}{-b}{}x{a-}{-b}{}xx`			
`10em`	`3em`	`0em`	`6em`

`123 xxxxxx 123`	`123`
	`xxa-`
	`-bxa-`
	`-bxa-`
	`-bxx`
	`123`
		`123`
		`xa-`
		`-bxa-`
		`-bxa-`
		`-bxa-`
		`-bxx`
		`123`
			`123 xxxxxx`
			`xxxxxx xxa-`
			`-bxxxx xxa-`
			`-bxxxx 123`

x{a-}{-b}{}x{a-}{-b}{}[3]x{a-}{-b}{}[1]x{a-}{-b}{}xx			
10em	3em	0em	6em
123 xxxxxx 123	123 xa- -bxxxa- -bxx 123	123 xa- -bxxxa- -bxx 123	123 xxxxa- -bxx xxxxxx xxxxxx xa- -bxxxxx 123

z{a-}{-b}{z}{a-}{-b}{z}{a-}{-b}{z}{a-}{-b}{z}z			
10em	3em	0em	6em
123 zzzzzz 123	123 za- -bza- -bza- -b 123	123 za- -bza- -bza- -b a- -b23	123 zzzzzz zzzzzz zzza- -bzz zzzzzz 123

z{a-}{-b}{z}{a-}{-b}{z}[3]{a-}{-b}{z}[1]{a-}{-b}{z}z			
10em	3em	0em	6em
123 zzzzzz 123	123 za- -bzzzz 123	123 za- -bzzzz a- -b23	123 zzzzzz zzzzzz za- -bzzzz a- -bzzzzz 123

5.4 Applying hyphenation

The internal structures LuaTEX 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 pattern files generated by patgen, but LuaTEX 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.

There are a few differences between LuaTEX and TEX82 that are a direct result of the implementation:

- LuaTEX happily hyphenates the full 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 `texmf.cnf` are ignored.
- Because there is no 'trie preparation' stage, language patterns never become frozen. This means that the primitive `\patterns` (and its Lua counterpart `lang.patterns`) can be used at any time, not only in iniTEX.

- Only the string representation of `\patterns` and `\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.
- LuaTEX uses the language-specific variables `\prehyphenchar` and `\posthyphenchar` in the creation of implicit discretionaries, instead of TEX82's `\hyphenchar`, and the values of the language-specific variables `\preexhyphenchar` and `\postexhyphenchar` for explicit discretionaries (instead of TEX82's empty discretionary).
- The value of the two counters related to hyphenation, `\hyphenpenalty` and `\exhyphenpenalty`, are now stored in the discretionary nodes. This permits a local overload for explicit `\discretionary` commands. The value current when the hyphenation pass is applied is used. When no callbacks are used this is compatible with traditional TEX. When you apply the Lua `lang.hyphenate` function the current values are used.
- The hyphenation exception dictionary is maintained as key-value hash, and that is also dynamic, so the `hyph_size` setting is not used either.

Because we store penalties in the disc node the `\discretionary` command has been extended to accept an optional penalty specification, so you can do the following:

```
\hsize1mm
1:foo{\hyphenpenalty 10000\discretionary{}{}{}}bar\par
2:foo\discretionary penalty 10000 {}{}{}bar\par
3:foo\discretionary{}{}{}bar\par
```

This results in:

1:foobar

2:foobar

3:foo
bar

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 `\setlanguage` command forces a word boundary).

All languages start out with `\prehyphenchar=`\-`, `\posthyphenchar=0`, `\preexhyphenchar=0` and `\postexhyphenchar=0`. When you assign the values of one of these four parameters, you are actually changing the settings for the current `\language`, this behaviour is compatible with `\patterns` and `\hyphenation`.

LuaTEX also hyphenates the first word in a paragraph. Words can be up to 256 characters long (up from 64 in TEX82). Longer words are ignored 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 behaviour cannot be controlled).

If you are using the Lua function `lang.hyphenate`, you should be aware that this function expects to receive a list of 'character' nodes. It will not operate properly in the presence of 'glyph', 'ligature', or 'ghost' nodes, nor does it know how to deal with kerning.

5.5 Applying ligatures and kerning

After all possible hyphenation points have been inserted in the list, LuaTEX will process the list to convert the 'character' nodes into 'glyph' and 'ligature' nodes. This is actually done in two stages: first all ligatures are processed, then all kerning information is applied to the result list. But those two stages are somewhat dependent on each other: If the used font makes it possible to do so, the ligaturing stage adds virtual 'character' nodes to the word boundaries in the list. While doing so, it removes and interprets `\noboundary` nodes. The kerning stage deletes those word boundary items after it is done with them, and it does the same for 'ghost' nodes. Finally, at the end of the kerning stage, all remaining 'character' nodes are converted to 'glyph' nodes.

This word 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 LuaTEX itself in order to make sure that the other, non-overruled, routine continues to function properly.

Although we could improve the situation the reality is that in modern OpenType fonts ligatures can be constructed in many ways: by replacing a sequence of characters by one glyph, or by selectively replacing individual glyphs, or by kerning, or any combination of this. Add to that contextual analysis and it will be clear that we have to let Lua do that job instead. The generic font handler that we provide (which is part of ConTEXt) distinguishes between base mode (which essentially is what we describe here and which delegates the task to TEX) and node mode (which deals with more complex fonts.

Let's look at an example. Take the word `office`, hyphenated `of-fice`, using a 'normal' font with all the `f-f` and `f-i` type ligatures:

initial	`{o}{f}{f}{i}{c}{e}`
after hyphenation	`{o}{f}{{-},{},{}}{f}{i}{c}{e}`
first ligature stage	`{o}{{f-},{f},{<ff>}}{i}{c}{e}`
final result	`{o}{{f-},{<fi>},{<ffi>}}{c}{e}`

That's bad enough, but let us assume that there is also a hyphenation point between the `f` and the `i`, to create `of-f-ice`. Then the final result should be:

```
{o}{{f-},
    {{f-},
     {i},
     {<fi>}},
    {{<ff>-},
     {i},
     {<ffi>}}}{c}{e}
```

with discretionaries in the post-break text as well as in the replacement text of the top-level discretionary that resulted from the first hyphenation point.

Here is that nested solution again, in a different representation:

	PRE		POST		REPLACE	
topdisc	f-	(1)		sub 1		sub 2
sub 1	f-	(2)	i	(3)	<fi>	(4)
sub 2	<ff>-	(5)	i	(6)	<ffi>	(7)

When line breaking is choosing its breakpoints, the following fields will eventually be selected:

```
of-f-ice    f-    (1)
            f-    (2)
             i    (3)
of-fice     f-    (1)
           <fi>   (4)
off-ice    <ff>-  (5)
             i    (6)
office     <ffi>  (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 `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:

FIELD	DESCRIPTION	
disc1.pre	f-	(1)
disc1.post	<fi>	(4)
disc1.replace	<ffi>	(7)
disc2.pre	f-	(2)
disc2.post	i	(3,6)
disc2.replace	<ff>-	(5)

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

```
{o}{{f-},
    {<fi>},
    {<ffi>}}
   {{f-},
    {i},
    {<ff>-}}{c}{e}
```

The two discretionaries have different subtypes from a discretionary appearing on its own: the first has subtype 4, and the second has subtype 5. The need for these special subtypes stems

from the fact that not all of the fields appear in their 'normal' location. The second discretionary especially looks odd, with things like the `<ff>-` appearing in `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.

Of course there is still that relationship with fonts: ligatures can be implemented by mapping a sequence of glyphs onto one glyph, but also by selective replacement and kerning. This means that the above examples are just representing the traditional approach.

5.6 Breaking paragraphs into lines

This code is 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, as patterns evolve and font handling can influence discretionaries, you need to be aware of the fact that long term consistency is not an engine matter only.

But that situation is now fairly common in LuaTEX, due to the changes to the ligaturing mechanism. And also, the LuaTEX discretionary nodes are implemented slightly different from the TEX82 nodes: the `no_break` text is now embedded inside the disc node, where previously these nodes kept their place in the horizontal list. In traditional TEX the discretionary node contains a counter indicating how many nodes to skip, but in LuaTEX we store the pre, post and replace text in the discretionary node.

The combined effect of these two differences is that LuaTEX does not always use all of the potential breakpoints in a paragraph, especially when fonts with many ligatures are used. Of course kerning also complicates matters here.

5.7 The `lang` library

This library provides the interface to LuaTEX's structure representing a language, and the associated functions.

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

This function creates a new userdata object. An object of type `<language>` is the first argument to most of the other functions in the `lang` library. These functions can also be used as if they were object methods, using the colon syntax. Without an argument, the next available internal id number will be assigned to this object. With argument, an object will be created that links to the internal language with that id number.

```
<number> n = lang.id(<language> l)
```

The number returned is the internal `\language` id number this object refers to.

```
<string> n = lang.hyphenation(<language> l)
```


```
lang.hyphenation(<language> l, <string> n)
```

This either returns the current hyphenation exceptions for this language, or adds new ones. The syntax of the string is explained in section 5.3.

```
lang.clear_hyphenation(<language> l)
```

This call clears the exception dictionary (string) for this language.

```
<string> n = lang.clean(<language> l, <string> o)
<string> n = lang.clean(<string> o)
```

This function creates a hyphenation key from the supplied hyphenation value. The syntax of the argument string is explained in section 5.3. This function is useful if you want to do something else based on the words in a dictionary file, like spell-checking.

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

This adds additional patterns for this language object, or returns the current set. The syntax of this string is explained in section 5.3.

```
lang.clear_patterns(<language> l)
```

This can be used to clear the pattern dictionary for a language.

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

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

These two are used to get or set the ‘pre-break’ and ‘post-break’ hyphen characters for implicit hyphenation in this language. The intial values are decimal 45 (hyphen) and decimal 0 (indicating emptiness).

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

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

These gets or set the ‘pre-break’ and ‘post-break’ hyphen characters for explicit hyphenation in this language. Both are initially decimal 0 (indicating emptiness).

```
<boolean> success = lang.hyphenate(<node> head)
<boolean> success = lang.hyphenate(<node> head, <node> tail)
```

This call 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 5.1 for more details.

The following two commands can be used to set or query hj codes:

```
lang.sethjcode(<language> l, <number> char, <number> usedchar)
<number> usedchar = lang.gethjcode(<language> l, <number> char)
```

When you set a hjcode the current sets get initialized unless the set was already initialized due to `\savinghyphcodes` being larger than zero.

6 Font structure

6.1 The font tables

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 'vf' means that this key will be created by the `font.read_vf()` routine, '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:

KEY	VF	TFM	USED	VALUE TYPE	DESCRIPTION
`name`	yes	yes	yes	string	metric (file) name
`area`	no	yes	yes	string	(directory) location, typically empty
`used`	no	yes	yes	boolean	indicates usage (initial: false)
`characters`	yes	yes	yes	table	the defined glyphs of this font
`checksum`	yes	yes	no	number	default: 0
`designsize`	no	yes	yes	number	expected size (default: 655360 == 10pt)
`direction`	no	yes	yes	number	default: 0
`encodingbytes`	no	no	yes	number	default: depends on `format`
`encodingname`	no	no	yes	string	encoding name
`fonts`	yes	no	yes	table	locally used fonts
`psname`	no	no	yes	string	This is the PostScript fontname in the incoming font source, and it's used as fontname identifier in the pdf output. This has to be a valid string, e.g. no spaces and such, as the backend will not do a cleanup. This gives complete control to the loader.
`fullname`	no	no	yes	string	output font name, used as a fallback in the pdf output if the `psname` is not set
`header`	yes	no	no	string	header comments, if any
`hyphenchar`	no	no	yes	number	default: TEX's `\hyphenchar`
`parameters`	no	yes	yes	hash	default: 7 parameters, all zero
`size`	no	yes	yes	number	the required scaling (by default the same as designsize)
`skewchar`	no	no	yes	number	default: TEX's `\skewchar`
`type`	yes	no	yes	string	basic type of this font
`format`	no	no	yes	string	disk format type
`embedding`	no	no	yes	string	pdf inclusion
`filename`	no	no	yes	string	the name of the font on disk
`tounicode`	no	yes	yes	number	When this is set to 1 LuaTEX assumes per-glyph tounicode entries are present in the font.

stretch	no	no	yes	number	the ‘stretch’ value from \expandglyphsinfont
shrink	no	no	yes	number	the ‘shrink’ value from \expandglyphsinfont
step	no	no	yes	number	the ‘step’ value from \expandglyphsinfont
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 where yes means: use a reference to the table that is passed to LuaTEX (this is the default), and no means: don’t store the table reference, don’t cache any Lua data for this font while renew means: 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 the font.
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.
oldmath	no	no	yes	boolean	This key flags a font as representing an old school TEX math font and disables the OpenType code path.
slant	no	no	yes	number	This parameter will tilt the font and does the same as SlantFont in the map file for Type1 fonts.
extend	no	no	yes	number	This parameter will scale the font horizontally and does the same as ExtendFont in the map file for Type1 fonts.
squeeze	no	no	yes	number	This parameter will scale the font vertically and has no equivalent in the map file.
width	no	no	yes	number	The backend will inject pdf operators that set the penwidth. The value is (as usual in TEX) divided by 1000. It works with the mode file.
mode	no	no	yes	number	The backend will inject pdf operators that relate to the drawing mode with 0 being a fill, 1 being an outline, 2 both draw and fill and 3 no painting at all.

The saved reference in the cache option 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.

The key name is always required. The keys stretch, shrink, step only have meaning when used

together: they can be used to replace a post-loading `\expandglyphsinfont` command. The `auto_expand` option is not supported in LuaTEX. In fact, the primitives that create expanded or protruding copies are probably only useful when used with traditional fonts because all these extra OpenType properties are kept out of the picture. 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.

Because we store the actual state of expansion with each glyph and don't have special font instances, we can change some font related parameters before lines are constructed, like:

```
font.setexpansion(font.current(),100,100,20)
```

This is mostly meant for experiments (or an optimizing routing written in Lua) so there is no primitive.

The key `attributes` can be used to set font attributes in the pdf file. 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:

#	DIR	#	DIR	#	DIR	#	DIR
0	LT	4	RT	8	TT	12	BT
1	LL	5	RL	9	TL	13	BL
2	LB	6	RB	10	TB	14	BB
3	LR	7	RR	11	TR	15	BR

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. Keep in mind that LuaTEX has a bit different directional model so these values are not used for anything.

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:

NAME	REMAPPING
slant	1
space	2
space_stretch	3
space_shrink	4
x_height	5
quad	6
extra_space	7

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).

Each character hash itself is a hash. For example, here is the character 'f' (decimal 102) in the font `cmr10 at 10pt`. The numbers that represent dimensions are in scaled points.

```
[102] = {
    ["width"]  = 200250,
    ["height"] = 455111,
    ["depth"]  = 0,
    ["italic"] = 50973,
    ["kerns"]  = {
        [63] = 50973,
        [93] = 50973,
        [39] = 50973,
        [33] = 50973,
        [41] = 50973
    },
    ["ligatures"] = {
        [102] = { ["char"] = 11, ["type"] = 0 },
        [108] = { ["char"] = 13, ["type"] = 0 },
        [105] = { ["char"] = 12, ["type"] = 0 }
    }
}
```

The following top-level keys can be present inside a character hash:

KEY	VF	TFM	USED	TYPE	DESCRIPTION
`width`	yes	yes	yes	number	character's width, in sp (default 0)
`height`	no	yes	yes	number	character's height, in sp (default 0)
`depth`	no	yes	yes	number	character's depth, in sp (default 0)
`italic`	no	yes	yes	number	character's italic correction, in sp (default zero)
`top_accent`	no	no	maybe	number	character's top accent alignment place, in sp (default zero)
`bot_accent`	no	no	maybe	number	character's bottom accent alignment place, in sp (default zero)
`left_protruding`	no	no	maybe	number	character's `\lpcode`
`right_protruding`	no	no	maybe	number	character's `\rpcode`
`expansion_factor`	no	no	maybe	number	character's `\efcode`
`tounicode`	no	no	maybe	string	character's Unicode equivalent(s), in utf-16BE hexadecimal format
`next`	no	yes	yes	number	the 'next larger' character index
`extensible`	no	yes	yes	table	the constituent parts of an extensible recipe
`vert_variants`	no	no	yes	table	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	string	the character (PostScript) name
index	no	no	yes	number	the (OpenType or TrueType) font glyph index
used	no	yes	yes	boolean	typeset already (default: false)
mathkern	no	no	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 page 95 in this manual for details. The values of left_protruding and right_protruding are used only when \protrudechars 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 \adjustspacing.

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. The string format is exactly the format that is expected by Adobe CMap files (utf-16BE in hexadecimal encoding), minus the enclosing angle brackets. For instance the tounicode for a fi ligature would be 00660069. When you pass a number the conversion will be done for you.

A math character can have a next field that points to a next larger shape. However, the presence of extensible will overrule next, if that is also present. The extensible field in turn can be overruled by vert_variants, the OpenType version. 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	The maximum overlap at the starting side (in scaled points).
end	number	The maximum overlap at the ending side (in scaled points).
advance	number	The total advance width of this item. It 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 of 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	`|=:>`	`l|n`
`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.

6.2 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` are:

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. This is true for traditional TEX fonts that communicate via tfm files.
- The type of the disk font file, for instance a bitmap file or an outline Type1, TrueType or OpenType font.
- The level of embedding requested, although in most cases a subset of characters is embedded. The times when nothing got embedded are (in our opinion at least) basically gone.

A font that uses anything other than an 8-bit encoding vector has to be written to the pdf in a different way. When the font table has `encodingbytes` set to 2, then it 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 falls back to the mapfile-based solution used by pdfTEX and dvips, so that legacy fonts are supported transparently. If a 'wide' font is used, 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. These extra fields are: `format`, `embedding`, `fullname`, `cidinfo` (as explained above), `filename`, and the `index` key in the separate characters.

The `format` variable can have the following values. `type3` fonts are provided for backward compatibility only, and do not support the new wide encoding options.

VALUE	DESCRIPTION
`type1`	this is a PostScript Type1 font
`type3`	this is a bitmapped (pk) font
`truetype`	this is a TrueType or TrueType-based OpenType font
`opentype`	this is a PostScript-based OpenType font

Valid values for the `embedding` variable are:

VALUE	DESCRIPTION
`no`	don't embed the font at all
`subset`	include and atttempt to subset the font
`full`	include this font in its entirety

The other fields are used as follows. The `fullname` will be the PostScript/pdf font name. The `cidinfo` will be used as the character set: the CID `/Ordering` and `/Registry` keys. The `filename` points to the actual font file. If you include the full path in the `filename` or if the file is in the local directory, LuaTEX will run a little bit more efficient because it will not have to re-run the `find_*_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 to a more unique one.

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. One way to get the correct `index` numbers for Type1 fonts is by loading the font via `fontloader.open` and use the table indices as `index` fields.

In order to make sure that cut and paste of the final document works okay you can best make sure that there is a `tounicode` vector enforced. Not all pdf viewers handle this right so take Acrobat as reference.

6.3 Virtual fonts

6.3.1 The structure

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`. In most cases it's optional because we look at the `commands` entry anyway.
- Make sure there is at least one valid entry in `fonts` (see below), although recent versions of LuaTEX add a default entry when this table is missing.
- Add a `commands` array to those characters that matter. A virtual character can itself point to virtual characters but be careful with nesting as you can create loops and overflow the stack (which often indicates an error anyway).

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. This also works 'in reverse': if you are absolutely certain that a font is not a virtual font, assigning the value `real` to `type` will inhibit LuaTEX from looking for a virtual font file, thereby saving you a disk search. This only matters when we load a tfm file.

The `fonts` is an (indexed) Lua table. 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. So, a table looks like this:

```
fonts = {
  { name = "ptmr8a", size = 655360 },
  { name = "psyr", size = 600000 },
  { id = 38 }
}
```

The first referenced font (at 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 a previously defined font that is known to LuaTEX as font id 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:

COMMAND	ARGUMENTS	TYPE	DESCRIPTION
`font`	1	number	select a new font from the local `fonts` table
`char`	1	number	typeset this character number from the current font, and move right by the character's width
`node`	1	node	output this node (list), and move right by the width of this list

slot	2	2 numbers	a shortcut for the combination of a font and char command
push	0		save current position
nop	0		do nothing
pop	0		pop position
rule	2	2 numbers	output a rule $ht * wd$, and move right.
down	1	number	move down on the page
right	1	number	move right on the page
special	1	string	output a \special command
pdf	2	2 strings	output a pdf literal, the first string is one of origin, page, text, font, direct or raw; if you have one string only origin is assumed
lua	1	string, function	execute a Lua script when the glyph is embedded; in case of a function it gets the font id and character code passed
image	1	image	output an image (the argument can be either an <image> variable or an image_spec table)
comment	any	any	the arguments of this command are ignored

When a font id is set to 0 then it will be replaced by the currently assigned font id. This prevents the need for hackery with future id's. Normally one could use font.nextid but when more complex fonts are built in the meantime other instances could have been loaded.

The pdf option also accepts a mode keyword in which case the third argument sets the mode. That option will change the mode in an efficient way (passing an empty string would result in an extra empty lines in the pdf file. This option only makes sense for virtual fonts. The font mode only makes sense in virtual fonts. Modes are somewhat fuzzy and partially inherited from pdfTEX.

MODE	DESCRIPTION
origin	enter page mode and set the position
page	enter page mode
text	enter text mode
font	enter font mode (kind of text mode, only in virtual fonts)
always	finish the current string and force a transform if needed
raw	finish the current string

You always need to check what pdf code is generated because there can be all kind of interferences with optimization in the backend and fonts are complicated anyway. Here is a rather elaborate glyph commands example using such keys:

```
...
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')
 -- { "slot", 2, 97 },           -- an alternative for the previous two
```


```
    { "pop" },                      -- go all the way back
    { "down", -200000 },            -- move upwards by about 3pt
    { "special", "pdf: 1 0 0 rg" }  -- switch to red color
 -- { "pdf", "origin", "1 0 0 rg" } -- switch to red color (alternative)
    { "rule", 500000, 20000 }       -- draw a bar
    { "special", "pdf: 0 g" }       -- back to black
 -- { "pdf", "origin", "0 g" }      -- back to black (alternative)
}
...
```

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 created 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.

The special can have a `pdf:`, `pdf:origin:`, `pdf:page:`, `pdf:direct:` or `pdf:raw:` prefix. When you have to concatenate strings using the `pdf` command might be more efficient.

6.3.2 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).

6.3.3 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
        local 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" },
              }
          end
        end
        return f
      else
        return font.read_tfm(name,size)
      end
    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
```

6.4 The vf library

The `vf` library can be used when Lua code, as defined in the `commands` of the font, is executed. The functions provided are similar as the commands: `char`, `down`, `fontid`, `image`, `node`, `nop`, `pop`, `push`, `right`, `rule`, `special` and `pdf`. This library has been present for a while but not been advertised and tested much, if only because it's easy to define an invalid font (or mess up the pdf stream). Keep in mind that the Lua snippets are executed each time when a character is output.

6.5 The font library

The font library provides the interface into the internals of the font system, and it also 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.

6.5.1 Loading a TFM file

The behaviour 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.

6.5.2 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.

```
<table> vf_fnt =
    font.read_vf(<string> name, <number> s)
```

The meaning of the number `s` and the format of the returned table are similar to the ones in the `read_tfm` function.

6.5.3 The fonts array

The whole table of TEX fonts is accessible from Lua using a virtual array.

```
font.fonts[n] = { ... }
<table> f = font.fonts[n]
```

Because this is a virtual array, you cannot call `pairs` on it, but see below for the `font.each` iterator.

The two metatable functions implementing the virtual array are:

```
<table> f = font.getfont(<number> n)
font.setfont(<number> n, <table> f)
```

Note that at the moment, each access to the `font.fonts` or call to `font.getfont` creates a Lua table for the whole font. This process can be quite slow. In a later version of 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 `font.fonts` quite a lot, a less stringent ruleset will likely be implemented later.

6.5.4 Checking a font’s status

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

```
<boolean> f =
    font.frozen(<number> n)
```

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

6.5.5 Defining a font directly

You can define your own font into `font.fonts` by calling this function:

```
<number> i =
    font.define(<table> f)
```

The return value is the internal id number of the defined font (the index into `font.fonts`). If the font creation fails, an error is raised. The table is a font structure. An alternative call is:

```
<number> i =
    font.define(<number> n, <table> f)
```

Where the first argument is a reserved font id (see below).

6.5.6 Extending a font

Within reasonable bounds you can extend a font after it has been defined. Because some properties are best left unchanged this is limited to adding characters.

```
font.addcharacters(<number n>, <table> f)
```

The table passed can have the fields `characters` which is a (sub)table like the one used in define, and for virtual fonts a `fonts` table can be added. The characters defined in the `characters` table are added (when not yet present) or replace an existing entry. Keep in mind that replacing can have side effects because a character already can have been used. Instead of posing restrictions we expect the user to be careful. (The `setfont` helper is a more drastic replacer.)

6.5.7 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. If you pass `true` as argument, the id gets reserved and you can pass to `font.define` as first argument. This can be handy when you create complex virtual fonts.

```
<number> i =
    font.nextid(true)
```

6.5.8 Font ids

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

This returns the font id associated with `csname`, or -1 if `csname` is not defined.

```
<number> i =
    font.max()
```

This is the largest used index in `font.fonts`.

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

This gets or sets the currently used font number.

6.5.9 Iterating over all fonts

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

This is an iterator over each of the defined 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.

7 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 or considered in the past that are now added to the engine.

7.1 Math styles

7.1.1 `\mathstyle`

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: `\mathstyle`. This is a 'convert command' like e.g. `\romannumeral`: its value can only be read, not set.

The returned value is between 0 and 7 (in math mode), or −1 (all other modes). For easy testing, the eight math style commands have been altered so that they can be used as numeric values, so you can write code like this:

```
\ifnum\mathstyle=\textstyle
    \message{normal text style}
\else \ifnum\mathstyle=\crampedtextstyle
    \message{cramped text style}
\fi \fi
```

Sometimes you won't get what you expect so a bit of explanation might help to understand what happens. When math is parsed and expanded it gets turned into a linked list. In a second pass the formula will be build. This has to do with the fact that in order to determine the automatically chosen sizes (in for instance fractions) following content can influence preceding sizes. A side effect of this is for instance that one cannot change the definition of a font family (and thereby reusing numbers) because the number that got used is stored and used in the second pass (so changing `\fam 12` mid-formula spoils over to preceding use of that family).

The style switching primitives like `\textstyle` are turned into nodes so the styles set there are frozen. The `\mathchoice` primitive results in four lists being constructed of which one is used in the second pass. The fact that some automatic styles are not yet known also means that the `\mathstyle` primitive expands to the current style which can of course be different from the one really used. It's a snapshot of the first pass state. As a consequence in the following example you get a style number (first pass) typeset that can actually differ from the used style (second pass). In the case of a math choice used ungrouped, the chosen style is used after the choice too, unless you group.

```
[a:\mathstyle]\quad
```


```
\bgroup
\mathchoice
    {\bf \scriptstyle       (x:d :\mathstyle)}
    {\bf \scriptscriptstyle (x:t :\mathstyle)}
    {\bf \scriptscriptstyle (x:s :\mathstyle)}
    {\bf \scriptscriptstyle (x:ss:\mathstyle)}
\egroup
\quad[b:\mathstyle]\quad
\mathchoice
    {\bf \scriptstyle       (y:d :\mathstyle)}
    {\bf \scriptscriptstyle (y:t :\mathstyle)}
    {\bf \scriptscriptstyle (y:s :\mathstyle)}
    {\bf \scriptscriptstyle (y:ss:\mathstyle)}
\quad[c:\mathstyle]\quad
\bgroup
\mathchoice
    {\bf \scriptstyle       (z:d :\mathstyle)}
    {\bf \scriptscriptstyle (z:t :\mathstyle)}
    {\bf \scriptscriptstyle (z:s :\mathstyle)}
    {\bf \scriptscriptstyle (z:ss:\mathstyle)}
\egroup
\quad[d:\mathstyle]
```

This gives:

$[a : 0]$ $(\mathbf{x}:\mathbf{d}:\mathbf{4})$ $[b : 0]$ $(\mathbf{y}:\mathbf{d}:\mathbf{4})$ $[c:0]$ $(\mathbf{z}:\mathbf{s}:\mathbf{6})$ $[d:0]$

$[a : 2]$ $(\mathbf{x}:\mathbf{t}:\mathbf{6})$ $[b : 2]$ $(\mathbf{y}:\mathbf{t}:\mathbf{6})$ $[c:2]$ $(\mathbf{z}:\mathbf{ss}:\mathbf{6})$ $[d:2]$

Using `\begingroup` ... `\endgroup` instead gives:

$[a : 0]$ $(\mathbf{x}:\mathbf{d}:\mathbf{4})$ $[b:0]$ $(\mathbf{y}:\mathbf{s}:\mathbf{6})$ $[c:0]$ $(\mathbf{z}:\mathbf{ss}:\mathbf{6})$ $[d:0]$

$[a : 2]$ $(\mathbf{x}:\mathbf{t}:\mathbf{6})$ $[b:2]$ $(\mathbf{y}:\mathbf{ss}:\mathbf{6})$ $[c:2]$ $(\mathbf{z}:\mathbf{ss}:\mathbf{6})$ $[d:2]$

This might look wrong but it's just a side effect of `\mathstyle` expanding to the current (first pass) style and the number being injected in the list that gets converted in the second pass. It all makes sense and it illustrates the importance of grouping. In fact, the math choice style being effective afterwards has advantages. It would be hard to get it otherwise.

7.1.2 \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 (`\over`, `\atop`, `\overwithdelims`, `\atopwithdelims`) would therefore normally return wrong values for `\mathstyle`. To fix this, LuaTEX introduces a special prefix command: `\Ustack`:

```
$\Ustack {a \over b}$
```

The `\Ustack` command will scan the next brace and start a new math group with the correct (numerator) math style.

7.2 Unicode math characters

Character handling is now extended up to the full Unicode range (the `\U` prefix), which is compatible with XeTeX.

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.

The input for such primitives would look like this:

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

The altered TeX82 primitives are:

PRIMITIVE	MIN	MAX		MIN	MAX
`\mathcode`	0	10FFFF	=	0	8000
`\delcode`	0	10FFFF	=	0	FFFFFF

The unaltered ones are:

PRIMITIVE	MIN	MAX
`\mathchardef`	0	8000
`\mathchar`	0	7FFF
`\mathaccent`	0	7FFF
`\delimiter`	0	7FFFFFF
`\radical`	0	7FFFFFF

For practical reasons `\mathchardef` will silently accept values larger that `0x8000` and interpret it as `\Umathcharnumdef`. This is needed to satisfy older macro packages.

The following new primitives are compatible with XeTeX:

PRIMITIVE	MIN	MAX		MIN	MAX
`\Umathchardef`	0+0+0	7+FF+10FFFF			
`\Umathcharnumdef`[5]	-80000000	7FFFFFFF			
`\Umathcode`	0	10FFFF	=	0+0+0	7+FF+10FFFF
`\Udelcode`	0	10FFFF	=	0+0	FF+10FFFF
`\Umathchar`	0+0+0	7+FF+10FFFF			
`\Umathaccent`	0+0+0	7+FF+10FFFF			
`\Udelimiter`	0+0+0	7+FF+10FFFF			
`\Uradical`	0+0	FF+10FFFF			
`\Umathcharnum`	-80000000	7FFFFFFF			

\Umathcodenum	0	10FFFF	=	-80000000	7FFFFFFF
\Udelcodenum	0	10FFFF	=	-80000000	7FFFFFFF

Specifications typically look like:

```
\Umathchardef\xx="1"0"456
\Umathcode   123="1"0"789
```

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 font via the `\Umathoperatorsize` parameter.

For some of these primitives, all information is packed into a single signed integer. For the first two (`\Umathcharnum` and `\Umathcodenum`), the lowest 21 bits are the character code, the 3 bits above that represent the math class, and the family data is kept in the topmost bits. This means that the values for math families 128–255 are actually negative. For `\Udelcodenum` there is no math class. The math family information is stored in the bits directly on top of the character code. Using these three commands is not as natural as using the two- and three-value commands, so unless you know exactly what you are doing and absolutely require the speedup resulting from the faster input scanning, it is better to use the verbose commands instead.

The `\Umathaccent` command accepts optional keywords to control various details regarding math accents. See section 7.12 below for details.

There are more new primitives and all of these will be explained in following sections:

PRIMITIVE	VALUE RANGE (IN HEX)
\Uroot	0 + 0–FF + 10FFFF
\Uoverdelimiter	0 + 0–FF + 10FFFF
\Uunderdelimiter	0 + 0–FF + 10FFFF
\Udelimiterover	0 + 0–FF + 10FFFF
\Udelimiterunder	0 + 0–FF + 10FFFF

7.3 Cramped math styles

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.

In Eijkhouts "TEX by Topic" the rules for handling styles in scripts are described as follows:

- In any style superscripts and subscripts are taken from the next smaller style. Exception: in display style they are in script style.
- Subscripts are always in the cramped variant of the style; superscripts are only cramped if the original style was cramped.

- In an `..\over..` formula in any style the numerator and denominator are taken from the next smaller style.
- The denominator is always in cramped style; the numerator is only in cramped style if the original style was cramped.
- Formulas under a `\sqrt` or `\overline` are in cramped style.

In LuaTEX one can set the styles in more detail which means that you sometimes have to set both normal and cramped styles to get the effect you want. (Even) if we force styles in the script using `\scriptstyle` and `\crampedscriptstyle` we get this:

STYLE	EXAMPLE
default	$b^{x=x\,x}_{x=x\,x}$
script	$b^{x=x\,x}_{x=x\,x}$
crampedscript	$b^{x=x\,x}_{x=x\,x}$

Now we set the following parameters

```
\Umathordrelspacing\scriptstyle=30mu
\Umathordordspacing\scriptstyle=30mu
```

This gives a different result:

STYLE	EXAMPLE
default	$b^{x\quad=x\quad x}_{x=x\,x}$
script	$b^{x\quad=x\quad x}_{x\quad=x\quad x}$
crampedscript	$b^{x=x\,x}_{x=x\,x}$

But, as this is not what is expected (visually) we should say:

```
\Umathordrelspacing\scriptstyle=30mu
\Umathordordspacing\scriptstyle=30mu
\Umathordrelspacing\crampedscriptstyle=30mu
\Umathordordspacing\crampedscriptstyle=30mu
```

Now we get:

STYLE	EXAMPLE
default	$b^{x\quad=x\quad x}_{x\quad=x\quad x}$
script	$b^{x\quad=x\quad x}_{x\quad=x\quad x}$
crampedscript	$b^{x\quad=x\quad x}_{x\quad=x\quad x}$

7.4 Math parameter settings

In LuaTEX, the font dimension parameters that TEX used in math typesetting are now accessible via primitive commands. In fact, refactoring of the math engine has resulted in many more parameters than were not accessible before.

PRIMITIVE NAME	DESCRIPTION
\Umathquad	the width of 18 mu's
\Umathaxis	height of the vertical center axis of the math formula above the baseline
\Umathoperatorsize	minimum size of large operators in display mode
\Umathoverbarkern	vertical clearance above the rule
\Umathoverbarrule	the width of the rule
\Umathoverbarvgap	vertical clearance below the rule
\Umathunderbarkern	vertical clearance below the rule
\Umathunderbarrule	the width of the rule
\Umathunderbarvgap	vertical clearance above the rule
\Umathradicalkern	vertical clearance above the rule
\Umathradicalrule	the width of the rule
\Umathradicalvgap	vertical clearance below the rule
\Umathradicaldegreebefore	the forward kern that takes place before placement of the radical degree
\Umathradicaldegreeafter	the backward kern that takes place after placement of the radical degree
\Umathradicaldegreeraise	this is the percentage of the total height and depth of the radical sign that the degree is raised by; it is expressed in `percents`, so 60% is expressed as the integer 60
\Umathstackvgap	vertical clearance between the two elements in a `\atop` stack
\Umathstacknumup	numerator shift upward in `\atop` stack
\Umathstackdenomdown	denominator shift downward in `\atop` stack
\Umathfractionrule	the width of the rule in a `\over`
\Umathfractionnumvgap	vertical clearance between the numerator and the rule
\Umathfractionnumup	numerator shift upward in `\over`
\Umathfractiondenomvgap	vertical clearance between the denominator and the rule
\Umathfractiondenomdown	denominator shift downward in `\over`
\Umathfractiondelsize	minimum delimiter size for `\...withdelims`
\Umathlimitabovevgap	vertical clearance for limits above operators
\Umathlimitabovebgap	vertical baseline clearance for limits above operators
\Umathlimitabovekern	space reserved at the top of the limit
\Umathlimitbelowvgap	vertical clearance for limits below operators
\Umathlimitbelowbgap	vertical baseline clearance for limits below operators
\Umathlimitbelowkern	space reserved at the bottom of the limit
\Umathoverdelimitervgap	vertical clearance for limits above delimiters
\Umathoverdelimiterbgap	vertical baseline clearance for limits above delimiters
\Umathunderdelimitervgap	vertical clearance for limits below delimiters
\Umathunderdelimiterbgap	vertical baseline clearance for limits below delimiters
\Umathsubshiftdrop	subscript drop for boxes and subformulas
\Umathsubshiftdown	subscript drop for characters
\Umathsupshiftdrop	superscript drop (raise, actually) for boxes and subformulas
\Umathsupshiftup	superscript raise for characters
\Umathsubsupshiftdown	subscript drop in the presence of a superscript

`\Umathsubtopmax`	the top of standalone subscripts cannot be higher than this above the baseline
`\Umathsupbottommin`	the bottom of standalone superscripts cannot be less than this above the baseline
`\Umathsupsubbottommax`	the bottom of the superscript of a combined super- and subscript be at least as high as this above the baseline
`\Umathsubsupvgap`	vertical clearance between super- and subscript
`\Umathspaceafterscript`	additional space added after a super- or subscript
`\Umathconnectoroverlapmin`	minimum overlap between parts in an extensible recipe

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

```
\Umathquad\displaystyle=1em
```

they obey grouping, and you can use `\the\Umathquad\displaystyle` if needed.

7.5 Skips around display math

The injection of `\abovedisplayskip` and `\belowdisplayskip` is not symmetrical. An above one is always inserted, also when zero, but the below is only inserted when larger than zero. Especially the latter makes it sometimes hard to fully control spacing. Therefore LuaTEX comes with a new directive: `\mathdisplayskipmode`. The following values apply:

VALUE	MEANING
0	normal TEX behaviour
1	always (same as 0)
2	only when not zero
3	never, not even when not zero

7.6 Font-based Math Parameters

While it is nice to have these math parameters available for tweaking, it would be tedious to have to set each of them by hand. For this reason, LuaTEX initializes a bunch of these parameters whenever you assign a font identifier to a math family based on either the traditional math font dimensions in the font (for assignments to math family 2 and 3 using tfm-based fonts like `cmsy` and `cmex`), or based on the named values in a potential `MathConstants` table when the font is loaded via Lua. If there is a `MathConstants` table, this takes precedence over font dimensions, and in that case no attention is paid to which family is being assigned to: the `MathConstants` tables in the last assigned family sets all parameters.

In the table below, the one-letter style abbreviations and symbolic tfm font dimension names match those used in the TEXbook. Assignments to `\textfont` set the values for the cramped and uncramped display and text styles, `\scriptfont` sets the script styles, and `\scriptscriptfont` sets the scriptscript styles, so we have eight parameters for three font sizes. In the tfm case, assignments only happen in family 2 and family 3 (and of course only for the parameters for which there are font dimensions).

Besides the parameters below, LuaTEX also looks at the ‘space’ font dimension parameter. For math fonts, this should be set to zero.

VARIABLE / STYLE	TFM / OPENTYPE
\Umathaxis	axis_height AxisHeight
[6] **\Umathoperatorsize** D, D’	– DisplayOperatorMinHeight
[9] **\Umathfractiondelsize** D, D’	delim1 FractionDelimiterDisplayStyleSize
[9] **\Umathfractiondelsize** T, T’, S, S’, SS, SS’	delim2 FractionDelimiterSize
\Umathfractiondenomdown D, D’	denom1 FractionDenominatorDisplayStyleShiftDown
\Umathfractiondenomdown T, T’, S, S’, SS, SS’	denom2 FractionDenominatorShiftDown
\Umathfractiondenomvgap D, D’	3*default_rule_thickness FractionDenominatorDisplayStyleGapMin
\Umathfractiondenomvgap T, T’, S, S’, SS, SS’	default_rule_thickness FractionDenominatorGapMin
\Umathfractionnumup D, D’	num1 FractionNumeratorDisplayStyleShiftUp
\Umathfractionnumup T, T’, S, S’, SS, SS’	num2 FractionNumeratorShiftUp
\Umathfractionnumvgap D, D’	3*default_rule_thickness FractionNumeratorDisplayStyleGapMin
\Umathfractionnumvgap T, T’, S, S’, SS, SS’	default_rule_thickness FractionNumeratorGapMin
\Umathfractionrule	default_rule_thickness FractionRuleThickness
\Umathskewedfractionhgap	math_quad/2 SkewedFractionHorizontalGap
\Umathskewedfractionvgap	math_x_height SkewedFractionVerticalGap
\Umathlimitabovebgap	big_op_spacing3 UpperLimitBaselineRiseMin
[1] **\Umathlimitabovekern**	big_op_spacing5 0
\Umathlimitabovevgap	big_op_spacing1 UpperLimitGapMin
\Umathlimitbelowbgap	big_op_spacing4 LowerLimitBaselineDropMin
[1] **\Umathlimitbelowkern**	big_op_spacing5

		0
	\Umathlimitbelowvgap	big_op_spacing2 LowerLimitGapMin
	\Umathoverdelimitervgap	big_op_spacing1 StretchStackGapBelowMin
	\Umathoverdelimiterbgap	big_op_spacing3 StretchStackTopShiftUp
	\Umathunderdelimitervgap	big_op_spacing2 StretchStackGapAboveMin
	\Umathunderdelimiterbgap	big_op_spacing4 StretchStackBottomShiftDown
	\Umathoverbarkern	default_rule_thickness OverbarExtraAscender
	\Umathoverbarrule	default_rule_thickness OverbarRuleThickness
	\Umathoverbarvgap	3*default_rule_thickness OverbarVerticalGap
[1]	**\Umathquad**	math_quad <font_size(f)>
	\Umathradicalkern	default_rule_thickness RadicalExtraAscender
[2]	**\Umathradicalrule**	<not set> RadicalRuleThickness
[3]	**\Umathradicalvgap**	default_rule_thickness+abs(math_x_height)/4
	D, D’	RadicalDisplayStyleVerticalGap
[3]	**\Umathradicalvgap**	default_rule_thickness+abs(default_rule_thickness)/4
	T, T’, S, S’, SS, SS’	RadicalVerticalGap
[2]	**\Umathradicaldegreebefore**	<not set> RadicalKernBeforeDegree
[2]	**\Umathradicaldegreeafter**	<not set> RadicalKernAfterDegree
[2,7]	**\Umathradicaldegreeraise**	<not set> RadicalDegreeBottomRaisePercent
[4]	**\Umathspaceafterscript**	script_space SpaceAfterScript
	\Umathstackdenomdown	denom1
	D, D’	StackBottomDisplayStyleShiftDown
	\Umathstackdenomdown	denom2
	T, T’, S, S’, SS, SS’	StackBottomShiftDown
	\Umathstacknumup	num1
	D, D’	StackTopDisplayStyleShiftUp
	\Umathstacknumup	num3

	T, T', S, S', SS, SS'	StackTopShiftUp
	\Umathstackvgap	7*default_rule_thickness
	D, D'	StackDisplayStyleGapMin
	\Umathstackvgap	3*default_rule_thickness
	T, T', S, S', SS, SS'	StackGapMin
	\Umathsubshiftdown	sub1
		SubscriptShiftDown
	\Umathsubshiftdrop	sub_drop
		SubscriptBaselineDropMin
[8]	**\Umathsubsupshiftdown**	–
		SubscriptShiftDownWithSuperscript
	\Umathsubtopmax	abs(math_x_height*4)/5
		SubscriptTopMax
	\Umathsubsupvgap	4*default_rule_thickness
		SubSuperscriptGapMin
	\Umathsupbottommin	abs(math_x_height/4)
		SuperscriptBottomMin
	\Umathsupshiftdrop	sup_drop
		SuperscriptBaselineDropMax
	\Umathsupshiftup	sup1
	D	SuperscriptShiftUp
	\Umathsupshiftup	sup2
	T, S, SS,	SuperscriptShiftUp
	\Umathsupshiftup	sup3
	D', T', S', SS'	SuperscriptShiftUpCramped
	\Umathsupsubbottommax	abs(math_x_height*4)/5
		SuperscriptBottomMaxWithSubscript
	\Umathunderbarkern	default_rule_thickness
		UnderbarExtraDescender
	\Umathunderbarrule	default_rule_thickness
		UnderbarRuleThickness
	\Umathunderbarvgap	3*default_rule_thickness
		UnderbarVerticalGap
[5]	**\Umathconnectoroverlapmin**	0
		MinConnectorOverlap

Note 1: OpenType fonts set `\Umathlimitabovekern` and `\Umathlimitbelowkern` to zero and set `\Umathquad` to the font size of the used font, because these are not supported in the `MATH` table,

Note 2: Traditional tfm fonts do not set `\Umathradicalrule` because TEX82 uses the height of the radical instead. When this parameter is indeed not set when LuaTEX 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 `\Umathradicaldegreebefore`, `\Umathradicaldegreeafter`, and `\Umathradicaldegreeraise`. These are then automatically initialized to 5/18quad, −10/18quad, and 60.

Note 3: If tfm fonts are used, then the `\Umathradicalvgap` is not set until the first time LuaTEX has to typeset a formula because this needs parameters from both family 2 and family 3. This provides a partial backward compatibility with TEX82, but that compatibility is only partial: once the `\Umathradicalvgap` is set, it will not be recalculated any more.

Note 4: When tfm fonts are used a similar situation arises with respect to `\Umathspaceafterscript`: it is not set until the first time LuaTEX has to typeset a formula. This provides some backward compatibility with TEX82. But once the `\Umathspaceafterscript` is set, `\scriptspace` will never be looked at again.

Note 5: Traditional tfm fonts set `\Umathconnectoroverlapmin` to zero because TEX82 always stacks extensibles without any overlap.

Note 6: The `\Umathoperatorsize` is only used in `\displaystyle`, and is only set in OpenType fonts. In tfm font mode, it is artificially set to one scaled point more than the initial attempt's size, so that always the 'first next' will be tried, just like in TEX82.

Note 7: The `\Umathradicaldegreeraise` is a special case because it is the only parameter that is expressed in a percentage instead of a number of scaled points.

Note 8: `SubscriptShiftDownWithSuperscript` does not actually exist in the 'standard' OpenType math font Cambria, but it is useful enough to be added.

Note 9: `FractionDelimiterDisplayStyleSize` and `FractionDelimiterSize` do not actually exist in the 'standard' OpenType math font Cambria, but were useful enough to be added.

7.7 Nolimit correction

There are two extra math parameters `\Umathnolimitsupfactor` and `\Umathnolimitsubfactor` that were added to provide some control over how limits are spaced (for example the position of super and subscripts after integral operators). They relate to an extra parameter `\mathnolimitsmode`. The half corrections are what happens when scripts are placed above and below. The problem with italic corrections is that officially that correction italic is used for above/below placement while advanced kerns are used for placement at the right end. The question is: how often is this implemented, and if so, do the kerns assume correction too. Anyway, with this parameter one can control it.

	$\int_1^0$	$\int_1^0$	$\int_1^0$	$\int_1^0$	$\int_1^0$	${}_1\int^0$
mode	`0`	`1`	`2`	`3`	`4`	`8000`
superscript	0	font	0	0	+ic/2	0
subscript	-ic	font	0	-ic/2	-ic/2	8000ic/1000

When the mode is set to one, the math parameters are used. This way a macro package writer can decide what looks best. Given the current state of fonts in ConTEXt we currently use mode 1 with factor 0 for the superscript and 750 for the subscripts. Positive values are used for both parameters but the subscript shifts to the left. A `\mathnolimitsmode` larger that 15 is considered to be a factor for the subscript correction. This feature can be handy when experimenting.

7.8 Math italic mess

The \mathitalicsmode parameter can be set to 1 to force italic correction before noads that represent some more complex structure (read: everything that is not an ord, bin, rel, open, close, punct or inner). We show a Cambria example.

\mathitalicsmode = 0 $|T^1|$ $|T|$ $T+1$ $T_{\frac{1}{2}}$ $T\sqrt{1}$
\mathitalicsmode = 1 $|T^1|$ $|T|$ $T+1$ $T_{\frac{1}{2}}$ $T\sqrt{1}$

This kind of parameters relate to the fact that italic correction in OpenType math is bound to fuzzy rules. So, control is the solution.

7.9 Script and kerning

If you want to typeset text in math macro packages often provide something \text which obeys the script sizes. As the definition can be anything there is a good chance that the kerning doesn't come out well when used in a script. Given that the first glyph ends up in a \hbox we have some control over this. And, as a bonus we also added control over the normal sublist kerning. The \mathscriptboxmode parameter defaults to 1.

VALUE	MEANING
0	forget about kerning
1	kern math sub lists with a valid glyph
2	also kern math sub boxes that have a valid glyph
2	only kern math sub boxes with a boundary node present

Here we show some examples. Of course this doesn't solve all our problems, if only because some fonts have characters with bounding boxes that compensate for italics, while other fonts can lack kerns.

	$T_{\tf fluff}$	$T_{\tf fluff}$	T_{fluff}	T_{fluff}	$T_{\text{\boundary1 fluff}}$
	mode 0	**mode 1**	**mode 1**	**mode 2**	**mode 3**
modern	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}
lucidaot	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}
pagella	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}
cambria	T_{fluff}	T_{fluff}	$T_{\textit{fluff}}$	$T_{\textit{fluff}}$	$T_{\textit{fluff}}$
dejavu	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}	T_{fluff}

Kerning between a character subscript is controlled by \mathscriptcharmode which also defaults to 1.

Here is another example. Internally we tag kerns as italic kerns or font kerns where font kerns result from the staircase kern tables. In 2018 fonts like Latin Modern and Pagella rely on cheats with the boundingbox, Cambria uses staircase kerns and Lucida a mixture. Depending on how fonts evolve we might add some more control over what one can turn on and off.

normal	modern	T_f	γ_e	γ_{ee}	T_{fluff}
	pagella	T_f	γ_e	γ_{ee}	T_{fluff}

	cambria	T_f	γ_e	γ_{ee}	T_{fluff}
	lucidaot	T_f	γ_e	γ_{ee}	T_{fluff}
bold	modern	$\boldsymbol{T_f}$	$\boldsymbol{\gamma_e}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	pagella	$\boldsymbol{T_f}$	$\boldsymbol{\gamma_e}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	cambria	$\boldsymbol{T_f}$	$\boldsymbol{\gamma_e}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$
	lucidaot	$\boldsymbol{T_f}$	$\boldsymbol{\gamma_e}$	$\boldsymbol{\gamma_{ee}}$	$\boldsymbol{T_{\mathrm{fluff}}}$

7.10 Unscaled fences

The `\mathdelimitersmode` primitive is experimental and deals with the following (potential) problems. Three bits can be set. The first bit prevents an unwanted shift when the fence symbol is not scaled (a cambria side effect). The second bit forces italic correction between a preceding character ordinal and the fenced subformula, while the third bit turns that subformula into an ordinary so that the same spacing applies as with unfenced variants. Here we show Cambria (with `\mathitalicsmode` enabled).

```
\mathdelimitersmode = 0
\mathdelimitersmode = 1
\mathdelimitersmode = 2
\mathdelimitersmode = 3
\mathdelimitersmode = 4
\mathdelimitersmode = 5
\mathdelimitersmode = 6
\mathdelimitersmode = 7
```

So, when set to 7 fenced subformulas with unscaled delimiters come out the same as unfenced ones. This can be handy for cases where one is forced to use `\left` and `\right` always because of unpredictable content. As said, it's an experimental feature (which somehow fits in the exceptional way fences are dealt with in the engine). The full list of flags is given in the next table:

VALUE	MEANING
"01	don't apply the usual shift
"02	apply italic correction when possible
"04	force an ordinary subformula
"08	no shift when a base character
"10	only shift when an extensible

The effect can depend on the font (and for Cambria one can use for instance "16).

7.11 Math spacing setting

Besides the parameters mentioned in the previous sections, there are also 64 new primitives to control the math spacing table (as explained in Chapter 18 of the TEXbook). The primitive names

are a simple matter of combining two math atom types, but for completeness' sake, here is the whole list:

```
\Umathordordspacing
\Umathordopspacing
\Umathordbinspacing
\Umathordrelspacing
\Umathordopenspacing
\Umathordclosespacing
\Umathordpunctspacing
\Umathordinnerspacing
\Umathopordspacing
\Umathopopspacing
\Umathopbinspacing
\Umathoprelspacing
\Umathopopenspacing
\Umathopclosespacing
\Umathoppunctspacing
\Umathopinnerspacing
\Umathbinordspacing
\Umathbinopspacing
\Umathbinbinspacing
\Umathbinrelspacing
\Umathbinopenspacing
\Umathbinclosespacing
\Umathbinpunctspacing
\Umathbininnerspacing
\Umathrelordspacing
\Umathrelopspacing
\Umathrelbinspacing
\Umathrelrelspacing
\Umathrelopenspacing
\Umathrelclosespacing
\Umathrelpunctspacing
\Umathrelinnerspacing
\Umathopenordspacing
\Umathopenopspacing
\Umathopenbinspacing
\Umathopenrelspacing
\Umathopenopenspacing
\Umathopenclosespacing
\Umathopenpunctspacing
\Umathopeninnerspacing
\Umathcloseordspacing
\Umathcloseopspacing
\Umathclosebinspacing
\Umathcloserelspacing
\Umathcloseopenspacing
\Umathcloseclosespacing
\Umathclosepunctspacing
\Umathcloseinnerspacing
\Umathpunctordspacing
\Umathpunctopspacing
\Umathpunctbinspacing
\Umathpunctrelspacing
\Umathpunctopenspacing
\Umathpunctclosespacing
\Umathpunctpunctspacing
\Umathpunctinnerspacing
\Umathinnerordspacing
\Umathinneropspacing
\Umathinnerbinspacing
\Umathinnerrelspacing
\Umathinneropenspacing
\Umathinnerclosespacing
\Umathinnerpunctspacing
\Umathinnerinnerspacing
```

These parameters are of type `\muskip`, so setting a parameter can be done like this:

```
\Umathopordspacing\displaystyle=4mu plus 2mu
```

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 specially. 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.

7.12 Math accent handling

LuaTEX 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:

```
\Umathaccent both fixed 0 0 "20D7 fixed 0 0 "20D7 {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 TEX82.

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 TEXbook), but if a value turns out to be zero and the font had a `MathConstants` table, then `AccentBaseHeight` is used instead.

The vertical placement of a bottom accent is straight below the accentee, no correction takes place.

Possible locations are `top`, `bottom`, `both` and `center`. When no location is given `top` is assumed. An additional parameter `fraction` can be specified followed by a number; a value of for instance 1200 means that the criterium is 1.2 times the width of the nucleus. The fraction only applies to the stepwise selected shapes and is mostly meant for the `overlay` location. It also works for the other locations but then it concerns the width.

7.13 Math root extension

The new primitive `\Uroot` allows the construction of a radical noad including a degree field. Its syntax is an extension of `\Uradical`:

```
\Uradical <fam integer> <char integer> <radicand>
```


```
\Uroot    <fam integer> <char integer> <degree> <radicand>
```

The placement of the degree is controlled by the math parameters `\Umathradicaldegreebefore`, `\Umathradicaldegreeafter`, and `\Umathradicaldegreeraise`. The degree will be typeset in `\scriptscriptstyle`.

7.14 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, LuaTEX checks whether the super- or subscript and the nucleus are both simple character items. If they are, and if the fonts of both character items are OpenType fonts (as opposed to legacy TEX fonts), then LuaTEX will use the OpenType math algorithm for deciding on the horizontal placement of the super- or subscript.

This works as follows:

- The vertical position of the script is calculated.
- The default horizontal position is flat next to the base character.
- For superscripts, the italic correction of the base character is added.
- For a superscript, two vertical values are calculated: the bottom of the script (after shifting up), and the top of the base. For a subscript, the two values are the top of the (shifted down) script, and the bottom of the base.
- For each of these two locations:
  - find the math kern 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 math kern 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 math kern 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 math kern pairs at all).

7.15 Scripts on horizontally extensible items like arrows

The primitives `\Uunderdelimiter` and `\Uoverdelimiter` allow the placement of a subscript or superscript on an automatically extensible item and `\Udelimiterunder` and `\Udelimiterover` allow the placement of an automatically extensible item as a subscript or superscript on a nucleus. The input:

```
$\Uoverdelimiter  0 "2194 {\hbox{\strut  overdelimiter}}$
$\Uunderdelimiter 0 "2194 {\hbox{\strut underdelimiter}}$
$\Udelimiterover  0 "2194 {\hbox{\strut  delimiterover}}$
```

```
$\Udelimiterunder 0 "2194 {\hbox{\strut delimiterunder}}$
```

will render this:

$$\overset{\text{overdelimiter}}{\longleftrightarrow}\underset{\text{underdelimiter}}{\longleftrightarrow}\overset{\longleftrightarrow}{\text{delimiterover}}\underset{\longleftrightarrow}{\text{delimiterunder}}$$

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.

These primitives accepts an option `width` specification. When used the also optional keywords `left`, `middle` and `right` will determine what happens when a requested size can't be met (which can happen when we step to successive larger variants).

An extra primitive `\Uhextensible` is available that can be used like this:

```
$\Uhextensible width 10cm 0 "2194$
```

This will render this:

$$\longleftrightarrow$$

Here you can also pass options, like:

```
$\Uhextensible width 1pt middle 0 "2194$
```

This gives:

↔

LuaTEX internally uses a structure that supports OpenType 'MathVariants' as well as tfm 'extensible recipes'. In most cases where font metrics are involved we have a different code path for traditional fonts end OpenType fonts.

7.16 Extracting values

You can extract the components of a math character. Say that we have defined:

```
\Umathcode 1 2 3 4
```

then

```
[\Umathcharclass1] [\Umathcharfam1] [\Umathcharslot1]
```

will return:

```
[2] [3] [4]
```

These commands are provides as convenience. Before they come available you could do the following:

```
\def\Umathcharclass{\directlua{tex.print(tex.getmathcode(token.scan_int())[1])}}
\def\Umathcharfam  {\directlua{tex.print(tex.getmathcode(token.scan_int())[2])}}
\def\Umathcharslot {\directlua{tex.print(tex.getmathcode(token.scan_int())[3])}}
```

7.17 fractions

The `\abovewithdelims` command accepts a keyword `exact`. When issued the extra space relative to the rule thickness is not added. One can of course use the `\Umathfraction..gap` commands to influence the spacing. Also the rule is still positioned around the math axis.

```
$$ { {a} \abovewithdelims() exact 4pt {b} }$$
```

The math parameter table contains some parameters that specify a horizontal and vertical gap for skewed fractions. Of course some guessing is needed in order to implement something that uses them. And so we now provide a primitive similar to the other fraction related ones but with a few options so that one can influence the rendering. Of course a user can also mess around a bit with the parameters `\Umathskewedfractionhgap` and `\Umathskewedfractionvgap`.

The syntax used here is:

```
{ {1} \Uskewed / <options> {2} }
{ {1} \Uskewedwithdelims / () <options> {2} }
```

where the options can be `noaxis` and `exact`. By default we add half the axis to the shifts and by default we zero the width of the middle character. For Latin Modern the result looks as follows:

	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$
`exact`	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$
`noaxis`	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$
`exact noaxis`	$x + {}^{a}\!/_{b} + x$	$x + {}^{1}\!/_{2} + x$	$x + ({}^{a}\!/_{b}) + x$	$x + ({}^{1}\!/_{2}) + x$

7.18 Last lines

There is a new primitive to control the overshoot in the calculation of the previous line in mid-paragraph display math. The default value is 2 times the em width of the current font:

```
\predisplaygapfactor=2000
```

If you want to have the length of the last line independent of math i.e. you don't want to revert to a hack where you insert a fake display math formula in order to get the length of the last line, the following will often work too:

```
\def\lastlinelength{\dimexpr
    \directlua {tex.sprint (
        (nodes.dimensions(node.tail(tex.lists.page_head).list))
    )}sp
\relax}
```

7.19 Other Math changes

7.19.1 Verbose versions of single-character math commands

LuaTEX defines six new primitives that have the same function as ^, _, $, and $$:

PRIMITIVE	EXPLANATION
\Usuperscript	duplicates the functionality of ^
\Usubscript	duplicates the functionality of _
\Ustartmath	duplicates the functionality of $, when used in non-math mode.
\Ustopmath	duplicates the functionality of $, when used in inline math mode.
\Ustartdisplaymath	duplicates the functionality of $$, when used in non-math mode.
\Ustopdisplaymath	duplicates the functionality of $$, when used in display math mode.

The \Ustopmath and \Ustopdisplaymath 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).

7.19.2 Script commands \Unosuperscript and \Unosubscript

These two commands result in super- and subscripts but with the current style (at the time of rendering). So,

```
$\Uhextensible width 1pt middle 0 "2194$
```

results in $x_2^1 = x_2^1 = x_2^1 = x_2^1$.

7.19.3 Allowed math commands in non-math modes

The commands \mathchar, and \Umathchar and control sequences that are the result of \mathchardef or \Umathchardef are also acceptable in the horizontal and vertical modes. In those cases, the \textfont from the requested math family is used.

7.20 Math surrounding skips

Inline math is surrounded by (optional) \mathsurround spacing but that is a fixed dimension. There is now an additional parameter \mathsurroundskip. When set to a non-zero value (or zero with some stretch or shrink) this parameter will replace \mathsurround. By using an additional parameter instead of changing the nature of \mathsurround, we can remain compatible. In the meantime a bit more control has been added via \mathsurroundmode. This directive can take 6 values with zero being the default behaviour.

```
\mathsurround    10pt
\mathsurroundskip20pt
```

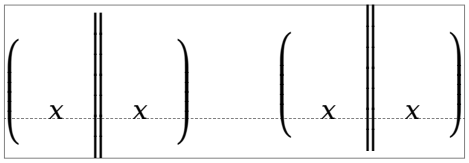

MODE	XXX	X X X	EFFECT
0	x*x*x	x *x* x	obey \mathsurround when \mathsurroundskip is 0pt
1	x*x*x	x *x* x	only add skip to the left
2	x*x*x	x *x* x	only add skip to the right
3	x*x*x	x *x* x	add skip to the left and right
4	x*x*x	x *x* x	ignore the skip setting, obey \mathsurround
5	x*x*x	x *x* x	disable all spacing around math
6	x*x*x	x *x* x	only apply \mathsurroundskip when also spacing
7	x*x*x	x *x* x	only apply \mathsurroundskip when no spacing

Method six omits the surround glue when there is (x)spacing glue present while method seven does the opposite, the glue is only applied when there is (x)space glue present too. Anything more fancy, like checking the begining or end of a paragraph (or edges of a box) would not be robust anyway. If you want that you can write a callback that runs over a list and analyzes a paragraph. Actually, in that case you could also inject glue (or set the properties of a math node) explicitly. So, these modes are in practice mostly useful for special purposes and experiments (they originate in a tracker item). Keep in mind that this glue is part of the math node and not always treated as normal glue: it travels with the begin and end math nodes. Also, method 6 and 7 will zero the skip related fields in a node when applicable in the first occasion that checks them (linebreaking or packaging).

7.20.1 Delimiters: \Uleft, \Umiddle and \Uright

Normally you will force delimiters to certain sizes by putting an empty box or rule next to it. The resulting delimiter will either be a character from the stepwise size range or an extensible. The latter can be quite differently positioned than the characters as it depends on the fit as well as the fact if the used characters in the font have depth or height. Commands like (plain TEXs) \big need use this feature. In LuaTEX we provide a bit more control by three variants that support optional parameters height, depth and axis. The following example uses this:

```
\Uleft   height 30pt depth 10pt      \Udelimiter "0 "0 "000028
\quad x\quad
\Umiddle height 40pt depth 15pt      \Udelimiter "0 "0 "002016
\quad x\quad
\Uright  height 30pt depth 10pt      \Udelimiter "0 "0 "000029
\quad \quad \quad
\Uleft   height 30pt depth 10pt axis \Udelimiter "0 "0 "000028
\quad x\quad
\Umiddle height 40pt depth 15pt axis \Udelimiter "0 "0 "002016
\quad x\quad
\Uright  height 30pt depth 10pt axis \Udelimiter "0 "0 "000029
```



The keyword `exact` can be used as directive that the real dimensions should be applied when the criteria can't be met which can happen when we're still stepping through the successively larger variants. When no dimensions are given the `noaxis` command can be used to prevent shifting over the axis.

You can influence the final class with the keyword `class` which will influence the spacing. The numbers are the same as for character classes.

7.20.2 Fixed scripts

We have three parameters that are used for this fixed anchoring:

PARAMETER	REGISTER
d	`\Umathsubshiftdown`
u	`\Umathsupshiftup`
s	`\Umathsubsupshiftdown`

When we set `\mathscriptsmode` to a value other than zero these are used for calculating fixed positions. This is something that is needed for instance for chemistry. You can manipulate the mentioned variables to achieve different effects.

MODE	DOWN	UP	EXAMPLE
0	dynamic	dynamic	$CH_2 + CH_2^+ + CH_2^2$
1	d	u	$CH_2 + CH_2^+ + CH_2^2$
2	s	u	$CH_2 + CH_2^+ + CH_2^2$
3	s	$u + s - d$	$CH_2 + CH_2^+ + CH_2^2$
4	$d + (s - d)/2$	$u + (s - d)/2$	$CH_2 + CH_2^+ + CH_2^2$
5	d	$u + s - d$	$CH_2 + CH_2^+ + CH_2^2$

The value of this parameter obeys grouping but applies to the whole current formula.

7.20.3 Penalties: `\mathpenaltiesmode`

Only in inline math penalties will be added in a math list. You can force penalties (also in display math) by setting:

```
\mathpenaltiesmode = 1
```

This primnitive is not really needed in LuaTEX because you can use the callback `mlist_to_hlist` to force penalties by just calling the regular routine with forced penalties. However, as part of opening up and control this primitive makes sense. As a bonus we also provide two extra penalties:

```
\prebinoppenalty = -100 % example value
\prerelpenalty   =  900 % example value
```

They default to inifinite which signals that they don't need to be inserted. When set they are injected before a binop or rel noad. This is an experimental feature.

7.20.4 Equation spacing: \matheqnogapstep

By default TEX will add one quad between the equation and the number. This is hard coded. A new primitive can control this:

```
\matheqnogapstep = 1000
```

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.

7.20.5 Flattening: \mathflattenmode

The TEX math engine collapses `ord` noads without sub- and superscripts and a character as nucleus. and which has the side effect that in OpenType mode italic corrections are applied (given that they are enabled).

```
\switchtobodyfont[modern]
$V \mathbin{\mathbin{v}} V$\par
$V \mathord{\mathord{v}} V$\par
```

This renders as:

$V v V$
$V v V$

When we set `\mathflattenmode` to 31 we get:

$V v V$
$V v V$

When you see no difference, then the font probably has the proper character dimensions and no italic correction is needed. For Latin Modern (at least till 2018) there was a visual difference. In that respect this parameter is not always needed unless of course you want efficient math lists anyway.

You can influence flattening by adding the appropriate number to the value of the mode parameter. The default value is 1.

MODE	CLASS
1	ord
2	bin
4	rel
8	punct
16	inner

7.20.6 Tracing

Because there are quite some math related parameters and values, it is possible to limit tracing. Only when `tracingassigns` and/or `tracingrestores` are set to 2 or more they will be traced.

7.20.7 Math options

The logic in the math engine is rather complex and there are often no universal solutions (read: what works out well for one font, fails for another). Therefore some variations in the implementation are driven by parameters (modes). In addition there is a new primitive `\mathoption` which will be used for testing.

7.20.7.1 `\mathoption old`

This option was introduced for testing purposes when the math engine got split code paths and it forces the engine to treat new fonts as old ones with respect to italic correction etc. There are no guarantees given with respect to the final result and unexpected side effects are not seen as bugs as they relate to font properties.

The `oldmath` boolean flag in the Lua font table is the official way to force old treatment as it's bound to fonts. Like with all options we may temporarily introduce with this command this feature is not meant for production.

7.20.7.2 Obsolete options

The following options are gone: `noitaliccompensation`, `nocharitalic`, `useoldfractionscaling`, and `umathcodemeaning`.

8 Nodes

8.1 LUA node representation

TEX's nodes are represented in Lua as userdata objects with a variable set of fields. In the following syntax tables, such as the type of such a userdata object is represented as ⟨node⟩.

The current return value of `node.types()` is: `hlist` (0), `vlist` (1), `rule` (2), `ins` (3), `mark` (4), `adjust` (5), `boundary` (6), `disc` (7), `whatsit` (8), `local_par` (9), `dir` (10), `math` (11), `glue` (12), `kern` (13), `penalty` (14), `unset` (15), `style` (16), `choice` (17), `noad` (18), `radical` (19), `fraction` (20), `accent` (21), `fence` (22), `math_char` (23), `sub_box` (24), `sub_mlist` (25), `math_text_char` (26), `delim` (27), `margin_kern` (28), `glyph` (29), `align_record` (30), `pseudo_file` (31), `pseudo_line` (32), `page_insert` (33), `split_insert` (34), `expr_stack` (35), `nested_list` (36), `span` (37), `attribute` (38), `glue_spec` (39), `attribute_list` (40), `temp` (41), `align_stack` (42), `movement_stack` (43), `if_stack` (44), `unhyphenated` (45), `hyphenated` (46), `delta` (47), `passive` (48), `shape` (49).

The `\lastnodetype` primitive is ε-TEX compliant. The valid range is still $[-1, 15]$ and glyph nodes (formerly known as char nodes) have number 0 while ligature nodes are mapped to 7. That way macro packages can use the same symbolic names as in traditional ε-TEX. Keep in mind that these ε-TEX node numbers are different from the real internal ones and that there are more ε-TEX node types than 15.

You can ask for a list of fields with the `node.fields` and for valid subtypes with `node.subtypes`.

The `node.values` function reports some used values. Valid arguments are `dir`, `direction`, `glue`, `pdf_literal`, `pdf_action`, `pdf_window` and `color_stack`. Keep in mind that the setters normally expect a number, but this helper gives you a list of what numbers matter. For practical reason the `pagestate` values are also reported with this helper.

8.1.1 Attributes

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.

8.1.1.1 attribute_list nodes

An `attribute_list` item is used as a head pointer for a list of attribute items. It has only one user-visible field:

FIELD	TYPE	EXPLANATION
`next`	node	pointer to the first attribute

8.1.1.2 attr nodes

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.

FIELD	TYPE	EXPLANATION
next	node	pointer to the next attribute
number	number	the attribute type id
value	number	the attribute value

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.

8.1.2 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:

FIELD	TYPE	EXPLANATION
next	node	the next node in a list, or nil
id	number	the node's type (id) number
subtype	number	the node subtype identifier

The subtype is sometimes just a dummy entry because 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 a 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. By now most of TEX's node processing makes sure that the prev nodes are valid but there can be exceptions, especially when the internal magic uses a leading temp nodes to temporarily store a state.

8.1.2.1 hlist nodes

FIELD	TYPE	EXPLANATION
subtype	number	0 = unknown, 1 = line, 2 = box, 3 = indent, 4 = alignment, 5 = cell, 6 = equation, 7 = equationnumber
attr	node	list of attributes
width	number	the width of the box
height	number	the height of the box
depth	number	the depth of the box
shift	number	a displacement perpendicular to the character progression direction
glue_order	number	a number in the range [0, 4], indicating the glue order
glue_set	number	the calculated glue ratio

glue_sign	number	0 = normal, 1 = stretching, 2 = shrinking
head/list	node	the first node of the body of this list
dir	string	the direction of this box, see 8.1.2.15

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 field name head and list are both valid. Sometimes it makes more sense to refer to a list by head, sometimes list makes more sense.

8.1.2.2 vlist nodes

This node is similar to hlist, except that 'shift' is a displacement perpendicular to the line progression direction, and 'subtype' only has the values 0, 4, and 5.

8.1.2.3 rule nodes

Contrary to traditional TEX, LuaTEX has more \rule subtypes because we also use rules to store reuseable objects and images. User nodes are invisible and can be intercepted by a callback.

FIELD	TYPE	EXPLANATION
subtype	number	0 = normal, 1 = box, 2 = image, 3 = empty, 4 = user, 5 = over, 6 = under, 7 = fraction, 8 = radical, 9 = outline
attr	node	list of attributes
width	number	the width of the rule where the special value −1073741824 is used for 'running' glue dimensions
height	number	the height of the rule (can be negative)
depth	number	the depth of the rule (can be negative)
left	number	shift at the left end (also subtracted from width)
right	number	(subtracted from width)
dir	string	the direction of this rule, see 8.1.2.15
index	number	an optional index that can be referred to
transform	number	an private variable (also used to specify outline width)

The left and type right keys are somewhat special (and experimental). When rules are auto adapting to the surrounding box width you can enforce a shift to the right by setting left. The value is also subtracted from the width which can be a value set by the engine itself and is not entirely under user control. The right is also subtracted from the width. It all happens in the backend so these are not affecting the calculations in the frontend (actually the auto settings also happen in the backend). For a vertical rule left affects the height and right affects the depth. There is no matching interface at the TEX end (although we can have more keywords for rules it would complicate matters and introduce a speed penalty.) However, you can just construct a rule node with Lua and write it to the TEX input. The outline subtype is just a convenient variant and the transform field specifies the width of the outline.

8.1.2.4 ins nodes

This node relates to the \insert primitive.

FIELD	TYPE	EXPLANATION
subtype	number	the insertion class
attr	node	list of attributes
cost	number	the penalty associated with this insert
height	number	height of the insert
depth	number	depth of the insert
head/list	node	the first node of the body of this insert

There is a set of extra fields that concern the associated glue: `width`, `stretch`, `stretch_order`, `shrink` and `shrink_order`. These are all numbers.

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. You can use `list` instead (often in functions you want to use local variable with similar names and both names are equally sensible).

8.1.2.5 `mark` nodes

This one relates to the `\mark` primitive.

FIELD	TYPE	EXPLANATION
subtype	number	unused
attr	node	list of attributes
class	number	the mark class
mark	table	a table representing a token list

8.1.2.6 `adjust` nodes

This node comes from `\vadjust` primitive.

FIELD	TYPE	EXPLANATION
subtype	number	0 = `normal`, 1 = `pre`
attr	node	list of attributes
head/list	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 the result.

8.1.2.7 `disc` nodes

The `\discretionary` and `\-`, the `-` character but also the hyphenation mechanism produces these nodes.

FIELD	TYPE	EXPLANATION
subtype	number	0 = `discretionary`, 1 = `explicit`, 2 = `automatic`, 3 = `regular`, 4 = `first`, 5 = `second`
attr	node	list of attributes
pre	node	pointer to the pre-break text

post	node	pointer to the post-break text
replace	node	pointer to the no-break text
penalty	number	the penalty associated with the break, normally \hyphenpenalty or \exhyphenpenalty

The subtype numbers 4 and 5 belong to the 'of-f-ice' explanation given elsewhere. These disc nodes are kind of special as at some point they also keep information about breakpoints and nested ligatures.

The pre, post and replace fields at the Lua end are in fact indirectly accessed and have a prev pointer that is not nil. This means that when you mess around with the head of these (three) lists, you also need to reassign them because that will restore the proper prev pointer, so:

```
pre = d.pre
-- change the list starting with pre
d.pre = pre
```

Otherwise you can end up with an invalid internal perception of reality and LuaTEX might even decide to crash on you. It also means that running forward over for instance pre is ok but backward you need to stop at pre. And you definitely must not mess with the node that prev points to, if only because it is not really a node but part of the disc data structure (so freeing it again might crash LuaTEX).

8.1.2.8 math nodes

Math nodes represent the boundaries of a math formula, normally wrapped into $ signs.

FIELD	TYPE	EXPLANATION
subtype	number	0 = beginmath, 1 = endmath
attr	node	list of attributes
surround	number	width of the \mathsurround kern

There is a set of extra fields that concern the associated glue: width, stretch, stretch_order, shrink and shrink_order. These are all numbers.

8.1.2.9 glue nodes

Skips are about the only type of data objects in traditional TEX that are not a simple value. They are inserted when TEX sees a space in the text flow but also by \hskip and \vskip. The structure that represents the glue components of a skip is called a glue_spec, and it has the following accessible fields:

FIELD	TYPE	EXPLANATION
width	number	the horizontal or vertical displacement
stretch	number	extra (positive) displacement or stretch amount
stretch_order	number	factor applied to stretch amount
shrink	number	extra (negative) displacement or shrink amount
shrink_order	number	factor applied to shrink amount

The effective width of some glue subtypes depends on the stretch or shrink needed to make the encapsulating box fit its dimensions. For instance, in a paragraph lines normally have glue representing spaces and these stretch or shrink to make the content fit in the available space. The `effective_glue` function that takes a glue node and a parent (hlist or vlist) returns the effective width of that glue item.

A `glue_spec` node is a special kind of node that is used for storing a set of glue values in registers. Originally they were also used to store properties of glue nodes (using a system of reference counts) but we now keep these properties in the glue nodes themselves, which gives a cleaner interface to Lua.

The indirect spec approach was in fact an optimization in the original TEX code. First of all it can save quite some memory because all these spaces that become glue now share the same specification (only the reference count is incremented), and zero testing is also a bit faster because only the pointer has to be checked (this is no longer true for engines that implement for instance protrusion where we really need to ensure that zero is zero when we test for bounds). Another side effect is that glue specifications are read-only, so in the end copies need to be made when they are used from Lua (each assignment to a field can result in a new copy). So in the end the advantages of sharing are not that high (and nowadays memory is less an issue, also given that a glue node is only a few memory words larger than a spec).

FIELD	TYPE	EXPLANATION
`subtype`	number	0 = `userskip`, 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 = `mathskip`, 17 = `thinmuskip`, 18 = `medmuskip`, 19 = `thickmuskip`, 98 = `conditionalmathskip`, 99 = `muglue`, 100 = `leaders`, 101 = `cleaders`, 102 = `xleaders`, 103 = `gleaders`
`attr`	node	list of attributes
`leader`	node	pointer to a box or rule for leaders

In addition there are the `width`, `stretch` `stretch_order`, `shrink`, and `shrink_order` fields. Note that we use the key `width` in both horizontal and vertical glue. This suits the TEX internals well so we decided to stick to that naming.

A regular word space also results in a `spaceskip` subtype (this used to be a `userskip` with subtype zero).

8.1.2.10 kern nodes

The `\kern` command creates such nodes but for instance the font and math machinery can also add them.

FIELD	TYPE	EXPLANATION
`subtype`	number	0 = `fontkern`, 1 = `userkern`, 2 = `accentkern`, 3 = `italiccorrection`
`attr`	node	list of attributes
`kern`	number	fixed horizontal or vertical advance

8.1.2.11 penalty nodes

The \penalty command is one that generates these nodes.

FIELD	TYPE	EXPLANATION
subtype	number	0 = userpenalty, 1 = linebreakpenalty, 2 = linepenalty, 3 = wordpenalty, 4 = finalpenalty, 5 = noadpenalty, 6 = beforedisplaypenalty, 7 = afterdisplaypenalty, 8 = equationnumberpenalty
attr	node	list of attributes
penalty	number	the penalty value

The subtypes are just informative and TEX itself doesn't use them. When you run into an linebreakpenalty you need to keep in mind that it's a accumulation of club, widow and other relevant penalties.

8.1.2.12 glyph nodes

These are probably the mostly used nodes and although you can push them in the current list with for instance \char TEX will normally do it for you when it considers some input to be text.

FIELD	TYPE	EXPLANATION
subtype	number	bit field
attr	node	list of attributes
char	number	the character index in the font
font	number	the font identifier
lang	number	the language identifier
left	number	the frozen \lefthyphenmnin value
right	number	the frozen \righthyphenmnin value
uchyph	boolean	the frozen \uchyph value
components	node	pointer to ligature components
xoffset	number	a virtual displacement in horizontal direction
yoffset	number	a virtual displacement in vertical direction
width	number	the (original) width of the character
height	number	the (original) height of the character
depth	number	the (original) depth of the character
expansion_factor	number	the to be applied expansion_factor

The width, height and depth values are read-only. The expansion_factor is assigned in the par builder and used in the backend.

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
1	ligature
2	ghost

3	left
4	right

See section 5.1 for a detailed description of the `subtype` field.

The `expansion_factor` has been introduced as part of the separation between font- and backend. It is the result of extensive experiments with a more efficient implementation of expansion. Early versions of LuaTEX already replaced multiple instances of fonts in the backend by scaling but contrary to pdfTEX in LuaTEX we now also got rid of font copies in the frontend and replaced them by expansion factors that travel with glyph nodes. Apart from a cleaner approach this is also a step towards a better separation between front- and backend.

The `is_char` function checks if a node is a glyph node with a subtype still less than 256. This function can be used to determine if applying font logic to a glyph node makes sense. The value `nil` gets returned when the node is not a glyph, a character number is returned if the node is still tagged as character and `false` gets returned otherwise. When nil is returned, the id is also returned. The `is_glyph` variant doesn't check for a subtype being less than 256, so it returns either the character value or nil plus the id. These helpers are not always faster than separate calls but they sometimes permit making more readable tests. The `uses_font` helpers takes a node and font id and returns true when a glyph or disc node references that font.

8.1.2.13 `boundary` nodes

This node relates to the `\noboundary`, `\boundary`, `\protrusionboundary` and `\wordboundary` primitives.

FIELD	TYPE	EXPLANATION
`subtype`	number	0 = `cancel`, 1 = `user`, 2 = `protrusion`, 3 = `word`
`attr`	node	list of attributes
`value`	number	values 0–255 are reserved

8.1.2.14 `local_par` nodes

This node is inserted at the start of a paragraph. You should not mess too much with this one.

FIELD	TYPE	EXPLANATION
`attr`	node	list of attributes
`pen_inter`	number	local interline penalty (from `\localinterlinepenalty`)
`pen_broken`	number	local broken penalty (from `\localbrokenpenalty`)
`dir`	string	the direction of this par. see 8.1.2.15
`box_left`	node	the `\localleftbox`
`box_left_width`	number	width of the `\localleftbox`
`box_right`	node	the `\localrightbox`
`box_right_width`	number	width of the `\localrightbox`

A warning: never assign a node list to the `box_left` or `box_right` field unless you are sure its internal link structure is correct, otherwise an error may result.

8.1.2.15 dir nodes

Direction nodes mark parts of the running text that need a change of direction and the \textdir command generates them.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
dir	string	the direction (but see below)
level	number	nesting level of this direction whatsit

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 nodes, the representation of dir is not a three-letter but a combination of numbers. When printed the direction is indicated by a + or -, indicating whether the value is pushed or popped from the direction stack.

8.1.2.16 marginkern nodes

Margin kerns result from protrusion.

FIELD	TYPE	EXPLANATION
subtype	number	0 = left, 1 = right
attr	node	list of attributes
width	number	the advance of the kern
glyph	node	the glyph to be used

8.1.3 Math noads

These are the so–called 'noad's and the nodes that are specifically associated with math processing. Most of these nodes contain subnodes so that the list of possible fields is actually quite small. First, the subnodes:

8.1.3.1 Math kernel subnodes

Many object fields in math mode are either simple characters in a specific family or math lists or node lists. There are four associated subnodes that represent these cases (in the following node descriptions these are indicated by the word <kernel>).

The next and prev fields for these subnodes are unused.

8.1.3.1.1 `math_char` and `math_text_char` subnodes

FIELD	TYPE	EXPLANATION
`attr`	node	list of attributes
`char`	number	the character index
`fam`	number	the family number

The `math_char` is the simplest subnode field, it contains the character and family for a single glyph object. The `math_text_char` is a special case that you will not normally encounter, it arises temporarily during math list conversion (its sole function is to suppress a following italic correction).

8.1.3.1.2 `sub_box` and `sub_mlist` subnodes

FIELD	TYPE	EXPLANATION
`attr`	node	list of attributes
`head/list`	node	list of nodes

These two subnode types are used for subsidiary list items. For `sub_box`, the `head` points to a 'normal' vbox or hbox. For `sub_mlist`, the `head` points to a math list that is yet to be converted.

A warning: never assign a node list to the `head` field unless you are sure its internal link structure is correct, otherwise an error is triggered.

8.1.3.1.3 `delim` subnodes

There is a fifth subnode type that is used exclusively for delimiter fields. As before, the `next` and `prev` fields are unused.

FIELD	TYPE	EXPLANATION
`attr`	node	list of attributes
`small_char`	number	character index of base character
`small_fam`	number	family number of base character
`large_char`	number	character index of next larger character
`large_fam`	number	family number of next larger character

The fields `large_char` and `large_fam` can be zero, in that case the font that is set for the `small_fam` is expected to provide the large version as an extension to the `small_char`.

8.1.3.2 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.

Some noads have an option field. The values in this bitset are common:

MEANING	BITS
set	0x08
internal	0x00 + 0x08
internal	0x01 + 0x08
axis	0x02 + 0x08
no axis	0x04 + 0x08
exact	0x10 + 0x08
left	0x11 + 0x08
middle	0x12 + 0x08
right	0x14 + 0x08
no sub script	0x21 + 0x08
no super script	0x22 + 0x08
no script	0x23 + 0x08

8.1.3.2.1 simple noad nodes

FIELD	TYPE	EXPLANATION
subtype	number	0 = ord, 1 = opdisplaylimits, 2 = oplimits, 3 = opnolimits, 4 = bin, 5 = rel, 6 = open, 7 = close, 8 = punct, 9 = inner, 10 = under, 11 = over, 12 = vcenter
attr	node	list of attributes
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
options	number	bitset of rendering options

8.1.3.2.2 accent nodes

FIELD	TYPE	EXPLANATION
subtype	number	0 = bothflexible, 1 = fixedtop, 2 = fixedbottom, 3 = fixedboth
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
accent	kernel node	top accent
bot_accent	kernel node	bottom accent
fraction	number	larger step criterium (divided by 1000)

8.1.3.2.3 style nodes

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 be prefixed by cramped.

8.1.3.2.4 choice nodes

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
display	node	list of display size alternatives
text	node	list of text size alternatives
script	node	list of scriptsize alternatives
scriptscript	node	list of scriptscriptsize alternatives

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 can occur.

8.1.3.2.5 radical nodes

FIELD	TYPE	EXPLANATION
subtype	number	0 = radical, 1 = uradical, 2 = uroot, 3 = uunderdelimiter, 4 = uoverdelimiter, 5 = udelimiterunder, 6 = udelimiterover
attr	node	list of attributes
nucleus	kernel node	base
sub	kernel node	subscript
sup	kernel node	superscript
left	delimiter node	
degree	kernel node	only set by \Uroot
width	number	required width
options	number	bitset of rendering options

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 can be triggered.

8.1.3.2.6 fraction nodes

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
width	number	(optional) width of the fraction
num	kernel node	numerator
denom	kernel node	denominator
left	delimiter node	left side symbol
right	delimiter node	right side symbol
middle	delimiter node	middle symbol
options	number	bitset of rendering options

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 can result.

8.1.3.2.7 fence nodes

FIELD	TYPE	EXPLANATION
subtype	number	0 = unset, 1 = left, 2 = middle, 3 = right, 4 = no

attr	node	list of attributes
delim	delimiter node	delimiter specification
italic	number	italic correction
height	number	required height
depth	number	required depth
options	number	bitset of rendering options
class	number	spacing related class

Warning: some of these fields are used by the renderer and might get adapted in the process.

8.1.4 whatsit nodes

Whatsit nodes come in many subtypes that you can ask for them by running `node.whatsits()`: `open` (0), `write` (1), `close` (2), `special` (3), `save_pos` (6), `late_lua` (7), `user_defined` (8), `pdf_literal` (16), `pdf_refobj` (17), `pdf_annot` (18), `pdf_start_link` (19), `pdf_end_link` (20), `pdf_dest` (21), `pdf_action` (22), `pdf_thread` (23), `pdf_start_thread` (24), `pdf_end_thread` (25), `pdf_thread_data` (26), `pdf_link_data` (27), `pdf_colorstack` (28), `pdf_setmatrix` (29), `pdf_save` (30), `pdf_restore` (31).

8.1.4.1 front-end whatsits

8.1.4.1.1 open

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
stream	number	TEX's stream id number
name	string	file name
ext	string	file extension
area	string	file area (this may become obsolete)

8.1.4.1.2 write

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
stream	number	TEX's stream id number
data	table	a table representing the token list to be written

8.1.4.1.3 close

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
stream	number	TEX's stream id number

8.1.4.1.4 user_defined

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 whatsits without ever looking at the contents.

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
user_id	number	id number
type	number	type of the value
value	number	a Lua number
	node	a node list
	string	a Lua string
	table	a Lua table

The type can have one of six distinct values. The number is the ascii value if the first character of the type name (so you can use string.byte("l") instead of 108).

VALUE	MEANING	EXPLANATION
97	a	list of attributes (a node list)
100	d	a Lua number
108	l	a Lua value (table, number, boolean, etc)
110	n	a node list
115	s	a Lua string
116	t	a Lua token list in Lua table form (a list of triplets)

8.1.4.1.5 save_pos

FIELD	TYPE	EXPLANATION
attr	node	list of attributes

8.1.4.1.6 late_lua

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
data	string	data to execute
string	string	data to execute
name	string	the name to use for Lua error reporting

The difference between data and string is that on assignment, the data field is converted to a token list, cf. use as \latelua. The string version is treated as a literal string.

8.1.4.2 DVI backend whatsits

8.1.4.2.1 special

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
data	string	the \special information

8.1.4.3 PDF backend whatsits

8.1.4.3.1 pdf_literal

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
mode	number	the 'mode' setting of this literal
data	string	the \pdfliteral information

Possible mode values are:

VALUE	KEYWORD
0	origin
1	page
2	direct
3	raw
4	text

The higher the number, the less checking and the more you can run into trouble. Especially the raw variant can produce bad pdf so you can best check what you generate.

8.1.4.3.2 pdf_refobj

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
objnum	number	the referenced pdf object number

8.1.4.3.3 pdf_annot

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
width	number	the width (not used in calculations)
height	number	the height (not used in calculations)
depth	number	the depth (not used in calculations)
objnum	number	the referenced pdf object number
data	string	the annotation data

8.1.4.3.4 pdf_start_link

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
width	number	the width (not used in calculations)
height	number	the height (not used in calculations)
depth	number	the depth (not used in calculations)
objnum	number	the referenced pdf object number
link_attr	table	the link attribute token list
action	node	the action to perform

8.1.4.3.5 pdf_end_link

FIELD	TYPE	EXPLANATION
attr	node	

8.1.4.3.6 pdf_dest

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
width	number	the width (not used in calculations)
height	number	the height (not used in calculations)
depth	number	the depth (not used in calculations)
named_id	number	is the dest_id a string value?
dest_id	number	the destination id
	string	the destination name
dest_type	number	type of destination
xyz_zoom	number	the zoom factor (times 1000)
objnum	number	the pdf object number

8.1.4.3.7 pdf_action

These are a special kind of items that only appear inside pdf start link objects.

FIELD	TYPE	EXPLANATION
action_type	number	the kind of action involved
action_id	number or string	token list reference or string
named_id	number	the index of the destination
file	string	the target filename
new_window	number	the window state of the target
data	string	the name of the destination

Valid action types are:

VALUE	MEANING
0	page

1	goto
2	thread
3	user

Valid window types are:

VALUE	MEANING
0	notset
1	new
2	nonew

8.1.4.3.8 pdf_thread

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
width	number	the width (not used in calculations)
height	number	the height (not used in calculations)
depth	number	the depth (not used in calculations)
named_id	number	is tread_id a string value?
tread_id	number	the thread id
	string	the thread name
thread_attr	number	extra thread information

8.1.4.3.9 pdf_start_thread

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
width	number	the width (not used in calculations)
height	number	the height (not used in calculations)
depth	number	the depth (not used in calculations)
named_id	number	is tread_id a string value?
tread_id	number	the thread id
	string	the thread name
thread_attr	number	extra thread information

8.1.4.3.10 pdf_end_thread

FIELD	TYPE	EXPLANATION
attr	node	

8.1.4.3.11 pdf_colorstack

FIELD	TYPE	EXPLANATION
attr	node	list of attributes

stack	number	colorstack id number
command	number	command to execute
data	string	data

8.1.4.3.12 pdf_setmatrix

FIELD	TYPE	EXPLANATION
attr	node	list of attributes
data	string	data

8.1.4.3.13 pdf_save

FIELD	TYPE	EXPLANATION
attr	node	list of attributes

8.1.4.3.14 pdf_restore

FIELD	TYPE	EXPLANATION
attr	node	list of attributes

8.2 The node library

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

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

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

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

The other available fields depend on the id (and for 'whatsits', the subtype) of the node.

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

Nodes can be compared to each other, but: you are actually comparing indices into the node memory. This means that equality tests can only be trusted under very limited conditions. It will

not work correctly in any situation where one of the two nodes has been freed and/or reallocated: in that case, there will be false positives.

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

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

8.2.1 Node handling functions

8.2.1.1 node.is_node

```
<boolean|integer> t =
    node.is_node(<any> item)
```

This function returns a number (the internal index of the node) if the argument is a userdata object of type `<node>` and false when no node is passed.

8.2.1.2 node.types

```
<table> 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.

8.2.1.3 node.whatsits

```
<table> t =
    node.whatsits()
```

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

8.2.1.4 node.id

```
<number> id =
    node.id(<string> type)
```

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

8.2.1.5 node.subtype

```
<number> subtype =
    node.subtype(<string> type)
```

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

8.2.1.6 node.type

```
<string> type =
    node.type(<any> n)
```

In the argument is a number, then this function converts an internal numeric representation to an external string representation. Otherwise, it will return the string `node` if the object represents a node, and `nil` otherwise.

8.2.1.7 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.

8.2.1.8 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.

8.2.1.9 node.new

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

The `new` function creates a new node. All its fields are initialized to either zero or `nil` except for `id` and `subtype`. Instead of numbers you can also use strings (names). If you create a new `whatsit` node the second argument is required. As with all node functions, this function creates a node at the TEX level.

8.2.1.10 node.free and node.flush_node

```
<node> next =
    node.free(<node> n)
flush_node(<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.

The `free` function returns the next field of the freed node, while the `flush_node` alternative returns nothing.

8.2.1.11 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.

8.2.1.12 node.copy

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

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

8.2.1.13 node.copy_list

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

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

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

8.2.1.14 node.next

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


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

8.2.1.15 node.prev

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

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

8.2.1.16 node.current_attr

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

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

The intended usage of `current_attr` is as follows:

```
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.

8.2.1.17 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. The second return value is the badness of the generated box.

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!

8.2.1.18 node.vpack

```
<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.

The second return value is the badness of the generated box.

See the description of `node.hpack()` for a few memory allocation caveats.

8.2.1.19 node.dimensions, node.rangedimensions

```
<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)
```

This function calculates the natural in-line dimensions of the node list starting at node `n` and terminating just before node `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:

```
<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)
```

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 in between the `a` and `b` as it would be if `\box0` was used as-is:

```
\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)
)) }
```

You need to keep in mind that this is one of the few places in TEX where floats are used, which means that you can get small differences in rounding when you compare the width reported by `hpack` with `dimensions`.

The second alternative saves a few lookups and can be more convenient in some cases:

```
<number> w, <number> h, <number> d  =
    node.rangedimensions(<node> parent, <node> first)
<number> w, <number> h, <number> d  =
    node.rangedimensions(<node> parent, <node> first, <node> last)
```

8.2.1.20 node.mlist_to_hlist

```
<node> 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 `n` into the horizontal list `h`. The interface is exactly the same as for the callback `mlist_to_hlist`.

8.2.1.21 node.slide

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

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

8.2.1.22 node.tail

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

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

8.2.1.23 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 `n`. If `m` is also supplied it stops at `m` instead of at the end of the list. The node `m` is not counted.

8.2.1.24 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 `n` that have a matching `id` field. If `m` is also supplied, counting stops at `m` instead of at the end of the list. The node `m` is not counted.

This function also accept string `id`'s.

8.2.1.25 node.traverse

```
<node> t, id, subtype =
    node.traverse(<node> n)
```

This is a Lua iterator that loops over the node list that starts at `n`. Typically code looks like this:

```
for n in node.traverse(head) do
   ...
end
```

is functionally equivalent to:

```
do
  local n
  local function f (head,var)
    local t
    if var == nil then
       t = head
```


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

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

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

8.2.1.26 node.traverse_id

```
<node> t, subtype =
    node.traverse_id(<number> id, <node> n)
```

This is an iterator that loops over all the nodes in the list that starts at `n` that have a matching `id` field.

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

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

8.2.1.27 node.traverse_char

This iterator loops over the `glyph` nodes in a list. Only nodes with a subtype less than 256 are seen.

```
<node> n, font, char =
```

```
    node.traverse_char(<node> n)
```

8.2.1.28 node.traverse_glyph

This iterator loops over a list and returns the list and filters all glyphs:

```
<node> n, font, char =
    node.traverse_glyph(<node> n)
```

8.2.1.29 node.traverse_list

This iterator loops over the `hlist` and `vlist` nodes in a list.

```
<node> n, id, subtype, list =
    node.traverse_list(<node> n)
```

The four return values can save some time compared to fetching these fields but in practice you seldom need them all. So consider it a (side effect of experimental) convenience.

8.2.1.30 node.has_glyph

This function returns the first glyph or disc node in the given list:

```
<node> n =
    node.has_glyph(<node> n)
```

8.2.1.31 node.end_of_math

```
<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 end node this helper returns that node, else it follows the list and returns the next math endnote. If no such node is found nil is returned.

8.2.1.32 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.

8.2.1.33 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`.

8.2.1.34 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`.

8.2.1.35 node.first_glyph

```
<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 `nil`. If `m` is given, processing stops at (but including) that node, otherwise processing stops at the end of the list.

8.2.1.36 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).

8.2.1.37 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 node list. 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).

8.2.1.38 node.unprotect_glyphs and node.unprotect_glyph

```
node.unprotect_glyph(<node> n)
node.unprotect_glyphs(<node> n,[<node> n])
```

Subtracts 256 from all glyph node subtypes. This and the next function are helpers to convert from `characters` to `glyphs` during node processing. The second argument is optional and indicates the end of a range.

8.2.1.39 node.protect_glyphs and node.protect_glyph

```
node.protect_glyph(<node> n)
node.protect_glyphs(<node> n,[<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 `characters` will become `glyphs` after subtraction of 256. A single character can be marked by the singular call. The second argument is optional and indicates the end of a range.

8.2.1.40 node.last_node

```
<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.

8.2.1.41 node.write

```
node.write(<node> n)
```

This function that will append a node list to TEX's 'current list'. The node list is not deep-copied! There is no error checking either! You mignt need to enforce horizontal mode in order for this to work as expected.

8.2.1.42 node.protrusion_skippable

```
<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.

8.2.2 Glue handling

8.2.2.1 node.setglue

You can set the properties of a glue in one go. If you pass no values, the glue will become a zero glue.

```
node.setglue(<node> n)
node.setglue(<node> n,width,stretch,shrink,stretch_order,shrink_order)
```

When you pass values, only arguments that are numbers are assigned so

```
node.setglue(n,655360,false,65536)
```

will only adapt the width and shrink.

When a list node is passed, you set the glue, order and sign instead.

8.2.2.2 node.getglue

The next call will return 5 values or nothing when no glue is passed.

```
<integer> width, <integer> stretch, <integer> shrink, <integer> stretch_order,
    <integer> shrink_order = node.getglue(<node> n)
```

When the second argument is false, only the width is returned (this is consistent with `tex.get`).

When a list node is passed, you get back the glue that is set, the order of that glue and the sign.

8.2.2.3 node.is_zero_glue

This function returns `true` when the width, stretch and shrink properties are zero.

```
<boolean> isglue =
    node.is_zero_glue(<node> n)
```

8.2.3 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.

8.2.3.1 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`.

8.2.3.2 node.get_attribute

```
<number> v =
    node.get_attribute(<node> n, <number> id)
```

Tests if a node has an attribute with number `id` set. It returns the value, or, if no match is found, `nil`. If no `id` is given then the zero attributes is assumed.

8.2.3.3 node.find_attribute

```
<number> v, <node> n =
    node.find_attribute(<node> n, <number> id)
```

Finds the first node that has attribute with number `id` set. It returns the value and the node if there is a match and otherwise nothing.

8.2.3.4 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.

8.2.3.5 node.unset_attribute

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

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

If the attribute was actually deleted, returns its old value. Otherwise, returns `nil`.

8.2.3.6 node.slide

This helper makes sure that the node lists is double linked and returns the found tail node.

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

After some callbacks automatic sliding takes place. This feature can be turned off with `node.fix_node_lists(false)` but you better make sure then that you don't mess up lists. In most cases TEX itself only uses `next` pointers but your other callbacks might expect proper `prev` pointers too. Future versions of LuaTEX can add more checking but this will not influence usage.

8.2.3.7 node.check_discretionary and node.check_discretionaries

When you fool around with disc nodes you need to be aware of the fact that they have a special internal data structure. As long as you reassign the fields when you have extended the lists it's ok because then the tail pointers get updated, but when you add to list without reassigning you might end up in trouble when the linebreak routien kicks in. You can call this function to check the list for issues with disc nodes.

```
node.check_discretionary(<node> n)
node.check_discretionaries(<node> head)
```

The plural variant runs over all disc nodes in a list, the singular variant checks one node only (it also checks if the node is a disc node).

8.2.3.8 node.flatten_discretionaries

This function will remove the discretionaries in the list and inject the replace field when set.

```
<node> head, count = node.flatten_discretionaries(<node> n)
```

8.2.3.9 node.family_font

When you pass a proper family identifier the next helper will return the font currently associated with it. You can normally also access the font with the normal font field or getter because it will resolve the family automatically for noads.

```
<integer> id =
    node.family_font(<integer> fam)
```

8.2.3.10 node.set_synctex_fields and node.get_synctex_fields

You can set and query the synctex fields, a file number aka tag and a line number, for a glue, kern, hlist, vlist, rule and math nodes as well as glyph nodes (although this last one is not used in native synctex).

```
node.set_synctex_fields(<integer> f, <integer> l)
<integer> f, <integer> l =
    node.get_synctex_fields(<node> n)
```

Of course you need to know what you're doing as no checking on sane values takes place. Also, the synctex interpreter used in editors is rather peculiar and has some assumptions (heuristics).

8.3 Two access models

Deep down in TEX a node has a number which is a numeric 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 too. Each node gets a number that is in fact an index in the memory table and that number often is reported when you print node related information.

There are two access models, a robust one using a so called user data object that provides a virtual interface to the internal nodes, and a more direct access which uses the node numbers directly. The first model provides key based access while the second always accesses fields via functions:

```
nodeobject.char
getfield(nodenumber,"char")
```

If you use the direct model, even if you know that you deal with numbers, you should not depend on that property but treat it as 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, multiplying a node number makes no sense.

So our advice is: use the indexed (table) approach when possible and investigate the direct one when speed might be a real issue. For that reason LuaTEX also provide the `get*` and `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 nodes are accessed 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:

```
local next = current.next
if next then
    -- do something
end
```

Here `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 `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
```

Some accessors are used frequently and for these we provide more efficient helpers:

FUNCTION	EXPLANATION
`getnext`	parsing nodelist always involves this one

`getprev`	used less but a logical companion to `getnext`
`getboth`	returns the next and prev pointer of a node
`getid`	consulted a lot
`getsubtype`	consulted less but also a topper
`getfont`	used a lot in OpenType handling (glyph nodes are consulted a lot)
`getchar`	idem and also in other places
`getwhd`	returns the `width`, `height` and `depth` of a list, rule or (unexpanded) glyph as well as glue (its spec is looked at) and unset nodes
`getdisc`	returns the `pre`, `post` and `replace` fields and optionally when true is passed also the tail fields
`getlist`	we often parse nested lists so this is a convenient one too
`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)
`getbox`	gets the given box (a list node)

In the direct namespace there are more such helpers and most of them are accompanied by setters. The getters and setters are clever enough to see what node is meant. We don't deal with whatsit nodes: their fields are always accessed by name. It doesn't make sense to add getters for all fields, we just identifier the most likely candidates. In complex documents, many node and fields types never get seen, or seen only a few times, but for instance glyphs are candidates for such optimization. The `node.direct` interface has some more helpers.[2]

The `setdisc` helper takes three (optional) arguments plus an optional fourth indicating the subtype. Its `getdisc` takes an optional boolean; when its value is `true` the tail nodes will also be returned. The `setfont` helper takes an optional second argument, it being the character. The directmode setter `setlink` takes a list of nodes and will link them, thereby ignoring `nil` entries. The first valid node is returned (beware: for good reason it assumes single nodes). For rarely used fields no helpers are provided and there are a few that probably are used seldom too but were added for consistency. You can of course always define additional accessors using `getfield` and `setfield` with little overhead.

FUNCTION	NODE	DIRECT
`check_discretionaries`	+	+
`copy_list`	+	+
`copy`	+	+
`count`	+	+
`current_attr`	+	+
`dimensions`	+	+
`effective_glue`	+	+
`end_of_math`	+	+
`family_font`	+	−
`fields`	+	−
`find_attribute`	+	+

[2] We can define the helpers in the node namespace with `getfield` which is about as efficient, so at some point we might provide that as module.

first_glyph	+	+
flush_list	+	+
flush_node	+	+
free	+	+
get_attribute	+	+
getattributelist	−	+
getboth	+	+
getbox	−	+
getchar	+	+
getcomponents	−	+
getdepth	−	+
getdir	−	+
getdisc	+	+
getfam	−	+
getfield	+	+
getfont	+	+
getglue	+	+
getheight	−	+
getid	+	+
getkern	−	+
getlang	−	+
getleader	+	+
getlist	+	+
getnext	+	+
getnucleus	−	+
getoffsets	−	+
getpenalty	−	+
getprev	+	+
getproperty	+	+
getshift	−	+
getwidth	−	+
getwhd	−	+
getsub	−	+
getsubtype	+	+
getsup	−	+
has_attribute	+	+
has_field	+	+
has_glyph	+	+
hpack	+	+
id	+	−
insert_after	+	+
insert_before	+	+
is_char	+	+
is_direct	−	+
is_glue_zero	+	+
is_glyph	+	+

is_node	+	+
kerning	+	+
last_node	+	+
length	+	+
ligaturing	+	+
mlist_to_hlist	+	−
new	+	+
next	+	−
prev	+	−
protect_glyphs	+	+
protect_glyph	+	+
protrusion_skippable	+	+
rangedimensions	+	+
remove	+	+
set_attribute	−	+
setattributelist	−	+
setboth	−	+
setbox	−	+
setchar	−	+
setcomponents	−	+
setdepth	−	+
setdir	−	+
setdisc	−	+
setfield	+	+
setfont	−	+
setglue	+	+
setheight	−	+
setid	−	+
setkern	−	+
setlang	−	+
setleader	−	+
setlist	−	+
setnext	−	+
setnucleus	−	+
setoffsets	−	+
setpenalty	−	+
setprev	−	+
setproperty	−	+
setshift	−	+
setwidth	−	+
setwhd	−	+
setsub	−	+
setsubtype	−	+
setsup	−	+
slide	+	+
subtypes	+	−

subtype	+	−
tail	+	+
todirect	+	+
tonode	+	+
tostring	+	+
traverse_char	+	+
traverse_id	+	+
traverse	+	+
types	+	−
type	+	−
unprotect_glyphs	+	+
unset_attribute	+	+
usedlist	+	+
uses_font	+	+
vpack	+	+
whatsitsubtypes	+	−
whatsits	+	−
write	+	+
set_synctex_fields	+	+
get_synctex_fields	+	+

The `node.next` and `node.prev` functions will stay but for consistency there are variants called `getnext` 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.

Some nodes have indirect references. For instance a math character refers to a family instead of a font. In that case we provide a virtual font field as accessor. So, `getfont` and `.font` can be used on them. The same is true for the `width`, `height` and `depth` of glue nodes. These actually access the spec node properties, and here we can set as well as get the values.

156 Nodes

9 LUA callbacks

9.1 Registering callbacks

This library has functions that register, find and list callbacks. Callbacks are Lua functions that are called in well defined places. There are two kind of callbacks: those that mix with existing functionality, and those that (when enabled) replace functionality. In mosty cases the second category is expected to behave similar to the built in functionality because in a next step specific data is expected. For instance, you can replace the hyphenation routine. The function gets a list that can be hyphenated (or not). The final list should be valid and is (normally) used for constructing a paragraph. Another function can replace the ligature builder and/or kerner. Doing something else is possible but in the end might not give the user the expected outcome.

The first thing you need to do is registering a callback:

```
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)
```

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

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

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

For some minor speed gain, you can assign the boolean `false` to the non-file related callbacks, doing so will prevent LuaTEX from executing whatever it would execute by default (when no callback function is registered at all). Be warned: this may cause all sorts of grief unless you know *exactly* what you are doing!

```
<table> info =
    callback.list()
```

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

```
<function> f = callback.find (callback_name)
```

If the callback is not set, `callback.find` returns `nil`.

9.2 File discovery callbacks

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

9.2.1 find_read_file and find_write_file

Your callback function should have the following conventions:

```
<string> actual_name =
    function (<number> id_number, <string> asked_name)
```

Arguments:

id_number
: This number is zero for the log or `\input` files. For TEX's `\read` or `\write` the number is incremented by one, so `\read0` becomes 1.

asked_name
: This is the user-supplied filename, as found by `\input`, `\openin` or `\openout`.

Return value:

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 `actual_name` that has an extension and that is suitable for use as `jobname`. If you don't, you will have to manually fix the name of the log file and output file after LuaTEX is finished, and an eventual format filename will become mangled. That is because these file names depend on the jobname.
  You have to return `nil` if the file cannot be found.

9.2.2 find_font_file

Your callback function should have the following conventions:

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

The `asked_name` is an otf or tfm font metrics file.

Return `nil` if the file cannot be found.

9.2.3 find_output_file

Your callback function should have the following conventions:

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

The `asked_name` is the pdf or dvi file for writing.

9.2.4 find_format_file

Your callback function should have the following conventions:

```
<string> actual_name =
```

```
    function (<string> asked_name)
```

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

9.2.5 find_vf_file

Like `find_font_file`, but for virtual fonts. This applies to both Aleph's ovf files and traditional Knuthian vf files.

9.2.6 find_map_file

Like `find_font_file`, but for map files.

9.2.7 find_enc_file

Like `find_font_file`, but for enc files.

9.2.8 find_pk_file

Like `find_font_file`, but for pk bitmap files. This callback takes two arguments: `name` and `dpi`. In your callback you can decide to look for:

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

but other strategies are possible. It is up to you to find a 'reasonable' bitmap file to go with that specification.

9.2.9 find_data_file

Like `find_font_file`, but for embedded files (`\pdfobj file '...'`).

9.2.10 find_opentype_file

Like `find_font_file`, but for OpenType font files.

9.2.11 find_truetype_file and find_type1_file

Your callback function should have the following conventions:

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

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

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

9.2.12 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 hard disk, so make sure you return something that is considered the name of a valid file by your operating system.

9.2.13 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.

9.2.14 open_read_file

Your callback function should have the following conventions:

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

Argument:

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

Return value:

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

9.2.14.1 reader

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

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

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

9.2.14.2 `close`

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

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

Your function should not return any value.

9.2.15 General file readers

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

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

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

success
: Return `false` when a fatal error occurred (e.g. when the file cannot be found, after all).

data
: The bytes comprising the file.

data_size
: The length of the `data`, in bytes.

Return an empty string and zero if the file was found but there was a reading problem.

The list of functions is:

FUNCTION	USAGE
`read_font_file`	ofm or tfm files
`read_vf_file`	virtual fonts
`read_map_file`	map files
`read_enc_file`	encoding files
`read_pk_file`	pk bitmap files
`read_data_file`	embedded files (as is possible with pdf objects)
`read_truetype_file`	TrueType font files
`read_type1_file`	Type1 font files
`read_opentype_file`	OpenType font files

9.3 Data processing callbacks

9.3.1 process_input_buffer

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

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

If you return `nil`, LuaTEX 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.

9.3.2 process_output_buffer

This callback allows you to change the contents of the line output buffer just before LuaTEX 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 so called `\write` 18 calls).

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

If you return `nil`, LuaTEX 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.

9.3.3 process_jobname

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`, LuaTEX will pretend your callback never happened. This callback does not replace any internal code.

9.4 Node list processing callbacks

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

9.4.1 contribute_filter

This callback is called when LuaTEX adds contents to list:

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

The string reports the group code. From this you can deduce from what list you can give a treat.

VALUE	EXPLANATION
`pre_box`	interline material is being added
`pre_adjust`	`\vadjust` material is being added
`box`	a typeset box is being added (always called)
`adjust`	`\vadjust` material is being added

9.4.2 buildpage_filter

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

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

The string `extrainfo` gives some additional information about what TEX's state is with respect to the 'current page'. The possible values for the `buildpage_filter` callback are:

VALUE	EXPLANATION
`alignment`	a (partial) alignment is being added
`after_output`	an output routine has just finished
`new_graf`	the beginning of a new paragraph
`vmode_par`	`\par` was found in vertical mode
`hmode_par`	`\par` was found in horizontal mode
`insert`	an insert is added
`penalty`	a penalty (in vertical mode)
`before_display`	immediately before a display starts
`after_display`	a display is finished
`end`	LuaTEX is terminating (it's all over)

9.4.3 build_page_insert

This callback is called when the pagebuilder adds an insert. There is not much control over this mechanism but this callback permits some last minute manipulations of the spacing before an insert, something that might be handy when for instance multiple inserts (types) are appended in a row.

```
function(<number> n, <number> i)
    return <number> register
end
```

with

VALUE	EXPLANATION
n	the insert class
i	the order of the insert

The return value is a number indicating the skip register to use for the prepended spacing. This permits for instance a different top space (when `i` equals one) and intermediate space (when `i` is larger than one). Of course you can mess with the insert box but you need to make sure that LuaTEX is happy afterwards.

9.4.4 pre_linebreak_filter

This callback is called just before LuaTEX starts converting a list of nodes into a stack of `\hboxes`, after the addition of `\parfillskip`.

```
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.

VALUE	EXPLANATION
<empty>	main vertical list
hbox	`\hbox` in horizontal mode
adjusted_hbox	`\hbox` in vertical mode
vbox	`\vbox`
vtop	`\vtop`
align	`\halign` or `\valign`
disc	discretionaries
insert	packaging an insert
vcenter	`\vcenter`
local_box	`\localleftbox` or `\localrightbox`
split_off	top of a `\vsplit`
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
- <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.

9.4.5 linebreak_filter

This callback replaces LuaTEX's line breaking algorithm.

```
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 <node>, LuaTEX will apply the internal linebreak algorithm on the list that starts at <head>. Otherwise, the <node> you return is supposed to be the head of a list of nodes that are all allowed in vertical mode, and at least one of those has to represent a hbox. Failure to do so will result in a fatal error.

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

9.4.6 append_to_vlist_filter

This callback is called whenever LuaTEX adds a box to a vertical list:

```
function(<node> box, <string> locationcode, <number prevdepth>,
    <boolean> mirrored)
    return list, prevdepth
end
```

It is ok to return nothing in which case you also need to flush the box or deal with it yourself. The prevdepth is also optional. Locations are `box`, `alignment`, `equation`, `equation_number` and `post_linebreak`.

9.4.7 post_linebreak_filter

This callback is called just after LuaTEX has converted a list of nodes into a stack of `\hbox`es.

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

This callback does not replace any internal code.

9.4.8 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.

```
function(<node> head, <string> groupcode, <number> size,
         <string> packtype [, <string> direction] [, <node> attributelist])
```


```
    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`.

This callback does not replace any internal code.

9.4.9 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.

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

This callback does not replace any internal code.

9.4.10 hpack_quality

This callback can be used to intercept the overfull messages that can result from packing a horizontal list (as happens in the par builder). The function takes a few arguments:

```
function(<string> incident, <number> detail, <node> head, <number> first,
         <number> last)
    return <node> whatever
end
```

The incident is one of `overfull`, `underfull`, `loose` or `tight`. The detail is either the amount of overflow in case of `overfull`, or the badness otherwise. The head is the list that is constructed (when protrusion or expansion is enabled, this is an intermediate list). Optionally you can return a node, for instance an overfull rule indicator. That node will be appended to the list (just like TEX's own rule would).

9.4.11 vpack_quality

This callback can be used to intercept the overfull messages that can result from packing a vertical list (as happens in the page builder). The function takes a few arguments:

```
function(<string> incident, <number> detail, <node> head, <number> first,
         <number> last)
```

```
end
```

The incident is one of `overfull`, `underfull`, `loose` or `tight`. The detail is either the amount of overflow in case of `overfull`, or the badness otherwise. The head is the list that is constructed.

9.4.12 process_rule

This is an experimental callback. It can be used with rules of subtype 4 (user). The callback gets three arguments: the node, the width and the height. The callback can use `pdf.print` to write code to the pdf file but beware of not messing up the final result. No checking is done.

9.4.13 pre_output_filter

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

```
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.

9.4.14 hyphenate

```
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.

9.4.15 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.

You must not ruin the node list. For instance, the head normally is a local par node, and the tail a glue. Messing too much can push LuaTEX into panic mode.

9.4.16 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.

You must not ruin the node list. For instance, the head normally is a local par node, and the tail a glue. Messing too much can push LuaTEX into panic mode.

9.4.17 insert_local_par

Each paragraph starts with a local par node that keeps track of for instance the direction. You can hook a callback into the creator:

```
function(<node> local_par, <string> location)
end
```

There is no return value and you should make sure that the node stays valid as otherwise TEX can get confused.

9.4.18 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.

9.5 Information reporting callbacks

9.5.1 pre_dump

```
function()
end
```

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

9.5.2 start_run

```
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.

9.5.3 stop_run

```
function()
end
```

This callback replaces the code that prints LuaTEX's statistics and 'output written to' messages. The engine can still do housekeeping and therefore you should not rely on this hook for postprocessing the pdf or log file.

9.5.4 start_page_number

```
function()
end
```

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

9.5.5 stop_page_number

```
function()
end
```

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

9.5.6 show_error_hook

```
function()
end
```

This callback is run from inside the TEX error function, and the idea is to allow you to do some extra reporting on top of what TEX already does (none of the normal actions are removed). You may find some of the values in the `status` table useful. This callback does not replace any internal code.

9.5.7 show_error_message

```
function()
end
```

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

9.5.8 show_lua_error_hook

```
function()
end
```

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

9.5.9 start_file

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

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

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

9.5.10 stop_file

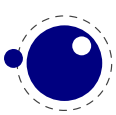
```
function(category)
end
```

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

9.5.11 call_edit

```
function(filename,linenumber)
end
```

This callback replaces the call to an external editor when 'E' is pressed in reply to an error message. Processing will end immediately after the callback returns control to the main program.



9.5.12 finish_synctex

This callback can be used to wrap up alternative synctex methods. It kicks in after the normal synctex finalizer (that happens to remove the synctex files after a run when native synctex is not enabled).

9.5.13 wrapup_run

This callback is called after the pdf and log files are closed. Use it at your own risk.

9.6 PDF-related callbacks

9.6.1 finish_pdffile

```
function()
end
```

This callback is called when all document pages are already written to the pdf file and LuaTEX 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.

9.6.2 finish_pdfpage

```
function(shippingout)
end
```

This callback is called after the pdf page stream has been assembled and before the page object gets finalized.

9.7 Font-related callbacks

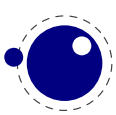
9.7.1 define_font

```
function(<string> name, <number> size, <number> id)
    return <table> font | <number> id
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 6. That table is saved internally, so you can put extra fields in the table for your later Lua code to use. In alternative, `retval` can be a previously defined fontid. This is useful if a previous definition can be reused instead of creating a whole new font structure.

Setting this callback to `false` is pointless as it will prevent font loading completely but will nevertheless generate errors.

9.7.2 glyph_not_found

This callback kicks in when the backend cannot insert a glyph. When no callback is defined a message is written to the log.

```
function(<number> id, <number> char)
    -- do something with font id and char code
end
```

10 The TEX related libraries

10.1 The lua library

10.1.1 LUA version

This library contains one read-only item:

```
<string> s = lua.version
```

This returns the Lua version identifier string. The value is currently Lua 5.3.

10.1.2 LUA bytecode registers

Lua registers can be used to store Lua code 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. The function must not contain any upvalues. 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 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 iniTEX.

10.1.3 LUA chunk name registers

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

```
lua.name[<number> n] = <string> s
<string> s = lua.name[<number> n]
```

If you want to unset a Lua name, you can assign `nil` to it.

10.2 The `status` library

This contains a number of run-time configuration items that you may find useful in message reporting, as well as an iterator function that gets all of the names and values as a table.

```
<table> info = status.list()
```

The keys in the table are the known items, the value is the current value. Almost all of the values in `status` are fetched through a metatable at run-time whenever they are accessed, so you cannot use `pairs` on `status`, but you *can* use `pairs` on `info`, of course. If you do not need the full list, you can also ask for a single item by using its name as an index into `status`.

The current list is:

KEY	EXPLANATION
`banner`	terminal display banner
`best_page_break`	the current best break (a node)
`buf_size`	current allocated size of the line buffer
`callbacks`	total number of executed callbacks so far
`cs_count`	number of control sequences
`dest_names_size`	pdf destination table size
`dvi_gone`	written dvi bytes
`dvi_ptr`	not yet written dvi bytes
`dyn_used`	token (multi-word) memory in use
`filename`	name of the current input file
`fix_mem_end`	maximum number of used tokens
`fix_mem_min`	minimum number of allocated words for tokens
`fix_mem_max`	maximum number of allocated words for tokens
`font_ptr`	number of active fonts
`hash_extra`	extra allowed hash
`hash_size`	size of hash
`indirect_callbacks`	number of those that were themselves a result of other callbacks (e.g. file readers)
`ini_version`	`true` if this is an iniTEX run
`init_pool_ptr`	iniTEX string pool index
`init_str_ptr`	number of iniTEX strings
`input_ptr`	the level of input we're at
`inputid`	numeric id of the current input
`largest_used_mark`	max referenced marks class
`lasterrorcontext`	last error context string (with newlines)
`lasterrorstring`	last TEX error string
`lastluaerrorstring`	last Lua error string
`lastwarningstring`	last warning tag, normally an indication of in what part

`lastwarningtag`	last warning string
`linenumber`	location in the current input file
`log_name`	name of the log file
`luabytecode_bytes`	number of bytes in Lua bytecode registers
`luabytecodes`	number of active Lua bytecode registers
`luastate_bytes`	number of bytes in use by Lua interpreters
`luatex_engine`	the LuaTEX engine identifier
`luatex_hashchars`	length to which Lua hashes strings (2^n)
`luatex_hashtype`	the hash method used (in LuajitTEX)
`luatex_version`	the LuaTEX version number
`luatex_revision`	the LuaTEX revision string
`max_buf_stack`	max used buffer position
`max_in_stack`	max used input stack entries
`max_nest_stack`	max used nesting stack entries
`max_param_stack`	max used parameter stack entries
`max_save_stack`	max used save stack entries
`max_strings`	maximum allowed strings
`nest_size`	nesting stack size
`node_mem_usage`	a string giving insight into currently used nodes
`obj_ptr`	max pdf object pointer
`obj_tab_size`	pdf object table size
`output_active`	`true` if the `\output` routine is active
`output_file_name`	name of the pdf or dvi file
`param_size`	parameter stack size
`pdf_dest_names_ptr`	max pdf destination pointer
`pdf_gone`	written pdf bytes
`pdf_mem_ptr`	max pdf memory used
`pdf_mem_size`	pdf memory size
`pdf_os_cntr`	max pdf object stream pointer
`pdf_os_objidx`	pdf object stream index
`pdf_ptr`	not yet written pdf bytes
`pool_ptr`	string pool index
`pool_size`	current size allocated for string characters
`save_size`	save stack size
`shell_escape`	`0` means disabled, `1` means anything is permitted, and `2` is restricted
`safer_option`	`1` means safer is enforced
`kpse_used`	`1` means that kpse is used
`stack_size`	input stack size
`str_ptr`	number of strings
`total_pages`	number of written pages
`var_mem_max`	number of allocated words for nodes
`var_used`	variable (one-word) memory in use
`lc_collate`	the value of `LC_COLLATE` at startup time (becomes `C` at startup)
`lc_ctype`	the value of `LC_CTYPE` at startup time (becomes `C` at startup)
`lc_numeric`	the value of `LC_NUMERIC` at startup time

The error and warning messages can be wiped with the `resetmessages` function. A return value

can be set with setexitcode.

10.3 The tex library

The tex table contains a large list of virtual internal TEX parameters that are partially writable.

The designation 'virtual' means that these items are not properly defined in Lua, but are only frontends that are handled by a metatable that operates on the actual TEX values. As a result, most of the Lua table operators (like pairs and #) do not work on such items.

At the moment, it is possible to access almost every parameter that you can use after \the, is a single tokens or is sort of special in TEX. This excludes parameters that need extra arguments, like \the\scriptfont. The subset comprising simple integer and dimension registers are writable as well as readable (like \tracingcommands and \parindent).

10.3.1 Internal parameter values

For all the parameters in this section, it is possible to access them directly using their names as index in the tex table, or by using one of the functions tex.get and tex.set.

The exact parameters and return values differ depending on the actual parameter, and so does whether tex.set has any effect. For the parameters that *can* be set, it is possible to use global as the first argument to tex.set; this makes the assignment global instead of local.

```
tex.set (["global",] <string> n, ...)
... = tex.get (<string> n)
```

Glue is kind of special because there are five values involved. The return value is a glue_spec node but when you pass false as last argument to tex.get you get the width of the glue and when you pass true you get all five values. Otherwise you get a node which is a copy of the internal value so you are responsible for its freeing at the Lua end. When you set a glue quantity you can either pass a glue_spec or upto five numbers. If you pass true to get you get 5 values returned for a glue and when you pass false you only get the width returned.

For the registers you can use getskip (node), getglue (numbers) setskip (node) and setglue (numbers). If you pass false as second argument to getglue you only get the width returned.

There are also dedicated setters, getters and checkers:

```
local d = tex.getdimen("foo")
if tex.isdimen("bar") then
    tex.setdimen("bar",d)
end
```

There are such helpers for dimen, count, skip, box and attribute registers.

10.3.1.1 Integer parameters

The integer parameters accept and return Lua numbers.

Read-write:

```
tex.adjdemerits
tex.binoppenalty
tex.brokenpenalty
tex.catcodetable
tex.clubpenalty
tex.day
tex.defaulthyphenchar
tex.defaultskewchar
tex.delimiterfactor
tex.displaywidowpenalty
tex.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
tex.mag
tex.maxdeadcycles
tex.month
tex.newlinechar
tex.outputpenalty
tex.pausing
tex.postdisplaypenalty
tex.predisplaydirection
tex.predisplaypenalty
tex.pretolerance
tex.relpenalty
tex.righthyphenmin
tex.savinghyphcodes
tex.savingvdiscards
tex.showboxbreadth
tex.showboxdepth
tex.time
tex.tolerance
tex.tracingassigns
tex.tracingcommands
tex.tracinggroups
tex.tracingifs
tex.tracinglostchars
tex.tracingmacros
tex.tracingnesting
tex.tracingonline
tex.tracingoutput
tex.tracingpages
tex.tracingparagraphs
tex.tracingrestores
tex.tracingscantokens
tex.tracingstats
tex.uchyph
tex.vbadness
tex.widowpenalty
tex.year
```

Read-only:

```
tex.deadcycles          tex.parshape            tex.spacefactor
tex.insertpenalties     tex.prevgraf
```

10.3.1.2 Dimension parameters

The dimension parameters accept Lua numbers (signifying scaled points) or strings (with included dimension). The result is always a number in scaled points.

Read-write:

```
tex.boxmaxdepth             tex.mathsurround           tex.parindent
tex.delimitershortfall      tex.maxdepth               tex.predisplaysize
tex.displayindent           tex.nulldelimiterspace     tex.scriptspace
tex.displaywidth            tex.overfullrule           tex.splitmaxdepth
tex.emergencystretch        tex.pagebottomoffset       tex.vfuzz
tex.hangindent              tex.pageheight             tex.voffset
tex.hfuzz                   tex.pageleftoffset         tex.vsize
tex.hoffset                 tex.pagerightoffset        tex.prevdepth
tex.hsize                   tex.pagetopoffset          tex.prevgraf
tex.lineskiplimit           tex.pagewidth              tex.spacefactor
```

Read-only:

```
tex.pagedepth               tex.pagefilstretch         tex.pagestretch
tex.pagefilllstretch        tex.pagegoal               tex.pagetotal
tex.pagefillstretch         tex.pageshrink
```

Beware: as with all Lua tables you can add values to them. So, the following is valid:

```
tex.foo = 123
```

When you access a TEX parameter a look up takes place. For read–only variables that means that you will get something back, but when you set them you create a new entry in the table thereby making the original invisible.

There are a few special cases that we make an exception for: `prevdepth`, `prevgraf` and `spacefactor`. These normally are accessed via the `tex.nest` table:

```
tex.nest[tex.nest.ptr].prevdepth   = p
tex.nest[tex.nest.ptr].spacefactor = s
```

However, the following also works:

```
tex.prevdepth   = p
tex.spacefactor = s
```

Keep in mind that when you mess with node lists directly at the Lua end you might need to update the top of the nesting stack's `prevdepth` explicitly as there is no way LuaTEX can guess your intentions. By using the accessor in the `tex` tables, you get and set the values at the top of the nesting stack.

10.3.1.3 Direction parameters

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

```
tex.bodydir                 tex.pagedir                tex.textdir
tex.mathdir                 tex.pardir
```

10.3.1.4 Glue parameters

The glue parameters accept and return a userdata object that represents a `glue_spec` node.

```
tex.abovedisplayshortskip   tex.leftskip            tex.spaceskip
tex.abovedisplayskip        tex.lineskip            tex.splittopskip
tex.baselineskip            tex.parfillskip         tex.tabskip
tex.belowdisplayshortskip   tex.parskip             tex.topskip
tex.belowdisplayskip        tex.rightskip           tex.xspaceskip
```

10.3.1.5 Muglue parameters

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

```
tex.medmuskip               tex.thickmuskip         tex.thinmuskip
```

10.3.1.6 Tokenlist parameters

The tokenlist parameters accept and return Lua strings. Lua strings are converted to and from token lists using `\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.

```
tex.errhelp                 tex.everyhbox           tex.everyvbox
tex.everycr                 tex.everyjob            tex.output
tex.everydisplay            tex.everymath
tex.everyeof                tex.everypar
```

10.3.2 Convert commands

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

```
tex.eTeXVersion                     tex.fontname(number)
tex.eTeXrevision                    tex.uniformdeviate(number)
tex.formatname                      tex.number(number)
tex.jobname                         tex.romannumeral(number)
tex.luatexbanner                    tex.fontidentifier(number)
tex.luatexrevision
```

If you are wondering why this list looks haphazard; these are all the cases of the 'convert' internal command that do not require an argument, as well as the ones that require only a simple numeric value. The special (Lua-only) case of `tex.fontidentifier` returns the `csname` string that matches a font id number (if there is one).

10.3.3 Last item commands

All 'last item' commands are read-only and return a number. The supported commands at this moment are:

```
tex.lastpenalty          tex.lastypos             tex.currentgrouptype
tex.lastkern             tex.randomseed           tex.currentiflevel
tex.lastskip             tex.luatexversion        tex.currentiftype
tex.lastnodetype         tex.eTeXminorversion     tex.currentifbranch
tex.inputlineno          tex.eTeXversion
tex.lastxpos             tex.currentgrouplevel
```

10.3.4 Attribute, count, dimension, skip and token registers

TEX's attributes (`\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:

```
tex.attribute            tex.dimen                tex.toks
tex.count                tex.skip
```

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

```
tex.count.scratchcounter = 0
enormous = tex.dimen['maxdimen']
```

In this case, LuaTEX 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; `em` and `ex` and `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 `\the \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:

```
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)
```

We have similar setters for `count`, `dimen`, `muskip`, and `toks`. Counters and dimen are represented by numbers, skips and muskips by nodes, and toks by strings. For tokens registers we have an alternative where a catcode table is specified:

```
tex.scantoks(0,3,"$e=mc^2$")
tex.scantoks("global",0,"$\int\limits^1_2$")
```

In the function-based interface, it is possible to define values globally by using the string `global` as the first function argument.

There are four extra skip related helpers:

```
tex.setglue (["global"], <number> n,
    width, stretch, shrink, stretch_order, shrink_order)
tex.setglue (["global"], <string> s,
    width, stretch, shrink, stretch_order, shrink_order)
width, stretch, shrink, stretch_order, shrink_order =
    tex.getglue (<number> n)
width, stretch, shrink, stretch_order, shrink_order =
    tex.getglue (<string> s)
```

The other two are `tex.setmuglue` and `tex.getmuglue`.

10.3.5 Character code registers

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

```
tex.lccode                  tex.sfcode                  tex.mathcode
tex.uccode                  tex.catcode                 tex.delcode
```

The function call interfaces are roughly as above, but there are a few twists. `sfcodes` are the simple ones:

```
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)
tex.setuccode (["global"], <number> n, <number> uc)
tex.setuccode (["global"], <number> n, <number> uc, <number> lc)
<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> class,
    <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
}
```

You can also avoid the table:

```
tex.setmathcode (["global"], <number> n, <number> class,
    <number> family, <number> character)
class, family, char =
    tex.getmathcodes (<number> n)
tex.setdelcode (["global"], <number> n, <number> smallfam,
    <number> smallchar, <number> largefam, <number> largechar)
smallfam, smallchar, largefam, largechar =
    tex.getdelcodes (<number> n)
```

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.

10.3.6 Box registers

It is possible to set and query actual boxes, coming for instance from `\hbox`, `\vbox` or `\vtop`, using the node interface as defined in the `node` library:

```
tex.box
```

for array access, or

```
tex.setbox(["global",] <number> n, <node> s)
tex.setbox(["global",] <string> cs, <node> s)
<node> n = tex.getbox(<number> n)
<node> n = tex.getbox(<string> cs)
```

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

```
tex.box[0] = tex.box[2]
```

does not copy the node list, it just duplicates a node pointer. If `\box2` will be cleared by TEX commands later on, the contents of `\box0` becomes invalid as well. To prevent this from happening, always use `node.copy_list()` unless you are assigning to a temporary variable:

```
tex.box[0] = node.copy_list(tex.box[2])
```

The following function will register a box for reuse (this is modelled after so called xforms in pdf). You can (re)use the box with `\useboxresource` or by creating a rule node with subtype 2.

```
local index = tex.saveboxresource(n,attributes,resources,immediate,type,margin)
```

The optional second and third arguments are strings, the fourth is a boolean. The fifth argument is a type. When set to non-zero the `/Type` entry is omitted. A value of 1 or 3 still writes a `/BBox`, while 2 or 3 will write a `/Matrix`. The sixth argument is the (virtual) margin that extends beyond the effective boundingbox as seen by TEX.

You can generate the reference (a rule type) with:

```
local reused = tex.useboxresource(n,wd,ht,dp)
```

The dimensions are optional and the final ones are returned as extra values. The following is just a bonus (no dimensions returned means that the resource is unknown):

```
local w, h, d, m = tex.getboxresourcedimensions(n)
```

This returns the width, height, depth and margin of the resource.

You can split a box:

```
local vlist = tex.splitbox(n,height,mode)
```

The remainder is kept in the original box and a packaged vlist is returned. This operation is comparable to the `\vsplit` operation. The mode can be `additional` or `exactly` and concerns the split off box.

10.3.7 Math parameters

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

```
tex.setmath(["global",] <string> n, <string> t, <number> n)
<number> n = 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:

```
quad                 axis                 operatorsize
overbarkern          overbarrule          overbarvgap
underbarkern         underbarrule         underbarvgap
radicalkern          radicalrule          radicalvgap
radicaldegreebefore  radicaldegreeafter   radicaldegreeraise
stackvgap            stacknumup           stackdenomdown
fractionrule         fractionnumvgap      fractionnumup
fractiondenomvgap    fractiondenomdown    fractiondelsize
limitabovevgap       limitabovebgap       limitabovekern
limitbelowvgap       limitbelowbgap       limitbelowkern
underdelimitervgap   underdelimiterbgap
overdelimitervgap    overdelimiterbgap
subshiftdrop         supshiftdrop         subshiftdown
subsupshiftdown      subtopmax            supshiftup
supbottommin         supsubbottommax      subsupvgap
spaceafterscript     connectoroverlapmin
ordordspacing        ordopspacing         ordbinspacing     ordrelspacing
ordopenspacing       ordclosespacing      ordpunctspacing   ordinnerspacing
opordspacing         opopspacing          opbinspacing      oprelspacing
opopenspacing        opclosespacing       oppunctspacing    opinnerspacing
binordspacing        binopspacing         binbinspacing     binrelspacing
binopenspacing       binclosespacing      binpunctspacing   bininnerspacing
relordspacing        relopspacing         relbinspacing     relrelspacing
relopenspacing       relclosespacing      relpunctspacing   relinnerspacing
openordspacing       openopspacing        openbinspacing    openrelspacing
openopenspacing      openclosespacing     openpunctspacing  openinnerspacing
closeordspacing      closeopspacing       closebinspacing   closerelspacing
closeopenspacing     closeclosespacing    closepunctspacing closeinnerspacing
punctordspacing      punctopspacing       punctbinspacing   punctrelspacing
punctopenspacing     punctclosespacing    punctpunctspacing punctinnerspacing
innerordspacing      inneropspacing       innerbinspacing   innerrelspacing
inneropenspacing     innerclosespacing    innerpunctspacing innerinnerspacing
```

The values for the style parameter are:

```
display       crampeddisplay
text          crampedtext
script        crampedscript
scriptscript  crampedscriptscript
```

The value is either a number (representing a dimension or number) or a glue spec node representing a muskip for `ordordspacing` and similar spacing parameters.

10.3.8 Special list heads

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

FIELD	EXPLANATION
`page_ins_head`	circular list of pending insertions
`contrib_head`	the recent contributions
`page_head`	the current page content
`hold_head`	used for held-over items for next page
`adjust_head`	head of the current `\vadjust` list
`pre_adjust_head`	head of the current `\vadjust pre` list
`page_discards_head`	head of the discarded items of a page break
`split_discards_head`	head of the discarded items in a vsplit

10.3.9 Semantic nest levels

The virtual table `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 `tex.nest.ptr`, which gives the highest available index. Neither the array items in `tex.nest[]` nor `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. `tex.nest[tex.nest.ptr].prevdepth`.

`tex.nest[tex.nest.ptr]` is the current nest state, `tex.nest[0]` the outermost (main vertical list) level.

The known fields are:

KEY	TYPE	MODES	EXPLANATION
`mode`	number	all	a number representing the main mode at this level: `0` = no mode (this happens during `\write`), `1` = vertical, `127` = horizontal, `253` = display math, `-1` = internal vertical, `-127` = restricted horizontal, `-253` = inline math
`modeline`	number	all	source input line where this mode was entered in, negative inside the output routine
`head`	node	all	the head of the current list
`tail`	node	all	the tail of the current list
`prevgraf`	number	vmode	number of lines in the previous paragraph
`prevdepth`	number	vmode	depth of the previous paragraph
`spacefactor`	number	hmode	the current space factor
`dirs`	node	hmode	used for temporary storage by the line break algorithm
`noad`	node	mmode	used for temporary storage of a pending fraction numerator, for `\over` etc.

delimptr	node	mmode	used for temporary storage of the previous math delimiter, for \middle
mathdir	boolean	mmode	true when during math processing the \mathdir is not the same as the surrounding \textdir
mathstyle	number	mmode	the current \mathstyle

10.3.10 Print functions

The 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.

10.3.10.1 tex.print

```
tex.print(<string> s, ...)
tex.print(<number> n, <string> s, ...)
tex.print(<table> t)
tex.print(<number> n, <table> t)
```

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).

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.

10.3.10.2 tex.sprint

```
tex.sprint(<string> s, ...)
tex.sprint(<number> n, <string> s, ...)
tex.sprint(<table> t)
tex.sprint(<number> n, <table> t)
```

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

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

the space before `in between` 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).

The optional argument sets the catcode regime, as with `tex.print()`. This influences the string arguments (or numbers turned into strings).

Although this needs to be used with care, you can also pass token or node userdata objects. These get injected into the stream. Tokens had best be valid tokens, while nodes need to be around when they get injected. Therefore it is important to realize the following:

- When you inject a token, you need to pass a valid token userdata object. This object will be collected by Lua when it no longer is referenced. When it gets printed to TEX the token itself gets copied so there is no interference with the Lua garbage collection. You manage the object yourself. Because tokens are actually just numbers, there is no real extra overhead at the TEX end.
- When you inject a node, you need to pass a valid node userdata object. The node related to the object will not be collected by Lua when it no longer is referenced. It lives on at the TEX end in its own memory space. When it gets printed to TEX the node reference is used assuming that node stays around. There is no Lua garbage collection involved. Again, you manage the object yourself. The node itself is freed when TEX is done with it.

If you consider the last remark you might realize that we have a problem when a printed mix of strings, tokens and nodes is reused. Inside TEX the sequence becomes a linked list of input buffers. So, `"123"` or `"\foo{123}"` gets read and parsed on the fly, while `<token userdata>` already is tokenized and effectively is a token list now. A `<node userdata>` is also tokenized into a token list but it has a reference to a real node. Normally this goes fine. But now assume that you store the whole lot in a macro: in that case the tokenized node can be flushed many times. But, after the first such flush the node is used and its memory freed. You can prevent this by using copies which is controlled by setting `\luacopyinputnodes` to a non-zero value. This is one of these fuzzy areas you have to live with if you really mess with these low level issues.

10.3.10.3 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.

10.3.10.4 tex.cprint

This function takes a number indicating the to be used catcode, plus either a table of strings or an argument list of strings that will be pushed into the input stream.

```
tex.cprint( 1," 1: $&{\\foo}") tex.print("\\par") -- a lot of \bgroup s
tex.cprint( 2," 2: $&{\\foo}") tex.print("\\par") -- matching \egroup s
```


```
tex.cprint( 9," 9: $&{\\foo}") tex.print("\\par") -- all get ignored
tex.cprint(10,"10: $&{\\foo}") tex.print("\\par") -- all become spaces
tex.cprint(11,"11: $&{\\foo}") tex.print("\\par") -- letters
tex.cprint(12,"12: $&{\\foo}") tex.print("\\par") -- other characters
tex.cprint(14,"12: $&{\\foo}") tex.print("\\par") -- comment triggers
```

10.3.10.5 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).

10.3.11 Helper functions

10.3.11.1 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 too big' error as well.

10.3.11.2 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 too big' error(s) as well.

Note: the precision of the output of this function will depend on your computer's architecture and operating system, so use with care! An interface to LuaTEX's internal, 100% portable scale function will be added at a later date.

10.3.11.3 tex.sp

```
<number> n = tex.sp(<number> o)
<number> n = tex.sp(<string> s)
```

Converts the number `o` or a string `s` that represents an explicit dimension into an integer number of scaled points.

For parsing the string, the same scanning and conversion rules are used that LuaTEX would use if it was scanning a dimension specifier in its TEX-like input language (this includes generating errors for bad values), expect for the following:

1. only explicit values are allowed, control sequences are not handled
2. infinite dimension units (`fil...`) are forbidden
3. `mu` units do not generate an error (but may not be useful either)

10.3.11.4 tex.definefont

```
tex.definefont(<string> csname, <number> fontid)
tex.definefont(<boolean> global, <string> csname, <number> fontid)
```

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

10.3.11.5 tex.getlinenumber and tex.setlinenumber

You can mess with the current line number:

```
local n = tex.getlinenumber()
tex.setlinenumber(n+10)
```

which can be shortcut to:

```
tex.setlinenumber(10,true)
```

This might be handy when you have a callback that read numbers from a file and combines them in one line (in which case an error message probably has to refer to the original line). Interference with TEX's internal handling of numbers is of course possible.

10.3.11.6 tex.error

```
tex.error(<string> s)
tex.error(<string> s, <table> help)
```

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

The array part of the `help` table has to contain strings, one for each line of error help.

10.3.11.7 tex.hashtokens

```
for i,v in pairs (tex.hashtokens()) do ... end
```

Returns a list of names. 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. You can use `token.create` to inspect properties, for instance when the `command` key in a created table equals 123, you have the `cmdname` value `undefined_cs`.

10.3.12 Functions for dealing with primitives

10.3.12.1 tex.enableprimitives

```
tex.enableprimitives(<string> prefix, <table> primitive names)
```

This function accepts a prefix string and an array of primitive names.

For each combination of 'prefix' and 'name', the `tex.enableprimitives` first verifies that 'name' is an actual primitive (it must be returned by one of the `tex.extraprimitives()` calls explained below, or part of TEX82, or `\directlua`). If it is not, `tex.enableprimitives` does nothing and skips to the next pair.

But if it is, then it will construct a csname variable by concatenating the 'prefix' and 'name', unless the 'prefix' is already the actual prefix of 'name'. In the latter case, it will discard the 'prefix', and just use 'name'.

Then it will check for the existence of the constructed csname. If the csname is currently undefined (note: that is not the same as `\relax`), it will globally define the csname to have the meaning: run code belonging to the primitive 'name'. If for some reason the csname is already defined, it does nothing and tries the next pair.

An example:

```
tex.enableprimitives('LuaTeX', {'formatname'})
```

will define `\LuaTeXformatname` with the same intrinsic meaning as the documented primitive `\formatname`, provided that the control sequences `\LuaTeXformatname` is currently undefined.

When LuaTEX is run with `--ini` only the TEX82 primitives and `\directlua` are available, so no extra primitives **at all**.

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.

10.3.12.2 tex.extraprimitives

```
<table> t = tex.extraprimitives(<string> s, ...)
```

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 given in the following table. In addition the somewhat special primitives '\ ', '\/' and '-' are defined.

NAME	VALUES
tex	vskip write vsize boundary unhcopy output unskip unvbox boxmaxdepth muskipdef string toksdef floatingpenalty righthyphenmin voffset escapechar topmark splitfirstmark vsplit everydisplay badness xleaders textfont showlists language mathchoice topskip abovedisplayshortskip underline tracinglostchars pagefillstretch unvcopy splitbotmark finalhyphendemerits atopwithdelims pretolerance fi dp setlanguage ht mathchardef nulldelimiterspace or wd pagegoal advance chardef catcode mathchar scriptscriptfont mathcode leftskip pageshrink pagefilstretch delcode fontname brokenpenalty lastkern belowdisplayshortskip tolerance mathopen exhyphenpenalty maxdepth futurelet abovewithdelims hangindent lastskip linepenalty everyjob xspaceskip globaldefs everypar scriptfont delimiter afterassignment firstmark wordboundary lineskiplimit lineskip def fam day iffalse textstyle end mag box belowdisplayskip ifx let errmessage exhyphenchar hss expandafter the displaywidth Uright mathsurround pagedepth looseness leaders vss ifhmode botmark displaystyle accent immediate ifmmode meaning abovedisplayskip medmuskip emergencystretch rightskip mathclose hangafter hoffset aftergroup cleaders romannumeral hbadness mathbin showboxbreadth ifvmode jobname vbadness patterns nonstopmode errhelp predisplaypenalty endlinechar mathinner lastbox showboxdepth postdisplaypenalty mathrel holdinginserts radical mathord pagetotal everycr adjdemerits halign defaultskewchar errorcontextlines splitmaxdepth Uleft ifcase noindent tracingmacros moveright predisplaysize tracingrestores message ifhbox deadcycles interlinepenalty mathpunct lccode noboundary displayindent nonscript everyhbox global penalty tracingcommands everymath nolimits noalign inputlineno pagestretch parskip indent dimendef widowpenalty ifvbox above spaceskip middle displaylimits pausing everyvbox iftrue moveleft mathop endcsname dimen ifcat clubpenalty splittopskip doublehyphendemerits ifdim limits ifeof ignorespaces insert delimitershortfall ifodd insertpenalties tracingpages hpack vadjust tracingonline count ifnum edef char begingroup tracingparagraphs hyphenation hfuzz openout leqno hyphenpenalty vcenter hfil thickmuskip maxdeadcycles mkern hbox overfullrule else hsize raise thinmuskip spacefactor input hrule left eqno parfillskip font valign dump relax prevdepth read shipout batchmode right setbox baselineskip special mskip endgroup uchyph binoppenalty endinput omit pagefilllstretch overwithdelims newlinechar vfilneg time tpack skip vfill span prevgraf over show vbox tracingstats year defaulthyphenchar nullfont muskip vpack toks outer multiply tracingoutput firstvalidlanguage parindent protrusionboundary displaywidowpenalty unhbox lefthyphenmin vtop mathaccent vfuzz overline unkern closeout showthe showbox uppercase lowercase closein openin errorstopmode scrollmode skewchar hyphenchar sfcode uccode skipdef countdef glet xdef gdef long Umiddle atop scriptscriptstyle scriptstyle discretionary unpenalty copy lower kern vfil hfilneg hfill hskip crcr cr ifvoid ifinner if number lastpenalty par vrule

	parshape noexpand mark fontdimen divide csname scriptspace outputpenalty month delimiterfactor relpenalty tabskip
core	directlua
etex	unless botmarks currentiftype pagediscards mutoglue displaywidowpenalties fontcharic fontchardp fontcharht fontcharwd widowpenalties tracingifs iffontchar eTeXVersion protected topmarks showgroups glueexpr splitfirstmarks predisplaydirection everyeof eTeXversion clubpenalties savingvdiscards splitbotmarks showtokens tracingassigns dimexpr parshapedimen readline tracingscantokens tracingnesting ifdefined currentifbranch firstmarks lastnodetype marks currentgrouplevel interlinepenalties muexpr unexpanded ifcsname parshapeindent showifs parshapelength splitdiscards gluetomu glueshrink gluestretch glueshrinkorder gluestretchorder numexpr scantokens interactionmode detokenize currentiflevel currentgrouptype savinghyphcodes lastlinefit tracinggroups eTeXrevision eTeXminorversion
luatex	Umathcloseopspacing textdir Umathordpunctspacing Udelimiterunder mathsurroundmode Uskewedwithdelims bodydirection Umathopenpunctspacing pagebottomoffset luabytecodecall mathsurroundskip endlocalcontrol Umathordinnerspacing Umathbinclosespacing toksapp rightghost Umathlimitbelowbgap Umathopeninnerspacing textdirection tokspre Umathnolimitsubfactor Uoverdelimiter Umathpunctpunctspacing Umathclosepunctspacing mathdisplayskipmode saveimageresource mathrulesfam Umathrelordspacing Umathsupbottommin Umathlimitbelowkern copyfont pagedirection Umathstackdenomdown localrightbox Umathfractionrule Umathcharfam Umathcloseinnerspacing Umathopenrelspacing Uhextensible Umathsupsubbottommax leftmarginkern Umathcloserelspacing linedirection ifincsname Umathcharnum Umathinnerordspacing synctex luabytecode formatname letterspacefont boxdirection pdfextension Umathrelinnerspacing Umathsubtopmax randomseed suppressoutererror Umathsubsupshiftdown Umathopbinspacing Umathordbinspacing Umathrelopspacing Umathopenbinspacing suppressprimitiveerror Umathoverdelimiterbgap localleftbox alignmark Uunderdelimiter hyphenationmin Umathclosebinspacing Umathcodenum dvifeedback outputmode luafunction compoundhyphenmode Umathpunctopenspacing luacopyinputnodes Umathconnectoroverlapmin crampedscriptscriptstyle csstring Umathradicaldegreeafter uniformdeviate luatexversion Umathfractionnumup rightmarginkern Umathopclosespacing mathrulesmode explicithyphenpenalty Umathordclosespacing Umathoverdelimitervgap etokspre expanded suppressmathparerror Udelcode bodydir immediateassigned shapemode attribute Umathsubshiftdrop Umathsubshiftdown matheqnogapstep Umathpunctrelspacing lastsavedimageresourceindex lastsavedimageresourcepages mathoption Umathradicaldegreeraise adjustspacing Umathsupshiftdrop Umathcharslot Umathcloseclosespacing luatexrevision insertht localinterlinepenalty useboxresource explicitdiscretionary Umathchar Udelimiterover Ustack Umathcode mathdelimitersmode saveboxresource Udelcodenum gtoksapp suppresslongerror ignoreligaturesinfont Umathaxis Umathfractionnumvgap gtokspre mathflattenmode Umathskewedfractionhgap Umathrelclosespacing Umathpunctbinspacing Ustopdisplaymath quitvmode crampedscriptstyle letcharcode setrandomseed hyphenationbounds crampedtextstyle pagedir Umathbinrelspacing Umathopordspacing dvivariable

attributedef mathdirection Umathordordspacing pdffeedback Umathskewedfractionvgap Umathopenordspacing mathitalicsmode mathdir outputbox Umathcloseordspacing Umathnolimitsupfactor pagewidth Ustopmath aligntab prehyphenchar dviextension luafunctioncall Umathpunctopspacing breakafterdirmode Umathsubsupvgap luaescapestring prerelpenalty begincsname Umathradicalrule Umathunderbarrule postexhyphenchar Umathradicaldegreebefore Umathstacknumup normaldeviate Umathbinopspacing xtoksapp boxdir Ustartdisplaymath savecatcodetable Umathbinpunctspacing mathscriptboxmode tagcode Uroot lastsavedboxresourceindex Unosuperscript Umathoperatorsize xtokspre Uradical mathstyle Umathopopenspacing Umathordopenspacing automatichyphenpenalty Umathbininnerspacing Umathinnerrelspacing clearmarks Umathoverbarvgap fontid Umathopenopenspacing immediateassignment Umathunderdelimiterbgap Umathoverbarrule setfontid crampeddisplaystyle ifabsdim Umathlimitabovebgap Umathcharclass Umathstackvgap Umathinneropspacing Umathrelbinspacing Umathcloseopenspacing ifcondition pardir initcatcodetable nokerns pageleftoffset luadef tracingfonts nospaces Umathrelopenspacing Umathlimitabovekern Udelimiter savepos nohrule mathrulethicknessmode localbrokenpenalty Umathfractiondelsize exceptionpenalty automaticdiscretionary gleaders Umathunderdelimitervgap Umathinnerbinspacing noligs hyphenpenaltymode draftmode automatichyphenmode prebinoppenalty Usubscript Umathcharnumdef rpcode mathpenaltiesmode mathscriptcharmode Umathaccent pagetopoffset pageheight catcodetable Umathspaceafterscript predisplaygapfactor primitive Umathinneropenspacing Uskewed pxdimen Umathordopspacing Umathopenopspacing ifabsnum scantextokens mathnolimitsmode mathscriptsmode suppressifcsnameerror suppressfontnotfounderror pardirection pdfvariable latelua useimageresource pagerightoffset linedir efcode lpcode hjcode preexhyphenchar posthyphenchar Umathinnerinnerspacing Umathinnerpunctspacing Umathinnerclosespacing Umathpunctinnerspacing Umathpunctclosespacing Umathpunctordspacing Umathopenclosespacing Umathrelpunctspacing Umathrelrelspacing Umathbinopenspacing Umathbinbinspacing Umathbinordspacing Umathopinnerspacing Umathoppunctspacing Umathoprelspacing Umathopopspacing Umathordrelspacing Umathsupshiftup Umathlimitbelowvgap Umathlimitabovevgap Umathfractiondenomdown Umathfractiondenomvgap Umathradicalvgap Umathradicalkern Umathunderbarvgap Umathunderbarkern Umathoverbarkern Umathquad Umathchardef Uvextensible Unosubscript Usuperscript Ustartmath ifprimitive Uchar luatexbanner lastypos lastxpos novrule etoksapp leftghost expandglyphsinfont lastnamedcs protrudechars

---

Note that `luatex` does not contain `directlua`, as that is considered to be a core primitive, along with all the TEX82 primitives, so it is part of the list that is returned from `'core'`.

Running `tex.extraprimitives()` will give you the complete list of primitives `-ini` startup. It is exactly equivalent to `tex.extraprimitives("etex","luatex")`.

10.3.12.3 tex.primitives

```
<table> t = tex.primitives()
```


This function returns a list of all primitives that LuaTEX knows about.

10.3.13 Core functionality interfaces

10.3.13.1 tex.badness

```
<number> b = tex.badness(<number> t, <number> s)
```

This helper function is useful during linebreak calculations. `t` and `s` are scaled values; the function returns the badness for when total `t` is supposed to be made from amounts that sum to `s`. The returned number is a reasonable approximation of $100(t/s)^3$;

10.3.13.2 tex.resetparagraph

This function resets the parameters that TEX normally resets when a new paragraph is seen.

10.3.13.3 tex.linebreak

```
local <node> nodelist, <table> info =
       tex.linebreak(<node> listhead, <table> parameters)
```

The understood parameters are as follows:

NAME	TYPE	EXPLANATION
pardir	string	
pretolerance	number	
tracingparagraphs	number	
tolerance	number	
looseness	number	
hyphenpenalty	number	
exhyphenpenalty	number	
pdfadjustspacing	number	
adjdemerits	number	
pdfprotrudechars	number	
linepenalty	number	
lastlinefit	number	
doublehyphendemerits	number	
finalhyphendemerits	number	
hangafter	number	
interlinepenalty	number or table	if a table, then it is an array like \interlinepenalties
clubpenalty	number or table	if a table, then it is an array like \clubpenalties
widowpenalty	number or table	if a table, then it is an array like \widowpenalties
brokenpenalty	number	
emergencystretch	number	in scaled points
hangindent	number	in scaled points

hsize	number	in scaled points
leftskip	glue_spec node	
rightskip	glue_spec node	
parshape	table	

Note that there is no interface for `\displaywidowpenalties`, you have to pass the right choice for `widowpenalties` yourself.

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:

NAME	EXPLANATION
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.

10.3.13.4 tex.shipout

`tex.shipout(<number> n)`

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

10.3.13.5 tex.getpagestate

This helper reports the current page state: `empty`, `box_there` or `inserts_only` as integer value.

10.3.14 Functions related to synctex

The next helpers only make sense when you implement your own synctex logic. Keep in mind that the library used in editors assumes a certain logic and is geared for plain and LaTeX, so after a decade users expect a certain behaviour.

NAME	EXPLANATION
set_synctex_mode	0 is the default and used normal synctex logic, 1 uses the values set by the next helpers while 2 also sets these for glyph nodes; 3 sets glyphs and glue and 4 sets only glyphs
set_synctex_tag	set the current tag (file) value (obeys save stack)
set_synctex_line	set the current line value (obeys save stack)
force_synctex_tag	overload the tag (file) value (0 resets)
force_synctex_line	overload the line value (0 resets)
get_synctex_tag	get the currently set value of tag (file)
get_synctex_line	get the currently set value of line
set_synctex_no_files	disable synctex file logging

The last one is somewhat special. Due to the way files are registered in SyncTEX we need to explicitly disable that feature if we provide our own alternative if we want to avoid that overhead. Passing a value of 1 disables registering.

This mechanism is somewhat experimental.

10.4 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.

KEY	TYPE	DEFAULT	EXPLANATION
kpse_init	boolean	true	false totally disables kpathsea initialisation, and enables interpretation of the following numeric key–value pairs. (only ever unset this if you implement *all* file find callbacks!)
shell_escape	string	'f'	Use 'y' or 't' or '1' to enable \write 18 unconditionally, 'p' to enable the commands that are listed in shell_escape_commands
shell_escape_commands	string		Comma-separated list of command names that may be executed by \write 18 even if shell_escape is set to 'p'. Do *not* use spaces around commas, separate any required command arguments by using a space, and use the ascii double quote (") for any needed argument or path quoting
string_vacancies	number	75000	cf. web2c docs
pool_free	number	5000	cf. web2c docs
max_strings	number	15000	cf. web2c docs
strings_free	number	100	cf. web2c docs
nest_size	number	50	cf. web2c docs
max_in_open	number	15	cf. web2c docs
param_size	number	60	cf. web2c docs
save_size	number	4000	cf. web2c docs
stack_size	number	300	cf. web2c docs
dvi_buf_size	number	16384	cf. web2c docs

error_line	number	79	cf. web2c docs
half_error_line	number	50	cf. web2c docs
max_print_line	number	79	cf. web2c docs
hash_extra	number	0	cf. web2c docs
pk_dpi	number	72	cf. web2c docs
trace_file_names	boolean	true	false disables TEX's normal file open-close feedback (the assumption is that callbacks will take care of that)
file_line_error	boolean	false	do file:line style error messages
halt_on_error	boolean	false	abort run on the first encountered error
formatname	string		if no format name was given on the command line, this key will be tested first instead of simply quitting
jobname	string		if no input file name was given on the command line, this key will be tested first instead of simply giving up

Note: the numeric values that match web2c parameters are only used if kpse_init is explicitly set to false. In all other cases, the normal values from texmf.cnf are used.

10.5 The texio library

This library takes care of the low-level I/O interface: writing to the log file and/or console.

10.5.1 texio.write

```
texio.write(<string> target, <string> s, ...)
texio.write(<string> s, ...)
```

Without the target argument, writes all given strings to the same location(s) TEX writes messages to at this moment. If \batchmode is in effect, it writes only to the log, otherwise it writes to the log and the terminal. The optional target can be one of three possibilities: term, log or term and log.

Note: If several strings are given, and if the first of these strings is or might be one of the targets above, the target must be specified explicitly to prevent Lua from interpreting the first string as the target.

10.5.2 texio.write_nl

```
texio.write_nl(<string> target, <string> s, ...)
texio.write_nl(<string> s, ...)
```

This function behaves like 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.

10.5.3 texio.setescape

You can disable ^^ escaping of control characters by passing a value of zero.

10.6 The token library

10.6.1 The scanner

The token library provides means to intercept the input and deal with it at the Lua level. The 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 functions provided in the `token` namespace are given in the next table:

FUNCTION	ARGUMENT	RESULT
`is_token`	token	checks if the given argument is a token
`get_next`		returns the next token in the input
`scan_keyword`	string	returns true if the given keyword is gobbled
`scan_int`		returns an integer
`scan_real`		returns a number from e.g. `1`, `1.1`, `.1` with optional collapsed signs
`scan_float`		returns a number from e.g. `1`, `1.1`, `.1`, `1.1E10`, , `.1e-10` with optional collapsed signs
`scan_dimen`	infinity, mu-units	returns a number representing a dimension and or two numbers being the filler and order
`scan_glue`	mu-units	returns a glue spec node
`scan_toks`	definer, expand	returns a table of tokens tokens
`scan_code`	bitset	returns a character if its category is in the given bitset (representing catcodes)
`scan_string`		returns a string given between `{}`, as `\macro` or as sequence of characters with catcode 11 or 12
`scan_word`		returns a sequence of characters with catcode 11 or 12 as string
`scan_csname`		returns `foo` after scanning `\foo`
`set_macro`	see below	assign a macro
`create`		returns a userdata token object of the given control sequence name (or character); this interface can change

The scanners can be considered stable apart from the one scanning for a token. 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:

NAME	RESULT	
\directlua{token.scan_string()}{foo}	foo	full expansion
\directlua{token.scan_string()}foo	foo	letters and others
\directlua{token.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 since there we do expand.

The scan_word scanner can be used to implement for instance a number scanner:

```
function token.scan_number(base)
    return tonumber(token.scan_word(),base)
end
```

This scanner accepts any valid Lua number so it is a way to pick up floats in the input.

The creator function can be used as follows:

```
local t = token.create("relax")
```

This gives back a token object that has the properties of the \relax primitive. The possible properties of tokens are:

NAME	EXPLANATION
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.

You can ask for a list of commands:

```
local t = token.commands()
```

The id of a token class can be queried as follows:

```
local id = token.command_id("math_shift")
```

If you really know what you're doing you can create character tokens by not passing a string but a number:

```
local letter_x = token.create(string.byte("x"))
local other_x  = token.create(string.byte("x"),12)
```

Passing weird numbers can give side effects so don't expect too much help with that. As said, you need to know what you're doing. The best way to explore the way these internals work is to just look at how primitives or macros or `\chardef`'d commands are tokenized. Just create a known one and inspect its fields. A variant that ignores the current catcode table is:

```
local whatever = token.new(123,12)
```

You can test if a control sequence is defined with `is_defined`, which accepts a string and returns a boolean:

```
local okay = token.is_defined("foo")
```

More interesting are the scanners. You can use the Lua interface as follows:

```
\directlua {
    function mymacro(n)
        ...
    end
}

\def\mymacro#1{%
    \directlua {
        mymacro(\number\dimexpr#1)
    }%
}

\mymacro{12pt}
\mymacro{\dimen0}
```

You can also do this:

```
\directlua {
    function mymacro()
        local d = token.scan_dimen()
        ...
    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, then it is 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).

10.6.2 Macros

The `set_macro` function can get upto 4 arguments:

```
set_macro("csname","content")
set_macro("csname","content","global")
set_macro("csname")
```

You can pass a catcodetable identifier as first argument:

```
set_macro(catcodetable,"csname","content")
set_macro(catcodetable,"csname","content","global")
set_macro(catcodetable,"csname")
```

The results are like:

```
 \def\csname{content}
\gdef\csname{content}
 \def\csname{}
```

The `get_macro` function can be used to get the content of a macro while the `get_meaning` function gives the meaning including the argument specification (as usual in TEX separated by `->`).

The `set_char` function can be used to do a `\chardef` at the Lua end, where invalid assignments are silently ignored:

```
set_char("csname",number)
```


```
set_char("csname",number,"global")
```

A special one is the following:

```
set_lua("mycode",id)
set_lua("mycode",id,"global","protected")
```

This creates a token that refers to a Lua function with an entry in the table that you can access with `lua.get_functions_table()`. It is the companion to `\luadef`.

10.6.3 Pushing back

There is a (for now) experimental putter:

```
local t1 = token.get_next()
local t2 = token.get_next()
local t3 = token.get_next()
local t4 = token.get_next()
-- watch out, we flush in sequence
token.put_next { t1, t2 }
-- but this one gets pushed in front
token.put_next ( t3, t4 )
```

When we scan `wxyz!` we get `yzwx!` back. The argument is either a table with tokens or a list of tokens. The `token.expand` function will trigger expansion but what happens really depends on what you're doing where.

10.6.4 Nota bene

When scanning for the next token you need to keep in mind that we're not scanning like TEX does: expanding, changing modes and doing things as it goes. When we scan with Lua we just pick up tokens. Say that we have:

```
\bar
```

but `\bar` is undefined. Normally TEX will then issue an error message. However, when we have:

```
\def\foo{\bar}
```

We get no error, unless we expand `\foo` while `\bar` is still undefined. What happens is that as soon as TEX sees an undefined macro it will create a hash entry and when later it gets defined that entry will be reused. So, `\bar` really exists but can be in an undefined state.

bar : bar
foo : foo
myfirstbar :

This was entered as:

```
bar        : \directlua{tex.print(token.scan_csname())}\bar
foo        : \directlua{tex.print(token.scan_csname())}\foo
myfirstbar : \directlua{tex.print(token.scan_csname())}\myfirstbar
```

The reason that you see `bar` reported and not `myfirstbar` is that `\bar` was already used in a previous paragraph.

If we now say:

```
\def\foo{}
```

we get:

bar : bar
foo : foo
myfirstbar :

And if we say

```
\def\foo{\bar}
```

we get:

bar : bar
foo : foo
myfirstbar :

When scanning from Lua we are not in a mode that defines (undefined) macros at all. There we just get the real primitive undefined macro token.

1774360 536941998
1774389 536941998
1773841 536941998

This was generated with:

```
\directlua{local t = token.get_next() tex.print(t.id.." "..t.tok)}\myfirstbar
\directlua{local t = token.get_next() tex.print(t.id.." "..t.tok)}\mysecondbar
\directlua{local t = token.get_next() tex.print(t.id.." "..t.tok)}\mythirdbar
```

So, we do get a unique token because after all we need some kind of Lua object that can be used and garbage collected, but it is basically the same one, representing an undefined control sequence.

10.7 The kpse library

This library provides two separate, but nearly identical interfaces to the kpathsea file search functionality: there is a 'normal' procedural interface that shares its kpathsea instance with LuaTEX itself, and an object oriented interface that is completely on its own.

10.7.1 kpse.set_program_name and kpse.new

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

First, when LuaTEX is used to typeset documents, this initialization happens automatically and the kpathsea executable and program names are set to `luatex` (that is, unless explicitly prohibited by the user's startup script. See section 4.1 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.

10.7.2 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> mustex-
ist)
<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: `MetaPost support, PostScript header, TeX system documentation, TeX system sources, Troff fonts, afm, base, bib, bitmap font, bst, cid maps, clua, cmap files, cnf, cweb, dvips config, enc files, fmt, font feature files, gf, graphic/figure, ist, lig files, ls-R, lua, map, mem, mf, mfpool, mft, misc fonts, mlbib, mlbst, mp, mppool, ocp,`

ofm, opentype fonts, opl, other binary files, other text files, otp, ovf, ovp, pdftex config, pk, subfont definition files, tex, texmfscripts, texpool, tfm, truetype fonts, type1 fonts, type42 fonts, vf, web, web2c files
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.

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.

10.7.3 lookup

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

```
<string> f, ... = kpse.lookup(<string> filename, <table> options)
```

The options match commandline arguments from kpsewhich:

KEY	TYPE	EXPLANATION
debug	number	set debugging flags for this lookup
format	string	use specific file type (see list above)
dpi	number	use this resolution for this lookup; default 600
path	string	search in the given path
all	boolean	output all matches, not just the first
mustexist	boolean	search the disk as well as ls-R if necessary
mktexpk	boolean	disable/enable mktexpk generation for this lookup
mktextex	boolean	disable/enable mktextex generation for this lookup
mktexmf	boolean	disable/enable mktexmf generation for this lookup
mktextfm	boolean	disable/enable mktextfm generation for this lookup
subdir	string or table	only output matches whose directory part ends with the given string(s)

10.7.4 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)
```

10.7.5 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.

10.7.6 expand_path

Like kpsewhich's `-expand-path`:

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

10.7.7 expand_var

Like kpsewhich's `-expand-var`:

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

10.7.8 expand_braces

Like kpsewhich's `-expand-braces`:

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

10.7.9 show_path

Like kpsewhich's `-show-path`:

```
<string> r = kpse.show_path(<string> ftype)
```

10.7.10 var_value

Like kpsewhich's `-var-value`:

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

10.7.11 version

Returns the kpathsea version string.

```
<string> r = kpse.version()
```

11 The graphic libraries

11.1 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 6.3.

11.1.1 new

```
<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

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

followed by

```
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:

```
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 LuaTEX 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 boundingbox (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.

You can use pdf.setignoreunknownimages(1) (or at the TEX end the \pdfvariable ignoreunknownimages) to get around a quit when no known image type is found (based on name or preamble). Beware: this will not catch invalid images and we cannot guarantee side effects.

A zero dimension image is still included when requested. No special flags are set. A proper workflow will not rely in such a catch but make sure that images are valid.

11.1.2 keys

```
<table> keys = img.keys()
```

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

FIELD NAME	TYPE	DESCRIPTION
`attr`	string	the image attributes for LuaTEX
`bbox`	table	table with 4 boundingbox dimensions `llx`, `lly`, `urx` and `ury` overruling the `pagebox` entry
`colordepth`	number	the number of bits used by the color space
`colorspace`	number	the color space object number
`depth`	number	the image depth for LuaTEX
`filename`	string	the image file name
`filepath`	string	the full (expanded) file name of the image
`height`	number	the image height for LuaTEX
`imagetype`	string	one of `pdf`, `png`, `jpg`, `jp2` or `jbig2`
`index`	number	the pdf image name suffix
`objnum`	number	the pdf image object number
`page`	number	the identifier for the requested image page
`pagebox`	string	the requested bounding box, one of `none`, `media`, `crop`, `bleed`, `trim`, `art`
`pages`	number	the total number of available pages
`rotation`	number	the image rotation from included pdf file, in multiples of 90 deg.
`stream`	string	the raw stream data for an `/Xobject /Form` object
`transform`	number	the image transform, integer number 0..7
`orientation`	number	the (jpeg) image orientation, integer number 1..8 (0 for unset)
`width`	number	the image width for LuaTEX
`xres`	number	the horizontal natural image resolution (in dpi)
`xsize`	number	the natural image width
`yres`	number	the vertical natural image resolution (in dpi)
`ysize`	number	the natural image height
`visiblefilename`	string	when set, this name will find its way in the pdf file as PTEX specification; when an empty string is assigned nothing is written to file; otherwise the natural filename is taken
`userpassword`	string	the userpassword needed for opening a pdf file
`ownerpassword`	string	the ownerpassword needed for opening a pdf file
`keepopen`	boolean	keep the pdf file open
`nolength`	boolean	don't add length key nor compress for streams

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.

11.1.3 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 a

```
<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.

11.1.4 copy

```
<image> var = img.copy(<image> var)
```


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

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

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

11.1.5 write

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

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

Again you can do a terse call like

```
img.write { filename = "foo.png" }
```

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

11.1.6 immediatewrite

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

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

Also here you can do a terse call like

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

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

11.1.7 node

```
<node> n = img.node(<image> var)
<node> n = img.node(<table> image_spec)
```

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:

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

This example outputs an image:

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

11.1.8 types

```
<table> types = 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`.

11.1.9 boxes

```
<table> boxes = 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.

11.2 The mplib library

The MetaPost library interface registers itself in the table `mplib`. It is based on mplib version 2.00.

11.2.1 new

To create a new MetaPost instance, call

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

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

NAME	TYPE	DESCRIPTION	DEFAULT
`error_line`	number	error line width	79
`print_line`	number	line length in ps output	100
`random_seed`	number	the initial random seed	variable
`math_mode`	string	the number system to use: `double`, `scaled`, `binary` or `decimal`	`scaled`
`interaction`	string	the interaction mode: `batch`, `nonstop`, `scroll` or `errorstop`	`errorstop`
`job_name`	string	`--jobname`	`mpout`
`find_file`	function	a function to find files	only local files

The `find_file` function should be of this form:

```
<string> found = finder (<string> name, <string> mode, <string> type)
```

with:

NAME	THE REQUESTED FILE
mode	the file mode: r or w
type	the kind of file, one of: mp, tfm, map, pfb, enc

Return either the full path name 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.

The pdf file is kept open after its properties are determined. After inclusion, which happens when the page that references the image is flushed, the file is closed. This means that when you have thousands of images on one page, your operating system might decide to abort the run. When you include more than one page from a pdf file you can set the keepopen flag when you allocate an image object, or pass the keepopen directive when you refer to the image with \useimageresource. This only makes sense when you embed many pages. An \immediate applied to \saveimageresource will also force a close after inclusion.

```
\immediate\useimageresource{foo.pdf}%
          \saveimageresource         \lastsavedimageresourceindex % closed
          \useimageresource{foo.pdf}%
          \saveimageresource         \lastsavedimageresourceindex % kept open
          \useimageresource{foo.pdf}%
          \saveimageresource keepopen\lastsavedimageresourceindex % kept open

\directlua{img.write(img.scan{ file = "foo.pdf" })}                  % closed
\directlua{img.write(img.scan{ file = "foo.pdf", keepopen = true })} % kept open
```

11.2.2 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:

FIELD	TYPE	EXPLANATION
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.

11.2.3 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 stand alone `mpost` command, there is *no* implied 'input' at the start of the first chunk.

11.2.4 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.

11.2.5 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).

FIELD	TYPE	EXPLANATION
`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:

FIELD	TYPE	EXPLANATION
`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 stand alone mode
`width`	function	the `fontcharwd` value
`height`	function	the `fontcharht` value
`depth`	function	the `fontchardp` value
`italcorr`	function	the `fontcharit` 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 a 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.

11.2.5.1 fill

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

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

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

FIELD	TYPE	EXPLANATION
`width`	number	width of the pen
`sx`	number	x scale
`rx`	number	$x\,y$ multiplier
`ry`	number	$y\,x$ multiplier

sy	number	*y* scale
tx	number	*x* offset
ty	number	*y* offset

11.2.5.2 outline

FIELD	TYPE	EXPLANATION
path	table	the list of knots
pen	table	knots of the pen
color	table	the object's color
linejoin	number	line join style (bare number)
miterlimit	number	miterlimit
linecap	number	line cap style (bare number)
dash	table	representation of a dash list
prescript	string	the prescript text
postscript	string	the postscript text

The entry dash is optional.

11.2.5.3 text

FIELD	TYPE	EXPLANATION
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

11.2.5.4 special

FIELD	TYPE	EXPLANATION
prescript	string	special text

11.2.5.5 start_bounds, start_clip

FIELD	TYPE	EXPLANATION
path	table	the list of knots

11.2.5.6 stop_bounds, stop_clip

Here are no fields available.

11.2.6 Subsidiary table formats

11.2.6.1 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.

FIELD	TYPE	EXPLANATION
`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 stringB Avalued key `type` with value `elliptical` besides the array part containing the knot list.

11.2.6.2 Colors

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

FIELD	TYPE	EXPLANATION
`0`	marking only	no values
`1`	greyscale	one value in the range $(0, 1)$, 'black' is 0
`3`	rgb	three values in the range $(0, 1)$, 'black' is $0, 0, 0$
`4`	cmyk	four values in the range $(0, 1)$, 'black' is $0, 0, 0, 1$

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

11.2.6.3 Transforms

Each transform is a six-item array.

INDEX	TYPE	EXPLANATION
`1`	number	represents x
`2`	number	represents y
`3`	number	represents xx
`4`	number	represents yx
`5`	number	represents xy
`6`	number	represents yy

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

11.2.6.4 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.

FIELD	TYPE	EXPLANATION
`dashes`	hash	an array of on-off numbers
`offset`	number	the starting offset value

11.2.7 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.

11.2.7.1 mp:char_width

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

11.2.7.2 mp:char_height

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

11.2.7.3 mp:char_depth

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


12 The fontloader

The fontloader library is sort of independent of the rest in the sense that it can load font into a Lua table that then can be converted into a table suitable for TEX. The library is an adapted subset of FontForge and as such gives a similar view on a font (which has advantages when you want to debug). We will not discuss OpenType in detail here as the Microsoft website offers enough information about it. The tables returned by the loader are not that far from the standard. We have no plans to extend the loader (it may even become an external module at some time).

12.1 Getting quick information on a font

When you want to locate font by name you need some basic information that is hidden in the font files. For that reason we provide an efficient helper that gets the basic information without loading all of the font. Normally this helper is used to create a font (name) database.

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

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

KEY	TYPE	EXPLANATION
`fontname`	string	the PostScript name of the font
`fullname`	string	the formal name of the font
`familyname`	string	the family name this font belongs to
`weight`	string	a string indicating the color value of the font
`version`	string	the internal font version
`italicangle`	float	the slant angle
`units_per_em`	number	1000 for PostScript-based fonts, usually 2048 for TrueType
`pfminfo`	table	(see section 12.6.6)

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.

12.2 Loading an OPENTYPE or TRUETYPE file

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

```
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')
```

The main function call is

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

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`.

```
<table> f = fontloader.to_table(<userdata> font)
```

This table cannot be used directly by LuaTEX and should be turned into another one as described in chapter 6. Do not forget to store the `fontname` value in the `psname` field of the metrics table to be returned to LuaTEX, otherwise the font inclusion backend will not be able to find the correct font in the collection.

See section 12.5 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:

```
fontloader.close(<userdata> font)
```

12.3 Applying a ‘feature file’

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

```
<table> errors = fontloader.apply_featurefile(<userdata> font, <string> file-
name)
```

A ‘feature file’ is a textual representation of the features in an OpenType font. See

```
http://www.adobe.com/devnet/opentype/afdko/topic_feature_file_syntax.html
```

and

```
http://fontforge.sourceforge.net/featurefile.html
```

for a more detailed description of feature files.

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

12.4 Applying an ‘AFM file’

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

```
<table> 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.

12.5 Fontloader font tables

As mentioned earlier, the return value of `fontloader.open()` is a userdata object. One way to have access to the actual metrics is to call `fontloader.to_table()` on this object, returning the table structure that is explained in the following sections. In teh following sections we will not explain each field in detail. Most fields are self descriptive and for the more technical aspects you need to consult the relevant font references.

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 it is possible to access the userdata object directly.

- 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`:

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

Only the top-level font, the subfont table entries, and the glyphs are virtual objects, everything else still produces normal Lua values and tables.

If you want to know the valid fields in a font or glyph structure, call the `fields` function on an object of a particular type (either glyph or font):

```
<table> fields = fontloader.fields(<userdata> font)
<table> fields = fontloader.fields(<userdata> font_glyph)
```

For instance:

```
local fields = fontloader.fields(f)
local fields = fontloader.fields(f.glyphs[0])
```

12.6 Table types

12.6.1 Top-level

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

KEY	TYPE	explanation
table_version	number	indicates the metrics version (currently 0.3)
fontname	string	PostScript font name
fullname	string	official (human-oriented) font name
familyname	string	family name
weight	string	weight indicator
copyright	string	copyright information
filename	string	the file name
version	string	font version
italicangle	float	slant angle
units_per_em	number	1000 for PostScript-based fonts, usually 2048 for TrueType
ascent	number	height of ascender in units_per_em
descent	number	depth of descender in units_per_em
upos	float	
uwidth	float	
uniqueid	number	
glyphs	array	
glyphcnt	number	number of included glyphs
glyphmax	number	maximum used index the glyphs array
glyphmin	number	minimum used index the glyphs array
notdef_loc	number	location of the .notdef glyph or -1 when not present
hasvmetrics	number	
onlybitmaps	number	
serifcheck	number	
isserif	number	
issans	number	
encodingchanged	number	
strokedfont	number	
use_typo_metrics	number	
weight_width_slope_only	number	
head_optimized_for_cleartype	number	
uni_interp	enum	unset, none, adobe, greek, japanese, trad_chinese, simp_chinese, korean, ams
origname	string	the file name, as supplied by the user
map	table	
private	table	
xuid	string	
pfminfo	table	
names	table	
cidinfo	table	
subfonts	array	
commments	string	
fontlog	string	
cvt_names	string	
anchor_classes	table	

ttf_tables	table	
ttf_tab_saved	table	
kerns	table	
vkerns	table	
texdata	table	
lookups	table	
gpos	table	
gsub	table	
mm	table	
chosenname	string	
macstyle	number	
fondname	string	
fontstyle_id	number	
fontstyle_name	table	
strokewidth	float	
mark_classes	table	
creationtime	number	
modificationtime	number	
os2_version	number	
sfd_version	number	
math	table	
validation_state	table	
horiz_base	table	
vert_base	table	
extrema_bound	number	
truetype	boolean	signals a TrueType font

12.6.2 Glyph items

The glyphs is an array containing the per-character information (quite a few of these are only present if non-zero).

KEY	TYPE	EXPLANATION
name	string	the glyph name
unicode	number	unicode code point, or -1
boundingbox	array	array of four numbers, see note below
width	number	only for horizontal fonts
vwidth	number	only for vertical fonts
tsidebearing	number	only for vertical ttf/otf fonts, and only if non-zero
lsidebearing	number	only if non-zero and not equal to boundingbox[1]
class	string	one of "none", "base", "ligature", "mark", "component" (if not present, the glyph class is ‘automatic’)
kerns	array	only for horizontal fonts, if set
vkerns	array	only for vertical fonts, if set
dependents	array	linear array of glyph name strings, only if nonempty
lookups	table	only if nonempty

ligatures	table	only if nonempty
anchors	table	only if set
comment	string	only if set
tex_height	number	only if set
tex_depth	number	only if set
italic_correction	number	only if set
top_accent	number	only if set
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 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 :

```
['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!

12.6.3 map table

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

KEY	TYPE	EXPLANATION
`enccount`	number	
`encmax`	number	
`backmax`	number	
`remap`	table	
`map`	array	non-linear array of mappings
`backmap`	array	non-linear array of backward mappings
`enc`	table	

The `remap` table is very small:

KEY	TYPE	EXPLANATION
`firstenc`	number	
`lastenc`	number	
`infont`	number	

The `enc` table is a bit more verbose:

KEY	TYPE	EXPLANATION
`enc_name`	string	
`char_cnt`	number	
`char_max`	number	
`unicode`	array	of Unicode position numbers
`psnames`	array	of PostScript glyph names
`builtin`	number	
`hidden`	number	
`only_1byte`	number	
`has_1byte`	number	
`has_2byte`	number	
`is_unicodebmp`	number	only if non-zero
`is_unicodefull`	number	only if non-zero
`is_custom`	number	only if non-zero
`is_original`	number	only if non-zero
`is_compact`	number	only if non-zero
`is_japanese`	number	only if non-zero
`is_korean`	number	only if non-zero
`is_tradchinese`	number	only if non-zero [name?]
`is_simplechinese`	number	only if non-zero

low_page	number
high_page	number
iconv_name	string
iso_2022_escape	string

12.6.4 private table

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

12.6.5 cidinfo table

KEY	TYPE	EXPLANATION
registry	string	
ordering	string	
supplement	number	
version	number	

12.6.6 pfminfo table

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

KEY	TYPE	EXPLANATION
pfmset	number	
winascent_add	number	
windescent_add	number	
hheadascent_add	number	
hheaddescent_add	number	
typoascent_add	number	
typodescent_add	number	
subsuper_set	number	
panose_set	number	
hheadset	number	
vheadset	number	
pfmfamily	number	
weight	number	
width	number	
avgwidth	number	
firstchar	number	
lastchar	number	
fstype	number	
linegap	number	
vlinegap	number	
hhead_ascent	number	
hhead_descent	number	
os2_typoascent	number	

os2_typodescent	number	
os2_typolinegap	number	
os2_winascent	number	
os2_windescent	number	
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:

KEY	TYPE	EXPLANATION
familytype	string	Values as in the OpenType font specification: Any, No Fit, Text and Display, Script, Decorative, Pictorial
serifstyle	string	See the OpenType font specification for values
weight	string	idem
proportion	string	idem
contrast	string	idem
strokevariation	string	idem
armstyle	string	idem
letterform	string	idem
midline	string	idem
xheight	string	idem

12.6.7 names table

Each item has two top-level keys:

KEY	TYPE	EXPLANATION
lang	string	language for this entry
names	table	

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

12.6.8 anchor_classes table

The anchor_classes classes:

KEY	TYPE	EXPLANATION
name	string	a descriptive id of this anchor class
lookup	string	
type	string	one of mark, mkmk, curs, mklg

12.6.9 gpos table

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

KEY	TYPE	EXPLANATION
type	string	one of gpos_single, gpos_pair, gpos_cursive, gpos_mark2base, gpos_mark2ligature, gpos_mark2mark, gpos_context, gpos_contextchain
flags	table	
name	string	
features	array	
subtables	array	

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

KEY	TYPE	EXPLANATION
r2l	boolean	
ignorebaseglyphs	boolean	
ignoreligatures	boolean	
ignorecombiningmarks	boolean	
mark_class	string	

The features subtable items of gpos have:

KEY	TYPE	EXPLANATION
tag	string	
scripts	table	

The scripts table within features has:

KEY	TYPE	EXPLANATION
script	string	
langs	array of strings	

The subtables table has:

KEY	TYPE	EXPLANATION
name	string	
suffix	string	(only if used)
anchor_classes	number	(only if used)
vertical_kerning	number	(only if used)
kernclass	table	(only if used)

The kernclass with subtables table has:

KEY	TYPE	EXPLANATION
firsts	array of strings	
seconds	array of strings	
lookup	string or array	associated lookup(s)
offsets	array of numbers	

Note: the kernclass (as far as we can see) always has one entry so it could be one level deep instead. Also the seconds start at [2] which is close to the fontforge internals so we keep that too.

12.6.10 gsub table

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

KEY	TYPE	EXPLANATION
type	string	one of gsub_single, gsub_multiple, gsub_alternate, gsub_ligature, gsub_context, gsub_contextchain, gsub_reversecontextchain

12.6.11 ttf_tables and ttf_tab_saved tables

KEY	TYPE	EXPLANATION
tag	string	
len	number	
maxlen	number	
data	number	

12.6.12 mm table

KEY	TYPE	EXPLANATION
axes	table	array of axis names

instance_count	number	
positions	table	array of instance positions (#axes * instances)
defweights	table	array of default weights for instances
cdv	string	
ndv	string	
axismaps	table	

The axismaps:

KEY	TYPE	EXPLANATION
blends	table	an array of blend points
designs	table	an array of design values
min	number	
def	number	
max	number	

12.6.13 mark_classes table

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

12.6.14 math table

The math table has the variables that are also discussed in the chapter aboout math: 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 and FractionDelimiterDisplayStyleSize.

12.6.15 validation_state table

This is just a bonus table with keys: `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` and `bad_sfnt_header`.

12.6.16 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.

12.6.17 altuni table

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

KEY	TYPE	EXPLANATION
`unicode`	number	this glyph is also used for this unicode
`variant`	number	the alternative is driven by this unicode selector

12.6.18 vert_variants and horiz_variants table

KEY	TYPE	EXPLANATION
`variants`	string	
`italic_correction`	number	
`parts`	table	

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

KEY	TYPE	EXPLANATION
component	string	
extender	number	
start	number	
end	number	
advance	number	

12.6.19 mathkern table

KEY	TYPE	EXPLANATION
top_right	table	
bottom_right	table	
top_left	table	
bottom_left	table	

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

KEY	TYPE	EXPLANATION
height	number	
kern	number	

12.6.20 kerns table

Substructure is identical to the per-glyph subtable.

12.6.21 vkerns table

Substructure is identical to the per-glyph subtable.

12.6.22 texdata table

KEY	TYPE	EXPLANATION
type	string	unset, text, math, mathext
params	array	22 font numeric parameters

12.6.23 lookups table

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

KEY	TYPE	EXPLANATION
type	string	
format	enum	one of glyphs, class, coverage, reversecoverage
tag	string	

current_class	array	
before_class	array	
after_class	array	
rules	array	an array of rule items

Rule items have one common item and one specialized item:

KEY	TYPE	EXPLANATION
lookups	array	a linear array of lookup names
glyphs	array	only if the parent's format is glyphs
class	array	only if the parent's format is class
coverage	array	only if the parent's format is coverage
reversecoverage	array	only if the parent's format is reversecoverage

A glyph table is:

KEY	TYPE	EXPLANATION
names	string	
back	string	
fore	string	

A class table is:

KEY	TYPE	EXPLANATION
current	array	of numbers
before	array	of numbers
after	array	of numbers

for coverage:

KEY	TYPE	EXPLANATION
current	array	of strings
before	array	of strings
after	array	of strings

and for reversecoverage:

KEY	TYPE	EXPLANATION
current	array	of strings
before	array	of strings
after	array	of strings
replacements	string	

13 The backend libraries

13.1 The pdf library

This library contains variables and functions that are related to the pdf backend. You can find more details about the expected values to setters in section 3.2.

13.1.1 mapfile, mapline

```
pdf.mapfile(<string> map file)
pdf.mapline(<string> map line)
```

These two functions can be used to replace primitives `\pdfmapfile` and `\pdfmapline` inherited from pdfTEX. They expect a string as only parameter and have no return value. The first character in a map line can be `-`, `+` or `=` which means as much as remove, add or replace this line.

13.1.2 [set|get][catalog|info|names|trailer]

These functions complement the corresponding pdf backend token lists dealing with metadata. The value types are strings and they are written to the pdf file directly after the token registers set at the TEX end are written.

13.1.3 [set|get][pageattributes|pageresources|pagesattributes]

These functions complement the corresponding pdf backend token lists dealing with page resources. The variables have no interaction with the corresponding pdf backend token register. They are written to the pdf file directly after the token registers set at the TEX end are written.

13.1.4 [set|get][xformattributes|xformresources]

These functions complement the corresponding pdf backend token lists dealing with reuseable boxes and images. The variables have no interaction with the corresponding pdf backend token register. They are written to the pdf file directly after the token registers set at the TEX end are written.

13.1.5 getversion and [set|get]minorversion

The version is frozen in the binary but you can set the minor version. What minor version you set depends on what pdf features you use. This is out of control of LuaTEX.

13.1.6 getcreationdate

This function returns a string with the date in the format that ends up in the pdf file, in this case it's: `D:20180624224452+02'00'`.

13.1.7 [set|get]inclusionerrorlevel, [set|get]ignoreunknownimages

These variable control how error in included image are treated. They are modeled after the pdfTEX equivalents.

13.1.8 [set|get]suppressoptionalinfo

This bitset determines what kind of info gets flushed. By default we flush all.

13.1.9 [set|get]trailerid

You can set your own trailer id. This has to be a valid array string with checksums.

13.1.10 [set|get]compresslevel

These two functions set the level of compression. The minimum value is 0, the maximum is 9.

13.1.11 [set|get]objcompresslevel

These two functions set the level of compression. The minimum value is 0, the maximum is 9.

13.1.12 [set|get]recompress

When set to 1 compressed objects will be decompressed and when compresslevel is larger than zero be recompressed. This is mostly a debugging feature and should not be relied upon.

13.1.13 [set|get]gentounicode

This flag enables tounicode generation (like in pdfTEX).

13.1.14 [set|get]omitcidset

This flag disables inclusion of a so called CIDSet which can be handy when aiming at some of the many pdf substandards.

13.1.15 [set|get]decimaldigits

These two functions set the accuracy of floats written to the pdffile. You can set any value but the backend will not go below 3 and above 6.

13.1.16 [set|get]pkresolution

These setter takes two arguments: the resolution and an optional zero or one that indicates if this is a fixed one. The getter returns these two values.

13.1.17 getlast[obj|link|annot] and getretval

These status variables are similar to the ones traditionally used in the backend interface at the TEX end.

13.1.18 maxobjnum and objtype, fontname, fontobjnum, fontsize, xformname

.

These (and some other) introspective helpers were moved from the `tex` namespace to the `pdf` namespace but kept their original names. They are mostly used when you construct pdf objects yourself and need for instance information about a (to be) embedded font.

13.1.19 [set|get]origin

This one is used to set the horizonal and/or vertical offset, a traditional backend property.

```
pdf.setorigin() -- sets both to 0pt
pdf.setorigin(tex.sp("1in")) -- sets both to 1in
pdf.setorigin(tex.sp("1in"),tex.sp("1in"))
```

The counterpart of this function returns two values.

13.1.20 [set|get]imageresolution

These two functions relate to the imageresolution that is used when the image itself doesn't provide a non-zero x or y resolution.

13.1.21 [set|get][link|dest|thread|xform]margin

These functions can be used to set and retrieve the margins that are added to the natural bounding boxes of the respective objects.

13.1.22 get[pos|hpos|vpos]

These functions get current location on the output page, measured from its lower left corner. The values return scaled points as units.

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

13.1.23 [has|get]matrix

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

```
if pdf.hasmatrix() then
```


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

13.1.24 print

You can print a string to the pdf document from within a `\latelua` call. This function is not to be used inside `\directlua` unless you know *exactly* what you are doing.

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

The optional parameter can be used to mimic the behavior of pdf literals: the `type` is `direct` or `page`.

13.1.25 immediateobj

This function creates a pdf object and immediately writes it to the pdf file. It is modelled after pdfTEX's `\immediate \pdfobj` primitives. All function variants return the object number of the newly generated object.

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

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

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

```
<number> n =
    pdf.immediateobj(<integer> n, <string> objtext)
<number> n =
    pdf.immediateobj(<integer> n, "file", <string> filename)
<number> n =
```

```
    pdf.immediateobj(<integer> n, "stream", <string> streamtext, <string> attr-
text)
<number> n =
    pdf.immediateobj(<integer> n, "streamfile", <string> filename, <string> at-
trtext)
```

13.1.26 obj

This function creates a pdf object, which is written to the pdf file only when referenced, e.g., by `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 pdfTEX's `\pdfobj` primitive.

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

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

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

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

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


The type field can have the values raw and stream, this field is required, the others are optional (within constraints). When nolength is set, there will be no /Length entry added to the dictionary.

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

13.1.27 refobj

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

```
pdf.refobj(<integer> n)
```

This function works in both the \directlua and \latelua environment. Inside \directlua a new whatsit node 'pdf_refobj' is created, which will be marked for flushing during page output and the object is then written directly after the page, when also the resources objects are written to the pdf file. Inside \latelua the object will be marked for flushing.

This function has no return values.

13.1.28 reserveobj

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

```
<number> n = pdf.reserveobj()
<number> n = pdf.reserveobj("annot")
```

13.1.29 registerannot

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 \latelua.

```
pdf.registerannot (<number> objnum)
```

13.1.30 newcolorstack

This function allocates a new color stack and returns it's id. The arguments are the same as for the similar backend extension primitive.

```
pdf.newcolorstack("0 g","page",true) -- page|direct|origin
```

13.1.31 setfontattributes

This function will force some additional code into the font resource. It can for instance be used to add a custom ToUnicode vector to a bitmap file.

```
pdf.setfontattributes(<number> font id, <string> pdf code)
```

13.2 The pdfe library

The pdfe library replaces the epdf library and provides an interface to pdf files. It uses the same code as is used for pdf image inclusion. The pplib library by Paweł Jackowski replaces the poppler (derived from xpdf) library.

A pdf file is basically a tree of objects and one descends into the tree via dictionaries (key/value) and arrays (index/value). There are a few topmost dictionaries that start at root that are accessed more directly.

Although everything in pdf is basically an object we only wrap a few in so called userdata Lua objects.

pdf	**Lua**
null	nil
boolean	boolean
integer	integer
float	number
name	string
string	string
array	array userdatum
dictionary	dictionary userdatum
stream	stream userdatum (with related dictionary)
reference	reference userdatum

The regular getters return these Lua data types but one can also get more detailed information.

A document is laded from a file or string

```
<pdfe document> = pdfe.open(filename)
<pdfe document> = pdfe.new(somestring,somelength)
```

Such a document is closed with:

```
pdfe.close(<pdfe document>)
```

You can check if a document opened well by:

```
pdfe.status(<pdfe document>)
```

The returned codes are:

VALUE	EXPLANATION
-2	the document failed to open
-1	the document is (still) protected
0	the document is not encrypted
2	the document has been unencrypted

An encrypted document can be unencrypted by the next command where instead of either password you can give `nil`:

```
pdfe.unencrypt(<pdfe document>,userpassword,ownerpassword)
```

A successfully opened document can provide some information:

```
bytes = size(<pdfe document>)
major, minor = version(<pdfe document>)
n = nofobjects(<pdfe document>)
n = nofpages(<pdfe document>)
bytes, waste = nofpages(<pdfe document>)
```

For accessing the document structure you start with the so called catalog, a dictionary:

```
<pdfe dictionary> = pdfe.getcatalog(<pdfe document>)
```

The other two root dictionaries are accessed with:

```
<pdfe dictionary> = pdfe.gettrailer(<pdfe document>)
<pdfe dictionary> = pdfe.getinfo(<pdfe document>)
```

A specific page can conveniently be reached with the next command, which returns a dictionary.

```
<pdfe dictionary> = pdfe.getpage(<pdfe document>,pagenumber)
```

Another convenience command gives you the (bounding) box of a (normally page) which can be inheritted from the document itself. An example of a valid box name is `MediaBox`.

```
pages = pdfe.getbox(<pdfe document>,boxname)
```

Common values in dictionaries and arrays are strings, integers, floats, booleans and names (which are also strings) and these are also normal Lua objects:

```
s = getstring (<pdfe array|dictionary>,index|key)
i = getinteger(<pdfe array|dictionary>,index|key)
n = getnumber (<pdfe array|dictionary>,index|key)
b = getboolean(<pdfe array|dictionary>,index|key)
n = getname   (<pdfe array|dictionary>,index|key)
```

Normally you will use an index in an array and key in a dictionary but dictionaries also accept an index. The size of an array or dictionary is available with the usual # operator.

```
<pdfe dictionary>   = getdictionary(<pdfe array|dictionary>,index|key)
<pdfe array>        = getarray     (<pdfe array|dictionary>,index|key)
<pdfe stream>,
<pdfe dictionary>   = getstream    (<pdfe array|dictionary>,index|key)
```

These commands return dictionaries, arrays and streams, which are dictionaries with a blob of data attached.

Before we come to an alternative access mode, we mention that the objects provide access in a different way too, for instance this is valid:

```
print(pdfe.open("foo.pdf").Catalog.Type)
```

At the topmost level there are `Catalog`, `Info`, `Trailer` and `Pages`, so this is also okay:

```
print(pdfe.open("foo.pdf").Pages[1])
```

Streams are sort of special. When your index or key hits a stream you get back a stream object and dictionary object. The dictionary you can access in the usual way and for the stream there are the following methods:

```
okay   = openstream(<pdfe stream>,[decode])
         closestream(<pdfe stream>)
str, n = readfromstream(<pdfe stream>)
str, n = readwholestream(<pdfe stream>,[decode])
```

You either read in chunks, or you ask for the whole. When reading in chunks, you need to open and close the stream yourself. The `n` value indicates the length read. The `decode` parameter controls if the stream data gets uncompressed.

As with dictionaries, you can access fields in a stream dictionary in the usual Lua way too. You get the content when you 'call' the stream. You can pass a boolean that indicates if the stream has to be decompressed.

In addition to the interface described before, there is also a bit lower level interface available.

```
key, type, value, detail = getfromdictionary(<pdfe dictionary>,index)
type, value, detail = getfromarray(<pdfe array>,index)
```

TYPE	MEANING	VALUE	DETAIL
0	none	nil	
1	null	nil	
2	boolean	boolean	
3	boolean	integer	
4	number	float	
5	name	string	
6	string	string	hex
7	array	arrayobject	size
8	dictionary	dictionaryobject	size
9	stream	streamobject	dictionary size
10	reference	integer	

A `hex` string is (in the pdf file) surrounded by <> while plain strings are bounded by <>.

All entries in a dictionary or table can be fetched with the following commands where the return values are a hashed or indexed table.

```
hash = dictionarytotable(<pdfe dictionary>)
list = arraytotable(<pdfe array>)
```


You can get a list of pages with:

```
{ { <pdfe dictionary>, size, objnum }, ... } = pagestotable(<pdfe document>)
```

Because you can have unresolved references, a reference object can be resolved with:

```
<pdfe dictionary|array|stream> = getfromreference(<pdfe reference>)
```

There is an experimental function `pdfe.new` that takes three arguments:

VALUE	EXPLANATION
`stream`	this is a (in low level Lua speak) light userdata object, i.e. a pointer to a sequence of bytes
`length`	this is the length of the stream in bytes (the stream can have embedded zeros)
`name`	optional, this is a unique identifier that is used for hashing the stream, so that multiple doesn't use more memory

The third argument is optional. When it is not given the function will return an `pdfe` document object as with a regular file, otherwise it will return a filename that can be used elsewhere (e.g. in the image library) to reference the stream as pseudo file.

Instead of a light userdata stream (which is actually fragile but handy when you come from a library) you can also pass a Lua string, in which case the given length is (at most) the string length.

The function returns an `pdfe` object and a string. The string can be used in the `img` library instead of a filename. You need to prevent garbage collection of the object when you use it as image (for instance by storing it somewhere).

Both the memory stream and it's use in the image library is experimental and can change. In case you wonder where this can be used: when you use the swiglib library for `graphicmagick`, it can return such a userdata object. This permits conversion in memory and passing the result directly to the backend. This might save some runtime in one-pass workflows. This feature is currently not meant for production and we might come up with a better implementation.

13.3 The `pdfscanner` library

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

```
pdfscanner.scan (<pdfe stream>, <table> operatortable, <table> info)
pdfscanner.scan (<pdfe array>, <table> operatortable, <table> info)
pdfscanner.scan (<string>, <table> operatortable, <table> info)
```

The first argument should be a Lua string or a stream or array onject coming from the `pdfe` library. The second argument, `operatortable`, should be a Lua table where the keys are pdf operator name strings and the values are Lua functions (defined by you) that are used to process those operators. The functions are called whenever the scanner finds one of these pdf operators

in the content stream(s). The functions are called with two arguments: the `scanner` object itself, and the `info` table that was passed are the third argument to `pdfscanner.scan`.

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

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

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

```
local operatortable = { }

operatortable.Do = function(scanner,info)
    local resources = info.resources
    if resources then
        local val     = scanner:pop()
        local name    = val[2]
        local xobject = resources.XObject
        print(info.space .. "Uses XObject " .. name)
        local resources = xobject.Resources
        if resources then
            local newinfo =  {
                space     = info.space .. "  ",
                resources = resources,
            }
            pdfscanner.scan(entry, operatortable, newinfo)
        end
    end
end

local function Analyze(filename)
    local doc = pdfe.open(filename)
    if doc then
        local pages = doc.Pages
        for i=1,#pages do
            local page = pages[i]
            local info = {
              space     = "  " ,
              resources = page.Resources,
            }
            print("Page " .. i)
         -- pdfscanner.scan(page.Contents,operatortable,info)
            pdfscanner.scan(page.Contents(),operatortable,info)
        end
    end
```


```
end

Analyze("foo.pdf")
```

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 `pdfe` functions, let's focus on the type `pdfscanner` specific code instead.

From the bottom up, the following line runs the scanner with the pdf page's top-level content given in the first argument.

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

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

The function `Do` prints the name of the current `XObject`, and then starts a new scanner for that object's content stream, under the condition that the `XObject` is in fact a `/Form`. That nested scanner is called with new `info` argument with an updated `space` value so that the indentation of the output nicely nests, and with a new `resources` field to help the next iteration down to properly process any other, embedded `XObjects`.

Of course, this is not a very useful example in practice, but for the purpose of demonstrating `pdfscanner`, it is just long enough. It makes use of only one `scanner` method: `scanner:pop()`. That function pops the top operand of the internal stack, and returns a Lua table where the object at index one is a string representing the type of the operand, and object two is its value.

The list of possible operand types and associated Lua value types is:

TYPES	TYPE
`integer`	<number>
`real`	<number>
`boolean`	<boolean>
`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 you 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 and in case of `dict`, the table keys are pdf name strings and the values are `pop` return values.

There are a few more methods defined that you can ask `scanner`:

METHOD	EXPLANATION
`pop`	see 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 ends at the end of the stream, possibly wasting cpu cycles.

Topics

Primitives

This register contains the primitives that are mentioned in the manual. There are of course many more primitives. The LuaTEX primitives are typeset in bold. The primitives from pdfTEX are not supported that way but mentioned anyway.

Callbacks

Nodes

This register contains the nodes that are known to LuaTEX. The primary nodes are in bold, whatsits that are determined by their subtype are normal. The names prefixed by `pdf_` are backend specific.

Statistics

The following fonts are used in this document:

used	filesize	version	filename
22	1.622.732	5.960	cambria.ttc
4	827.080	5.960	cambriai.ttf
11	163.452	1.802	LucidaBrightMathOT-Demi.otf
11	348.296	1.802	LucidaBrightMathOT.otf
4	73.284	1.801	LucidaBrightOT.otf
22	733.500	1.958	latinmodern-math.otf
1	64.684	2.004	lmmono10-regular.otf
1	64.160	2.004	lmmonoltcond10-regular.otf
4	111.536	2.004	lmroman10-regular.otf
22	525.008	1.106	texgyredejavu-math.otf
22	601.220	1.632	texgyrepagella-math.otf
4	144.472	2.004	texgyrepagella-regular.otf
1	693.876	2.340	DejaVuSans-Bold.ttf
1	741.536	2.340	DejaVuSans.ttf
4	318.392	2.340	DejaVuSansMono-Bold.ttf
3	335.068	2.340	DejaVuSansMono.ttf
9	345.364	2.340	DejaVuSerif-Bold.ttf
1	336.884	2.340	DejaVuSerif-BoldItalic.ttf
1	343.388	2.340	DejaVuSerif-Italic.ttf
4	367.260	2.340	DejaVuSerif.ttf
152	**8.761.192**		**20 files loaded**