luametatex

the manual

version 2.11.06

dev id 20241031

Contents

introduction

Introduction

The LuaMetaTEX manual that is a variant of the LuaTEX manual provides an overview similar to its parent. Instead of adding more and more to that one, an alternative take is provided. Here we start less form a historic perspective and treat the engine as independent development. The main reason for this is that we want to focus on ConTEXt, if only because that is the macro package that uses it and also drives the development.

In LuaMetaTEX we go further than in LuaTEX. We extend the language, refactor most subsystems and assume that the macro package adapts to that. Of course we are compatible as much as possible with predecessors but we also take the freedom to tune some default behavior. For instance, moving on with math rendering means that we can make assumptions with respect to fonts and because the math fonts have issues that never will be solved we assume that the macro package is not only to feed the engine with tweaked fonts that can use the engine to its maximum extend. The same is true for more mechanism, like for instance the par builder, where we introduce multiple paragraph line break passes using features not present in other engines. Although extensions like these are not discussed here we do have to describe the underlying mechanisms and interfaces and thereby assume usage as in ConTEXt.

A manual like this evolves over time and will take years to complete. These are volunteer efforts unless some project makes it possible to spend more time on it. In practice most work on TEX development is unpaid for and therefore mostly driven by the joy of playing with typesetting and coming up with solutions for problems that users present us. Keep that in mind when reading and wondering why the focus is not on what you expect or what is best for marketing.

This manual replaces the older LuaMetaTEX manual. It has some less and some more than its predecessor which was derived from the LuaTEX manual. It will take some time to 'complete'. Eventually I might add a few registers but it makes only sense when the manual is more stable and I have to be in the mood to spend time on it.

Author	Hans Hagen & friends
ConTEXt	2024.10.31 11:32
LuaMetaTEX	2.11.06 (dev id: 20241031)
Support	contextgarden.net & tug.org

engines

1 Engines

1.1 Introduction

There are good reasons why we started the LuaTEX and later LuaMetaTEX projects. Here I will go into some of them. It is just short wrap up of how it started, how other engines influenced the process and how we see usage. There are plenty of documents out there that go into more detail. The main objective of this section is to put documentation into perspective.

1.2 How it started

When we started with ConTEXt, hardware was rather limited compared to what we have today. A personal computer had some 640kB memory, possibly bumped to 1MB with help from a memory extender. This put some restrictions to how macro packages could be defined, also because that memory had to be shared with the baseline operating system. However, over time memory and runtime became less of an issue and the TEX engine could be configured to use whatever was available. Extending the program other than increasing the available memory became more feasible.

As with any program, there is always something to wish for which is why the ε-TEX variant came into view. Before those extensions could be used, pdfTEX showed up. That variant simplified the 'TEX plus separate backend driver' model to a one-step process. Eventually ε-TEX was merged into pdfTEX, and that became the de facto standard engine. There was never a follow up on ε-TEX, and more drastic deviations like Omega were never ready for production. At some point XƎTEX came around but that was mostly a font specific extension. We were kind of stuck with a wish-list that never would be fulfilled but we occasionally pondered a follow up. We drafted an extended ε-TEX proposal, played with some features related to pdf, improved a few things but that was it.

Having some experiences with Lua as extension language in SciTE, I wondered what something like that would bring to TEX and after discussing this with Hartmut he made variant of pdfTEX that has some basic interfaces: we could access properties of registers and print something to TEX as if it came from file. As is common with some variant, a new name was coined and LuaTEX came into existence. We're talking 2005.

Because Idris wanted to typeset high quality scholar manuscripts mixing Arabic and Latin we discussed how to do that in ConTEXt and his experiences with Omega were such that alternatives had to be considered: the Oriental TEX project was started and LuaTEX was the starting point. Taco merged some parts of Aleph (a somewhat stable variant of Omega) into the code base and stepwise some primitives were added. It was overall a rather large and serious project that took a lot of our time. It was not commercially driven, mostly for ConTEXt users and therefore also a lot of fun to do. As often with such projects, early adapters keep things going.

It took a while before LuaTEX was stable in the sense that nothing more was added. Because the engine was developed alongside what is called ConTEXt MkIV, we could easily adapt both to each other. Even better: users could use both in production. However, in order for other macro packages to use LuaTEX (per request) it had to be frozen, and that happened around 2015, some 10 years after we started. However, we were not done yet and in order not to violate this stability principle the follow up was called LuaMetaTEX. Because it was a more drastic extension project, and also a somewhat drastic separation of the code base from the complex LuaTEX one, the related ConTEXt code was also separated, this time tagged MkXL, or LMTX when we talk about the combination. The project started around 2019 and soon again entered a state of combined development and use in production and most users switched to this variant.

There are more complete wrap ups of these developments and we systematically reported on them in various documents that are available in the distribution and/or published in user group journals.

1.3 The engines

Of course all starts with original TEX. We want to be compatible so we keep that functionality. However, for practical reasons LuaMetaTEX omits two core components. Font loading is not present in the frontend and there is no backend. Both are supposed to be provided via Lua plugins. This makes sense because in the meantime font technologies have changed and keep changing and backend also are a moving target. In ConTEXt we already did all that in Lua, so there was no need to keep that font and pdf generation code around in the engine. There are a few more deviations, like dropping some system specific features (terminal related) and in former times practical features like outer and long macros that no longer made sense and complicated integrating new features unnecessarily.

As mentioned in the introduction pdfTEX is the basis for LuaTEX and LuaTEX is where we started with LuaMetaTEX. If we compare pdfTEX with traditional TEX the main additions are:

- There is an integrated pdf backend that also supports for instance hyperlinks and various annotations.
- Expansion of glyphs (aka hz) has been added to the engine and integrated in the par builder. The same is true for character protrusion (in the margin).
- There is, to some extend, support for inter-character kerning.
- There are some handy helpers, for instance for calculating hashes, randomization, etc.
- There is an extension to injection between lines (adjust).
- We have few more conditionals (like testing for a csname and absolute values).
- A few helpers like **`\quitvmode`** (that we liked to have in ConTEXt) were added.

Because pdfTEX was actively developed and maintained over many years, extensions showed up stepwise, also depending on usage and needs. That is also why the ε-TEX extensions were included:

- More that 256 registers, including marks.
- Access to discarded material in the vertical splitting code.
- Protection against expansion of macros (the **`\protected`** prefix).
- A simple right to left typesetting mechanism.
- Access to some states, a limited set of last nodes, etc.
- There are some additional tracing features.
- One can reprocess tokens and produce detokenized lists.

In LuaTEX we also looked at what Omega could bring:

- More that 256 registers.
- Multi-directional typesetting.
- Local boxes (in lines).
- Input processing.

If we combine these lists, we see font expansion and protrusion coming back in LuaMetaTEX. However, already in LuaTEX expansion and protrusion were dealt with a bit differently and even more so in LuaMetaTEX, while protection in LuaMetaTEX is implemented differently. We also kept injection of vertical material but in LuaMetaTEX that done quite differently. Most if ε-TEX is there but not right to left typesetting and the register approach. Of course we kept the additional conditionals but implemented them a bit different.

In LuaTEX we took the Omega enlarged register approach and directional typesetting although that has been stripped down and redone to right to left only. Local boxes are there but redone in LuaMetaTEX. There was no need for input processing because we have Lua. In the end there is little that we kept from the other engines which also means that one cannot take the manuals that come with these engines and simply assume that it is there.

We should of course mention MetaPost. That graphical subsystem was integrated in LuaTEX and on the one hand stripped down (less backend) and extended (remove bottlenecks and add some functionality) in LuaMetaTEX. With respect to Lua we moved to more recent versions and dropped support for just in time compilation.

There is of course a lot in LuaTEX that can be found also in LuaMetaTEX but the later one goes way beyond its predecessor. It actually provides what we always wanted (as ConTEXt developers) but never showed up. And this brings us to a next topic.

1.4 Usage

Why is it that there has been little fundamental development around TEX engines? One of the reasons is that macro packages have to be stable. New features can be added but if they are only available in one engine (and there are a few more around now, like XƎTEX and the cjk specific ones) a macro package has to provide ways around them when they are not available. Risking some criticism I dare to say that in order to use LuaTEX to its full potential, macro package has to be set up such that this is possible and ConTEXt does just that. When we talk backends it's relatively easy, and when we talk fonts it's doable. But if you are not willing to adapt the core of your code dramatically (and conceptually) all you get from LuaTEX is a built-in scripting language and some occasional messing around with node lists. In ConTEXt we could transition rather well because the user interfaces permitted to do so without users noticing. Of course there were changes, for instance because encodings matter less, which is also true for e.g. XƎTEX, and font technologies changed. But for macro packages other than ConTEXt just the availability of Lua might be enough reasons to use that engine. That also means that documentation of the more intricate features is less important: one can just learn by example and ConTEXt is that example.

With LuaMetaTEX we go further because here one really has to make some fundamental choices. Again this could be done within the existing user interfaces, but here we are not only talking of fundamental improvements, like rendering math or breaking paragraphs into lines, but also of more flexible handling of alignments, inserts, adjusts, marks, par and page building, etc. Basically all mechanism got extended and opened up. In order to profit from this you have to be able to throw away existing solution and use these extensions to come up with better ones. If one can put sentiments aside, this also takes quite some time.

A very important aspect (at least for me) is that I want the macro code to look nice and in that respect stick to the TEX syntax as much as possible. That means that we have more programming related primitives, enhanced macro argument parsing, more (flexible) conditionals, additional registers, extra expansion related features and so on. Instead of some intermediate layer (like the helpers in ConTEXt) we can stick closer to the language itself. Of course this is not something that most users will notice. What users might notice, is that on the average ConTEXt with LuaMetaTEX performs better than with

LuaTEX or even LuaMetaTEX. Even with more performance critical components delegated to Lua (like the backend pdf generation) we gain and often can compete performance wise well with the faster eight bit pdfTEX engine.

The fact that one has to make (and cannot make) drastic choices has a consequence for documentation. Most of what is new and interesting is discussed in articles and low level manuals. However, it is often discussed in the perspective of ConTEXt. Although we do discuss and show generic solutions it makes little sense to go into details there simply because in the end only ConTEXt will use them as intended. It's just a waste of time to implement variants that are more generic because they will never be used elsewhere, especially in situations where the solutions are considered ‘standard’ and will not change. In ConTEXt we always followed the principle that if we can do better, we will do better, and interfaces are such that this can be done.

Of course that brings up the question “How do you know that these are the best solutions” and the answer is that we don't. However, we're not talking of quick and dirty solutions. For instance it took years to enhance math support: experiments, discussion, reconsideration, documenting, writing articles, looking at usage, fonts, etc. A wider discussion would not have brought better solutions, if at all. If that were the case, there would already have been successors. The same is true for most extensions: there was little need for them outside the ConTEXt community. So in the end that's what those interested should look at: how is LuaMetaTEX used in ConTEXt. It is the combined development together with acceptance by users that makes this possible,

constructions

2 Constructions

2.1 Introduction

This is more a discussion of the way some constructs in for instance math work. It will never be exhausting and mostly is for our own usage. We don't discuss all the options bit many are interfaced in higher level macros in ConTEXt. This chapter will gradually grow, depending on time and mood.

2.2 Boxes

Boxes are very important in TEX. We have horizontal boxes and vertical boxes. When you look at a page of text, the page itself is a vertical box, and among other things it packs lines that are themselves horizontal boxes. The lines that make a paragraph are the result of breaking a long horizontal box in smaller pieces.

This is a vertical box. It has a few lines
of text that started out as one long line
but has been broken in pieces. Doing
this as good as possible is one of TEX's
virtues.

There is a low level manual on boxes so here we can limit the discussion to basics. A box is in TEX speak a node. In traditional TEX it has a width, height, depth and shift.

Here we see a box and the gray line is called the baseline, the height goes up and the depth goes down. Normally the height and depth are determined by what goes in the box but they can be changed as we like.

```
\setbox\scratchboxone\ruledhpack{SHIFT 1}
\setbox\scratchboxtwo\ruledhpack{SHIFT 2}

\boxshift\scratchboxtwo 1ex \dontleavehmode \box\scratchboxone\box\scratchboxtwo

\setbox\scratchboxone\ruledvpack{SHIFT 3}
\setbox\scratchboxtwo\ruledvpack{SHIFT 4}

\boxshift\scratchboxtwo 1ex \box\scratchboxone\box\scratchboxtwo
```

In this example you'll notice that the shift depends on the box being horizontal or vertical. The primitives `\raise`, `\lower`, `\moveleft` and `\moveright` can be used to shift a box.

SHIFT 1 SHIFT 2
SHIFT 3
SHIFT 4

The reason why we have the shift property is that it is more efficient than wrapping a box in another box and shifting with kerns. In that case we also have to go via a box register so that we can manipulate the final dimensions. Another advantage is that the engine can use shifts to position for instance elements in a math formula and even the par builder used shifts to deal with positioning the lines according to shape and margin. In LuaMetaTEX the later is no longer the case.

Inside a box there can be mark (think running headers), insert (think footnotes) and adjust (think injecting something before or after the current line) nodes. The par builder will move this from inside the box to between the lines but when boxes are nested too deeply this won't happen and they get lost. In LuaMetaTEX these objects do bubble up because we make them box properties. So, in addition to the dimensions and shift a box also has migration fields.

In the low level manuals you can find examples of accessing various properties of boxes so here we stick to a short description. The reason for mentioning them is that it gives you an idea of what goes on in the engine.

field	usage
`width`	the (used) width
`height`	the (used) height
`depth`	the (used) depth
`shift_amount`	the shift (right or down)
`list`	pointer to the content
`glue_order`	the calculated order of glue stretch of shrink
`glue_sign`	the determined sign of glue stretch of shrink
`glue_set`	the calculated multiplier for glue stretch or shrink
`geometry`	a bit set registering manipulations
`orientation`	positional manipulations
`w_offset`	used in horizontal movement calculations
`h_offset`	used in vertical movement calculations
`d_offset`	used in vertical movement calculations
`x_offset`	a horizontal shift independent of dimensions
`y_offset`	a vertical shift independent of dimensions
`axis`	the math axis
`dir`	the direction the box goes to (l2r or r2l)
`package_state`	a bitset indicating how the box came to be as it is
`index`	a (system dependent) identifier
`pre_migrated`	content bound to the box that eventually will be injected
`post_migrated`	idem
`pre_adjusted`	idem
`post_adjusted`	idem
`source_anchor`	an identifier bound to the box
`target_anchor`	idem
`anchor`	a bitset indicating where and how to anchor
`except`	carried information about additional virtual depth
`exdepth`	additional virtual depth taken into account in the page builder

We have the usual dimension but also extra ones that relate to **`\boxxoffset`** and **`\boxyoffset`** (these are virtual) as well as **`\boxxmove`** and **`\boxymove`** (these influence dimensions). The **`\boxorientation`** also gets registered. The state fields carry information that is used in various places, the pre and post fields relate to the mentioned embedded content. Anchors are just there so that a macro package can play with this and excepts refer to an additional dimensions that is looked at in the page builder, for instance in order to prevent a page break at an unlucky spot. It all gives an indication of what we are dealing with.

2.3 Math style variants

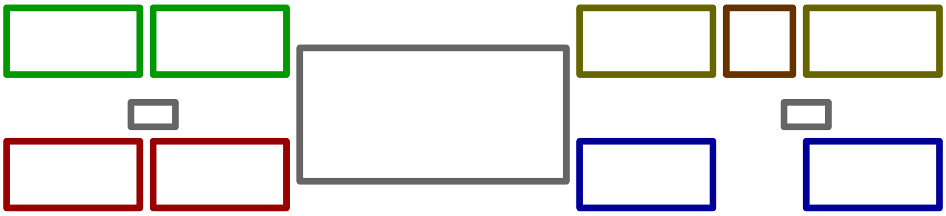
The LuaMetaTEX math engine is a follow up on the one in LuaTEX. That one gradually became more configurable in order to deal with both traditional fonts and OpenType fonts. In LuaMetaTEX much has been redone, opened up and extended. New mechanisms and constructs have been added. In the process hard coded heuristics with regards to math styles inside constructions were made configurable, a feature that is probably not used much, apart from experimenting. A side effect is that we can show how the engine is set up, so we do that when applicable.

construct	value	preset name
\Umathoverlinevariant	0x11335577	cramped
\Umathunderlinevariant	0x01234567	normal
\Umathoverdelimitervariant	0x45456767	small
\Umathunderdelimitervariant	0x45456767	small
\Umathdelimiterovervariant	0x01234567	normal
\Umathdelimiterundervariant	0x01234567	normal
\Umathhextensiblevariant	0x01234567	normal
\Umathvextensiblevariant	0x01234567	normal
\Umathfractionvariant	0x11335577	cramped
\Umathradicalvariant	0x11335577	cramped
\Umathaccentvariant	0x11335577	cramped
\Umathdegreevariant	0x67676767	doublesuperscript
\Umathtopaccentvariant	0x11335577	cramped
\Umathbottomaccentvariant	0x11335577	cramped
\Umathoverlayaccentvariant	0x11335577	cramped
\Umathnumeratorvariant	0x23456767	numerator
\Umathdenominatorvariant	0x33557777	denominator
\Umathsuperscriptvariant	0x45456767	small
\Umathsubscriptvariant	0x55557777	subscript
\Umathprimevariant	0x45456767	small
\Umathstackvariant	0x23456767	numerator

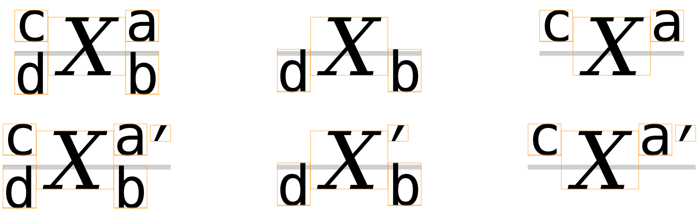
2.4 Math scripts

The basic components in a math formula are characters, accents, fractions, radicals and fences. They are represented in the to be processed node list as noads and eventually are converted in glyph, kern, glue and list nodes. Each noad carries similar but also specific information about its purpose and intended rendering. In LuaMetaTEX that is quite a bit more than in traditional TEX.

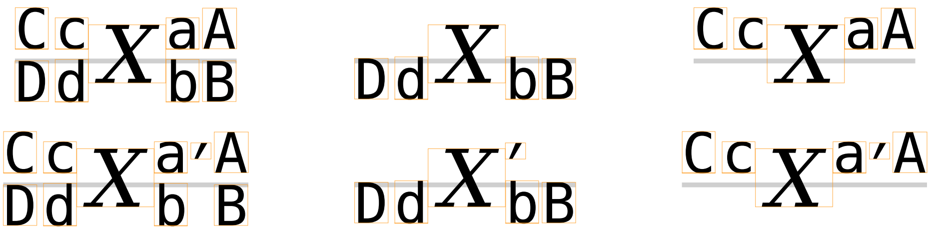
These noads are often called atoms. The center piece in a noad is called the nucleus. The fact that these noads also can have scripts attached makes them more like molecules. Scripts can be attached to the left and right, high or low. That makes fours of them: pre/post super/sub scripts. In LuaMetaTEX we also have a prime script, which comes on its own, above a post subscript or after the post superscript, if given.



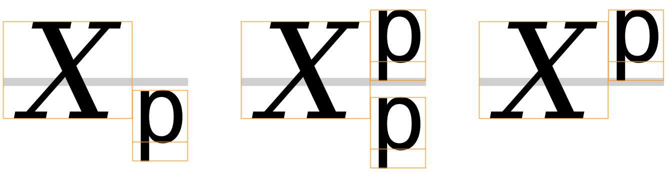
Here the raised rectangle represents the prime. The large center piece is the nucleus. Four scripts are attached to the nucleus. The two smaller center pieces indicate follow up atoms. They make it possible to have multiple pre- and postscripts. For single scripts we get combinations like these:



And for multiple (there can be more that two) we get this assembly:



It will be clear that there is quite a bit of code involved in dealing with this because these scripts are not only to be anchored relative to the nucleus but also to each other. The dimensions of the scripts determine for instance how close a combined super and subscript are positioned.



The rendering of scripts involves several parameters, of which some relate to font parameters. In LuaMetaTEX we have a few more variables and we also overload font parameters, if only because only a few make sense and it looks like font designers just copy values from the first available fonts so in the end we can as well use our own preferred values.

The following parameters play a role in rendering the shown assembly, The traditional TEX engine expects a math font to set quite some parameters for positioning the scripts but has no concept of prescripts and neither has OpenType. This is why we have extra parameters (and for completeness we also have them for the post scripts). One can wonder of font parameters make sense here because in the end we can decide for a better visual result with different ones. After all, assembling scripts is not really what fonts are about.

engine parameter	target	open type font	tex font
subscriptshiftdrop	post	SubscriptBaselineDropMin	subdrop
subscriptshiftdown	post	SubscriptShiftDown	sub1
subscriptsuperscriptshiftdown	post	SubscriptShiftDown[WithSuperscript]	sub2

	subscriptsuperscriptvgap	post	SubSuperscriptGapMin	4 rulethickness
	subscripttopmax	post	SubscriptTopMax	4/5 xheight
	superscriptshiftdrop	post	SuperscriptBaselineDropMax	supdrop
	superscriptbottommin	post	SuperscriptBottomMin	1/4 xheight
	superscriptshiftup	post	SuperscriptShiftUp[Cramped]	sup1 sup2 sup3
	superscriptsubscriptbottommax	post	SuperscriptBottomMaxWithSubscript	4/5 xheight
⋆	primeraise	prime	PrimeRaisePercent	
⋆	primeraisecomposed	prime	PrimeRaiseComposedPercent	
⋆	primeshiftup	prime	PrimeShiftUp[Cramped]	
⋆	primeshiftdrop	prime	PrimeBaselineDropMax	
⋆	primespaceafter	prime	PrimeSpaceAfter	
	spaceafterscript	post	SpaceAfterScript	**`\scriptspace`**
⋆	spacebeforescript	post	SpaceBeforeScript	
⋆	spacebetweenscript	multi	SpaceBetweenScript	
⋆	extrasuperscriptshift	pre		
⋆	extrasuperprescriptshift	pre		
⋆	extrasubscriptshift	pre		
⋆	extrasubprescriptshift	pre		
⋆	extrasuperscriptspace	post		
⋆	extrasubscriptspace	post		
⋆	extrasuperprescriptspace	pre		
⋆	extrasubprescriptspace	pre		

The parameters marked by a ⋆ are LuaMetaTEX specific. Some have an associated font parameter but that is not official OpenType. For a very long time we had only a few math fonts but even today most of these fonts seem to use values that are similar to the ones TEX uses. In that respect one can as well turn them into rendering specific ones. After all, changes are slim that a formula rendered by TEX or e.g. MS Word are metric compatible and with the advanced spacing options in LuaMetaTEX we're even further off. Also keep in mind that the TEX font parameters could be overloaded at the TEX end.

The spacing after a (combination of) postscript(s) is determined by 'space after script' and the spacing before a (combination of) prescript(s) by 'space before script'. If we have multi-scripts the 'space between script' kicks in and the space after the script is subtracted from it. The given space between is scaled with the **`\scriptspacebetweenfactor`** parameter.

The default style mapping that we use are the same as those (hard coded) in regular TEX and those for prime scripts are the same as for superscripts.

subscriptvariant

current style	mapping	used style
display	`0x55557777`	crampedscript
crampeddisplay	`0x55557777`	crampedscript
text	`0x55557777`	crampedscript
crampedtext	`0x55557777`	crampedscript
script	`0x55557777`	crampedscriptscript
crampedscript	`0x55557777`	crampedscriptscript
scriptscript	`0x55557777`	crampedscriptscript
crampedscriptscript	`0x55557777`	crampedscriptscript

superscriptvariant

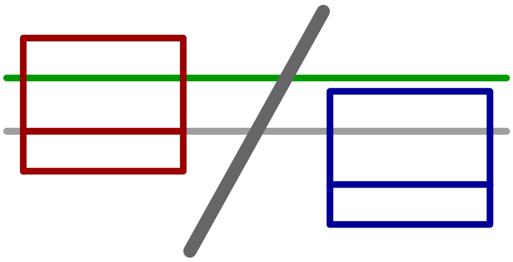
current style	mapping	used style
display	0x45456767	script
crampeddisplay	0x45456767	crampedscript
text	0x45456767	script
crampedtext	0x45456767	crampedscript
script	0x45456767	scriptscript
crampedscript	0x45456767	crampedscriptscript
scriptscript	0x45456767	scriptscript
crampedscriptscript	0x45456767	crampedscriptscript

primevariant

current style	mapping	used style
display	0x45456767	script
crampeddisplay	0x45456767	crampedscript
text	0x45456767	script
crampedtext	0x45456767	crampedscript
script	0x45456767	scriptscript
crampedscript	0x45456767	crampedscriptscript
scriptscript	0x45456767	scriptscript
crampedscriptscript	0x45456767	crampedscriptscript

2.5 Skewed fractions

Skewed fractions are native in LuaMetaTEX. Such a fraction is a horizontal construct with the numerator and denominator shifted up and down a bit. It looks like this:



The rendering is driven by some parameters that determine the horizontal and vertically shifts but we found that the ones given by the font make no sense (and are not that well defined in the standard either). The horizontal shift relates to the width (and angle) of the slash and the vertical relates to the math axis. We don't listen to 'skewed fraction hgap' nor to 'skewed fraction vgap' but use the width of the middle character, normally a slash, that can grow on demand and multiply that with a `hfactor` that can be passed with the fraction command. A `vfactor` is used a multiplier for the vertical shift over the axis. Examples of (more)) control can be found in the ConTEXt math manual. Here we just show a few examples that use `\vfrac` with its default values.

$${}^{1}/_{2} \qquad {}^{a}/_{b} \qquad {}^{b}/_{a}$$

$${}^{x^2}/_{x^3} \qquad {}^{(x+1)}/_{(x+2)} \qquad {}^{x+1}/_{x+2}$$

The quality of the slashes differs per font, some lack granularity in sizes, others have inconsistent angles between the base character and larger variants.

The following commands are used:

`\Uskewed`
`\Uskewedwithdelims`

There are some parameter involved:

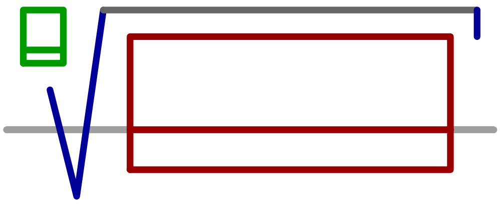
`\Umathskeweddelimitertolerance`
`\Umathskewedfractionhgap`
`\Umathskewedfractionvgap`

2.6 Math fractions

todo

2.7 Math radicals

Radicals indeed look like roots. But the radical mechanism basically is a wrapping construct: there's something at the left that in traditional TEX gets a rule appended. The left piece is an extensible, so it first grow with variant glyphs and when we run out if these we get an upward extensible with a repeated upward rule like symbol that then connect with the horizontal rule. In LuaMetaTEX the horizontal rule can be an extensible (repeated symbol) and we can also have a symbol at the right, which indeed can be a vertical extensible too.



Here are some aspects to take care of when rendering a radical like this:

- The radical symbol goes below the baseline of what it contains.
- There is some distance between the left symbol and the body.
- There is some distance between the top symbol and the body.
- There is some distance between the right symbol and the body.
- The degree has to be anchored properly and possibly can stick out left.
- The (upto) three elements have to overlap a little to avoid artifacts.
- Multiple radicals might have to be made consistent with respect to heights and depths.

Involved commands:

`\Uradical`
`\Uroot`
`\Urooted`

Relevant parameters:

`\Umathradicaldegreeafter`

```
\Umathradicaldegreebefore
\Umathradicaldegreeraise
\Umathradicalextensibleafter
\Umathradicalextensiblebefore
\Umathradicalkern
\Umathradicalrule
\Umathradicalvariant
\Umathradicalvgap
```

2.8 Math accents

todo

2.9 Math fences

todo

assumptions

3 Assumptions

3.1 Introduction

Because the engine provides no backend there is also no need to document it. However, in ConTEXt we assume some features to be supported by its own backend. These will be collected here. This chapter is rather ConTEXt specific, for instance we have extended what can be done with characters and that is pretty much up to a macro package to decide.

3.2 Virtual fonts

Virtual fonts are a nice extension to traditional TEX fonts that originally was independent from the engine, which only needs dimensions from a tfm file. In LuaTEX, because it has a backend built in, virtual fonts are handled by the engine but here we also can construct such fonts at runtime. The original set of commands is:

char	+	chr sx sy
right	+	amount
down	+	amount
push	+	
pop	+	
font	+	index
nop	+	
special	–	str
rule	+	v h

The pdfTEX engine added two more but these are not supported in ConTEXt:

pdf	–	str
pdfmode	–	n

The LuaTEX engine also added some but these are never found in loaded fonts, only in those constructed at runtime. Two are not supported in ConTEXt.

lua	+	code	f(font,char,posh,posv,sx,sy)
image	–	n	
node	+	n	
scale	–	sx sy	

The LuaMetaTEX engine has nothing on board and doesn't even carry the virtual commands around. The backend can just fetch them from the Lua end. An advantage is that we can easily extend the repertoire of commands:

slot	+	index chr csx csy
use	+	index chr … chr
left	+	amount
up	+	amount
offset	+	h v chr [csx [csy]]
stay	+	chr (push/pop)

compose	+	h v chr	
frame	+	wd jt dp line outline advance baseline color	
line	+	wd ht dp color	
inspect	+		
trace	+		
<plugin>	+		f(posh,posv,packet)

There are some manipulations that don't need the virtual mechanism. In addition to the character properties like `width`, `height` and `depth` we also have:

advance		the width used in the backend
scale		an additional scale factor
xoffset		horizontal shift
yoffset		vertical shift
effect	slant	factor used for tilting
	extend	horizontal scale
	squeeze	vertical scale
	mode	special effects like outline
	weight	pen stroke width

internals

4 Internals

4.1 Introduction

If you look at TEX as a programming language and are familiar with other languages, a natural question to ask is what data types there are and how is all managed. Here I will give a general overview of some concepts. The explanation below is not entirely accurate because it tries to avoid the sometimes messy details. More can be found in the other low level manuals. I assume that one knows at least how to process a simple document with a few commands.

It is not natural to start an explanation with how memory is laid out but by doing this it is easier to introduce the concepts. I will focus on what is called hash table, the stack, node memory and token memory. We leave fonts, languages, character properties, math, etc. out of the picture. There are details that we skip because it's the general picture that matters here.

I might add some more to this manual, depending on questions by users at meetings or on the mailing list. Some details might change over time but the principles remain the same.

4.2 A few basics

This is a reference manual and not a tutorial. This means that we discuss changes relative to traditional TEX and also present new (or extended) functionality. As a consequence we will refer to concepts that we assume to be known or that might be explained later. Because the LuaTEX and LuaMetaTEX engines open up TEX there's suddenly quite some more to explain, especially about the way a (to be) typeset stream moves through the machinery. However, discussing all that in detail makes not much sense, because deep knowledge is only relevant for those who write code not possible with regular TEX and who are already familiar with these internals (or willing to spend time on figuring it out).

So, 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” too.

But ... if you're here because of Lua, then all you need to know is that you can call it from within a run. If you want to learn the language, just read the well written Lua book. 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 don't know what this means, you definitely need to have a look at the Lua book first.

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 like this, where a `par` node stores some metadata and is followed by a `hlist` flagged as indent box:

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

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 and nodes can have many properties but that will be discussed later. Each node points back to a previous node or next node, given that these exist. Sometimes multiple characters are represented by one glyph (shape), so one can also get:

```
[par] <=> [hlist] <=> H <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```

And maybe some characters get positioned relative to each other, so we might see:

```
[par] <=> [hlist] <=> H <=> [kern] <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```

Actually, the above representation is one view, because in LuaMetaTEX we can choose for this:

```
[par] <=> [glue] <=> H <=> [kern] <=> i <=> [glue] <=> Th <=> e <=> r <=> e ...
```

where glue (currently fixed) is used instead of an empty hlist (think of a **\hbox**). Options like this are available because want a certain view on these lists from the Lua end and the result being predicable is part of that.

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. At some point horizontal blobs are wrapped into vertical ones. 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.

```
\setbox\scratchbox\vbox\bgroup
    line 1\par line 2
\egroup
```

```
\showbox\scratchbox
```

The above code produces the next log lines that reveal how the engines sees a paragraph (wrapped in a **\vbox**):

```
1:4: > \box257=
1:4: \vbox[normal][16=1,17=1,47=1], width 483.69687, height 27.58083, depth 0.1416, direction l2r
1:4: .\list
1:4: ..\hbox[line][16=1,17=1,47=1], width 483.69687, height 7.59766, depth 0.1416, glue 455.40097fil, direction l2r
1:4: ...\list
1:4: ....\glue[left hang][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[left][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[parfillleft][16=1,17=1,47=1] 0.0pt
1:4: ....\par[newgraf][16=1,17=1,47=1], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 3000, adjdemerits 10000, linepenalty 10
   , doublehyphendemerits 10000, finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, emergencystretch 12.0,
   parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
1:4: ....\glue[indent][16=1,17=1,47=1] 0.0pt
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +00006C l
```

```
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +000069 i
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +00006E n
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +000065 e
1:4: ....\glue[space][16=1,17=1,47=1] 3.17871pt plus 1.58936pt minus 1.05957pt, font 30
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +000031 1
1:4: ....\penalty[line][16=1,17=1,47=1] 10000
1:4: ....\glue[parfill][16=1,17=1,47=1] 0.0pt plus 1.0fil
1:4: ....\glue[right][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[right hang][16=1,17=1,47=1] 0.0pt
1:4: ..\glue[par][16=1,17=1,47=1] 5.44995pt plus 1.81665pt minus 1.81665pt
1:4: ..\glue[baseline][16=1,17=1,47=1] 6.79396pt
1:4: ..\hbox[line][16=1,17=1,47=1], width 483.69687, height 7.59766, depth 0.1416, glue 455.40097fil, direction l2r
1:4: ...\list
1:4: ....\glue[left hang][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[left][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[parfillleft][16=1,17=1,47=1] 0.0pt
1:4: ....\par[newgraf][16=1,17=1,47=1], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 3000, adjdemerits 10000, linepenalty 10
   , doublehyphendemerits 10000, finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, emergencystretch 12.0,
   parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
1:4: ....\glue[indent][16=1,17=1,47=1] 0.0pt
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +00006C l
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +000069 i
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +00006E n
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +000065 e
1:4: ....\glue[space][16=1,17=1,47=1] 3.17871pt plus 1.58936pt minus 1.05957pt, font 30
1:4: ....\glyph[32768][16=1,17=1,47=1], language (n=1,l=2,r=3), hyphenationmode 499519, options 128 , font <30: DejaVuSerif @ 10.0pt>, glyph U
   +000032 2
1:4: ....\penalty[line][16=1,17=1,47=1] 10000
1:4: ....\glue[parfill][16=1,17=1,47=1] 0.0pt plus 1.0fil
1:4: ....\glue[right][16=1,17=1,47=1] 0.0pt
1:4: ....\glue[right hang][16=1,17=1,47=1] 0.0pt
```

The LuaMetaTEX 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 behavior, which still is the benchmark. When we refer to 'callbacks' we means these hooks. The TEX engine itself is pretty well optimized but when you kick in much Lua code, you will notices that performance drops. Don't blame and bother the authors with performance issues. In ConTEXt over 50% of the time can be spent in Lua, but so far we didn't get many complaints about efficiency. Adding more callbacks makes no sense, also because at some point the performance hit gets too large. There are plenty of ways to achieve goals. For that reason: take remarks about LuaMetaTEX, features, potential, performance etc. with a natural grain of salt.

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 support, 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. Each successive step expects a certain result and if you mess around to much, the engine eventually might bark and quit. It can even crash, because testing everywhere for what users can do wrong is no real option.

When you read about nodes in the following chapters it's good to keep in mind what commands relate to them. 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
\textdirection	`dir`	a change in text direction

Whatever we feed into TEX at some point becomes a token which is either interpreted directly or stored in a linked list. A token is just a number that encodes a specific command (operator) and some value (operand) that further specifies what that command is supposed to do. In addition to an interface to nodes, there is an interface to tokens, as later chapters will demonstrate.

Text (interspersed with macros) comes from an input medium. This can be a file, token list, macro body cq. arguments, some internal quantity (like a number), Lua, etc. Macros get expanded. In the process TEX can enter a group. Inside the group, changes to registers get saved on a stack, and restored after leaving the group. When conditionals are encountered, another kind of nesting happens, and again there is a stack involved. Tokens, expansion, stacks, input levels are all terms used in the next chapters. Don't worry, they loose their magic once you use TEX a lot. You have access to most of the internals and when not, at least it is possible to query some state we're in or level we're at.

When we talk about pack(ag)ing it can mean two things. When TEX has consumed some tokens that represent text they are added to the current list. When the text is put into a so called **\hbox** (for instance a line in a paragraph) it (normally) first gets hyphenated, next ligatures are build, and finally kerns are added. Each of these stages can be overloaded using Lua code. When these three stages are finished, the dimension of the content is calculated and the box gets its width, height and depth. What happens with the box depends on what macros do with it.

The other thing that can happen is that the text starts a new paragraph. In that case some information is stored in a leading `par` node. Then indentation is appended and the paragraph ends with some glue. Again the three stages are applied but this time afterwards, the long line is broken into lines and the result is either added to the content of a box or to the main vertical list (the running text so to say). This is called par building. At some point TEX decides that enough is enough and it will trigger the page builder. So, building is another concept we will encounter. Another example of a builder is the one that turns an intermediate math list into something typeset.

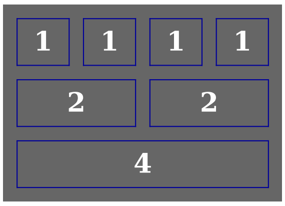
Wrapping something in a box is called packing. Adding something to a list is described in terms of contributing. The more complicated processes are wrapped into builders. For now this should be enough to enable you to understand the next chapters. The text is not as enlightening and entertaining as Don Knuths books, sorry.

4.3 Memory words

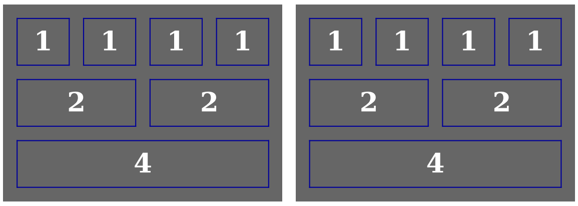
Before we come to know that TEX manages most of it memory itself. It allocates arrays of (pairs of) 32 bit integers because that is what TEX uses all over the place: integers. They store integer numbers of various ranges values, fixed point floats, pointers (indices in arrays), states, commands, and often groups of them travel around the system.

integer	:	mostly 8, 16, 24, 32 but we have odd packing too
fixed point float	:	16.16 used to represent dimensions
boolean	:	simple state variables
enumerations	:	a choice from a set, like operators and operands
strings	:	an index in a string pool (character array)

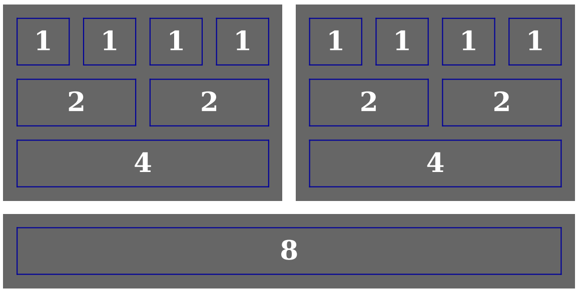
The main memory areas in TeX are therefore arrays integers or pairs of integers as we want to handle linked lists where in an element one integer has some data and the other points to another element. Keep in mind that when TeX showed up efficient memory management was best done by the application, especially when it had to be portable. This might seem odd now but is actually not that bad performance wise. One just has to get accustomed to the way TeX handles data.



Depending on usage we use four, two or one byte. Often a pair is used:



Such a pair is called a (memory) word and each component is a halfword that itself can have two quarterwords and four singlewords. In LuaMetaTeX we also can combine them:



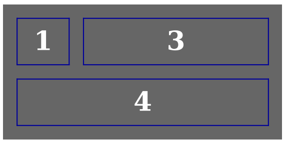
The eight byte field is used for pointers (to more dynamic structures) and double floats but that can only happen when multiple words are used as a combined data structure (as in a so called node, explained below). Quite often the second field is used as pointer to another pair. We could have changed that model in LuaTeX and LuaMetaTeX but there is little gain in that and we want to stay close to the well documented original as much as possible. It also has the side effect of simplifying the code and retain performance.[1]

4.4 Tokens

A token is a halfword, so a 32 bit integer as mentioned before. Here we use a one plus three model, not mentioned in the previous section. Sometimes we just look at the whole number, but quite often we

[1] In the source this is reflected in the names used: `vinfo` and `vlink` in these pairs but in LuaMetaTeX we often use more symbolic names.

look at the two smaller ones. The single byte is the so called command identifier (cmd), the second one traditionally is called character (chr), but what we're really talking about is an operator and operand kind of model. In a TeX engine source you can find variable names like `cur_cmd`, `cur_chr` and `cur_tok` were the third one combines the first two.



Tokens travel through the system as integers and when some action is required the command part is consulted which then triggers some action further defined by the character part. The combination can either directly trigger some action but often that action has to look ahead in order to get some more details.

Consider the following input:

```
\starttext
Hi there!

This is a \hbox{box}.
\stoptext
```

Every character falls in a category, and there are 16 of them. The `H` is a ‘letter’, the empty line a newline. The backslash is an ‘escape’ that tells the parser to scan for a command where the name is from letters. That command is then looked up and a token is created: in this case a ‘call’ command with as operand the memory address (an index in the to be discussed hash) where the start of a list of stored tokens can be found.

The characters in the text also become tokens and here we get two ‘letter’ commands (with the Unicode slots as operand), one ‘space’ command, five more letter commands and an ‘other’ command, and so on.

Here every token is fed into the interpreter. The **`\starttext`** is a macro (control sequence) so it gets expanded and its stored tokens get interpreted. The letters become (to be discussed) nodes in a linked list of content. In this case the tokens are not stored and discarded as we read on.

The **`\hbox`** is also a control sequence but a built in primitive. The operator is `make_box` and the operand is `hbox`. It will trigger making a box of the given kind by reading an optional specification, the left curly brace (begin group) collects content, and when the right curly brace (end group) is seen wraps up by packaging the result. Al that is hard coded, contrary to a macro, but one can of course define **`\hbox`** as macro, which normally is a bad idea.

As a side note: quite often TeX reads a token, and then puts if back into the input. For instance, when it expects a number or keyword it keeps reading till it is satisfied and when it ends up in the unexpected it has to wrap up and go one step back. However, when we read from file we can't go back, which is why TeX has a model of ‘input levels’. Pushing back boils down to creating a token list with this one token and then starts reading from that list. It is beyond this explanation to go into details but all you need to know is that TeX has various input sources, for instance files, token lists, arguments to commands (also token lists) and Lua output, but in the end all provide tokens.[2]

[2] We could use a double linked list in which case we would have a three integer element which is odd for TeX and has no real benefits as it would change the model completely.

1	**info**	**link**
n	**info**	**link**

So to wrap up tokens, we have either singular ones (just 32 bit integers encoding a command and value aka operator and operand) or a pair where the second one is a link. A token list starts at some index and the link is zero (end of list) or another index. Token memory is huge array of memory words like these. When token lists are constructed we take from this pool so there is an index indicating the first available token. When a list is discarded it gets appended to a list of free tokens. So in practice we first try to get a free token from this pool. In LuaMetaTeX it the token array will grow on demand with a configurable chunk size.

4.5 Nodes

We already mentioned nodes. These are slices from an array that hold some values that belong together. So again we have a large array of memory words but where a token is one pair a node is multiple. Nodes have different size. The first node starts at index 1 and when it needs four memory words the second node starts at index 5.

A character in the input that is typeset will become a glyph node of 112 bytes and a paragraph starts with a par node of 280 bytes. A space becomes a glue node of 56 bytes and every box that you (or TeX) make is 128 bytes. Most nodes are way larger in LuaMetaTeX than in traditional TeX but we don't have the memory constraints of those times.

Here it is worth noticing that where TeX has a dedicated subsystem for glue which make sharing space related glue efficient: the so called glue specifications are reference counted. In LuaTeX we made these normal nodes which is slightly less efficient but fits better in the opened up (Lua) interface and also has some other advantages (we leave it to reader to guess what).

For instance, a kern node at the time of this writing needs three memory words (as with other nodes we might add some more fields, like `options`).

3128	**type**	**subtype**	**next**
3129	**previous**		**attribute**
3130	**amount**		**expansion**

So here we take a slice of three memory words from the node array starting at index 3128. We mention this detail because sometimes (when tracing) you see these numbers. This doesn't mean that at that point we had 3128 nodes, because the next node taken from this pool will have number 3131. The numbers are indices!

In the source code we access thes enumber like this:

```
# define kern_amount(a)    vlink(a,2)
# define kern_expansion(a) vinfo(a,2)
```

So when $a = 3128$ the amount is found in the link field $a = 3128 + 2 = 3130$. The name link is somewhat weird here but that's the way these fields are called: `vlink` and `vinfo`. It could as well be `first` and `second` but by using macros we get away by abstraction. So now you can figure out what these references do:[3]

```
# define node_type(a)    vinfo0(a,0)
# define node_subtype(a) vinfo1(a,0)

# define node_next(a)    vlink(a,0)
# define node_prev(a)    vlink(a,1)
# define node_attr(a)    vinfo(a,1)
```

Not all nodes end up in a list that results in output, like paragraphs and pages. For instance **\parshape** and **\widowpenalties** also use nodes as storage container. Their common node is a specification node of 32 but with a pointer to a dynamically memory array.

Because the sizes differ one cannot simply have a list of free nodes (as with tokens) without some lookup mechanism that combines nodes when needed (they need to be next to each other) or split larger ones when we run out of nodes. In LuaTEX and LuaMetaTEX we keep a list of free nodes per size which in practice is more efficient and one seldom runs out of nodes because on the average a page has a similar distribution and when a page is flushed (or any box for that matter) nodes get freed. For instance right at this moment, we have 1120 nodes in use and 3178 glyphs in stock.[4]

4.6 The hash table

The engine has a lot of built-in commands and users can define additional ones. An example is macros, like the mentioned **\starttext** that refers to a token list that starts the typesetting process. When reading the input from file these commands and macros are looked up in a hash table. There are also built-in commands that generate a hash entry. For instance when you define a counter or a font, the given name becomes a hash entry that points to a memory location (again an index).

Here it gets more complex. A hash table is used to lookup primitive commands like **\hbox** and **\font** as well as **\starttext**. The string is converted into an integer within a specific range. That integer is then an index into a table like we saw before, with two halfwords per slot.

1	**next**	**string**
n	**next**	**string**

The hash value (integer calculated from string) point to a slot and the string is compared with the stored string. When the string is different, the next field points to a different slot (outside the hash range in the same table) and again the string is checked. When there is no next value set (zero), the index is used to determine what to do.

[3] In what order these two fields end up in memory depends on the cpu being little or big endian.
[4] And a while later (that is: here) these numbers are 1197 and 3101. These numbers can handly be called dramatic as a page can only have so many glyph nodes: 1301 and 2997 were the numbers after the colon.

1	**type**	**flags**	**level**	**value**
n	**type**	**flags**	**level**	**value**

This table is called the table of equivalents. In LuaMetaTEX this is implemented a bit different than in the other engines because we combine tables. The fields that you see here keep track of the type (so that we can optimize some bits and pieces), flags (so that we can implement overload protection), a level (so that we can restore values after the group ends and of course a value.

That value can be a a pointer to (index of) a token list, or a pointer to (index of) a node. It can also be just some value, like a dimension, character reference or register entry.

Although there are similarities, the memory mapping in LuaMetaTEX differs from LuaTEX and that one differs from pdfTEX which again differs from original TEX.

In original TEX table of equivalents is organized in six regions.

1. active characters
2. hash table
   font identifiers
3. glue
   muglue
4. token lists
   boxes
   font names
   math codes
   category codes
   lowercase codes
   uppercase codes
   space factors
5. integers
   delimiter codes
6. dimensions

The internal dimension, integer, skip, muskip, token and box registers are part of this and for users there are 256 registers of each category. There are 256 active characters, and the mentioned codes and factors also have 256 entries.

In LuaMetaTEX (like in LuaTEX) we use Unicode, so there it makes no sense to store values in the table of equivalents. We use dedicates hashes instead. So there we have different regions. In LuaTEX we roughly have this:

1. hash table
2. frozen control sequences
3. font identifiers
4. glue
5. muglue
6. tokens
7. boxes
8. integers
9. attributes
10. dimensions

As we moved forward, LuaMetaTEX has some more:

1. hash table
2. frozen control sequences
3. glue
4. muglue
5. tokens
6. boxes
7. integers
8. attributes
9. dimensions
10. posits
11. units
12. specifications

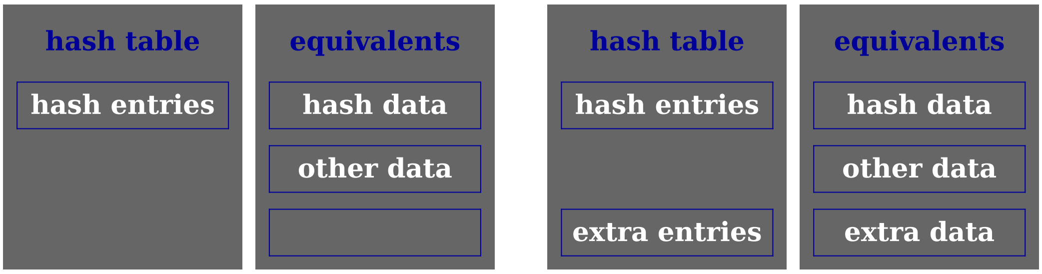
In case one wonders, on top of built-in units users can define their own. Specifications are for instance shape and penalty arrays. Fonts are not in here because we manage them in Lua.

In traditional TEX a delimiter code needs two integers so there it uses both fields in a memory word and saves the state in a parallel array with quarterwords. We don't need this in LuaMetaTEX because we store delimiters in a separate hash table (and actually don't need them at all, because we use OpenType fonts).

We need to keep some save/restore related state in the table but for integers and delimiter codes we need all four bytes of the value. Therefore original TEX has a separate parallel table for this, which as side effect spoils some memory. In LuaTEX we have way more registers so there the waste is larger.

In LuaMetaTEX we got rid of this. We could also use less space for the type and store some extra data. A side effect is that we keep the type information which is handy for tracing, sparse dumping, and optimizing save and restore. This is why with more functionality we don't need less more memory than one would expect.

The hash table in original TEX is a bit too small for larger macro packages which is why in practice engines took more than the default couple of thousands slots. But going too large makes no sense because one ends up with many misses and unused hash and equivalent space. That is why soon after TEX showed up support for extra hash space was introduced. That space is allocated at the end of normal hash space and can be configured when the format file is made. This means that the hash table also grows to the size of the equivalents table:



Too much extra hash space also means too much equivalent space as these arrays run in parallel. In LuaMetaTEX we can let hash memory grow on demand so there the penalty is less.

It makes sense to move the 'other data' to the beginning so that we can use a smaller hash but. That could potentially save 4MB memory, but when we decide to limit the maximum number of registers to 8K (instead of 64K) we are at 512KB so that might be easier as it avoids using offsets. And who knows how we can use the yet unused space later. Compared to LuaTEX we already save much memory elsewhere.

4.7 Save stack

I only mention this here because it relates to the table of equivalents. Whenever a quantity (register, parameter, macro, you name it) changes the engine registers the old value on the save stack when the assignment is local. The equivalent is replaced and when found in the save stack restored afterwards. In order to let the save stack not grow too much we try to only save a state when there is a real change. We can do that because we have a bit more information available and otherwise do a bit more testing. This is specific for LuaMetaTEX.

4.8 Data types

The long winding explanation explanation in the previous section shows that we have a curious mix of data to manage. We already saw tokens and nodes but here we also saw registers. However, integers, dimensions and attributes are all basically just 32 bit numbers. Even a posit (float) fits into that space. So if you enter `10pt` internally it becomes a so called scaled (dimension). The skip registers point to a glue node and the token and box registers to a node list and those pointers are also numbers. So, what the user sees as a data type internally is just a number and its type (the command field in a token) tells what to do with it.

When tracing is turned on there can be mentioning of save stack, input levels, fonts, languages, hyphenation, various character related properties and so on. Here we have specialized data structures that have their own memory layout and management. Where terms like token, node, integer (count), dimension and glue indicate something that the user should grasp, the entries in a save stack are never presented other than in an message.

Manipulating data types is explained in various low level manuals, some relate to programming, and some to typesetting. It makes no sense to repeat that here. Take for instance macros: then come in variants (think of **`\protected`** and/or **`\tolerant`** ones) can take arguments (which effectively are token lists) and the flags in the mentioned table of equivalents control take care of that.

One aspect of token lists is worth mentioning: they start with a so called head token. So a list of length one actually has two tokens. The head keeps track of the fact that a list is a copy. Because a macro is also a token list, in LuaMetaTEX the head also has some information that permits a more efficient code path. Because token lists are used all over the place in the engine, sharing makes sense.

Attributes attached to a node are node lists themselves and these are also shared which not only saves memory but also is more performing. There are many places where LuaMetaTEX differs from its predecessors: there are more primitives, there is more data moved around but it got compensated by optimizing mechanisms. But as much as possible we stayed within the same paradigms.

4.9 Time flies

For those curious about how different the engines are when it comes to memory usage, here is a quote from TEX the program:

> Since we are assuming 32-bit integers, a halfword must contain at least 16 bits, and a quarterword must contain at least 8 bits. But it doesn't hurt to have more bits; for example, with enough 36-bit words you might be able to have `mem_max` as large as 262142, which is eight times as much memory as anybody had during the first four years of TEX's existence.
>
> N.B.: Valuable memory space will be dreadfully wasted unless TEX is compiled by a Pascal that packs all of the `memory_word` variants into the space of a single integer. This means, for example, that `glue_ratio` words should be `short_real` instead of `real` on some computers. Some Pascal compilers will pack an integer whose subrange is `0 .. 255` into an eight-bit field, but others insist on allocating space for an additional sign bit; on such systems you can get 256 values into a quarterword only if the subrange is `128 .. 127`.
>
> The present implementation tries to accommodate as many variations as possible, so it makes few assumptions. If integers having the subrange `min_quarterword .. max_quarterword` can

> be packed into a quarterword, and if integers having the subrange `min_halfword .. max_halfword` can be packed into a halfword, everything should work satisfactorily.
>
> It is usually most efficient to have `min_quarterword = min_halfword = 0`, so one should try to achieve this unless it causes a severe problem. The values defined here are recommended for most 32-bit computers.

This still applies to pdfTEX although there a memory word is two 32 bit integer, so each halfword in there spans 32 bits, and a quarterword 16 bits. So what does that mean for nodes? Here is what the original code says about char nodes.

> A `char_node`, which represents a single character, is the most important kind of node because it accounts for the vast majority of all boxes. Special precautions are therefore taken to ensure that a `char_node` does not take up much memory space. Every such node is one word long, and in fact it is identifiable by this property, since other kinds of nodes have at least two words, and they appear in `mem` locations less than `hi_mem_min`. This makes it possible to omit the `type` field in a `char_node`, leaving us room for two bytes that identify a `font` and a `character` within that font.
>
> Note that the format of a `char_node` allows for up to 256 different fonts and up to 256 characters per font; but most implementations will probably limit the total number of fonts to fewer than 75 per job, and most fonts will stick to characters whose codes are less than 128 (since higher codes are more difficult to access on most keyboards).

So, in order to save space these single size nodes use little memory. Even more interesting is the follow up on that explanation:

> Extensions of TEX intended for oriental languages will need even more than 256×256 possible characters, when we consider different sizes and styles of type. It is suggested that Chinese and Japanese fonts be handled by representing such characters in two consecutive `char_node` entries: The first of these has `font = font_base`, and its `link` points to the second; the second identifies the font and the character dimensions. The saving feature about oriental characters is that most of them have the same box dimensions. The `character` field of the first `char_node` is a `charext` that distinguishes between graphic symbols whose dimensions are identical for typesetting purposes. (See the MetaFont manual.) Such an extension of TEX would not be difficult; further details are left to the reader.
>
> In order to make sure that the `character` code fits in a quarterword, TEX adds the quantity `min_quarterword` to the actual code.

What if that had been implemented right from the start? What if utf8 had been around at that time? Of course when 32 bit integers are used we can use these extra bit for a larger code range anyway.

When we flash forward to LuaTEX we don't see that optimization and there are reasons for it. First of all content related nodes have an attribute list pointer as well as a `prev` field; lists are double linked. That means we don't reuse the `type` and `subtype` fields. The macros that define a glyph are:

```
# define glyph_node_size      7
# define character(a)         vinfo((a)+2)
# define font(a)              vlink((a)+2)
# define lang_data(a)         vinfo((a)+3)
# define lig_ptr(a)           vlink((a)+3)
```

```
# define x_displace(a)        vinfo((a)+4)
# define y_displace(a)        vlink((a)+4)
# define ex_glyph(a)          vinfo((a)+5)  /* expansion factor (hz) */
# define glyph_node_data(a)   vlink((a)+5)
# define synctex_tag_glyph(a) vinfo((a)+6)
# define synctex_line_glyph(a) vlink((a)+6)
```

Instead of one memory word we use seven, and given the amount of characters on a page that adds quite a bit compared to the original. Of course it is irrelevant on todays machines. So how about LuaMetaTEX?

```
# define glyph_node_size      14
# define glyph_character(a)   vinfo(a,2)
# define glyph_font(a)        vlink(a,2)   /*tex can be quarterword */
# define glyph_data(a)        vinfo(a,3)   /*tex handy in context */
# define glyph_state(a)       vlink(a,3)   /*tex handy in context */
# define glyph_language(a)    vinfo0(a,4)
# define glyph_script(a)      vinfo1(a,4)
# define glyph_control(a)     vlink0(a,4)  /*tex we store 0xXXXX in the |\cccode| */
# define glyph_reserved(a)    vlink1(a,4)
# define glyph_options(a)     vinfo(a,5)
# define glyph_hyphenate(a)   vlink(a,5)
# define glyph_protected(a)   vinfo00(a,6)
# define glyph_lhmin(a)       vinfo01(a,6)
# define glyph_rhmin(a)       vinfo02(a,6)
# define glyph_discpart(a)    vinfo03(a,6)
# define glyph_expansion(a)   vlink(a,6)
# define glyph_x_scale(a)     vinfo(a,7)
# define glyph_y_scale(a)     vlink(a,7)
# define glyph_scale(a)       vinfo(a,8)
# define glyph_raise(a)       vlink(a,8)
# define glyph_left(a)        vinfo(a,9)
# define glyph_right(a)       vlink(a,9)
# define glyph_x_offset(a)    vinfo(a,10)
# define glyph_y_offset(a)    vlink(a,10)
# define glyph_weight(a)      vinfo(a,11)
# define glyph_slant(a)       vlink(a,11)
# define glyph_properties(a) vinfo0(a,12)  /*tex for math */
# define glyph_group(a)      vinfo1(a,12)  /*tex for math */
# define glyph_index(a)      vlink(a,12)   /*tex for math */
# define glyph_input_file(a) vinfo(a,13)
# define glyph_input_line(a) vlink(a,13)
```

We carry scaled, offsets, status information and various data around and consume twice what LuaTEX needs. In both cases there are the common fields:

```
# define node_type(a)    vinfo0(a,0)
# define node_subtype(a) vinfo1(a,0)
# define node_next(a)    vlink(a,0)
# define node_prev(a)    vlink(a,1)
```

```
# define node_attr(a)    vinfo(a,1)
```

As you see, we still use the original TEX `vinfo` and `vlink` identifications but in LuaMetaTEX we have node specific verbose accessors because we no longer use the same slots for (for instance) width, height and depth. This of course has impact on the code base because now `width(n)` becomes a different accessor per node it applies to. We get less compact code but gain readability and we often need to distinguish anyway. Where LuaTEX and predecessors we see:

```
w += width(n)
```

that covers boxes, glue and kerns. For glyphs we need to get the width from the font using the `font` and `char` fields. Actually, in TEX82 that can be done directly because we know that these values are okay. In LuaTEX however these values can be set in Lua and therefore we do need to check if they reference a loaded font and valid character slot. So in LuaTEX we do need a dedicated function to get the glyph width.

In LuaMetaTEX we have to be more granular and deal with each node type that has width independently:

```
switch (subtype(n) {
    case glyph_node:
        w += tex_glyph_width(s);
        break;
    case hlist_node:
    case vlist_node:
        w += box_width(n);
        break;
    case rule_node:
        w += rule_width(n);
        break;
    case glue_node:
        w += glue_amount(n);
        break;
    case kern_node:
        w += kern_amount(s);
        break;
    case math_node:
        if (tex_math_glue_is_zero(s)) {
            w += math_surround(s);
        } else {
            w -= math_amount(s);
        }
        break;

}
```

Because a glyph can have scaled set and similar features exist for glue we need to distinguish need to distinguish anyway. Watch the math node: we have to deal with either kern or glue.

primitives

5 Primitives

5.1 Introduction

Here I will discuss some of the new primitives in LuaTeX and LuaMetaTeX, the later being a successor that permits the ConTeXt folks to experiment with new features. The order is arbitrary. When you compare LuaTeX with pdfTeX, there are actually quite some differences. Some primitives that pdfTeX introduced have been dropped in LuaTeX because they can be done better in Lua. Others have been promoted to core primitives that no longer have a `pdf` prefix. Then there are lots of new primitives, some introduce new concepts, some are a side effect of for instance new math font technologies, and then there are those that are handy extensions to the macro language. The LuaMetaTeX engine drops quite some primitives, like those related to pdfTeX specific f(r)ont or backend features. It also adds some new primitives, mostly concerning the macro language.

We also discuss the primitives that fit into the macro programming scope that are present in traditional TeX and ε-TeX but there are for sure better of explanations out there already. Primitives that relate to typesetting, like those controlling math, fonts, boxes, attributes, directions, catcodes, Lua (functions) etc are not discussed or discussed in less detail here.

There are for instance primitives to create aliases to low level registers like counters and dimensions, as well as other (semi-numeric) quantities like characters, but normally these are wrapped into high level macros so that definitions can't clash too much. Numbers, dimensions etc can be advanced, multiplied and divided and there is a simple expression mechanism to deal with them. We don't go into these details here: it's mostly an overview of what the engine provides. If you are new to TeX, you need to play a while with its mixed bag of typesetting and programming features in order to understand the difference between this macro language and other languages you might be familiar with.

In this document the section titles that discuss the original TEX and ε-TEX primitives have a different color those explaining the LuaTEX and LuaMetaTEX primitives.

Primitives that extend typesetting related functionality, provide control over subsystems (like math), allocate additional data types and resources, deal with fonts and languages, manipulate boxes and glyphs, etc. are hardly discussed here, only mentioned. Math for instance is a topic of its own. In this document we concentrate on the programming aspects.

Most of the new primitives are discussed in specific manuals and often also original primitives are covered there but the best explanations of the traditional primitives can be found in The TEXbook by Donald Knuth and TEX by Topic from Victor Eijkhout. I see no need to try to improve on those.

5.2 Rationale

Some words about the why and how it came. One of the early adopters of ConTEXt was Taco Hoekwater and we spent numerous trips to TEX meetings all over the globe. He was also the only one I knew who had read the TEX sources. Because ConTEXt has always been on the edge of what is possible and at that time we both used it for rather advanced rendering, we also ran into the limitations. I'm not talking of TEX features here. Naturally old school TEX is not really geared for dealing with images of all kind, colors in all kind of color spaces, highly interactive documents, input methods like xml, etc. The nice thing is that it offers some escapes, like specials and writes and later execution of programs that opened up lots of possibilities, so in practice there were no real limitations to what one could do. But coming up with a consistent and extensible (multi lingual) user interface was non trivial, because it had an impact in memory usage and performance. A lot could be done given some programming, as ConTEXt MkII proves, but it was not always pretty under the hood. The move to LuaTEX and MkIV transferred some action to Lua, and because LuaTEX effectively was a ConTEXt related project, we could easily keep them in sync.

Our traveling together, meeting several times per year, and eventually email and intense LuaTEX developments (lots of Skype sessions) for a couple of years, gave us enough opportunity to discuss all kind of nice features not present in the engine. The previous century we discussed lots of them, rejected some, stayed with others, and I admit that forgot about most of the arguments already. Some that we did was already explored in `eetex`, some of those ended up in LuaTEX, and eventually what we have in LuaMetaTEX can been seen as the result of years of programming in TEX, improving macros, getting more performance and efficiency out of existing ConTEXt code and inspiration that we got out of the ConTEXt community, a demanding lot, always willing to experiment with us.

Once I decided to work on LuaMetaTEX and bind its source to the ConTEXt distribution so that we can be sure that it won't get messed up and might interfere with the ConTEXt expectations, some more primitives saw their way into it. It is very easy to come up with all kind of bells and whistles but it is equally easy to hurt performance of an engine and what might go unnoticed in simple tests can really affect a macro package that depends on stability. So, what I did was mostly looking at the ConTEXt code and wondering how to make some of the low level macros look more natural, also because I know that there are users who look into these sources. We spend a lot of time making them look consistent and nice and the nicer the better. Getting a better performance was seldom an argument because much is already as fast as can be so there is not that much to gain, but less clutter in tracing was an argument for some new primitives. Also, the fact that we soon might need to fall back on our phones to use TEX a smaller memory footprint and less byte shuffling also was a consideration. The LuaMetaTEX memory footprint is somewhat smaller than the LuaTEX footprint. By binding LuaMetaTEX to ConTEXt we can also guarantee that the combinations works as expected.

I'm aware of the fact that ConTEXt is in a somewhat unique position. First of all it has always been kind of cutting edge so its users are willing to experiment. There are users who immediately update and run tests, so bugs can and will be fixed fast. Already for a long time the community has an convenient infrastructure for updating and the build farm for generating binaries (also for other engines) is running smoothly.

Then there is the ConTEXt user interface that is quite consistent and permits extensions with staying backward compatible. Sometimes users run into old manuals or examples and then complain that ConTEXt is not compatible but that then involves obsolete technology: we no longer need font and input encodings and font definitions are different for OpenType fonts. We always had an abstract backend model, but nowadays pdf is kind of dominant and drives a lot of expectations. So, some of the MkII commands are gone and MkIV has some more. Also, as MetaPost evolved that department in ConTEXt also evolved. Think of it like cars: soon all are electric so one cannot expect a hole to poor in some fluid but gets a (often incompatible) plug instead. And buttons became touch panels. There is no need to use much force to steer or brake. Navigation is different, as are many controls. And do we need to steer ourselves a decade from now?

So, just look at TEX and ConTEXt in the same way. A system from the nineties in the previous century differs from one three decades later. Demands differ, input differs, resources change, editing and processing moves on, and so on. Manuals, although still being written are seldom read from cover to cover because online searching replaced them. And who buys books about programming? So LuaMetaTEX, while still being TEX also moves on, as do the way we do our low level coding. This makes sense because the original TEX ecosystem was not made with a huge and complex macro package in mind, that just happened. An author was supposed to make a style for each document. An often used argument for using another macro package over ConTEXt was that the later evolved and other macro packages would work the same forever and not change from the perspective of the user. In retrospect those arguments were somewhat strange because the world, computers, users etc. do change. Standards come and go, as do software politics and preferences. In many aspects the TEX community is not different from other large software projects, operating system wars, library devotees, programming language addicts, paradigm shifts. But, don't worry, if you don't like LuaMetaTEX and its new primitives, just forget about them. The other engines will be there forever and are a safe bet, although LuaTEX already stirred up the pot I guess. But keep in mind that new features in the latest greatest ConTEXt version will more and more rely on LuaMetaTEX being used; after all that is where it's made for. And this manual might help understand its users why, where and how the low level code differs between MkII, MkIV and LMTX.

Can we expect more new primitives than the ones introduced here? Given the amount of time I spent on experimenting and considering what made sense and what not, the answer probably is “no”, or at least “not that much”. As in the past no user ever requested the kind of primitives that were added, I don't expect users to come up with requests in the future either. Of course, those more closely related to ConTEXt development look at it from the other end. Because it's there where the low level action really is, demands might still evolve.

Basically there are wo areas where the engine can evolve: the programming part and the rendering. In this manual we focus on the programming and writing the manual sort of influences how details get filled in. Rendering in more complex because there heuristics and usage plays a more dominant role. Good examples are the math, par and page builder. They were extended and features were added over time but improved rendering came later. Not all extensions are critical, some are there (and got added) in order to write more readable code but there is only so much one can do in that area. Occasionally a feature pops up that is a side effect of a challenge. No matter what gets added it might not affect complexity too much and definitely not impact performance significantly!

5.3 Primitives

1 \<space>

This original TEX primitive is equivalent to the more verbose `\explicitspace`.

2 \-

This original TEX primitive is equivalent to the more verbose `\explicitdiscretionary`.

3 \/

This original TEX primitive is equivalent to the more verbose `\explicititaliccorrection`.

4 \Umathxscale

The `\Umathxscale` and `\Umathyscale` factors are applied to the horizontal and vertical parameters. They are set by style. There is no combined scaling primitive.

```
$\Umathxscale\textstyle  800 a + b + x + d + e = f $\par
$\Umathxscale\textstyle 1000 a + b + x + d + e = f $\par
```

```
$\Umathxscale\textstyle 1200 a + b + x + d + e = f $\blank

$\Umathyscale\textstyle  800 \sqrt[2]{x+1}$\quad
$\Umathyscale\textstyle 1000 \sqrt[2]{x+1}$\quad
$\Umathyscale\textstyle 1200 \sqrt[2]{x+1}$\blank
```

Normally only small deviations from 1000 make sense but here we want to show the effect and use a 20% scaling:

$a + b + x + d + e = f$

$a + b + x + d + e = f$

$a + b + x + d + e = f$

$\sqrt[2]{x+1}$ $\sqrt[2]{x+1}$ $\sqrt[2]{x+1}$

5 \Umathyscale

See `\Umathxscale]`

6 \above

This is a variant of `\over` that doesn't put a rule in between.

7 \abovedisplayshortskip

The glue injected before a display formula when the line above it is not overlapping with the formula. Watch out for interference with `\baselineskip`. It can be controlled by `\displayskipmode`.

8 \abovedisplayskip

The glue injected before a display formula. Watch out for interference with `\baselineskip`. It can be controlled by `\displayskipmode`.

9 \abovewithdelims

This is a variant of `\atop` but with delimiters. It has a more advanced upgrade in `\Uabovewithdelims`.

10 \accent

This primitive is kind of obsolete in wide engines and takes two arguments: the indexes of an accent and a base character.

11 \additionalpageskip

This quantity will be added to the current page goal, stretch and shrink after which it will be set to zero.

12 \adjdemerits

When TEX considers to lines to be incompatible it will add this penalty to its verdict when considering this breakpoint.

13 \adjustspacing

This parameter controls expansion (hz). A value 2 expands glyphs and font kerns and a value of 3 only glyphs. Expansion of kerns can have side effects when they are used for positioning by OpenType features.

14 \adjustspacingshrink

When set to a non zero value this overloads the shrink maximum in a font when expansion is applied. This is then the case for all fonts.

15 \adjustspacingstep

When set to a non zero value this overloads the expansion step in a font when expansion is applied. This is then the case for all fonts.

16 \adjustspacingstretch

When set to a non zero value this overloads the stretch maximum in a font when expansion is applied. This is then the case for all fonts.

17 \advance

Advances the given register by an also given value:

```
\advance\scratchdimen      10pt
\advance\scratchdimen      by 3pt
\advance\scratchcounterone \zerocount
\advance\scratchcounterone \scratchcountertwo
```

The `by` keyword is optional.

18 \advanceby

This is slightly more efficient variant of `\advance` that doesn't look for `by` and therefore, if one is missing, doesn't need to push back the last seen token. Using `\advance` with `by` is nearly as efficient but takes more tokens.

19 \afterassigned

The `\afterassignment` primitive stores a token to be injected (and thereby expanded) after an assignment has happened. Unlike `\aftergroup`, multiple calls are not accumulated, and changing that would be too incompatible. This is why we have `\afterassigned`, which can be used to inject a bunch of tokens. But in order to be consistent this one is also not accumulative.

```
\afterassigned{done}%
\afterassigned{{\bf done}}%
\scratchcounter=123
```

results in: **done** being typeset.

20 \afterassignment

The token following `\afterassignment`, a traditional TEX primitive, is saved and gets injected (and then expanded) after a following assignment took place.

```
\afterassignment !\def\MyMacro {}\quad
\afterassignment !\let\MyMacro ?\quad
\afterassignment !\scratchcounter 123\quad
\afterassignment !%
\afterassignment ?\advance\scratchcounter by 1
```

The `\afterassignments` are not accumulated, the last one wins:

! ! ! ?

21 \aftergroup

The traditional TEX `\aftergroup` primitive stores the next token and expands that after the group has been closed.

Multiple `\aftergroups` are combined:

```
before{ ! \aftergroup a\aftergroup f\aftergroup t\aftergroup e\aftergroup r}
```

before ! after

22 \aftergrouped

The in itself powerful `\aftergroup` primitives works quite well, even if you need to do more than one thing: you can either use it multiple times, or you can define a macro that does multiple things and apply that after the group. However, you can avoid that by using this primitive which takes a list of tokens.

```
regular
\bgroup
\aftergrouped{regular}%
\bf bold
\egroup
```

Because it happens after the group, we're no longer typesetting in bold.

regular **bold** regular

You can mix `\aftergroup` and `\aftergrouped`. Which one is more efficient depends on how many tokens are delayed. Picking up one token is faster than scanning a list.

```
{
    \aftergroup A \aftergroup B \aftergroup C
test 1 : }

{
    \aftergrouped{What comes next 1}
    \aftergrouped{What comes next 2}
    \aftergrouped{What comes next 3}
test 2 : }

{
    \aftergroup A \aftergrouped{What comes next 1}
    \aftergroup B \aftergrouped{What comes next 2}
    \aftergroup C \aftergrouped{What comes next 3}
test 3 : }

{
    \aftergrouped{What comes next 1} \aftergroup A
    \aftergrouped{What comes next 2} \aftergroup B
    \aftergrouped{What comes next 3} \aftergroup C
test 4 : }
```

This gives:

test 1 : ABC
test 2 : What comes next 1What comes next 2What comes next 3
test 3 : AWhat comes next 1BWhat comes next 2CWhat comes next 3
test 4 : What comes next 1AWhat comes next 2BWhat comes next 3C

23 \aliased

This primitive is part of the overload protection subsystem where control sequences can be tagged.

```
\permanent\def\foo{FOO}
          \let\ofo\foo
\aliased  \let\oof\foo

\meaningasis\foo
\meaningasis\ofo
\meaningasis\oof
```

gives:

```
\permanent \def \foo {FOO}
\def \ofo {FOO}
\permanent \def \oof {FOO}
```

When a something is `\let` the 'permanent', 'primitive' and 'immutable' flags are removed but the `\aliased` prefix retains them.

```
\let\relaxed\relax

\meaningasis\relax
```

```
\meaningasis\relaxed
```

So in this example the `\relaxed` alias is not flagged as primitive:

```
\primitive \relax
\relax
```

24 \aligncontent

This is equivalent to a hash in an alignment preamble. Contrary to `\alignmark` there is no need to duplicate inside a macro definition.

25 \alignmark

When you have the # not set up as macro parameter character cq. align mark, you can use this primitive instead. The same rules apply with respect to multiple such tokens in (nested) macros and alignments.

26 \alignmentcellsource

This sets the source id (a box property) of the current alignment cell.

27 \alignmentwrapsource

This sets the source id (a box property) of the current alignment row (in a `\halign`) or column (in a `\valign`).

28 \aligntab

When you have the & not set up as align tab, you can use this primitive instead. The same rules apply with respect to multiple such tokens in (nested) macros and alignments.

29 \allcrampedstyles

A symbolic representation of `\crampeddisplaystyle`, `\crampedtextstyle`, `\crampedscriptstyle` and `\crampedscriptscriptstyle`; integer representation: 17.

30 \alldisplaystyles

A symbolic representation of `\displaystyle` and `\crampeddisplaystyle`; integer representation: 8.

31 \allmainstyles

A symbolic representation of `\displaystyle`, `\crampeddisplaystyle`, `\textstyle` and `\cramped-textstyle`; integer representation: 13.

32 \allmathstyles

A symbolic representation of `\displaystyle`, `\crampeddisplaystyle`, `\textstyle`, `\cramped-textstyle`, `\scriptstyle`, `\crampedscriptstyle`, `\scriptscriptstyle` and `\crampedscriptscript-style`; integer representation: 12.

33 \allscriptscriptstyles

A symbolic representation of `\scriptscriptstyle` and `\crampedscriptscriptstyle`; integer representation: 11.

34 \allscriptstyles

A symbolic representation of `\scriptstyle` and `\crampedscriptstyle`; integer representation: 10.

35 \allsplitstyles

A symbolic representation of `\displaystyle` and `\textstyle` but not `\scriptstyle` and `\scriptscriptstyle`: set versus reset; integer representation: 14.

36 \alltextstyles

A symbolic representation of `\textstyle` and `\crampedtextstyle`; integer representation: 9.

37 \alluncrampedstyles

A symbolic representation of `\displaystyle`, `\textstyle`, `\scriptstyle` and `\scriptscriptstyle`; integer representation: 16.

38 \allunsplitstyles

A symbolic representation of `\scriptstyle` and `\scriptscriptstyle`; integer representation: 15.

39 \amcode

40 \associateunit

The TEX engine comes with some build in units, like `pt` (fixed) and `em` (adaptive). On top of that a macro package can add additional units, which is what we do in ConTEXt. In figure 5.1 we show the current repertoire.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
b																bp										
c			cc	cd				ch					cm										cw	cx		
d				dd							dk															
e													em						es		eu			ex		
f	fa		fc	fd				fh	fi						fo				fs	ft			fw			
h																			hs							
i														in												
l																							lw			
m	ma												mm				mq				mu			mx		
p			pc					ph	pi											pt			pw	px		
s				sd				sh								sp				st						
t								th											ts				tw			
u																					uu					
v																			vs							

tex pdftex luametatex context

Figure 5.1 Available units

When this primitive is used in a context where a number is expected it returns the origin of the unit (in the color legend running from 1 upto 4). A new unit is defined as:

```
\newdimen\MyDimenZA  \MyDimenZA=10pt

\protected\def\MyDimenAB{\dimexpr\hsize/2\relax}

\associateunit za \MyDimenZA
\associateunit zb \MyMacroZB
```

Possible associations are: macros that expand to a dimension, internal dimension registers, register dimensions (`\dimendef`, direct dimensions (`\dimensiondef`) and Lua functions that return a dimension.

One can run into scanning ahead issues where TEX expects a unit and a user unit gets expanded. This is why for instance in ConTEXt we define the `ma` unit as:

```
\protected\def\mathaxisunit{\scaledmathaxis\mathstyle\norelax}

\associateunit ma \mathaxisunit % or \newuserunit \mathaxisunit ma
```

So that it can be used in rule specifications that themselves look ahead for keywords and therefore are normally terminated by a `\relax`. Adding the extra `\norelax` will make the scanner see one that doesn't get fed back into the input. Of course a macro package has to manage extra units in order to avoid conflicts.

41 \atendoffile

The `\everyeof` primitive is kind of useless because you don't know if a file (which can be a tokenlist processed as pseudo file) itself includes a file, which then results in nested application of this token register. One way around this is:

```
\atendoffile\SomeCommand
```

This acts on files the same way as `\atendofgroup` does. Multiple calls will be accumulated and are bound to the current file.

42 \atendoffiled

This is the multi token variant of `\atendoffile`. Multiple invocations are accumulated and by default prepended to the existing list. As with grouping this permits proper nesting. You can force an append by the optional keyword reverse.

43 \atendofgroup

The token provided will be injected just before the group ends. Because these tokens are collected, you need to be aware of possible interference between them. However, normally this is managed by the macro package.

```
\bgroup
\atendofgroup\unskip
\atendofgroup )%
```

```
(but it works okay
\egroup
```

Of course these effects can also be achieved by combining (extra) grouping with `\aftergroup` calls, so this is more a convenience primitives than a real necessity: (but it works okay), as proven here.

44 \atendofgrouped

This is the multi token variant of `\atendofgroup`. Of course the next example is somewhat naive when it comes to spacing and so, but it shows the purpose.

```
\bgroup
\atendofgrouped{\bf QED}%
\atendofgrouped{ (indeed)}%
This sometimes looks nicer.
\egroup
```

Multiple invocations are accumulated: This sometimes looks nicer. **QED (indeed)**.

45 \atop

This one stack two math elements on top of each other, like a fraction but with no rule. It has a more advanced upgrade in `\Uatop`.

46 \atopwithdelims

This is a variant of `\atop` but with delimiters. It has a more advanced upgrade in `\Uatopwithdelims`.

47 \attribute

The following sets an attribute(register) value:

```
\attribute 999 = 123
```

An attribute is unset by assigning -2147483647 to it. A user needs to be aware of attributes being used now and in the future of a macro package and setting them this way is very likely going to interfere.

48 \attributedef

This primitive can be used to relate a control sequence to an attribute register and can be used to implement a mechanism for defining unique ones that won't interfere. As with other registers: leave management to the macro package in order to avoid unwanted side effects!

49 \automaticdiscretionary

This is an alias for the automatic hyphen trigger `-`.

50 \automatichyphenpenalty

The penalty injected after an automatic discretionary `-`, when `\hyphenationmode` enables this.

51 \automigrationmode

This bitset determines what will bubble up to an outer level:

0x01	mark
0x02	insert
0x04	adjust
0x08	pre
0x10	post

The current value is 0xFFFF.

52 \autoparagraphmode

A paragraph can be triggered by an empty line, a \par token or an equivalent of it. This parameter controls how **\par** is interpreted in different scenarios:

0x01	text
0x02	macro
0x04	continue

The current value is 0x1 and setting it to a non-zero value can have consequences for mechanisms that expect otherwise. The text option uses the same code as an empty line. The macro option checks a token in a macro preamble against the frozen \

token. The last option ignores the par token.

53 \badness

This one returns the last encountered badness value.

54 \baselineskip

This is the maximum glue put between lines. The depth of the previous and height of the next line are substracted.

55 \batchmode

This command disables (error) messages which can safe some runtime in situations where TEX's character-by-character log output impacts runtime. It only makes sense in automated workflows where one doesn't look at the log anyway.

56 \begincsname

The next code creates a control sequence token from the given serialized tokens:

```
\csname mymacro\endcsname
```

When \mymacro is not defined a control sequence will be created with the meaning \relax. A side effect is that a test for its existence might fail because it now exists. The next sequence will *not* create an controil sequence:

```
\begincsname mymacro\endcsname
```

This actually is kind of equivalent to:

```
\ifcsname mymacro\endcsname
    \csname mymacro\endcsname
\fi
```

57 \begingroup

This primitive starts a group and has to be ended with `\endgroup`. See `\beginsimplegroup` for more info.

58 \beginlocalcontrol

Once TEX is initialized it will enter the main loop. In there certain commands trigger a function that itself can trigger further scanning and functions. In LuaMetaTEX we can have local main loops and we can either enter it from the Lua end (which we don't discuss here) or at the TEX end using this primitive.

```
\scratchcounter100

\edef\whatever{
    a
    \beginlocalcontrol
        \advance\scratchcounter 10
        b
    \endlocalcontrol
    \beginlocalcontrol
        c
    \endlocalcontrol
    d
    \advance\scratchcounter 10
}

\the\scratchcounter
\whatever
\the\scratchcounter
```

A bit of close reading probably gives an impression of what happens here:

b c

110 a d 120

The local loop can actually result in material being injected in the current node list. However, where normally assignments are not taking place in an `\edef`, here they are applied just fine. Basically we have a local TEX job, be it that it shares all variables with the parent loop.

59 \beginmathgroup

In math mode grouping with `\begingroup` and `\endgroup` in some cases works as expected, but because the math input is converted in a list that gets processed later some settings can become persistent, like changes in style or family. The engine therefore provides the alternatives `\beginmathgroup` and `\endmathgroup` that restore some properties.

60 \beginsimplegroup

The original TEX engine distinguishes two kind of grouping that at the user end show up as:

```
\begingroup \endgroup
\bgroup \egroup { }
```

where the last two pairs are equivalent unless the scanner explicitly wants to see a left and/or right brace and not an equivalent. For the sake of simplify we use the aliases here. It is not possible to mix these pairs, so:

```
\bgroup xxx\endgroup
\begingroup xxx\egroup
```

will in both cases issue an error. This can make it somewhat hard to write generic grouping macros without somewhat dirty trickery. The way out is to use the generic group opener `\beginsimplegroup`.

Internally LuaMetaTEX is aware of what group it currently is dealing with and there we distinguish:

simple group	`\bgroup`	`\egroup`
semi simple group	`\begingroup`	`\endgroup \endsimplegroup`
also simple group	`\beginsimplegroup`	`\egroup \endgroup \endsimplegroup`
math simple group	`\beginmathgroup`	`\endmathgroup`

This means that you can say:

```
\beginsimplegroup xxx\endsimplegroup
\beginsimplegroup xxx\endgroup
\beginsimplegroup xxx\egroup
```

So a group started with `\beginsimplegroup` can be finished in three ways which means that the user (or calling macro) doesn't have take into account what kind of grouping was used to start with. Normally usage of this primitive is hidden in macros and not something the user has to be aware of.

61 \belowdisplayshortskip

The glue injected aftter a display formula when the line above it is not overlapping with the formula (TEX can't look ahead). Watch out for interference with `\baselineskip`. It can be controlled by `\displayskipmode`.

62 \belowdisplayskip

The glue injected after a display formula. Watch out for interference with `\baselineskip`. It can be controlled by `\displayskipmode`.

63 \binoppenalty

This internal quantity is a compatibility feature because normally we will use the inter atom spacing variables.

64 \botmark

This is a reference to the last mark on the current page, it gives back tokens.

65 \botmarks

This is a reference to the last mark with the given id (a number) on the current page, it gives back tokens.

66 \boundary

Boundaries are signals added to he current list. This primitive injects a user boundary with the given (integer) value. Such a boundary can be consulted at the Lua end or with `\lastboundary`.

67 \box

This is the box register accessor. While other registers have one property a box has many, like `\wd`, `\ht` and `\dp`. This primitive returns the box and resets the register.

68 \boxadapt

Adapting will recalculate the dimensions with a scale factor for the glue:

```
\setbox 0 \hbox       {test test test}
\setbox 2 \hbox {\red  test test test} \boxadapt 0  200
\setbox 4 \hbox {\blue test test test} \boxadapt 0 -200
\ruledhbox{\box0} \vskip-\lineheight
\ruledhbox{\box0} \vskip-\lineheight
\ruledhbox{\box0}
```

Like `\boxfreeze` and `\boxrepack` this primitive has been introduced for experimental usage, although we do use some in production code.

test test test

69 \boxanchor

This feature is part of an (experimental) mechanism that relates boxes. The engine just tags a box and it is up to the macro package to deal with it.

```
\setbox0\hbox anchor "01010202 {test}\tohexadecimal\boxanchor0
```

This gives: 1010202. Of course this feature is very macro specific and should not be used across macro packages without coordination. An anchor has two parts each not exceeding `0x0FFF`.

70 \boxanchors

This feature is part of an (experimental) mechanism that relates boxes. The engine just tags a box and it is up to the macro package to deal with it.

```
\setbox0\hbox anchors "0101 "0202 {test}\tohexadecimal\boxanchors0
```

This gives: 1010202. Of course this feature is very macro specific and should not be used across macro packages without coordination. An anchor has two parts each not exceeding `0x0FFF`.

71 \boxattribute

Every node, and therefore also every box gets the attributes set that are active at the moment of creation. Additional attributes can be set too:

```
\darkred
\setbox0\hbox attr 9999 1 {whatever}
\the\boxattribute 0 \colorattribute
\the\boxattribute 0 9998
\the\boxattribute 0 9999
```

A macro package should make provide a way define attributes that don't clash the ones it needs itself, like, in ConTEXt, the ones that can set a color

4
-2147483647
1

The number -2147483647 (-7FFFFFFF) indicates an unset attribute.

72 \boxdirection

The direction of a box defaults to `l2r` but can be explicitly set:

```
\setbox0\hbox direction 1 {this is a test}\textdirection1
\setbox2\hbox direction 0 {this is a test}\textdirection0
\the\boxdirection0: \box0
\the\boxdirection2: \box2
```

The `\textdirection` does not influence the box direction:

1: tset a si siht
0: this is a test

73 \boxfinalize

This is special version of `\boxfreeze` which we demonstrate with an example:

```
\boxlimitate 0 0   % don't recurse
\boxfreeze   2 0   % don't recurse
\boxfinalize 4 500 % scale glue multiplier by .50
\boxfinalize 6 250 % scale glue multiplier by .25
```

```
\boxfinalize 8 100 % scale glue multiplier by .10

\hpack\bgroup
    \copy0\quad\copy2\quad\copy4\quad\copy6\quad\copy8
\egroup
```

where the boxes are populated with:

```
\setbox0\ruledvbox to 3cm{\hsize 2cm test\vskip10pt plus 10pt test}
\setbox2\copy0\setbox4\copy0\setbox6\copy0\setbox8\copy0
```

test test test test test

test test test test test

74 \boxfreeze

Glue in a box has a fixed component that will always be used and stretch and shrink that kicks in when needed. The effective value (width) of the glue is driven by some box parameters that are set by the packaging routine. This is why we can unbox: the original value is kept. It is the backend that calculates the effective value. Te `\boxfreeze` primitive can do the same: turn the flexible glue into a fixed one.

```
\setbox    0 \hbox to 6cm {\hss frost}
\setbox    2 \hbox to 6cm {\hss frost}
\boxfreeze 2 0
\ruledhbox{\unhbox 0}
\ruledhbox{\unhbox 2}
```

The second parameter to `\boxfreeze` determines recursion. We don't recurse here so just freeze the outer level:

frost

frost

75 \boxgeometry

A box can have an orientation, offsets and/or anchors. These are stored independently but for efficiency reasons we register if one or more of these properties is set. This primitive accesses this state; it is a bitset:

`0x01`	offset
`0x02`	orientation
`0x04`	anchor

76 \boxlimit

This primitive will freeze the glue in a box but only when there is glue marked with the limit option.

77 \boxlimitate

This primitive will freeze the glue in a box. It takes two arguments, a box number and an number that when set to non-zero will recurse into nested lists.

78 \boxlimitmode

This variable controls if boxes with glue marked 'limit' will be checked and frozen.

79 \boxmaxdepth

You can limit the depth of boxes being constructed. It's one of these parameters that should be used with care because when that box is filled nested boxes can be influenced.

80 \boxorientation

The orientation field can take quite some values and is discussed in one of the low level ConTEXt manuals. Some properties are dealt with in the TEX engine because they influence dimensions but in the end it is the backend that does the work.

81 \boxrepack

When a box of to wide or tight we can tweak it a bit with this primitive. The primitive expects a box register and a dimension, where a positive number adds and a negatie subtracts from the current box with.

```
\setbox 0 \hbox        {test test test}
\setbox 2 \hbox {\red   test test test} \boxrepack0 +.2em
\setbox 4 \hbox {\green test test test} \boxrepack0 -.2em
\ruledhbox{\box0} \vskip-\lineheight
\ruledhbox{\box0} \vskip-\lineheight
\ruledhbox{\box0}
```

test test test

We can also use this primitive to check the natural dimensions of a box:

```
\setbox 0 \hbox spread 10pt {test test test}
\ruledhbox{\box0} (\the\boxrepack0,\the\wd0)
```

In this context only one argument is expected.
test test test

(0.0pt,0.0pt)

82 \boxshift

Returns or sets how much the box is shifted: up or down in horizontally mode, left or right in vertical mode.

83 \boxshrink

Returns the amount of shrink found (applied) in a box:

```
\setbox0\hbox to 4em {m m m m}
\the\boxshrink0
```

gives: 3.17871pt

84 \boxsource

This feature is part of an (experimental) mechanism that relates boxes. The engine just tags a box and it is up to the macro package to deal with it.

```
\setbox0\hbox source 123 {m m m m}
\the\boxsource0
```

This gives: 123. Of course this feature is very macro specific and should not be used across macro packages without coordination.

85 \boxstretch

Returns the amount of stretch found (applied) in a box:

```
\setbox0\hbox to 6em {m m m m}
\the\boxstretch0
```

gives: 4.76807pt

86 \boxtarget

This feature is part of an (experimental) mechanism that relates boxes. The engine just tags a box and it is up to the macro package to deal with it.

```
\setbox0\hbox source 123 {m m m m}
\the\boxsource0
```

This gives: 123. Of course this feature is very macro specific and should not be used across macro packages without coordination.

87 \boxtotal

Returns the total of height and depth of the given box.

88 \boxvadjust

When used as query this returns a bitset indicating the associated adjust and migration (marks and inserts) data:

`0x1` pre adjusted
`0x2` post adjusted

`0x4`	pre migrated
`0x8`	post migrated

When used as a setter it directly adds adjust data to the box and it accepts the same keywords as `\vadjust`.

89 `\boxxmove`

This will set the vertical offset and adapt the dimensions accordingly.

90 `\boxxoffset`

Returns or sets the horizontal offset of the given box.

91 `\boxymove`

This will set the vertical offset and adapt the dimensions accordingly.

92 `\boxyoffset`

Returns or sets the vertical offset of the given box.

93 `\brokenpenalties`

Together with `\widowpenalties` and `\clubpenalties` this one permits discriminating left- and right page (doublesided) penalties. For this one needs to also specify `\options 4` and provide penalty pairs. Where the others accept multiple pairs, this primitives expects a count value one.

94 `\brokenpenalty`

This penalty is added after a line that ends with a hyphen; it can help to discourage a page break (or split in a box).

95 `\catcode`

Every character can be put in a category, but this is typically something that the macro package manages because changes can affect behavior. Also, once passed as an argument, the catcode of a character is frozen. There are 16 different values:

`\escapecatcode`	0	`\begingroupcatcode`	1
`\endgroupcatcode`	2	`\mathshiftcatcode`	3
`\alignmentcatcode`	4	`\endoflinecatcode`	5
`\parametercatcode`	6	`\superscriptcatcode`	7
`\subscriptcatcode`	8	`\ignorecatcode`	9
`\spacecatcode`	10	`\lettercatcode`	11
`\othercatcode`	12	`\activecatcode`	13
`\commentcatcode`	14	`\invalidcatcode`	15

The first column shows the constant that ConTEXt provides and the name indicates the purpose. Here are two examples:

```
\catcode123=\begingroupcatcode
\catcode125=\endgroupcatcode
```

96 \catcodetable

The catcode table with the given index will become active.

97 \cccode

This is an experimental feature that can set some processing options. The character specific code is stored in the glyph node and consulted later. An example of such option is 'ignore twin', bit one, which we set for a few punctuation characters.

98 \cdef

This primitive is like `\edef` but in some usage scenarios is slightly more efficient because (delayed) expansion is ignored which in turn saves building a temporary token list.

```
\edef\FooA{this is foo} \meaningfull\FooA\crlf
\cdef\FooB{this is foo} \meaningfull\FooB\par
```

```
macro:this is foo
constant macro:this is foo
```

99 \cdefcsname

This primitive is like `\edefcsame` but in some usage scenarios is slightly more efficient because (delayed) expansion is ignored which in turn saves building a temporary token list.

```
\edefcsname FooA\endcsname{this is foo} \meaningasis\FooA\crlf
\cdefcsname FooB\endcsname{this is foo} \meaningasis\FooB\par
```

```
\def \FooA {this is foo}
\constant \def \FooB {this is foo}
```

100 \cfcode

This primitive is a companion to `\efcode` and sets the compression factor. It takes three values: font, character code, and factor.

101 \char

This appends a character with the given index in the current font.

102 \chardef

The following definition relates a control sequence to a specific character:

```
\chardef\copyrightsign"A9
```

However, because in a context where a number is expected, such a `\chardef` is seen as valid number, there was a time when this primitive was used to define constants without overflowing the by then limited pool of count registers. In ε-TEX aware engines this was less needed, and in LuaMetaTEX we have `\integerdef` as a more natural candidate.

103 \cleaders

See `\gleaders` for an explanation.

104 \clearmarks

This primitive is an addition to the multiple marks mechanism that originates in ε-TEX and reset the mark registers of the given category (a number).

105 \clubpenalties

This is an array of penalty put before the first lines in a paragraph. High values discourage (or even prevent) a lone line at the end of a page. This command expects a count value indicating the number of entries that will follow. The first entry is ends up after the first line.

106 \clubpenalty

This is the penalty put before a club line in a paragraph. High values discourage (or even prevent) a lone line at the end of a next page.

107 \constant

This prefix tags a macro (without arguments) as being constant. The main consequence is that in some cases expansion gets delayed which gives a little performance boost and less (temporary) memory usage, for instance in **`\csname`** like scenarios.

108 \constrained

See previous section about `\retained`.

109 \copy

This is the box register accessor that returns a copy of the box.

110 \copymathatomrule

This copies the rule bitset from the parent class (second argument) to the target class (first argument). The bitset controls the features that apply to atoms.

111 \copymathparent

This binds the given class (first argument) to another class (second argument) so that one doesn't need to define all properties.

112 \copymathspacing

This copies an class spacing specification to another one, so in

```
\copymathspacing 34 2
```

class 34 (a user one) get the spacing from class 2 (binary).

113 \count

This accesses a count register by index. This is kind of 'not done' unless you do it local and make sure that it doesn't influence macros that you call.

```
\count4023=10
```

In standard TEX the first 10 counters are special because they get reported to the console, and `\count0` is then assumed to be the page counter.

114 \countdef

This primitive relates a control sequence to a count register. Compare this to the example in the previous section.

```
\countdef\MyCounter4023
\MyCounter=10
```

However, this is also 'not done'. Instead one should use the allocator that the macro package provides.

```
\newcount\MyCounter
\MyCounter=10
```

In LuaMetaTEX we also have integers that don't rely on registers. These are assigned by the primitive `\integerdef`:

```
\integerdef\MyCounterA 10
```

Or better `\newinteger`.

```
\newinteger\MyCounterB
\MyCounterN10
```

There is a lowlevel manual on registers.

115 \cr

This ends a row in an alignment. It also ends an alignment preamble.

116 \crampeddisplaystyle

A less spacy alternative of `\displaystyle`; integer representation: 4.

117 \crampedscriptscriptstyle

A less spacy alternative of \scriptscriptstyle; integer representation: 6.

118 \crampedscriptstyle

A less spacy alternative of \scriptstyle; integer representation: 4.

119 \crampedtextstyle

A less spacy alternative of \textstyle; integer representation: 2.

120 \crcr

This ends a row in an alignment when it hasn't ended yet.

121 \csactive

Because LuaTEX (and LuaMetaTEX) are Unicode engines active characters are implemented a bit differently. They don't occupy a eight bit range of characters but are stored as control sequence with a special prefix U+FFFF which never shows up in documents. The \csstring primitive injects the name of a control sequence without leading escape character, the \csactive injects the internal name of the following (either of not active) character. As we cannot display the prefix: **\csactive**~ will inject the utf sequences for U+FFFF and U+007E, so here we get the bytes EFBFBF7E. Basically the next token is preceded by \string, so when you don't provide a character you are in for a surprise.

122 \csname

This original TEX primitive starts the construction of a control sequence reference. It does a lookup and when no sequence with than name is found, it will create a hash entry and defaults its meaning to \relax.

```
\csname letters and other characters\endcsname
```

123 \csnamestring

This is a companion of \lastnamedcs that injects the name of the found control sequence. When used inside a csname constructor it is more efficient than repeating a token list, compare:

```
\csname\ifcsname whatever\endcsname\csnamestring\endcsname % referenced
\csname\ifcsname whatever\endcsname     whatever\endcsname % scanned
```

124 \csstring

This primitive returns the name of the control sequence given without the leading escape character (normally a backslash). Of course you could strip that character with a simple helper but this is more natural.

```
\csstring\mymacro
```

We get the name, not the meaning: mymacro.

125 \currentgrouplevel

The next example gives: [1] [2] [3] [2] [1].

```
[\the\currentgrouplevel] \bgroup
    [\the\currentgrouplevel] \bgroup
        [\the\currentgrouplevel]
    \egroup [\the\currentgrouplevel]
\egroup [\the\currentgrouplevel]
```

126 \currentgrouptype

The next example gives: [22] [1] [22] [1] [1] [23] [1] [1].

```
[\the\currentgrouptype] \bgroup
    [\the\currentgrouptype] \begingroup
        [\the\currentgrouptype]
    \endgroup [\the\currentgrouptype]
    [\the\currentgrouptype] \beginmathgroup
        [\the\currentgrouptype]
    \endmathgroup [\the\currentgrouptype]
[\the\currentgrouptype] \egroup
```

The possible values depend in the engine and for LuaMetaTEX they are:

0	bottomlevel	9	output	18	mathoperator	27	mathnumber
1	simple	10	mathsubformula	19	mathradical	28	localbox
2	hbox	11	mathstack	20	mathchoice	29	splitoff
3	adjustedhbox	12	mathcomponent	21	alsosimple	30	splitkeep
4	vbox	13	discretionary	22	semisimple	31	preamble
5	vtop	14	insert	23	mathsimple	32	alignset
6	dbox	15	vadjust	24	mathfence	33	finishrow
7	align	16	vcenter	25	mathinline	34	lua
8	noalign	17	mathfraction	26	mathdisplay		

127 \currentifbranch

The next example gives: [0] [1] [-1] [1] [0].

```
[\the\currentifbranch] \iftrue
    [\the\currentifbranch] \iffalse
        [\the\currentifbranch]
    \else
        [\the\currentifbranch]
    \fi [\the\currentifbranch]
\fi [\the\currentifbranch]
```

So when in the 'then' branch we get plus one and when in the 'else' branch we end up with a minus one.

128 \currentiflevel

The next example gives: [0] [1][2] [3] [2] [1] [0].

```
[\the\currentiflevel] \iftrue
    [\the\currentiflevel]\iftrue
        [\the\currentiflevel] \iftrue
            [\the\currentiflevel]
        \fi [\the\currentiflevel]
    \fi [\the\currentiflevel]
\fi [\the\currentiflevel]
```

129 \currentiftype

The next example gives: [-1] [25][25] [25] [25] [25] [-1].

```
[\the\currentiftype] \iftrue
    [\the\currentiftype]\iftrue
        [\the\currentiftype] \iftrue
            [\the\currentiftype]
        \fi [\the\currentiftype]
    \fi [\the\currentiftype]
\fi [\the\currentiftype]
```

The values are engine dependent:

0	char	7	absfloat	14	odd	21	vbox	28	chknunber
1	cat	8	zerofloat	15	vmode	22	tok	29	numval
2	num	9	intervalfloat	16	hmode	23	cstoken	30	cmpnum
3	absnum	10	dim	17	mmode	24	x	31	chkdim
4	zeronum	11	absdim	18	inner	25	true	32	chkdimension
5	intervalnum	12	zerodim	19	void	26	false	33	dimval
6	float	13	intervaldim	20	hbox	27	chknum	34	cmpdim

130 \currentloopiterator

Here we show the different expanded loop variants:

```
\edef\testA{\expandedloop  1 10 1{!}}
\edef\testB{\expandedrepeat  10  {!}}
\edef\testC{\expandedendless     {\ifnum\currentloopiterator>10 \quitloop\else !\fi}}
\edef\testD{\expandedendless     {\ifnum#I>10 \quitloop\else !\fi}}
```

All these give the same result:

```
\def \testA {!!!!!!!!!!}
```

```
\def \testB {!!!!!!!!!!}
\def \testC {!!!!!!!!!!}
\def \testD {!!!!!!!!!!}
```

The #I is a shortcut to the current loop iterator; other shortcuts are #P for the parent iterator value and #G for the grand parent.

131 \currentloopnesting

This integer reports how many nested loops are currently active. Of course in practice the value only has meaning when you know at what outer level your nested loop started.

```
\expandedloop 1 10 1 {%
    \ifodd\currentloopiterator\else
      [\expandedloop 1 \currentloopiterator 1 {%
        \the\currentloopnesting
      }]
    \fi
}
```

Here we use the two numeric state primitives \currentloopiterator and \currentloopnesting. This results in:

[22] [2222] [222222] [22222222] [2222222222]

132 \currentlysetmathstyle

TODO

133 \currentmarks

Marks only get updated when a page is split off or part of a box using \vsplit gets wrapped up. This primitive gives access to the current value of a mark and takes the number of a mark class.

134 \currentstacksize

This is more diagnostic feature than a useful one but we show it anyway. There is some basic overhead when we enter a group:

```
\bgroup [\the\currentstacksize]
    \bgroup [\the\currentstacksize]
        \bgroup [\the\currentstacksize]
        [\the\currentstacksize] \egroup
    [\the\currentstacksize] \egroup
[\the\currentstacksize] \egroup
```

[180] [181] [182] [182] [181] [180]

As soon as we define something or change a value, the stack gets populated by information needed for recovery after the group ends.

```
\bgroup [\the\currentstacksize]
    \scratchcounter 1
    \bgroup [\the\currentstacksize]
        \scratchdimen 1pt
        \scratchdimen 2pt
        \bgroup [\the\currentstacksize]
            \scratchcounter 2
            \scratchcounter 3
        [\the\currentstacksize] \egroup
    [\the\currentstacksize] \egroup
[\the\currentstacksize] \egroup
```

[180] [182] [184] [185] [183] [181]

The stack also keeps some state information, for instance when a box is being built. In LuaMetaTEX that is is quite a bit more than in other engines but it is compensated by more efficient save stack handling elsewhere.

```
\hbox \bgroup [\the\currentstacksize]
    \hbox \bgroup [\the\currentstacksize]
        \hbox \bgroup [\the\currentstacksize]
        [\the\currentstacksize] \egroup
    [\the\currentstacksize] \egroup
[\the\currentstacksize] \egroup
```

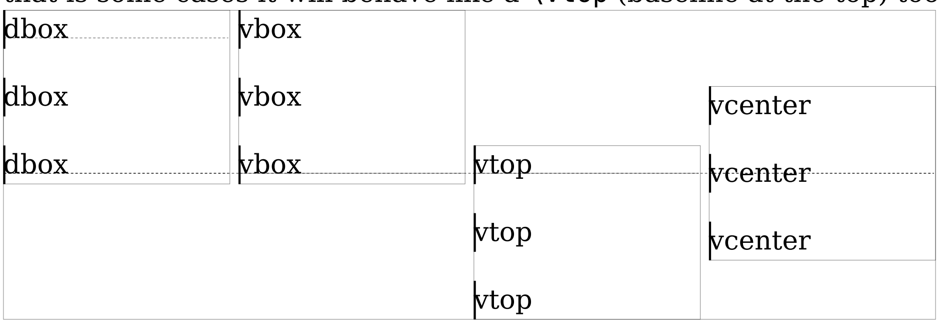
[189] [199] [209] [209] [199] [189]

135 \day

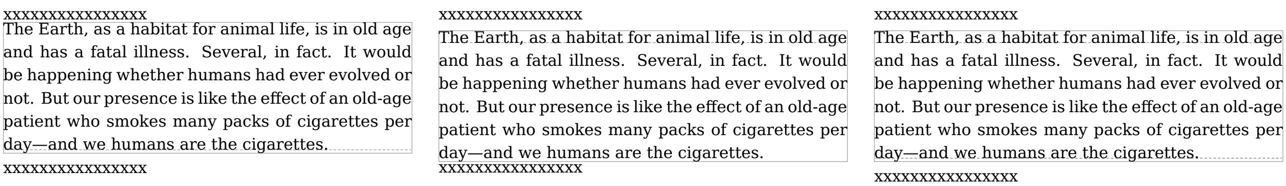
This internal number starts out with the day that the job started.

136 \dbox

A `\dbox` is just a `\vbox` (baseline at the bottom) but it has the property 'dual baseline' which means that is some cases it will behave like a `\vtop` (baseline at the top) too. Like:



A `\dbox` behaves like a `\vtop` when it's appended to a vertical list which means that the height of the first box or rule determines the (base)line correction that gets applied.



`\vbox` `\vtop` `\dbox`

137 \deadcycles

This counter is incremented every time the output routine is entered. When `\maxdeadcycles` is reached TEX will issue an error message, so you'd better reset its value when a page is done.

138 \def

This is the main definition command, as in:

```
\def\foo{l me}
```

with companions like `\gdef`, `\edef`, `\xdef`, etc. and variants like:

```
\def\foo#1{... #1...}
```

where the hash is used in the preamble and for referencing. More about that can be found in the low level manual about macros.

In the ConTEXt distribution you can find explanations about how LuaMetaTEX extends the argument parser. When defining a macro you can do this:

```
\def\foo(#1)#2{...}
```

Here the first argument between parentheses is mandate. But the magic prefix `\tolerant` makes that limitation go away:

```
\tolerant\def\foo(#1)#2{...}
```

A variant is this:

```
\tolerant\def\foo(#1)#*(#2){...}
```

Here we have two optional arguments, possibly be separated by spaces. There are more parsing options, that we just mention:

`+`	keep the braces
`-`	discard and don't count the argument
`/`	remove leading an trailing spaces and pars
`=`	braces are mandate
`_`	braces are mandate and kept
`^`	keep leading spaces
`1-9`	an argument
`0`	discard but count the argument
`*`	ignore spaces
`.`	ignore pars and spaces
`,`	push back space when no match
`:`	pick up scanning here
`;`	quit scanning

139 \defaulthyphenchar

When a font is loaded its hyphen character is set to this value. It can be changed afterwards. However, in LuaMetaTEX font loading is under Lua control so these properties can be set otherwise.

140 \defaultskewchar

When a font is loaded its skew character is set to this value. It can be changed afterwards. However, in LuaMetaTEX font loading is under Lua control so these properties can be set otherwise. Also, OpenType math fonts have top anchor instead.

141 \defcsname

We now get a series of log clutter avoidance primitives. It's fine if you argue that they are not really needed, just don't use them.

```
\expandafter\def\csname MyMacro:1\endcsname{...}
             \defcsname MyMacro:1\endcsname{...}
```

The fact that TEX has three (expanded and global) companions can be seen as a signal that less verbosity makes sense. It's just that macro packages use plenty of `\csname`'s.

142 \deferred

This is mostly a compatibility prefix and it can be checked at the Lua end when there is a Lua based assignment going on. It is the counterpart of `\immediate`. In the traditional engines a `\write` is normally deferred (turned into a node) and can be handled `\immediate`, while a `\special` does the opposite.

143 \delcode

This assigns delimiter properties to an eight bit character so it has little use in an OpenType math setup. WHen the assigned value is hex encoded, the first byte denotes the small family, then we have two bytes for the small index, followed by three similar bytes for the large variant.

144 \delimiter

This command inserts a delimiter with the given specification. In OpenType math we use a different command so it is unlikely that this primitive is used in LuaMetaTEX. It takes a number that can best be coded hexadecimal: one byte for the class, one for the small family, two for the small index, one for the large family and two for the large index. This demonstrates that it can't handle wide fonts. Also, in OpenType math fonts the larger sizes and extensible come from the same font as the small symbol. On top of that, in LuaMetaTEX we have more classes than fit in a byte.

145 \delimiterfactor

This is one of the parameters that determines the size of a delimiter: at least this factor times the formula height divided by 1000. In OpenType math different properties and strategies are used.

146 \delimitershortfall

This is one of the parameters that determines the size of a delimiter: at least the formula height minus this parameter. In OpenType math different properties and strategies are used.

147 \detokened

The following token will be serialized into characters with category ‘other’.

```
\toks0{123}
\def\foo{let's be \relax'd}
\def\oof#1{let's see #1}
\detokened\toks0
\detokened\foo
\detokened\oof
\detokened\setbox
\detokened X
```

Gives:

```
123
let's be \relax 'd
\oof
\setbox
X
```

Macros with arguments are not shown.

148 \detokenize

This ε-TEX primitive turns the content of the provides list will become characters, kind of verbatim.

```
\expandafter\let\expandafter\temp\detokenize{1} \meaning\temp
\expandafter\let\expandafter\temp\detokenize{A} \meaning\temp
```

```
the character U+0031 1
the character U+0041 A
```

149 \detokenized

The following (single) token will be serialized into characters with category ‘other’.

```
\toks0{123}
\def\foo{let's be \relax'd}
\def\oof#1{let's see #1}
\detokenized\toks0
\detokenized\foo
\detokenized\oof
\detokenized\setbox
\detokenized X
```

Gives:

```
\toks 0
\foo
\oof
\setbox
```

X

It is one of these new primitives that complement others like \detokened and such, and they are often mostly useful in experiments of some low level magic, which made them stay.

150 \dimen

Like \count this is a register accessor which is described in more detail in a low level manual.

```
\dimen0=10pt
```

While TEX has some assumptions with respect to the first ten count registers (as well as the one that holds the output, normally 255), all dimension registers are treated equal. However, you need to be aware of clashes with other usage. Therefore you can best use the predefined scratch registers or define dedicate ones with the **\newdimen** macro.

151 \dimendef

This primitive is used by the **\newdimen** macro when it relates a control sequence with a specific register. Only use it when you know what you're doing.

152 \dimensiondef

A variant of \integerdef is:

```
\dimensiondef\MyDimen = 1234pt
```

The properties are comparable to the ones described in the section \integerdef.

153 \dimexpr

This primitive is similar to of \numexpr but operates on dimensions instead. Integer quantities are interpreted as dimensions in scaled points.

```
\the\dimexpr (1pt + 2pt - 5pt) * 10 / 2 \relax
```

gives: -10.0pt. You can mix in symbolic integers and dimensions. This doesn't work:

because the engine scans for a dimension and only for an integer (or equivalent) after a * or /.

154 \dimexpression

This command is like \numexpression but results in a dimension instead of an integer. Where \dimexpr doesn't like 2 * 10pt this expression primitive is quite happy with it.

You can get an idea what the engines sees by setting \tracingexpressions to a value larger than zero. It shows the expression in rpn form.

```
\dimexpression 4pt * 2   + 6pt   \relax
\dimexpression 2   * 4pt + 6pt   \relax
\dimexpression 4pt * 2.5 + 6pt   \relax
```

```
\dimexpression 2.5 * 4pt + 6pt   \relax
\numexpression 2 * 4 + 6         \relax
\numexpression (1 + 2) * (3 + 4) \relax
```

The `\relax` is mandate simply because there are keywords involved so the parser needs to know where to stop scanning. It made no sense to be more clever and introduce fuzziness (so there is no room for exposing in-depth TEX insight and expertise here). In case you wonder: the difference in performance between the ε-TEX expression mechanism and the more extended variant will normally not be noticed, probably because they both use a different approach and because the ε-TEX variant also has been optimized.

155 \directlua

This is the low level interface to Lua:

Gives: "Greetings from the lua end!" as expected. In Lua we have access to all kind of internals of the engine. In LuaMetaTEX the interfaces have been polished and extended compared to LuaTEX. Although many primitives and mechanisms were added to the TEX frontend, the main extension interface remains Lua. More information can be found in documents that come with ConTEXt, in presentations and in articles.

156 \discretionary

The three snippets given with this command determine the pre, post and replace component of the injected discretionary node. The `penalty` keyword permits setting a penalty with this node. The `postword` keyword indicates that this discretionary starts a word, and `preword` ends it. With `break` the line break algorithm will prefer a pre or post component over a replace, and with `nobreak` replace will win over pre. With `class` you can set a math class that will determine spacing and such for discretionaries used in math mode.

157 \discretionaryoptions

Processing of discretionaries is controlled by this bitset:

`0x00000000`	normalword
`0x00000001`	preword
`0x00000002`	postword
`0x00000010`	preferbreak
`0x00000020`	prefernobreak
`0x00000040`	noitaliccorrection
`0x00000080`	nozeroitaliccorrection
`0x00010000`	userfirst
`0x40000000`	userlast

These can also be set on `\discretionary` using the `options` key.

158 \displayindent

The `\displaywidth`, `\displayindent` and `\predisplaysize` parameters are set by the line break routine (but can be adapted by the user), so that mid-par display formula can adapt itself to hanging indentation and par shapes. I order to calculate thee values and adapt the line break state afterwards such a display formula is assumed to occupy three lines, so basically a rather compact formula.

159 \displaylimits

By default in math display mode limits are place on top while in inline mode they are placed like scripts, after the operator. Placement can be forced with the `\limits` and `\nolimits` modifiers (after the operator). Because there can be multiple of these in a row there is `\displaylimits` that forces the default placement, so effectively it acts here as a reset modifier.

160 \displaystyle

One of the main math styles; integer representation: 0.

161 \displaywidowpenalties

This is a math specific variant of `\widowpenalties`.

162 \displaywidowpenalty

This is a math specific variant of `\widowpenalty`.

163 \displaywidth

This parameter determines the width of the formula and normally defaults to the `\hsize` unless we are in the middle of a paragraph in which case it is compensated for hanging indentation or the par shape.

164 \divide

The `\divide` operation can be applied to integers, dimensions, float, attribute and glue quantities. There are subtle rounding differences between the divisions in expressions and `\divide`:

```
\scratchcounter 1049 \numexpr\scratchcounter / 10\relax : 105
\scratchcounter 1049 \numexpr\scratchcounter : 10\relax : 104
\scratchcounter 1049 \divide\scratchcounter by 10       : 104
```

The `:` divider in `\dimexpr` is something that we introduced in LuaTEX.

165 \divideby

This is slightly more efficient variant of `\divide` that doesn't look for `by`. See previous section.

166 \doublehyphendemerits

This penalty will be added to the penalty assigned to a breakpoint that results in two lines ending with a hyphen.

167 \doublepenaltymode

When set to one this parameter signals the backend to use the alternative (left side) penalties of the pairs set on `\widowpenalties`, `\clubpenalties` and `\brokenpenalties`. For more information on this you can consult manuals (and articles) that come with ConTEXt.

168 \dp

Returns the depth of the given box.

169 \dpack

This does what `\dbox` does but without callback overhead.

170 \dsplit

This is the dual baseline variant of `\vsplit` (see `\dbox` for what that means).

171 \dump

This finishes an (ini) run and dumps a format (basically the current state of the engine).

172 \edef

This is the expanded version of `\def`.

```
\def \foo{foo}      \meaning\foo
\def \ofo{\foo\foo} \meaning\ofo
\edef\oof{\foo\foo} \meaning\oof
```

Because `\foo` is unprotected it will expand inside the body definition:

```
macro:foo
macro:\foo \foo
macro:foofoo
```

173 \edefcsname

This is the companion of `\edef`:

```
\expandafter\edef\csname MyMacro:1\endcsname{...}
             \edefcsname MyMacro:1\endcsname{...}
```

174 \edivide

When expressions were introduced the decision was made to round the divisions which is incompatible with the way `\divide` works. The expression scanners in LuaMetaTEX compensates that by providing a `:` for integer division. The `\edivide` does the opposite: it rounds the way expressions do.

```
\the\dimexpr .4999pt                     : 2 \relax           =.24994pt
```

```
\the\dimexpr .4999pt                    / 2 \relax          =.24995pt
\scratchdimen.4999pt \divide \scratchdimen 2 \the\scratchdimen=.24994pt
\scratchdimen.4999pt \edivide\scratchdimen 2 \the\scratchdimen=.24995pt
```

```
\the\numexpr   1001                       : 2 \relax            =500
\the\numexpr   1001                       / 2 \relax            =501
\scratchcounter1001  \divide \scratchcounter 2 \the\scratchcounter=500
\scratchcounter1001  \edivide\scratchcounter 2 \the\scratchcounter=501
```

Keep in mind that with dimensions we have a fractional part so we actually rounding applies to the fraction. For that reason we also provide `\rdivide`.

0.24994pt=.24994pt
0.24995pt=.24995pt
0.24994pt=.24994pt
0.24995pt=.24995pt

500=500
501=501
500=500
501=501

175 \edivideby

This the by-less variant of `\edivide`.

176 \efcode

This primitive originates in pdfTEX and can be used to set the expansion factor of a glyph (characters). This primitive is obsolete because the values can be set in the font specification that gets passed via Lua to TEX. Keep in mind that setting font properties at the TEX end is a global operation and can therefore influence related fonts. In LuaMetaTEX the `\cf` code can be used to specify the compression factor independent from the expansion factor. The primitive takes three values: font, character code, and factor.

177 \else

This traditional primitive is part of the condition testing mechanism. When a condition matches, TEX will continue till it sees an `\else` or `\or` or `\orelse` (to be discussed later). It will then do a fast skipping pass till it sees an `\fi`.

178 \emergencyextrastretch

This is one of the extended parbuilder parameters. You can you it so temporary increase the permitted stretch without knowing or messing with the normal value.

179 \emergencyleftskip

This is one of the extended parbuilder parameters (playground). It permits going ragged left in case of a too bad result.

180 \emergencyrightskip

This is one of the extended parbuilder parameters (playground). It permits going ragged right in case of a too bad result.

181 \emergencystretch

When set the par builder will run a third pass in order to fit the set criteria.

182 \end

This ends a TeX run, unless of course this primitive is redefined.

183 \endcsname

This primitive is used in combination with `\csname`, `\ifcsname` and `\begincsname` where its end the scanning for the to be constructed control sequence token.

184 \endgroup

This is the companion of the `\begingroup` primitive that opens a group. See `\beginsimplegroup` for more info.

185 \endinput

The engine can be in different input modes: reading from file, reading from a token list, expanding a macro, processing something that comes back from Lua, etc. This primitive quits reading from file:

```
this is seen
\endinput
here we're already quit
```

There is a catch. This is what the above gives:

this is seen

but how about this:

```
this is seen
before \endinput after
here we're already quit
```

Here we get:

this is seen before after

Because a token list is one line, the following works okay:

```
\def\quitrun{\ifsomething \endinput \fi}
```

but in a file you'd have to do this when you quit in a conditional:

```
\ifsomething
    \expandafter \endinput
\fi
```

While the one-liner works as expected:

```
\ifsomething \endinput \fi
```

186 \endlinechar

This is an internal integer register. When set to positive value the character with that code point will be appended to the line. The current value is 13. Here is an example:

```
\endlinechar\hyphenasciicode
line 1
line 2
```

line 1-line 2-

If the character is active, the property is honored and the command kicks in. The maximum value is 127 (the maximum character code a single byte utf character can carry.)

187 \endlocalcontrol

See `\beginlocalcontrol`.

188 \endmathgroup

This primitive is the counterpart of `\beginmathgroup`.

189 \endsimplegroup

This one ends a simple group, see `\beginsimplegroup` for an explanation about grouping primitives.

190 \enforced

The engine can be set up to prevent overloading of primitives and macros defined as `\permanent` or `\immutable`. However, a macro package might want to get around this in controlled situations, which is why we have a `\enforced` prefix. This prefix in interpreted differently in so called 'ini' mode when macro definitions can be dumped in the format. Internally they get an `always` flag as indicator that in these places an overload is possible.

```
\permanent\def\foo{original}

\def\oof         {\def\foo{fails}}
\def\oof{\enforced\def\foo{succeeds}}
```

Of course this only has an effect when overload protection is enabled.

191 \eofinput

This is a variant on `\input` that takes a token list as first argument. That list is expanded when the file ends. It has companion primitives `\atendoffile` (single token) and `\atendoffiled` (multiple tokens).

192 \eqno

This primitive stores the (typeset) content (presumably a number) and when the display formula is wrapped that number will end up right of the formula.

193 \errhelp

This is additional help information to `\errmessage` that triggers an error and shows a message.

194 \errmessage

This primitive expects a token list and shows its expansion on the console and/or in the log file, depending on how TEX is configured. After that it will enter the error state and either goes on or waits for input, again depending on how TEX is configured. For the record: we don't use this primitive in ConTEXt.

195 \errorcontextlines

This parameter determines the number on lines shown when an error is triggered.

196 \errorstopmode

This directive stops at every opportunity to interact. In ConTEXt we overload the actions in a callback and quit the run because we can assume that a successful outcome is unlikely.

197 \escapechar

This internal integer has the code point of the character that get prepended to a control sequence when it is serialized (for instance in tracing or messages).

198 \etoks

This assigns an expanded token list to a token register:

```
\def\temp{less stuff}
\etoks\scratchtoks{a bit \temp}
```

The orginal value of the register is lost.

199 \etoksapp

A variant of `\toksapp` is the following: it expands the to be appended content.

```
\def\temp{more stuff}
```

```
\etoksapp\scratchtoks{some \temp}
```

200 \etokspre

A variant of `\tokspre` is the following: it expands the to be prepended content.

```
\def\temp{less stuff}
\etokspre\scratchtoks{a bit \temp}
```

201 \eufactor

When we introduced the `es` (2.5cm) and `ts` (2.5mm) units as metric variants of the `in` we also added the `eu` factor. One `eu` equals one tenth of a `es` times the `\eufactor`. The `ts` is a convenient offset in test files, the `es` a convenient ones for layouts and image dimensions and the `eu` permits definitions that scale nicely without the need for dimensions. They also were a prelude to what later became possible with `\associateunit`.

202 \everybeforepar

This token register is expanded before a paragraph is triggered. The reason for triggering is available in `\lastpartrigger`.

203 \everycr

This token list gets expanded when a row ends in an alignment. Normally it will use `\noalign` as wrapper

```
{\everycr{\noalign{H}}             \halign{#\cr test\cr test\cr}}
{\everycr{\noalign{V}} \hsize 4cm \valign{#\cr test\cr test\cr}}
```

Watch how the `\cr` ending the preamble also get this treatment:

H
test

H
test

H

Vtest Vtest V

204 \everydisplay

This token list gets expanded every time we enter display mode. It is a companion of `\everymath`.

205 \everyeof

The content of this token list is injected when a file ends but it can only be used reliably when one is really sure that no other file is loaded in the process. So in the end it is of no real use in a more complex macro package.

206 \everyhbox

This token list behaves similar to `\everyvbox` so look there for an explanation.

207 \everyjob

This token list register is injected at the start of a job, or more precisely, just before the main control loop starts.

208 \everymath

Often math needs to be set up independent from the running text and this token list can be used to do that. There is also `\everydisplay`.

209 \everymathatom

When a math atom is seen this tokenlist is expanded before content is processed inside the atom body. It is basically a math companion for `\everyhbox` and friends and it is therefore probably just as useless. The next example shows how it works:

```
\everymathatom
  {\begingroup
   \scratchcounter\lastatomclass
   \everymathatom{}%
   \mathghost{\hbox to 0pt yoffset -1ex{\smallinfofont \setstrut\strut \the
     \scratchcounter\hss}}%
   \endgroup}
```

```
$ a = \mathatom class 4 {b} + \mathatom class 5 {c} $
```

We get a formula with open- and close atom spacing applied to b and c:

$a = \underset{4}{b} + \underset{5}{c}$

This example shows bit of all: we want the number to be invisible to the math machinery so we ghost it. So, we need to make sure we don't get recursion due to nested injection and expansion of `\everymathatom` and of course we need to store the number. The `\lastatomclass` state variable is only meaningful inside an explicit atom wrapper like `\mathatom` and `\mathatom`.

210 \everypar

When a paragraph starts this tokenlist is expanded before content is processed.

211 \everytab

This token list gets expanded every time we start a table cell in `\halign` or `\valign`.

212 \everyvbox

This token list gets expanded every time we start a vertical box. Like `\everyhbox` this is not that useful unless you are certain that there are no nested boxes that don't need this treatment. Of course you can wipe this register in this expansion, like:

```
\everyvbox{\kern10pt\everyvbox{}}
```

213 \exceptionpenalty

In exceptions we can indicate a penalty by `[digit]` in which case a penalty is injected set by this primitive, multiplied by the digit.

214 \exhyphenchar

The character that is used as pre component of the related discretionary.

215 \exhyphenpenalty

The penalty injected after `-` or `\-` unless `\hyphenationmode` is set to force the dedisated penalties.

216 \expand

Beware, this is not a prefix but a directive to ignore the protected characters of the following macro.

```
\protected \def \testa{\the\scratchcounter}
           \edef\testb{\testa}
           \edef\testc{\expand\testa}
```

The meaning of the three macros is:

```
protected macro:\the \scratchcounter
macro:\testa
macro:123
```

217 \expandactive

This a bit of an outlier and mostly there for completeness.

```
                              \meaningasis~
\edef\foo{~}                  \meaningasis\foo
\edef\foo{\expandactive~} \meaningasis\foo
```

There seems to be no difference but the real meaning of the first `\foo` is 'active character 126' while the second `\foo` 'protected call ' is.

\protected \def ~ {\nobreakspace }
\def \foo {~}
\def \foo {~}

Of course the definition of the active tilde is ConTEXt specific and situation dependent.

218 \expandafter

This original TEX primitive stores the next token, does a one level expansion of what follows it, which actually can be an not expandable token, and reinjects the stored token in the input. Like:

```
\expandafter\let\csname my weird macro name\endcsname{m w m n}
```

Without `\expandafter` the `\csname` primitive would have been let to the left brace (effectively then a begin group). Actually in this particular case the control sequence with the weird name is injected and when it didn't yet exist it will get the meaning `\relax` so we sort of have two assignments in a row then.

219 \expandafterpars

Here is another gobbler: the next token is reinjected after following spaces and par tokens have been read. So:

```
[\expandafterpars 1 2]
[\expandafterpars 3
4]
[\expandafterpars 5

6]
```

gives us: [12] [34] [56], because empty lines are like `\par` and therefore ignored.

220 \expandafterspaces

This is a gobbler: the next token is reinjected after following spaces have been read. Here is a simple example:

```
[\expandafterspaces 1 2]
[\expandafterspaces 3
4]
[\expandafterspaces 5

6]
```

We get this typeset: [12] [34] [5

6], because a newline normally is configured to be a space (and leading spaces in a line are normally being ingored anyway).

221 \expandcstoken

The rationale behind this primitive is that when we `\let` a single token like a character it is hard to compare that with something similar, stored in a macro. This primitive pushes back a single token alias created by `\let` into the input.

```
\let\tempA + \meaning\tempA

\let\tempB X \meaning\tempB \crlf
\let\tempC $ \meaning\tempC \par

\edef\temp         {\tempA} \doifelse{\temp}{+}{Y}{N} \meaning\temp \crlf
\edef\temp         {\tempB} \doifelse{\temp}{X}{Y}{N} \meaning\temp \crlf
\edef\temp         {\tempC} \doifelse{\temp}{X}{Y}{N} \meaning\temp \par
```

```
\edef\temp{\expandcstoken\tempA} \doifelse{\temp}{+}{Y}{N} \meaning\temp \crlf
\edef\temp{\expandcstoken\tempB} \doifelse{\temp}{X}{Y}{N} \meaning\temp \crlf
\edef\temp{\expandcstoken\tempC} \doifelse{\temp}{$}{Y}{N} \meaning\temp \par

\doifelse{\expandcstoken\tempA}{+}{Y}{N}
\doifelse{\expandcstoken\tempB}{X}{Y}{N}
\doifelse{\expandcstoken\tempC}{$}{Y}{N} \par
```

The meaning of the `\let` macros shows that we have a shortcut to a character with (in this case) catcode letter, other (here 'other character' gets abbreviated to 'character'), math shift etc.

```
the character U+002B 'plus sign'

the letter U+0058 X
math shift character U+0024 'dollar sign'

N macro:\tempA
N macro:\tempB
N macro:\tempC

Y macro:+
Y macro:X
Y macro:$

Y Y Y
```

Here we use the ConTEXt macro **`\doifelse`** which can be implemented in different ways, but the only property relevant to the user is that the expanded content of the two arguments is compared.

222 \expanded

This primitive complements the two expansion related primitives mentioned in the previous two sections. This time the content will be expanded and then pushed back into the input. Protected macros will not be expanded, so you can use this primitive to expand the arguments in a call. In ConTEXt you need to use **`\normalexpanded`** because we already had a macro with that name. We give some examples:

```
\def\A{!}
          \def\B#1{\string#1}                          \B{\A}
          \def\B#1{\string#1} \normalexpanded{\noexpand\B{\A}}
\protected\def\B#1{\string#1}                          \B{\A}
```

\A
!
\A

223 \expandedafter

The following two lines are equivalent:

```
\def\foo{123}
\expandafter[\expandafter[\expandafter\secondofthreearguments\foo]]
```

```
\expandedafter{[[\secondofthreearguments}\foo]]
```

In ConTEXt MkIV the number of times that one has multiple `\expandafters` is much larger than in ConTEXt LMTX thanks to some of the new features in LuaMetaTEX, and this primitive is not really used yet in the core code.

[[2]]
[[2]]

224 \expandeddetokenize

This is a companion to `\detokenize` that expands its argument:

```
\def\foo{12#H3}
\def\oof{\foo}
\detokenize        {\foo} \detokenize        {\oof}
\expandeddetokenize{\foo} \expandeddetokenize{\oof}
\edef\ofo{\expandeddetokenize{\foo}} \meaningless\ofo
\edef\ofo{\expandeddetokenize{\oof}} \meaningless\ofo
```

This is a bit more convenient than

```
\detokenize \expandafter {\normalexpanded {\foo}}
```

kind of solutions. We get:

\foo \oof
12#3 12#3
12#3
12#3

225 \expandedendless

This one loops forever but because the loop counter is not set you need to find a way to quit it.

226 \expandedloop

This variant of the previously introduced `\localcontrolledloop` doesn't enter a local branch but immediately does its work. This means that it can be used inside an expansion context like `\edef`.

```
\edef\whatever
  {\expandedloop 1 10 1
     {\scratchcounter=\the\currentloopiterator\relax}}

\meaningasis\whatever
```

`\def \whatever {\scratchcounter =1\relax \scratchcounter =2\relax \scratchcounter =3\relax \scratchcounter =4\relax \scratchcounter =5\relax \scratchcounter =6\relax \scratchcounter =7\relax \scratchcounter =8\relax \scratchcounter =9\relax \scratchcounter =10\relax }`

227 \expandedrepeat

This one takes one instead of three arguments which is sometimes more convenient.

228 \expandparameter

This primitive is a predecessor of `\parameterdef` so we stick to a simple example.

```
\def\foo#1#2%
  {\integerdef\MyIndexOne\parameterindex\plusone % 1
   \integerdef\MyIndexTwo\parameterindex\plustwo % 2
   \oof{P}\oof{Q}\oof{R}\norelax}

\def\oof#1%
  {<1:\expandparameter\MyIndexOne><1:\expandparameter\MyIndexOne>%
   #1%
   <2:\expandparameter\MyIndexTwo><2:\expandparameter\MyIndexTwo>}

\foo{A}{B}
```

In principle the whole parameter stack can be accessed but often one never knows if a specific macro is called nested. The original idea behind this primitive was tracing but it can also be used to avoid passing parameters along a chain of calls.

<1:A><1:A>P<2:B><2:B><1:A><1:A>Q<2:B><2:B><1:A><1:A>R<2:B><2:B>

229 \expandtoken

This primitive creates a token with a specific combination of catcode and character code. Because it assumes some knowledge of TeX we can show it using some `\expandafter` magic:

```
\expandafter\let\expandafter\temp\expandtoken 11 `X \meaning\temp
\expandafter\let\expandafter\temp\expandtoken 12 `X \meaning\temp
```

The meanings are:

```
the letter U+0058 X
the character U+0058 X
```

Using other catcodes is possible but the results of injecting them into the input directly (or here by injecting `\temp`) can be unexpected because of what TeX expects. You can get messages you normally won't get, for instance about unexpected alignment interference, which is a side effect of TeX using some catcode/character combinations as signals and there is no reason to change those internals. That said:

```
\xdef\tempA{\expandtoken  9 `X} \meaning\tempA
\xdef\tempB{\expandtoken 10 `X} \meaning\tempB
\xdef\tempC{\expandtoken 11 `X} \meaning\tempC
\xdef\tempD{\expandtoken 12 `X} \meaning\tempD
```

are all valid and from the meaning you cannot really deduce what's in there:

```
macro:X
macro:X
macro:X
macro:X
```

But you can be assured that:

```
[AB: \ifx\tempA\tempB Y\else N\fi]
[AC: \ifx\tempA\tempC Y\else N\fi]
[AD: \ifx\tempA\tempD Y\else N\fi]
[BC: \ifx\tempB\tempC Y\else N\fi]
[BD: \ifx\tempB\tempD Y\else N\fi]
[CD: \ifx\tempC\tempD Y\else N\fi]
```

makes clear that they're different: [AB: N] [AC: N] [AD: N] [BC: N] [BD: N] [CD: N], and in case you wonder, the characters with catcode 10 are spaces, while those with code 9 are ignored.

230 `\expandtoks`

This is a more efficient equivalent of `\the` applied to a token register, so:

```
\scratchtoks{just some tokens}
\edef\TestA{[\the       \scratchtoks]}
\edef\TestB{[\expandtoks\scratchtoks]}
[\the       \scratchtoks] [\TestA] \meaning\TestA
[\expandtoks\scratchtoks] [\TestB] \meaning\TestB
```

does the expected:

[just some tokens] [[just some tokens]] macro:[just some tokens]
[just some tokens] [[just some tokens]] macro:[just some tokens]

The `\expandtoken` primitive avoid a copy into the input when there is no need for it.

231 `\explicitdiscretionary`

This is the verbose alias for one of TEX's single character control sequences: `\-`.

232 `\explicithyphenpenalty`

The penalty injected after an automatic discretionary `\-`, when `\hyphenationmode` enables this.

233 `\explicititaliccorrection`

This is the verbose alias for one of TEX's single character control sequences: `\/`. Italic correction is a character property specific to TEX and the concept is not present in modern font technologies. There is a callback that hooks into this command so that a macro package can provide its own solution to this (or alternatively it can assign values to the italic correction field.

234 `\explicitspace`

This is the verbose alias for one of TEX's single character control sequences: `\`. A space is inserted with properties according the space related variables. There is look-back involved in order to deal with space factors.

When \nospaces is set to 1 no spaces are inserted, when its value is 2 a zero space is inserted.

235 \fam

In a numeric context it returns the current family number, otherwise it sets the given family. The number of families in a traditional engine is 16, in LuaTEX it is 256 and in LuaMetaTEX we have at most 64 families. A future version can lower that number when we need more classes.

236 \fi

This traditional primitive is part of the condition testing mechanism and ends a test. So, we have:

```
\ifsomething ... \else ... \fi
\ifsomething ... \or ... \or ... \else ... \fi
\ifsomething ... \orelse \ifsometing  ... \else ... \fi
\ifsomething ... \or ... \orelse \ifsometing  ... \else ... \fi
```

The \orelse is new in LuaMetaTEX and a continuation like we find in other programming languages (see later section).

237 \finalhyphendemerits

This penalty will be added to the penalty assigned to a breakpoint when that break results in a pre-last line ending with a hyphen.

238 \firstmark

This is a reference to the first mark on the (split off) page, it gives back tokens.

239 \firstmarks

This is a reference to the first mark with the given id (a number) on the (split off) page, it gives back tokens.

240 \firstvalidlanguage

Language id's start at zero, which makes it the first valid language. You can set this parameter to indicate the first language id that is actually a language. The current value is 1, so lower values will not trigger hyphenation.

241 \fitnessclasses

We can have more fitness classes than traditional TEX that has 'very loose', 'loose', 'decent' and 'tight'. In ConTEXt we have 'veryloose', 'loose', 'almostloose', 'barelyloose', 'decent', 'barelytight', 'almost-tight', 'tight' and 'verytight'. Although we can go up to 31 this is already more than enough. The default is the same as in regular TEX.

The \fitnessclasses can be used to set the criteria and like other specification primitives (like \parshape and \widowpenalties, it expects a count. With \adjacentdemerits one can set the demerits that are added depending on the distance between classes (in traditional TEX that is `adjdemerits` for all distances larger than one. With the double option the demerits come in pairs because we can go up or down in the list of fitness classes.

242 \float

In addition to integers and dimensions, which are fixed 16.16 integer floats we also have ‘native’ floats, based on 32 bit posit unums.

```
\float0 = 123.456           \the\float0
\float2 = 123.456           \the\float0
\advance \float0 by 123.456 \the\float0
\advance \float0 by \float2 \the\float0
\divideby\float0 3          \the\float0
```

They come with the same kind of support as the other numeric data types:

123.45600032806396484
123.45600032806396484
246.91200065612792969
370.36800384521484375
123.45600128173828125

We leave the subtle differences between floats and dimensions to the user to investigate:

```
\dimen00 = 123.456pt          \the\dimen0
\dimen02 = 123.456pt          \the\dimen0
\advance \dimen0 by 123.456pt \the\dimen0
\advance \dimen0 by \dimen2   \the\dimen0
\divideby\dimen0 3            \the\dimen0
```

The nature of posits is that they are more accurate around zero (or smaller numbers in general).

123.456pt
123.456pt
246.91199pt
370.36798pt
123.456pt

This also works:

```
\float0=123.456e4
\float2=123.456    \multiply\float2 by 10000
\the\float0
\the\float2
```

The values are (as expected) the same:

1234560
1234560

243 \floatdef

This primitive defines a symbolic (macro) alias to a float register, just like `\countdef` and friends do.

244 \floatexpr

This is the companion of `\numexpr`, `\dimexpr` etc.

```
\scratchcounter 200
\the   \floatexpr 123.456/456.123     \relax
\the   \floatexpr 1.2*\scratchcounter \relax
\the   \floatexpr \scratchcounter/3   \relax
\number\floatexpr \scratchcounter/3   \relax
```

Watch the difference between `\the` and `\number`:

0.2706638324563503265
240
66.666669845581054 69
67

245 \floatingpenalty

When an insertion is split (across pages) this one is added to to accumulated `\insertpenalties`. In LuaMetaTEX this penalty can be stored per insertion class.

246 \flushmarks

This primitive is an addition to the multiple marks mechanism that originates in ε-TEX and inserts a reset signal for the mark given category that will perform a clear operation (like `\clearmarks` which operates immediately).

247 \font

This primitive is either a symbolic reference to the current font or in the perspective of an assignment is used to trigger a font definitions with a given name (`cs`) and specification. In LuaMetaTEX the assignment will trigger a callback that then handles the definition; in addition to the filename an optional size specifier is checked (`at` or `scaled`).

In LuaMetaTEX *all* font loading is delegated to Lua, and there is no loading code built in the engine. Also, instead of `\font` in ConTEXt one uses dedicated and more advanced font definition commands.

248 \fontcharba

Fetches the bottom anchor of a character in the given font, so:

results in: 1.8275pt. However, this anchor is only available when it is set and it is not part of OpenType; it is something that ConTEXt provides for math fonts.

249 `\fontchardp`

Fetches the depth of a character in the given font, so:

results in: 2.22168pt.

250 `\fontcharht`

Fetches the width of a character in the given font, so:

results in: 5.33203pt.

251 `\fontcharic`

Fetches the italic correction of a character in the given font, but because it is not an OpenType property it is unlikely to return something useful. Although math fonts have such a property in ConTEXt we deal with it differently.

252 `\fontcharta`

Fetches the top anchor of a character in the given font, so:

results in: 1.8275pt. This is a specific property of math characters because in text mark anchoring is driven by a feature.

253 `\fontcharwd`

Fetches the width of a character in the given font, so:

results in: 6.40137pt.

254 `\fontdimen`

A traditional TEX font has a couple of font specific dimensions, we only mention the seven that come with text fonts:

1. The slant (slope) is an indication that we have an italic shape. The value divided by 65.536 is a fraction that can be compared with for instance the `slanted` operator in MetaPost. It is used for positioning accents, so actually not limited to oblique fonts (just like italic correction can be a property of any character). It is not relevant in the perspective of OpenType fonts where we have glyph specific top and bottom anchors.
2. Unless is it overloaded by `\spaceskip` this determines the space between words (or actually anything separated by a space).
3. This is the stretch component of `\fontdimen` 2(space).
4. This is the shrink component of `\fontdimen` 2(space).
5. The so called ex-height is normally the height of the ‘x’ and is also accessible as `em` unit.
6. The so called em-width or in TEX speak quad width is about the with of an ‘M’ but in many fonts just matches the font size. It is also accessible as `em` unit.
7. This is a very TEX specific property also known as extra space. It gets *added* to the regular space after punctuation when `\spacefactor` is 2000 or more. It can be overloaded by `\xspaceskip`.

This primitive expects a a number and a font identifier. Setting a font dimension is a global operation as it directly pushes the value in the font resource.

255 \fontid

Returns the (internal) number associated with the given font:

```
{\bf \xdef\MyFontA{\the\fontid\font}}
{\sl \xdef\MyFontB{\setfontid\the\fontid\font}}
```

with:

```
test {\setfontid\MyFontA test} test {\MyFontB test} test
```

gives: test **test** test *test* test.

256 \fontmathcontrol

The `\mathfontcontrol` parameter controls how the engine deals with specific font related properties and possibilities. It is set at the TEX end. It makes it possible to fine tune behavior in this mixed traditional and not perfect OpenType math font arena. One can also set this bitset when initializing (loading) the font (at the Lua end) and the value set there is available in `\fontmathcontrol`. The bits set in the font win over those in `\mathfontcontrol`. There are a few cases where we set these options in the (so called) goodie files. For instance we ignore font kerns in Libertinus, Antykwa and some more.

modern	0x0
pagella	0x0
antykwa	0x37EF3FF
libertinus	0x37EF3FF

257 \fontname

Depending on how the font subsystem is implemented this gives some information about the used font:

```
{\tf \fontname\font}
{\bf \fontname\font}
{\sl \fontname\font}
```

DejaVuSerif at 10.0pt
DejaVuSerif-Bold at 10.0pt
DejaVuSerif-Italic at 10.0pt

258 \fontspecdef

This primitive creates a reference to a specification that when triggered will change multiple parameters in one go.

```
\fontspecdef\MyFontSpec
    \fontid\font
```

```
    scale  1200
    xscale 1100
    yscale  800
    weight  200
    slant   500
\relax
```

is equivalent to:

```
\fontspecdef\MyFontSpec
    \fontid\font
    all 1200 1100 800 200 500
\relax
```

while

```
\fontspecdef\MyFontSpec
    \fontid\font
    all \glyphscale \glyphxscale \glyphyscale \glyphslant \glyphweight
\relax
```

is the same as

```
\fontspecdef\MyFontSpec
    \fontid\font
\relax
```

The engine adapts itself to these glyph parameters but when you access certain quantities you have to make sure that you use the scaled ones. The same is true at the Lua end. This is somewhat fundamental in the sense that when one uses these sort of dynamic features one also need to keep an eye on code that uses font specific dimensions.

259 \fontspecid

Internally a font reference is a number and this primitive returns the number of the font bound to the specification.

260 \fontspecifiedname

Depending on how the font subsystem is implemented this gives some information about the (original) definition of the used font:

```
{\tf \fontspecifiedname\font}
{\bf \fontspecifiedname\font}
{\sl \fontspecifiedname\font}
```

Serif sa 1
SerifBold sa 1
SerifSlanted sa 1

261 \fontspecifiedsize

Depending on how the font subsystem is implemented this gives some information about the (original) size of the used font:

```
{\tf  \the\fontspecifiedsize\font : \the\glyphscale}
{\bfa \the\fontspecifiedsize\font : \the\glyphscale}
{\slx \the\fontspecifiedsize\font : \the\glyphscale}
```

Depending on how the font system is setup, this is not the real value that is used in the text because we can use for instance `\glyphscale`. So the next lines depend on what font mode this document is typeset.

10.0pt: 1000
10.0pt: 1200
10.0pt: 800

262 \fontspecscale

This returns the scale factor of a fontspec where as usual 1000 means scaling by 1.

263 \fontspecslant

This returns the slant factor of a font specification, usually between zero and 1000 with 1000 being maximum slant.

264 \fontspecweight

This returns the weight of the font specification. Reasonable values are between zero and 500.

265 \fontspecxscale

This returns the scale factor of a font specification where as usual 1000 means scaling by 1.

266 \fontspecyscale

This returns the scale factor of a font specification where as usual 1000 means scaling by 1.

267 \fonttextcontrol

This returns the text control flags that are set on the given font, here `0x208`. Bits that can be set are:

`0x01` collapsehyphens
`0x02` baseligaturing
`0x04` basekerning
`0x08` noneprotected
`0x10` hasitalics
`0x20` autoitalics

268 \forcedleftcorrection

This is a callback driven left correction signal similar to italic corrections.

269 \forcedrightcorrection

This is a callback driven right correction signal similar to italic corrections.

270 \formatname

It is in the name: `cont-en`, but we cheat here by only showing the filename and not the full path, which in a ConTEXt setup can span more than a line in this paragraph.

271 \frozen

You can define a macro as being frozen:

```
\frozen\def\MyMacro{...}
```

When you redefine this macro you get an error:

```
! You can't redefine a frozen macro.
```

This is a prefix like `\global` and it can be combined with other prefixes.[5]

272 \futurecsname

In order to make the repertoire of `def`, `let` and `futurelet` primitives complete we also have:

```
\futurecsname MyMacro:1\endcsname\MyAction
```

273 \futuredef

We elaborate on the example of using `\futurelet` in the previous section. Compare that one with the next:

```
\def\MySpecialToken{[}
\def\DoWhatever{\ifx\NextToken\MySpecialToken YES\else NOP\fi : }
\futurelet\NextToken\DoWhatever [A]\crlf
\futurelet\NextToken\DoWhatever (A)\par
```

This time we get:

NOP: [A]
NOP: (A)

It is for that reason that we now also have `\futuredef`:

```
\def\MySpecialToken{[}
```

[5] The `\outer` and `\long` prefixes are no-ops in LuaMetaTEX and LuaTEX can be configured to ignore them.

```
\def\DoWhatever{\ifx\NextToken\MySpecialToken YES\else NOP\fi : }
\futuredef\NextToken\DoWhatever [A]\crlf
\futuredef\NextToken\DoWhatever (A)\par
```

So we're back to what we want:

YES: [A]
NOP: (A)

274 \futureexpand

This primitive can be used as an alternative to a \futurelet approach, which is where the name comes from.[6]

```
\def\variantone<#1>{(#1)}
\def\varianttwo#1{[#1]}
\futureexpand<\variantone\varianttwo<one>
\futureexpand<\variantone\varianttwo{two}
```

So, the next token determines which of the two variants is taken:

(one) [two]

Because we look ahead there is some magic involved: spaces are ignored but when we have no match they are pushed back into the input. The next variant demonstrates this:

```
\def\variantone<#1>{(#1)}
\def\varianttwo{}
\def\temp{\futureexpand<\variantone\varianttwo}
[\temp <one>]
[\temp {two}]
[\expandafter\temp\space <one>]
[\expandafter\temp\space {two}]
```

This gives us:

[(one)] [two] [(one)] [two]

275 \futureexpandis

We assume that the previous section is read. This variant will not push back spaces, which permits a consistent approach i.e. the user can assume that macro always gobbles the spaces.

```
\def\variantone<#1>{(#1)}
\def\varianttwo{}
\def\temp{\futureexpandis<\variantone\varianttwo}
[\temp <one>]
[\temp {two}]
[\expandafter\temp\space <one>]
[\expandafter\temp\space {two}]
```

---

6 In the engine primitives that have similar behavior are grouped in commands that are then dealt with together, code wise.

So, here no spaces are pushed back. This `is` in the name of this primitive means 'ignore spaces', but having that added to the name would have made the primitive even more verbose (after all, we also don't have `\expandeddef` but `\edef` and no `\globalexpandeddef` but `\xdef`.

[(one)] [two] [(one)] [two]

276 \futureexpandisap

This primitive is like the one in the previous section but also ignores par tokens, so `isap` means 'ignore spaces and paragraphs'.

277 \futurelet

The original TEX primitive `\futurelet` can be used to create an alias to a next token, push it back into the input and then expand a given token.

```
\let\MySpecialTokenL[
\let\MySpecialTokenR] % nicer for checker
\def\DoWhatever{\ifx\NextToken\MySpecialTokenL YES\else NOP\fi : }
\futurelet\NextToken\DoWhatever [A]\crlf
\futurelet\NextToken\DoWhatever (A)\par
```

This is typically the kind of primitive that most users will never use because it expects a sane follow up handler (here `\DoWhatever`) and therefore is related to user interfacing.

YES: [A]
NOP: (A)

278 \gdef

The is the global companion of `\def`.

279 \gdefcsname

As with standard TEX we also define global ones:

```
\expandafter\gdef\csname MyMacro:1\endcsname{...}
             \gdefcsname MyMacro:1\endcsname{...}
```

280 \givenmathstyle

This primitive expects a math style and returns it when valid or otherwise issues an error.

281 \gleaders

Leaders are glue with special property: a box, rule of (in LuaMetaTEX) glyph, like:

x MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMx
xx MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM xx

xMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMx
xx MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM xx

xMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMx
xxMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMxx

xMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMx
xx MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM xx

Leaders fill the available space. The `\leaders` command starts at the left edge and stops when there is no more space. The blobs get centered when we use `\cleaders`: excess space is distributed before and after a blob while `\xleaders` also puts space between the blobs.

When a rule is given the advance (width or height and depth) is ignored, so these are equivalent.

```
x\leaders \hrule           \hfill x
x\leaders \hrule width 1cm \hfill x
```

When a box is used one will normally have some alignment in that box.

```
x\leaders \hbox {\hss.\hss} \hfill            x
x\leaders \hbox {\hss.\hss} \hskip 6cm \relax x
```

The reference point is the left edge of the current (outer) box and the effective glue (when it has stretch or shrink) depends on that box. The `\gleaders` variant takes the page as reference. That makes it possible to 'align' across boxes.

282 `\glet`

This is the global companion of `\let`. The fact that it is not an original primitive is probably due to the expectation for it not it not being used (as) often (as in ConTEXt).

283 `\gletcsname`

Naturally LuaMetaTEX also provides a global variant:

```
\expandafter\global\expandafter\let\csname MyMacro:1\endcsname\relax
\expandafter                  \glet\csname MyMacro:1\endcsname\relax
                               \gletcsname MyMacro:1\endcsname\relax
```

So, here we save even more.

284 `\glettonothing`

This is the global companion of `\lettonothing`.

285 `\global`

This is one of the original prefixes that can be used when we define a macro of change some register.

```
\bgroup
       \def\MyMacroA{a}
```

```
\global\def\MyMacroB{a}
      \gdef\MyMacroC{a}
\egroup
```

The macro defined in the first line is forgotten when the groups is left. The second and third definition are both global and these definitions are retained.

286 \globaldefs

When set to a positive value, this internal integer will force all definitions to be global, and in a complex macro package that is not something a user will do unless it is very controlled.

287 \glueexpr

This is a more extensive variant of `\dimexpr` that also handles the optional stretch and shrink components.

288 \glueshrink

This returns the shrink component of a glue quantity. The result is a dimension so you need to apply `\the` when applicable.

289 \glueshrinkorder

This returns the shrink order of a glue quantity. The result is a integer so you need to apply `\the` when applicable.

290 \gluespecdef

A variant of `\integerdef` and `\dimensiondef` is:

```
\gluespecdef\MyGlue = 3pt plus 2pt minus 1pt
```

The properties are comparable to the ones described in the previous sections.

291 \gluestretch

This returns the stretch component of a glue quantity. The result is a dimension so you need to apply `\the` when applicable.

292 \gluestretchorder

This returns the stretch order of a glue quantity. The result is a integer so you need to apply `\the` when applicable.

293 \gluetomu

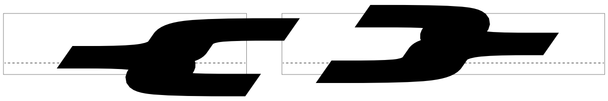
The sequence `\the\gluetomu 20pt plus 10pt minus 5pt` gives 20.0mu plus 10.0mu minus 5.0mu.

294 `\glyph`

This is a more extensive variant of `\char` that permits setting some properties if the injected character node.

```
\ruledhbox{\glyph
    scale 2000 xscale 9000 yscale 1200
    slant 700 weight 200
    xoffset 10pt yoffset -5pt left 10pt right 20pt
    123}
\quad
\ruledhbox{\glyph
    scale 2000 xscale 9000 yscale 1200
    slant 700 weight 200
    125}
```

In addition one can specify `font` (symbol), `id` (valid font id number), an `options` (bit set) and `raise`.



When no parameters are set, the current ones are used. More details and examples of usage can be found in the ConTEXt distribution.

295 `\glyphdatafield`

The value of this parameter is assigned to data field in glyph nodes that get injected. It has no meaning in itself but can be used at the Lua end.

296 `\glyphoptions`

The value of this parameter is assigned to the options field in glyph nodes that get injected.

`0x00000000`	normal	`0x00000400`	mathdiscretionary
`0x00000001`	noleftligature	`0x00000800`	mathsitalicstoo
`0x00000002`	norightligature	`0x00001000`	mathartifact
`0x00000004`	noleftkern	`0x00002000`	weightless
`0x00000008`	norightkern	`0x00004000`	spacefactoroverload
`0x00000010`	noexpansion	`0x00008000`	checktoddler
`0x00000020`	noprotrusion	`0x00010000`	checktwin
`0x00000040`	noitaliccorrection	`0x00020000`	istoddler
`0x00000080`	nozeroitaliccorrection	`0x00100000`	userfirst
`0x00000100`	applyxoffset	`0x40000000`	userlast
`0x00000200`	applyyoffset		

297 `\glyphscale`

An integer parameter defining the current glyph scale, assigned to glyphs (characters) inserted into the current list.

298 \glyphscriptfield

The value of this parameter is assigned to script field in glyph nodes that get injected. It has no meaning in itself but can be used at the Lua end.

299 \glyphscriptscale

This multiplier is applied to text font and glyph dimension properties when script style is used.

300 \glyphscriptscriptscale

This multiplier is applied to text font and glyph dimension properties when script script style is used.

301 \glyphslant

An integer parameter defining the current glyph slant, assigned to glyphs (characters) inserted into the current list.

302 \glyphstatefield

The value of this parameter is assigned to script state in glyph nodes that get injected. It has no meaning in itself but can be used at the Lua end.

303 \glyphtextscale

This multiplier is applied to text font and glyph dimension properties when text style is used.

304 \glyphweight

An integer parameter defining the current glyph weight, assigned to glyphs (characters) inserted into the current list.

305 \glyphxoffset

An integer parameter defining the current glyph x offset, assigned to glyphs (characters) inserted into the current list. Normally this will only be set when one explicitly works with glyphs and defines a specific sequence.

306 \glyphxscale

An integer parameter defining the current glyph x scale, assigned to glyphs (characters) inserted into the current list.

307 \glyphxscaled

This primitive returns the given dimension scaled by the `\glyphscale` and `\glyphxscale`.

308 \glyphyoffset

An integer parameter defining the current glyph x offset, assigned to glyphs (characters) inserted into the current list. Normally this will only be set when one explicitly works with glyphs and defines a specific sequence.

309 \glyphyscale

An integer parameter defining the current glyph y scale, assigned to glyphs (characters) inserted into the current list.

310 \glyphyscaled

This primitive returns the given dimension scaled by the `\glyphscale` and `\glyphyscale`.

311 \gtoksapp

This is the global variant of `\toksapp`.

312 \gtokspre

This is the global variant of `\tokspre`.

313 \halign

This command starts horizontally aligned material. Macro packages use this command in table mechanisms and math alignments. It starts with a preamble followed by entries (rows and columns). There are some related primitives, for instance `\alignmark` duplicates the functionality of `#` inside alignment preambles, while `\aligntab` duplicates the functionality of `&`. The `\aligncontent` primitive directly refers to an entry so that one does not get repeated.

Alignments can be traced with `\tracingalignments`. When set to 1 basics usage is shown, for instance of `\noalign` but more interesting is 2 or more: you then get the preambles reported.

The `\halign` (tested) and `\valign` (yet untested) primitives accept a few keywords in addition to `to` and `spread`:

keyword	explanation
`attr`	set the given attribute to the given value
`callback`	trigger the `alignment_filter` callback
`discard`	discard zero `\tabskip`'s
`noskips`	don't even process zero `\tabskip`'s
`reverse`	reverse the final rows

In the preamble the `\tabsize` primitive can be used to set the width of a column. By doing so one can avoid using a box in the preamble which, combined with the sparse tabskip features, is a bit easier on memory when you produce tables that span hundreds of pages and have a dozen columns.

The `\everytab` complements the `\everycr` token register but is sort of experimental as it might become more selective and powerful some day.

The two primitives `\alignmentcellsource` and `\alignmentwrapsource` that associate a source id (integer) to the current cell and row (line). Sources and targets are experimental and are being explored in ConTEXt so we'll see where that ends up in.

314 \hangafter

This parameter tells the par builder when indentation specified with `\hangindent` starts. A negative value does the opposite and starts indenting immediately. So, a value of −2 will make the first two lines indent.

315 \hangindent

This parameter relates to `\hangafter` and sets the amount of indentation. When larger than zero indentation happens left, otherwise it starts at the right edge.

316 \hbadness

This sets the threshold for reporting a horizontal badness value, its current value is 0.

317 \hbox

This constructs a horizontal box. There are a lot of optional parameters so more details can be found in dedicated manuals. When the content is packed a callback can kick in that can be used to apply for instance font features.

318 \hccode

The TEX engine is good at hyphenating but traditionally that has been limited to hyphens. Some languages however use different characters. You can set up a different `\hyphenchar` as well as pre and post characters, but there's also a dedicated code for controlling this.

```
\hccode"2013 "2013

\hsize 50mm test\char"2013test\par
\hsize  1mm test\char"2013test\par

\hccode"2013 `!

\hsize 50mm test\char"2013test\par
\hsize  1mm test\char"2013test\par
```

This example shows that we can mark a character as hyphen-like but also can remap it to something else:

test–test
test–
test
test–test
test!
test

319 \hfil

This is a shortcut for `\hskip plus 1 fil` (first order filler).

320 \hfill

This is a shortcut for `\hskip plus 1 fill` (second order filler).

321 \hfilneg

This is a shortcut for `\hskip plus - 1 fil` so it can compensate `\hfil`.

322 \hfuzz

This dimension sets the threshold for reporting horizontal boxes that are under- or overfull. The current value is 0.1pt.

323 \hjcode

The so called lowercase code determines if a character is part of a to-be-hyphenated word. In LuaTEX we introduced the 'hyphenation justification' code as replacement. When a language is saved and no `\hjcode` is set the `\lccode` is used instead. This code serves a second purpose. When the assigned value is greater than 0 but less than 32 it indicated the to be used length when checking for left- and righthyphenmin. For instance it make sense to set the code to 2 for characters like œ.

324 \hkern

This primitive is like `\kern` but will force the engine into horizontal mode if it isn't yet.

325 \hmcode

The `hm` stands for 'hyphenation math'. When bit 1 is set the characters will be repeated on the next line after a break. The second bit concerns italic correction but is of little relevance now that we moved to a different model in ConTEXt. Here are some examples, we also show an example of `\mathdiscretionary` because that is what this code triggers:

```
test $ \dorecurse {50} {
    a \discretionary class 2 {$\darkred +$}{$\darkgreen +$}{$\darkblue +$}
} b$

test $ a \mathdiscretionary class 1 {-}{-}{-} b$

\bgroup
    \hmcode"002B=1 % +
    \hmcode"002D=1 % -
    \hmcode"2212=1 % -
    test $ \dorecurse{50}{a + b - } c$
\egroup
```

test $a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a +$
$+ a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + a + b$

test $a - b$

test $a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b -$
$- a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a +$
$+ b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b -$
$- a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - a + b - c$

326 \holdinginserts

When set to a positive value inserts will be kept in the stream and not moved to the insert registers.

327 \holdingmigrations

When set to a positive value marks (and adjusts) will be kept in the stream and not moved to the outer level or related registers.

328 \hpack

This primitive is like `\hbox` but without the callback overhead.

329 \hpenalty

This primitive is like `\penalty` but will force the engine into horizontal mode if it isn't yet.

330 \hrule

This creates a horizontal rule. Unless the width is set it will stretch to fix the available width. In addition to the traditional `width`, `height` and `depth` specifiers some more are accepted. These are discussed in other manuals. To give an idea:

```
h\hrule width 10mm height 2mm depth 1mm \relax rule
h\hrule width 10mm height 2mm depth 1mm xoffset 30mm yoffset -10mm \relax rule
v\vrule width 10mm height 2mm depth 1mm \relax rule
v\vrule width 10mm height 2mm depth 1mm xoffset 30mm yoffset  10mm \relax rule
```

The `\relax` stops scanning and because we have more keywords we get a different error report than in traditional TEX when a lookahead confuses the engine. On separate lines we get the following.

h
rule
h
rule
v rule
v rule

331 \hsize

This sets (or gets) the current horizontal size.

```
\hsize 40pt \setbox0\vbox{x} hsize: \the\wd0
\setbox0\vbox{\hsize 40pt x} hsize: \the\wd0
```

In both cases we get the same size reported but the first one will also influence the current paragraph when used ungrouped.

hsize:
40.0pt
hsize:
40.0pt

332 \hskip

The given glue is injected in the horizontal list. If possible horizontal mode is entered.

333 \hss

In traditional TEX glue specifiers are shared. This makes a lot of sense when memory has to be saved. For instance spaces in a paragraph of text are often the same and a glue specification has at least an amount, stretch, shrink, stretch order and shrink order field plus a leader pointer; in LuaMetaTEX we have even more fields. In LuaTEX these shared (and therefore referenced) glue spec nodes became just copies.

```
x\hbox to 0pt{\hskip 0pt plus 1 fil minus 1 fil\relax test}x
x\hbox to 0pt{\hss test}x
x\hbox to 0pt{test\hskip 0pt plus 1 fil minus 1 fil\relax}x
x\hbox to 0pt{test\hss}x
```

The \hss primitives injects a glue node with one order stretch and one order shrink. In traditional TEX this is a reference to a shared specification, and in LuaTEX just a copy of a predefined specifier. The only gain is now in tokens because one could just be explicit or use a glue register with that value because we have plenty glue registers.

testxx
testxx
xxtest
xxtest

We could have this:

```
\permanent\protected\untraced\def\hss
  {\hskip0pt plus 1 fil minus 1 fil\relax}
```

or this:

```
\gluespecdef\hssglue 0pt plus 1 fil minus 1 fil

\permanent\protected\untraced\def\hss
  {\hskip\hssglue}
```

but we just keep the originals around.

334 \ht

Returns the height of the given box.

335 \hyphenation

The list passed to this primitive contains hyphenation exceptions that get bound to the current language. In LuaMetaTEX this can be managed at the Lua end. Exceptions are not stored in the format file.

336 \hyphenationmin

This property (that also gets bond to the current language) sets the minimum length of a word that gets hyphenated.

337 \hyphenationmode

TODO

338 \hyphenchar

This is one of the font related primitives: it returns the number of the hyphen set in the given font.

339 \hyphenpenalty

Discretionary nodes have a related default penalty. The `\hyphenpenalty` is injected after a regular discretionary, and `\exhyphenpenalty` after `\-` or `-`. The later case is called an automatic discretionary. In LuaMetaTEX we have two extra penalties: `\explicithyphenpenalty` and `\automatichyphenpenalty` and these are used when the related bits are set in `\hyphenationmode`.

340 \if

This traditional TEX conditional checks if two character codes are the same. In order to understand unexpanded results it is good to know that internally TEX groups primitives in a way that serves the implementation. Each primitive has a command code and a character code, but only for real characters the name character code makes sense. This condition only really tests for character codes when we have a character, in all other cases, the result is true.

```
\def\A{A}\def\B{B} \chardef\C=`C \chardef\D=`D \def\AA{AA}

[\if AA   YES \else NOP \fi] [\if AB   YES \else NOP \fi]
[\if \A\B YES \else NOP \fi] [\if \A\A YES \else NOP \fi]
[\if \C\D YES \else NOP \fi] [\if \C\C YES \else NOP \fi]
[\if \count\dimen YES \else NOP \fi] [\if \AA\A YES \else NOP \fi]
```

The last example demonstrates that the tokens get expanded, which is why we get the extra `A`:

[YES] [NOP] [NOP] [YES] [YES] [YES] [YES] [AYES]

341 \ifabsdim

This test will negate negative dimensions before comparison, as in:

```
\def\TestA#1{\ifdim   #1<2pt too small\orelse\ifdim   #1>4pt too large\else okay\fi}
\def\TestB#1{\ifabsdim#1<2pt too small\orelse\ifabsdim#1>4pt too large\else okay\fi}
```

```
\TestA {1pt}\quad\TestA {3pt}\quad\TestA {5pt}\crlf
\TestB {1pt}\quad\TestB {3pt}\quad\TestB {5pt}\crlf
\TestB{-1pt}\quad\TestB{-3pt}\quad\TestB{-5pt}\par
```

So we get this:

too small okay too large
too small okay too large
too small okay too large

342 \ifabsfloat

This test will negate negative floats before comparison, as in:

```
\def\TestA#1{\iffloat   #1<2.46 small\orelse\iffloat   #1>4.68 large\else medium\fi}
\def\TestB#1{\ifabsfloat#1<2.46 small\orelse\ifabsfloat#1>4.68 large\else medium\fi}
```

```
\TestA {1.23}\quad\TestA {3.45}\quad\TestA {5.67}\crlf
\TestB {1.23}\quad\TestB {3.45}\quad\TestB {5.67}\crlf
\TestB{-1.23}\quad\TestB{-3.45}\quad\TestB{-5.67}\par
```

So we get this:

small medium large
small medium large
small medium large

343 \ifabsnum

This test will negate negative numbers before comparison, as in:

```
\def\TestA#1{\ifnum   #1<100 too small\orelse\ifnum   #1>200 too large\else okay\fi}
\def\TestB#1{\ifabsnum#1<100 too small\orelse\ifabsnum#1>200 too large\else okay\fi}
```

```
\TestA {10}\quad\TestA {150}\quad\TestA {210}\crlf
\TestB {10}\quad\TestB {150}\quad\TestB {210}\crlf
\TestB{-10}\quad\TestB{-150}\quad\TestB{-210}\par
```

Here we get the same result each time:

too small okay too large
too small okay too large
too small okay too large

344 \ifarguments

This is a variant of `\ifcase` were the selector is the number of arguments picked up. For example:

```
\def\MyMacro#1#2#3{\ifarguments\0\or1\or2\or3\else ?\fi} \MyMacro{A}{B}{C}
\def\MyMacro#1#0#3{\ifarguments\0\or1\or2\or3\else ?\fi} \MyMacro{A}{B}{C}
\def\MyMacro#1#-#2{\ifarguments\0\or1\or2\or3\else ?\fi} \MyMacro{A}{B}{C}\par
```

Watch the non counted, ignored, argument in the last case. Normally this test will be used in combination with `\ignorearguments`.

3 3 2

345 \ifboolean

This tests a number (register or equivalent) and any nonzero value represents `true`, which is nicer than using an **\unless\ifcase**.

346 \ifcase

This numeric TEX conditional takes a counter (literal, register, shortcut to a character, internal quantity) and goes to the branch that matches.

```
\ifcase 3 zero\or one\or two\or three\or four\else five or more\fi
```

Indeed: three equals three. In later sections we will see some LuaMetaTEX primitives that behave like an `\ifcase`.

347 \ifcat

Another traditional TEX primitive: what happens with what gets read in depends on the catcode of a character, think of characters marked to start math mode, or alphabetic characters (letters) versus other characters (like punctuation).

```
\def\A{A}\def\B{,} \chardef\C=`C \chardef\D=`, \def\AA{AA}
```

```
[\ifcat $!   YES \else NOP \fi] [\ifcat ()   YES \else NOP \fi]
[\ifcat AA   YES \else NOP \fi] [\ifcat AB   YES \else NOP \fi]
[\ifcat \A\B YES \else NOP \fi] [\ifcat \A\A YES \else NOP \fi]
[\ifcat \C\D YES \else NOP \fi] [\ifcat \C\C YES \else NOP \fi]
[\ifcat \count\dimen YES \else NOP \fi] [\ifcat \AA\A YES \else NOP \fi]
```

Close reading is needed here:

[NOP] [YES] [YES] [YES] [NOP] [YES] [YES] [YES] [YES] [AYES]

This traditional TEX condition as a well as the one in the previous section are hardly used in ConTEXt, if only because they expand what follows and we seldom need to compare characters.

348 \ifchkdim

A variant on the checker in the previous section is a dimension checker:

```
\ifchkdim oeps        \or okay\else error\fi\quad
\ifchkdim 12          \or okay\else error\fi\quad
```

```
\ifchkdim 12pt        \or okay\else error\fi\quad
\ifchkdim 12pt or more\or okay\else error\fi
```

We get:

error error okay okay

349 \ifchkdimension

COntrary to `\ifchkdim` this test doesn't accept trailing crap:

```
\ifchkdimension oeps        \or okay\else error\fi\quad
\ifchkdimension 12          \or okay\else error\fi\quad
\ifchkdimension 12pt        \or okay\else error\fi\quad
\ifchkdimension 12pt or more\or okay\else error\fi
```

reports:

error error okay error

350 \ifchknum

In ConTEXt there are quite some cases where a variable can have a number or a keyword indicating a symbolic name of a number or maybe even some special treatment. Checking if a valid number is given is possible to some extend, but a native checker makes much sense too. So here is one:

```
\ifchknum oeps        \or okay\else error\fi\quad
\ifchknum 12          \or okay\else error\fi\quad
\ifchknum 12pt        \or okay\else error\fi\quad
\ifchknum 12pt or more\or okay\else error\fi
```

The result is as expected:

error okay okay okay

351 \ifchknumber

This check is more restrictive than `\ifchknum` discussed in the previous section:

```
\ifchknumber oeps        \or okay\else error\fi\quad
\ifchknumber 12          \or okay\else error\fi\quad
\ifchknumber 12pt        \or okay\else error\fi\quad
\ifchknumber 12pt or more\or okay\else error\fi
```

Here we get:

error okay error error

352 \ifcondition

The conditionals in TEX are hard coded as primitives and although it might look like `\newif` creates one, it actually just defined three macros.

```
\newif\ifMyTest
\meaning\MyTesttrue  \crlf
\meaning\MyTestfalse \crlf
\meaning\ifMyTest    \crlf \MyTesttrue
\meaning\ifMyTest    \par
```

```
protected macro:\always \let \ifMyTest \iftrue
protected macro:\always \let \ifMyTest \iffalse
\iffalse
\iftrue
```

This means that when you say:

```
\ifMytest ... \else ... \fi
```

You actually have one of:

```
\iftrue  ... \else ... \fi
\iffalse ... \else ... \fi
```

and because these are proper conditions nesting them like:

```
\ifnum\scratchcounter > 0 \ifMyTest A\else B\fi \fi
```

will work out well too. This is not true for macros, so for instance:

```
\scratchcounter = 1
\unexpanded\def\ifMyTest{\iftrue}
\ifnum\scratchcounter > 0 \ifMyTest A\else B\fi \fi
```

will make a run fail with an error (or simply loop forever, depending on your code). This is where `\ifcondition` enters the picture:

```
\def\MyTest{\iftrue} \scratchcounter0
\ifnum\scratchcounter > 0
    \ifcondition\MyTest A\else B\fi
\else
    x
\fi
```

This primitive is seen as a proper condition when TEX is in “fast skipping unused branches” mode but when it is expanding a branch, it checks if the next expanded token is a proper tests and if so, it deals with that test, otherwise it fails. The main condition here is that the `\MyTest` macro expands to a proper true or false test, so, a definition like:

```
\def\MyTest{\ifnum\scratchcounter<10 }
```

is also okay. Now, is that neat or not?

353 \ifcramped

Depending on the given math style this returns true of false:

```
\ifcramped\mathstyle        no  \fi
```

```
\ifcramped\crampedtextstyle yes \fi
\ifcramped\textstyle        no  \fi
\ifcramped\displaystyle     yes \fi
```

gives: yes.

354 \ifcsname

This is an ε-TEX conditional that complements the one on the previous section:

```
\expandafter\ifx\csname MyMacro\endcsname\relax ... \else ... \fi
            \ifcsname   MyMacro\endcsname       ... \else ... \fi
```

Here the first one has the side effect of defining the macro and defaulting it to `\relax`, while the second one doesn't do that. Just think of checking a few million different names: the first one will deplete the hash table and probably string space too.

In LuaMetaTEX the construction stops when there is no letter or other character seen (TEX expands on the go so expandable macros are dealt with). Instead of an error message, the match is simply false and all tokens till the `\endcsname` are gobbled.

355 \ifcstok

A variant on the primitive mentioned in the previous section is one that operates on lists and macros:

```
\def\a{a} \def\b{b} \def\c{a}
```

This:

```
\ifcstok\a\b   Y\else N\fi\space
\ifcstok\a\c   Y\else N\fi\space
\ifcstok{\a}\c Y\else N\fi\space
\ifcstok{a}\c  Y\else N\fi
```

will give us: N Y Y Y.

356 \ifdefined

In traditional TEX checking for a macro to exist was a bit tricky and therefore ε-TEX introduced a convenient conditional. We can do this:

```
\ifx\MyMacro\undefined ... \else ... \fi
```

but that assumes that **\undefined** is indeed undefined. Another test often seen was this:

```
\expandafter\ifx\csname MyMacro\endcsname\relax ... \else ... \fi
```

Instead of comparing with **\undefined** we need to check with `\relax` because the control sequence is defined when not yet present and defaults to `\relax`. This is not pretty.

357 \ifdim

Dimensions can be compared with this traditional TEX primitive.

```
\scratchdimen=1pt \scratchcounter=65536

\ifdim\scratchdimen=\scratchcounter sp YES \else NOP\fi
\ifdim\scratchdimen=1               pt YES \else NOP\fi
```

The units are mandate:

YES YES

358 \ifdimexpression

The companion of the previous primitive is:

This matches when the result is non zero, and you can mix calculations and tests as with normal expressions. Contrary to the number variant units can be used and precision kicks in.

359 \ifdimval

This conditional is a variant on \ifchkdim and provides some more detailed information about the value:

```
[-12pt : \ifdimval-12pt\or negative\or zero\or positive\else error\fi]\quad
[0pt   : \ifdimval  0pt\or negative\or zero\or positive\else error\fi]\quad
[12pt  : \ifdimval 12pt\or negative\or zero\or positive\else error\fi]\quad
[oeps  : \ifdimval oeps\or negative\or zero\or positive\else error\fi]
```

This gives:

[-12pt : negative]   [0pt : zero]   [12pt : positive]   [oeps : error]

360 \ifempty

This conditional checks if a control sequence is empty:

```
is \ifempty\MyMacro \else not \fi empty
```

It is basically a shortcut of:

```
is \ifx\MyMacro\empty \else not \fi empty
```

with:

```
\def\empty{}
```

Of course this is not empty at all:

```
\def\notempty#1{}
```

361 \iffalse

Here we have a traditional TEX conditional that is always false (therefore the same is true for any macro that is \let to this primitive).

362 \ifflags

This test primitive relates to the various flags that one can set on a control sequence in the perspective of overload protection and classification.

```
\protected\untraced\tolerant\def\foo[#1]{...#1...}
\permanent\constant         \def\oof{okay}
```

flag	\foo	\oof	flag	\foo	\oof
frozen	N	N	permanent	N	Y
immutable	N	N	mutable	N	N
noaligned	N	N	instance	N	N
untraced	Y	N	global	N	N
tolerant	Y	N	constant	N	Y
protected	Y	N	semiprotected	N	N

Instead of checking against a prefix you can test against a bitset made from:

0x1	frozen	0x2	permanent	0x4	immutable	0x8	primitive
0x10	mutable	0x20	noaligned	0x40	instance	0x80	untraced
0x100	global	0x200	tolerant	0x400	protected	0x800	overloaded
0x1000	aliased	0x2000	immediate	0x4000	conditional	0x8000	value
0x10000	semiprotected	0x20000	inherited	0x40000	constant	0x80000	deferred

363 \iffloat

This test does for floats what `\ifnum`, `\ifdim` do for numbers and dimensions: comparing two of them.

364 \iffontchar

This is an ε-TEX conditional. It takes a font identifier and a character number. In modern fonts simply checking could not be enough because complex font features can swap in other ones and their index can be anything. Also, a font mechanism can provide fallback fonts and characters, so don't rely on this one too much. It just reports true when the font passed to the frontend has a slot filled.

365 \ifhaschar

This one is a simplified variant of the above:

```
\ifhaschar !{this ! works} yes \else no \fi
```

and indeed we get: yes! Of course the spaces in this this example code are normally not present in such a test.

366 \ifhastok

This conditional looks for occurrences in token lists where each argument has to be a proper list.

```
\def\scratchtoks{x}
```

```
\ifhastoks{yz}         {xyz} Y\else N\fi\quad
\ifhastoks\scratchtoks {xyz} Y\else N\fi
```

We get:

Y Y

367 \ifhastoks

This test compares two token lists. When a macro is passed it's meaning gets used.

```
\def\x  {x}
\def\xyz{xyz}
```

```
(\ifhastoks  {x}  {xyz}Y\else N\fi)\quad
(\ifhastoks {\x}  {xyz}Y\else N\fi)\quad
(\ifhastoks  \x   {xyz}Y\else N\fi)\quad
(\ifhastoks  {y}  {xyz}Y\else N\fi)\quad
(\ifhastoks {yz}  {xyz}Y\else N\fi)\quad
(\ifhastoks {yz} {\xyz}Y\else N\fi)
```

(Y) (N) (Y) (Y) (Y) (N)

368 \ifhasxtoks

This primitive is like the one in the previous section but this time the given lists are expanded.

```
\def\x  {x}
\def\xyz{\x yz}
```

```
(\ifhasxtoks  {x}  {xyz}Y\else N\fi)\quad
(\ifhasxtoks {\x}  {xyz}Y\else N\fi)\quad
(\ifhastoks   \x   {xyz}Y\else N\fi)\quad
(\ifhasxtoks  {y}  {xyz}Y\else N\fi)\quad
(\ifhasxtoks {yz}  {xyz}Y\else N\fi)\quad
(\ifhasxtoks {yz} {\xyz}Y\else N\fi)
```

(Y) (Y) (Y) (Y) (Y) (Y)

This primitive has some special properties.

```
\edef\+{\expandtoken 9 `+}
```

```
\ifhasxtoks {xy}   {xyz}Y\else N\fi\quad
\ifhasxtoks {x\+y} {xyz}Y\else N\fi
```

Here the first argument has a token that has category code ‘ignore’ which means that such a character will be skipped when seen. So the result is:

Y Y

This permits checks like these:

```
\edef\,{\expandtoken 9 `,}

\ifhasxtoks{\,x\,} {,x,y,z,}Y\else N\fi\quad
\ifhasxtoks{\,y\,} {,x,y,z,}Y\else N\fi\quad
\ifhasxtoks{\,z\,} {,x,y,z,}Y\else N\fi\quad
\ifhasxtoks{\,x\,}  {,xy,z,}Y\else N\fi
```

I admit that it needs a bit of a twisted mind to come up with this, but it works ok:

Y Y Y N

369 `\ifhbox`

This traditional conditional checks if a given box register or internal box variable represents a horizontal box,

370 `\ifhmode`

This traditional conditional checks we are in (restricted) horizontal mode.

371 `\ifinalignment`

As the name indicates, this primitive tests for being in an alignment. Roughly spoken, the engine is either in a state of align, handling text or dealing with math.

372 `\ifincsname`

This conditional is sort of obsolete and can be used to check if we're inside a `\csname` or `\ifcsname` construction. It's not used in ConTEXt.

373 `\ifinner`

This traditional one can be confusing. It is true when we are in restricted horizontal mode (a box), internal vertical mode (a box), or inline math mode.

```
test \ifhmode \ifinner INNER\fi HMODE\fi\crlf
\hbox{test \ifhmode \ifinner INNER \fi HMODE\fi} \par

\ifvmode \ifinner INNER\fi VMODE \fi\crlf
\vbox{\ifvmode \ifinner INNER \fi VMODE\fi} \crlf
\vbox{\ifinner INNER \ifvmode VMODE \fi \fi} \par
```

Watch the last line: because we typeset `INNER` we enter horizontal mode:

test HMODE
test INNER HMODE

VMODE
INNER VMODE
INNER

374 \ifinsert

This is the equivalent of \ifvoid for a given insert class.

375 \ifintervaldim

This conditional is true when the intervals around the values of two dimensions overlap. The first dimension determines the interval.

```
[\ifintervaldim1pt 20pt 21pt \else no \fi overlap]
[\ifintervaldim1pt 18pt 20pt \else no \fi overlap]
```

So here: [overlap] [no overlap]

376 \ifintervalfloat

This one does with floats what we described under \ifintervaldim.

377 \ifintervalnum

This one does with integers what we described under \ifintervaldim.

378 \iflastnamedcs

When a \csname is constructed and succeeds the last one is remembered and can be accessed with \lastnamedcs. It can however be an undefined one. That state can be checked with this primitive. Of course it also works with the \ifcsname and \begincsname variants.

379 \ifmathparameter

This is an \ifcase where the value depends on if the given math parameter is zero, (0), set (1), or unset (2).

```
\ifmathparameter\Umathpunctclosespacing\displaystyle
    zero    \or
    nonzero \or
    unset   \fi
```

380 \ifmathstyle

This is a variant of \ifcase were the number is one of the seven possible styles: display, text, cramped text, script, cramped script, script script, cramped script script.

```
\ifmathstyle
  display
\or
  text
\or
  cramped text
\else
```

```
  normally smaller than text
\fi
```

381 \ifmmode

This traditional conditional checks we are in (inline or display) math mode mode.

382 \ifnum

This is a frequently used conditional: it compares two numbers where a number is anything that can be seen as such.

```
\scratchcounter=65 \chardef\A=65
```

```
\ifnum65=`A              YES \else NOP\fi
\ifnum\scratchcounter=65 YES \else NOP\fi
\ifnum\scratchcounter=\A YES \else NOP\fi
```

Unless a number is an unexpandable token it ends with a space or `\relax`, so when you end up in the true branch, you'd better check if TEX could determine where the number ends.

YES YES YES

On top of these ascii combinations, the engine also accepts some Unicode characters. This brings the full repertoire to:

character			operation
0x003C	$<$		less
0x003D	$=$		equal
0x003E	$>$		more
0x2208	$\in$		element of
0x2209	$\notin$		not element of
0x2260	$\neq$	!=	not equal
0x2264	$\leq$	!>	less equal
0x2265	$\geq$	!<	greater equal
0x2270	$\nleq$		not less equal
0x2271	$\ngeq$		not greater equal

This also applied to `\ifdim` although in the case of element we discard the fractional part (read: divide the numeric representation by 65536).

383 \ifnumexpression

Here is an example of a conditional using expressions:

This matches when the result is non zero, and you can mix calculations and tests as with normal expressions.

384 \ifnumval

This conditional is a variant on `\ifchknum`. This time we get some more detail about the value:

```
[-12  : \ifnumval  -12\or negative\or zero\or positive\else error\fi]\quad
[0    : \ifnumval    0\or negative\or zero\or positive\else error\fi]\quad
[12   : \ifnumval   12\or negative\or zero\or positive\else error\fi]\quad
[oeps : \ifnumval oeps\or negative\or zero\or positive\else error\fi]
```

This gives:

[-12 : negative] [0 : zero] [12 : positive] [oeps : error]

385 \ifodd

One reason for this condition to be around is that in a double sided layout we need test for being on an odd or even page. It scans for a number the same was as other primitives,

```
\ifodd65 YES \else NO\fi &
\ifodd`B YES \else NO\fi .
```

So: YES & NO.

386 \ifparameter

In a macro body `#1` is a reference to a parameter. You can check if one is set using a dedicated parameter condition:

```
\tolerant\def\foo[#1]#*[#2]%
  {\ifparameter#1\or one\else no one\fi\enspace
   \ifparameter#2\or two\else no two\fi\emspace}
```

```
\foo
\foo[1]
\foo[1][2]
```

We get:

no one no two   one no two   one two

387 \ifparameters

This is equivalent to an `\ifcase` with as value the number of parameters passed to the current macro.

388 \ifrelax

This is a convenient shortcut for `\ifx\relax` and the motivation for adding this one is (as with some others) to get less tracing.

389 \iftok

When you want to compare two arguments, the usual way to do this is the following:

```
\edef\tempA{#1}
\edef\tempb{#2}
```

```
\ifx\tempA\tempB
    the same
\else
    different
\fi
```

This works quite well but the fact that we need to define two macros can be considered a bit of a nuisance. It also makes macros that use this method to be not so called ‘fully expandable’. The next one avoids both issues:

```
\iftok{#1}{#2}
    the same
\else
    different
\fi
```

Instead of direct list you can also pass registers, so given:

```
\scratchtoks{a}%
\toks0{a}%
```

This:

```
\iftok 0 \scratchtoks          Y\else N\fi\space
\iftok{a}\scratchtoks          Y\else N\fi\space
\iftok\scratchtoks\scratchtoks Y\else N\fi
```

gives: Y Y Y.

390 `\iftrue`

Here we have a traditional TEX conditional that is always true (therefore the same is true for any macro that is `\let` to this primitive).

391 `\ifvbox`

This traditional conditional checks if a given box register or internal box variable represents a vertical box,

392 `\ifvmode`

This traditional conditional checks we are in (internal) vertical mode.

393 `\ifvoid`

This traditional conditional checks if a given box register or internal box variable has any content.

394 `\ifx`

We use this traditional TEX conditional a lot in ConTEXt. Contrary to `\if` the two tokens that are compared are not expanded. This makes it possible to compare the meaning of two macros. Depending on the need, these macros can have their content expanded or not. A different number of parameters results in false.

Control sequences are identical when they have the same command code and character code. Because a `\let` macro is just a reference, both let macros are the same and equal to `\relax`:

```
\let\one\relax \let\two\relax
```

The same is true for other definitions that result in the same (primitive) or meaning encoded in the character field (think of `\chardef`s and so).

395 \ifzerodim

This tests for a dimen (dimension) being zero so we have:

```
\ifdim<dimension>=0pt
\ifzerodim<dimension>
\ifcase<dimension register>
```

396 \ifzerofloat

As the name indicated, this tests for a zero float value.

```
[\scratchfloat\zerofloat \ifzerofloat\scratchfloat \else not \fi zero]
[\scratchfloat\plusone   \ifzerofloat\scratchfloat \else not \fi zero]
[\scratchfloat 0.01      \ifzerofloat\scratchfloat \else not \fi zero]
[\scratchfloat 0.0e0     \ifzerofloat\scratchfloat \else not \fi zero]
[\scratchfloat \zeropoint\ifzerofloat\scratchfloat \else not \fi zero]
```

So: [zero] [not zero] [not zero] [zero] [zero]

397 \ifzeronum

This tests for a number (integer) being zero so we have these variants now:

```
\ifnum<integer or equivalent>=0
\ifzeronum<integer or equivalent>
\ifcase<integer or equivalent>
```

398 \ignorearguments

This primitive will quit argument scanning and start expansion of the body of a macro. The number of grabbed arguments can be tested as follows:

```
\def\MyMacro[#1][#2][#3]%
 {\ifarguments zero\or one\or two\or three \else hm\fi}

\MyMacro          \ignorearguments \quad
\MyMacro       [1]\ignorearguments \quad
\MyMacro    [1][2]\ignorearguments \quad
\MyMacro [1][2][3]\ignorearguments \par
```

zero one two three

Todo: explain optional delimiters.

399 \ignoredepthcriterion

When setting the `\prevdepth` (either by TEX or by the current user) of the current vertical list the value 1000pt is a signal for special treatment of the skip between 'lines'. There is an article on that in the distribution. It also demonstrates that `\ignoredepthcriterion` can be used to change this special signal, just in case it is needed.

400 \ignorenestedupto

This primitive gobbles following tokens and can deal with nested 'environments', for example:

```
\def\startfoo{\ignorenestedupto\startfoo\stopfoo}
```

```
(before
\startfoo
    test \startfoo test \stopfoo
   {test \startfoo test \stopfoo}
\stopfoo
after)
```

delivers:

(before after)

401 \ignorepars

This is a variant of `\ignorespaces`: following spaces *and* `\par` equivalent tokens are ignored, so for instance:

```
one + \ignorepars

two = \ignorepars \par
three
```

renders as: one + two = three. Traditionally TEX has been sensitive to `\par` tokens in some of its building blocks. This has to do with the fact that it could indicate a runaway argument which in the times of slower machines and terminals was best to catch early. In LuaMetaTEX we no longer have long macros and the mechanisms that are sensitive can be told to accept `\par` tokens (and ConTEXt set them such that this is the case).

402 \ignorerest

An example shows what this primitive does:

```
\tolerant\def\foo[#1]#*[#2]%
  {1234
   \ifparameter#1\or\else
```

```
    \expandafter\ignorerest
  \fi
  /#1/
  \ifparameter#2\or\else
    \expandafter\ignorerest
  \fi
  /#2/ }
```

```
\foo test \foo[456] test \foo[456][789] test
```

As this likely makes most sense in conditionals you need to make sure the current state is properly finished. Because `\expandafter` bumps the input state, here we actually quit two levels; this is because so called 'backed up text' is intercepted by this primitive.

1234 test 1234 /456/ test 1234 /456/ /789/ test

403 \ignorespaces

This traditional TEX primitive signals the scanner to ignore the following spaces, if any. We mention it because we show a companion in the next section.

404 \ignoreupto

This ignores everything upto the given token, so

```
\ignoreupto \foo not this but\foo only this
```

will give: only this.

405 \immediate

This one has no effect unless you intercept it at the Lua end and act upon it. In original TEX immediate is used in combination with read from and write to file operations. So, this is an old primitive with a new meaning.

406 \immutable

This prefix flags what follows as being frozen and is usually applied to for instance `\integerdef`'d control sequences. In that respect is is like `\permanent` but it makes it possible to distinguish quantities from macros.

407 \indent

In engines other than LuaMetaTEX a paragraph starts with an indentation box. The width of that (empty) box is determined by `\parindent`. In LuaMetaTEX we can use a dedicated indentation skip instead (as part of paragraph normalization). An indentation can be zero'd with `\undent`.

408 \indexedsubprescript

This primitive (or ____) puts a flag on the script but renders the same:

```
$
    x \indexedsuperprescript{2} \subprescript      {2} +
    x \superprescript       {2} \indexedsubprescript{2} +
    x \superprescript       {2} ____               {2} =
    x \superprescript       {2} \subprescript      {2}
$
```

Gives: ${}^2_2x + {}^2_2x + {}^2_2x = {}^2_2x$.

409 \indexedsubscript

This primitive (or __) puts a flag on the script but renders the same:

```
$
    x \indexedsuperscript{2} \subscript      {2} +
    x \superscript       {2} \indexedsubscript{2} +
    x \superscript       {2} __              {2} =
    x \superscript       {2} \subscript      {2}
$
```

Gives: $x^2_2 + x^2_2 + x^2_2 = x^2_2$.

410 \indexedsuperprescript

This primitive (or ^^^^) puts a flag on the script but renders the same:

```
$
    x \indexedsuperprescript{2} \subprescript      {2} +
    x ^^^^                  {2} \subprescript      {2} +
    x \superprescript       {2} \indexedsubprescript{2} =
    x \superprescript       {2} \subprescript      {2}
$
```

Gives: ${}^2_2x + {}^2_2x + {}^2_2x = {}^2_2x$.

411 \indexedsuperscript

This primitive (or ^^) puts a flag on the script but renders the same:

```
$
    x \indexedsuperscript{2} \subscript      {2} +
    x ^^                 {2} \subscript      {2} +
    x \superscript       {2} \indexedsubscript{2} =
    x \superscript       {2} \subscript      {2}
$
```

Gives: $x^2_2 + x^2_2 + x^2_2 = x^2_2$.

412 \indexofcharacter

This primitive is more versatile variant of the backward quote operator, so instead of:

```
\number`|
\number`~
\number`\a
\number`\q
```

you can say:

```
\the\indexofcharacter |
\the\indexofcharacter ~
\the\indexofcharacter \a
\the\indexofcharacter \q
```

In both cases active characters and unknown single character control sequences are valid. In addition this also works:

```
\chardef    \foo 128
\mathchardef\oof 130

\the\indexofcharacter \foo
\the\indexofcharacter \oof
```

An important difference is that `\indexofcharacter` returns an integer and not a serialized number. A negative value indicates no valid character.

413 \indexofregister

You can use this instead of `\number` for determining the index of a register but it also returns a number when a register value is seen. The result is an integer, not a serialized number.

When you have defined a register with one of the `\...def` primitives but for some reasons needs to know the register index you can query that:

```
\the\indexofregister \scratchcounterone,
\the\indexofregister \scratchcountertwo,
\the\indexofregister \scratchwidth,
\the\indexofregister \scratchheight,
\the\indexofregister \scratchdepth,
\the\indexofregister \scratchbox
```

We lie a little here because in ConTEXt the box index `\scratchbox` is actually defined as: `\permanent\constant integer 257` but it still is a number so it fits in.

0, 0, 0, 0, 0, 257

414 \inherited

When this prefix is used in a definition using `\let` the target will inherit all the properties of the source.

415 \initcatcodetable

This initializes the catcode table with the given index.

416 \initialpageskip

When a page starts the value of this register are used to initialize `\pagetotal`, `\pagestretch` and `\pageshrink`. This make nicer code than using a `\topskip` with weird values.

417 \initialtopskip

When set this one will be used instead of `\topskip`. The rationale is that the `\topskip` is often also used for side effects and compensation.

418 \input

There are several ways to use this primitive:

```
\input  test
\input {test}
\input "test"
\input 'test'
```

When no suffix is given, TEX will assume the suffix is `.tex`. The second one is normally used.

419 \inputlineno

This integer holds the current linenumber but it is not always reliable.

420 \insert

This stores content in the insert container with the given index. In LuaMetaTEX inserts bubble up to outer boxes so we don't have the 'deeply buried insert issue'.

421 \insertbox

This is the accessor for the box (with results) of an insert with the given index. This is equivalent to the `\box` in the traditional method.

422 \insertcopy

This is the accessor for the box (with results) of an insert with the given index. It makes a copy so the original is kept. This is equivalent to a `\copy` in the traditional method.

423 \insertdepth

This is the (current) depth of the inserted material with the given index. It is comparable to the `\dp` in the traditional method.

424 \insertdistance

This is the space before the inserted material with the given index. This is equivalent to `\glue` in the traditional method.

425 \insertheight

This is the (current) depth of the inserted material with the given index. It is comparable to the \ht in the traditional method.

426 \insertheights

This is the combined height of the inserted material.

427 \insertlimit

This is the maximum height that the inserted material with the given index can get. This is equivalent to \dimen in the traditional method.

428 \insertmaxdepth

This is the maximum depth that the inserted material with the given index can get.

429 \insertmode

In traditional TEX inserts are controlled by a \box, \dimen, \glue and \count register with the same index. The allocators have to take this into account. When this primitive is set to one a different model is followed with its own namespace. There are more abstract accessors to interface to this.[7]

430 \insertmultiplier

This is the height (contribution) multiplier for the inserted material with the given index. This is equivalent to \count in the traditional method.

431 \insertpenalties

This dual purpose internal counter holds the sum of penalties for insertions that got split. When we're the output routine in reports the number of insertions that is kept in store.

432 \insertpenalty

This is the insert penalty associated with the inserted material with the given index.

433 \insertprogress

This returns the current accumulated insert height of the insert with the given index.

434 \insertstorage

The value passed will enable (one) or disable (zero) the insert with the given index.

---

[7] The old model might be removed at some point.

435 \insertstoring

The value passed will enable (one) or disable (zero) inserts.

436 \insertunbox

This is the accessor for the box (with results) of an insert with the given index. It makes a copy so the original is kept. The content is unpacked and injected. This is equivalent to an `\unvbox` in the traditional method.

437 \insertuncopy

This is the accessor for the box (with results) of an insert with the given index. It makes a copy so the original is kept. The content is unpacked and injected. This is equivalent to the `\unvcopy` in the traditional method.

438 \insertwidth

This is the (current) width of the inserted material with the given index. It is comparable to the `\wd` in the traditional method.

439 \instance

This prefix flags a macro as an instance which is mostly relevant when a macro package want to categorize macros.

440 \integerdef

You can alias to a count (integer) register with `\countdef`:

```
\countdef\MyCount134
```

Afterwards the next two are equivalent:

```
\MyCount    = 99
\count1234 = 99
```

where `\MyCount` can be a bit more efficient because no index needs to be scanned. However, in terms of storage the value (here 99) is always in the register so `\MyCount` has to get there. This indirectness has the benefit that directly setting the value is reflected in the indirect accessor.

```
\integerdef\MyCount = 99
```

This primitive also defines a numeric equivalent but this time the number is stored with the equivalent. This means that:

```
\let\MyCopyOfCount = \MyCount
```

will store the *current* value of `\MyCount` in `\MyCopyOfCount` and changing either of them is not reflected in the other.

The usual `\advance`, `\multiply` and `\divide` can be used with these integers and they behave like any number. But compared to registers they are actually more a constant.

441 \interactionmode

This internal integer can be used to set or query the current interaction mode:

\batchmode	0	omits all stops and terminal output
\nonstopmode	1	omits all stops
\scrollmode	2	omits error stops
\errorstopmode	3	stops at every opportunity to interact

442 \interlinepenalties

This is a more granular variant of `\interlinepenalty`: an array of penalties to be put between successive line from the start of a paragraph. The list starts with the number of penalties that gets passed.

443 \interlinepenalty

This is the penalty that is put between lines.

444 \jobname

This gives the current job name without suffix: `luametatex`.

445 \kern

A kern is injected with the given dimension. For variants that switch to a mode we have `\hkern` and `\vkern`.

446 \language

Sets (or returns) the current language, a number. In LuaTEX and LuaMetaTEX the current language is stored in the glyph nodes.

447 \lastarguments

```
\def\MyMacro    #1{\the\lastarguments (#1) }         \MyMacro{1}       \crlf
\def\MyMacro  #1#2{\the\lastarguments (#1) (#2)}     \MyMacro{1}{2}    \crlf
\def\MyMacro#1#2#3{\the\lastarguments (#1) (#2) (#3)} \MyMacro{1}{2}{3} \par

\def\MyMacro    #1{(#1)           \the\lastarguments} \MyMacro{1}       \crlf
\def\MyMacro  #1#2{(#1) (#2)      \the\lastarguments} \MyMacro{1}{2}    \crlf
\def\MyMacro#1#2#3{(#1) (#2) (#3) \the\lastarguments} \MyMacro{1}{2}{3} \par
```

The value of `\lastarguments` can only be trusted in the expansion until another macro is seen and expanded. For instance in these examples, as soon as a character (like the left parenthesis) is seen, horizontal mode is entered and `\everypar` is expanded which in turn can involve macros. You can see that in the second block (that is: unless we changed `\everypar` in the meantime).

1(1)
2(1) (2)
3(1) (2) (3)

(1) 0
(1) (2) 2
(1) (2) (3) 3

448 \lastatomclass

This returns the class number of the last atom seen in the math input parser.

449 \lastboundary

This primitive looks back in the list for a user boundary injected with `\boundary` and when seen it returns that value or otherwise zero.

450 \lastbox

When issued this primitive will, if possible, pull the last box from the current list.

451 \lastchkdimension

When the last check for a dimension with `\ifchkdimension` was successful this primitive returns the value.

452 \lastchknumber

When the last check for an integer with `\ifchknumber` was successful this primitive returns the value.

453 \lastkern

This returns the last kern seen in the list (if possible).

454 \lastleftclass

This variable registers the first applied math class in a formula.

455 \lastlinefit

The ε-TEX manuals explains this parameter in detail but in practice it is enough to know that when set to 1000 spaces in the last line might match those in the previous line. Basically it counters the strong push of a `\parfillskip`.

456 \lastloopiterator

In addition to `\currentloopiterator` we have a variant that stores the value in case an unexpanded loop is used:

```
\localcontrolledrepeat 8 { [\the\currentloopiterator\eq\the\lastloopiterator] }
\expandedrepeat        8 { [\the\currentloopiterator\eq\the\lastloopiterator] }
\unexpandedrepeat      8 { [\the\currentloopiterator\ne\the\lastloopiterator] }
```

[1=1] [2=2] [3=3] [4=4] [5=5] [6=6] [7=7] [8=8]
[1=1] [2=2] [3=3] [4=4] [5=5] [6=6] [7=7] [8=8]
[0≠1] [0≠2] [0≠3] [0≠4] [0≠5] [0≠6] [0≠7] [0≠8]

457 \lastnamedcs

The example code in the previous section has some redundancy, in the sense that there to be looked up control sequence name `mymacro` is assembled twice. This is no big deal in a traditional eight bit TEX but in a Unicode engine multi-byte sequences demand some more processing (although it is unlikely that control sequences have many multi-byte utf8 characters).

```
\ifcsname mymacro\endcsname
    \csname mymacro\endcsname
\fi
```

Instead we can say:

```
\ifcsname mymacro\endcsname
    \lastnamedcs
\fi
```

Although there can be some performance benefits another advantage is that it uses less tokens and parsing. It might even look nicer.

458 \lastnodesubtype

When possible this returns the subtype of the last node in the current node list. Possible values can be queried (for each node type) via Lua helpers.

459 \lastnodetype

When possible this returns the type of the last node in the current node list. Possible values can be queried via Lua helpers.

460 \lastpageextra

This reports the last applied (permitted) overshoot.

461 \lastparcontext

When a paragraph is wrapped up the reason is reported by this state variable. Possible values are:

`0x00`	normal	`0x04`	dbox	`0x08`	output	`0x0C`	math
`0x01`	vmode	`0x05`	vcenter	`0x09`	align	`0x0D`	lua
`0x02`	vbox	`0x06`	vadjust	`0x0A`	noalign	`0x0E`	reset
`0x03`	vtop	`0x07`	insert	`0x0B`	span		

462 `\lastpartrigger`

There are several reasons for entering a paragraphs and some are automatic and triggered by other commands that force TEX into horizontal mode.

`0x00`	normal	`0x04`	mathchar	`0x08`	math	`0x0C`	valign
`0x01`	force	`0x05`	char	`0x09`	kern	`0x0D`	vrule
`0x02`	indent	`0x06`	boundary	`0x0A`	hskip		
`0x03`	noindent	`0x07`	space	`0x0B`	unhbox		

463 `\lastpenalty`

This returns the last penalty seen in the list (if possible).

464 `\lastrightclass`

This variable registers the last applied math class in a formula.

465 `\lastskip`

This returns the last glue seen in the list (if possible).

466 `\lccode`

When the `\lowercase` operation is applied the lowercase code of a character is used for the replacement. This primitive is used to set that code, so it expects two character number. The code is also used to determine what characters make a word suitable for hyphenation, although in LuaTEX we introduced the `\hj` code for that.

467 `\leaders`

See `\gleaders` for an explanation.

468 `\left`

Inserts the given delimiter as left fence in a math formula.

469 `\lefthyphenmin`

This is the minimum number of characters after the last hyphen in a hyphenated word.

470 \leftmarginkern

The dimension returned is the protrusion kern that has been added (if at all) to the left of the content in the given box.

471 \leftskip

This skip will be inserted at the left of every line.

472 \lefttwindemerits

Additional demerits for a glyph sequence at the left edge when a previous line also has that sequence.

473 \leqno

This primitive stores the (typeset) content (presumably a number) and when the display formula is wrapped that number will end up left of the formula.

474 \let

Where a `\def` creates a new macro, either or not with argument, a `\let` creates an alias. You are not limited to aliasing macros, basically everything can be aliased.

475 \letcharcode

Assigning a meaning to an active character can sometimes be a bit cumbersome; think of using some documented uppercase magic that one tends to forget as it's used only a few times and then never looked at again. So we have this:

```
{\letcharcode 65 1 \catcode 65 13 A : \meaning A}\crlf
{\letcharcode 65 2 \catcode 65 13 A : \meaning A}\par
```

here we define `A` as an active charcter with meaning `1` in the first line and `2` in the second.

```
1 : the character U+0031 1
2 : the character U+0032 2
```

Normally one will assign a control sequence:

```
{\letcharcode 66 \bf \catcode 66 13 {B   bold}: \meaning B}\crlf
{\letcharcode 73 \it \catcode 73 13 {I italic}: \meaning I}\par
```

Of course `\bf` and `\it` are ConTEXt specific commands:

bold: protected macro:\ifmmode \expandafter \mathbf \else \expandafter \normalbf \fi
italic: protected macro:\ifmmode \expandafter \mathit \else \expandafter \normalit \fi

476 \letcsname

It is easy to see that we save two tokens when we use this primitive. As with the `..defcs..` variants it also saves a push back of the composed macro name.

```
\expandafter\let\csname MyMacro:1\endcsname\relax
            \letcsname MyMacro:1\endcsname\relax
```

477 \letfrozen

You can explicitly freeze an unfrozen macro:

```
\def\MyMacro{...}
\letfrozen\MyMacro
```

A redefinition will now give:

```
! You can't redefine a frozen macro.
```

478 \letmathatomrule

You can change the class for a specific style. This probably only makes sense for user classes. It's one of those features that we used when experimenting with more control.

```
\letmathatomrule 4 = 4 4 0 0
\letmathatomrule 5 = 5 5 0 0
```

This changes the classes 4 and 5 into class 0 in the two script styles and keeps them the same in display and text. We leave it to the reader to ponder how useful this is.

479 \letmathparent

This primitive takes five arguments: the target class, and four classes that determine the pre penalty class, post penalty class, options class and a dummy class for future use.

480 \letmathspacing

By default inter-class spacing inherits from the ordinary class but you can remap specific combinations is you want:

```
\letmathspacing \mathfunctioncode
    \mathordinarycode \mathordinarycode
    \mathordinarycode \mathordinarycode
```

The first value is the target class, and the nest four tell how it behaves in display, text, script and script script style. Here `\mathfunctioncode` is a ConTEXt specific class (26), one of the many.

481 \letprotected

Say that you have these definitions:

```
             \def  \MyMacroA{alpha}
\protected   \def  \MyMacroB{beta}
             \edef \MyMacroC{\MyMacroA\MyMacroB}
\letprotected      \MyMacroA
             \edef \MyMacroD{\MyMacroA\MyMacroB}
```

```
\meaning          \MyMacroC\crlf
\meaning          \MyMacroD\par
```

The typeset meaning in this example is:

```
macro:alpha\MyMacroB
macro:\MyMacroA \MyMacroB
```

482 \lettolastnamedcs

The `\lastnamedcs` primitive is somewhat special as it is a (possible) reference to a control sequence which is why we have a dedicated variant of `\let`.

```
\csname relax\endcsname\let                         \foo\lastnamedcs \meaning\foo
\csname relax\endcsname\expandafter\let\expandafter \oof\lastnamedcs \meaning\oof
\csname relax\endcsname\lettolastnamedcs            \ofo             \meaning\ofo
```

These give the following where the first one obviously is not doing what we want and the second one is kind of cumbersome.

\lastnamedcs
\relax
\relax

483 \lettonothing

This one let's a control sequence to nothing. Assuming that `\empty` is indeed empty, these two lines are equivalent.

```
\let         \foo\empty
\lettonothing\oof
```

484 \limits

This is a modifier: it flags the previous math atom to have its scripts above and below the (summation, product, integral etc.) symbol. In LuaMetaTEX this can be any atom (that is: any class). In display mode the location defaults to above and below.

Like any modifier it looks back for a math specific element. This means that the following will work well:

```
\sum \limits ^2 _3
\sum ^2 \limits _3
\sum ^2 _3 \limits
\sum ^2 _3 \limits \nolimits \limits
```

because scripts are bound to these elements so looking back just sees the element.

485 \linebreakchecks

The value of this parameter is passed to the linebreak callback so that one can act on it if needed.

486 \linebreakoptional

This selects the optional text range that is to be used. Optional content is marked with optionalboundary nodes.

487 \linebreakpasses

When set to a positive value it will apply additional line break runs defined with `\parpasses` until the criteria set in there are met.

488 \linedirection

This sets the text direction (1 for `r2l`) to the given value but keeps preceding glue into the range.

489 \linepenalty

Every line gets this penalty attached, so normally it is a small value, like here: 10.

490 \lineskip

This is the amount of glue that gets added when the distance between lines falls below `\lineskiplimit`.

491 \lineskiplimit

When the distance between two lines becomes less than `\lineskiplimit` a `\lineskip` glue item is added.

```
\ruledvbox{
    \lineskiplimit 0pt \lineskip3pt \baselineskip0pt
    \ruledhbox{line 1}
    \ruledhbox{line 2}
    \ruledhbox{\tx line 3}
}
```

Normally the `\baselineskip` kicks in first but here we've set that to zero, so we get two times a 3pt glue injected.

line 1
line 2
line 3

492 \localbreakpar

This forces a newline in a paragraph without side effects so that for instance `\widowpenalties` work as expected in scenarios where using a \

would have been the solution. This is an experimental primitive!

493 \localbrokenpenalty

TODO

494 \localcontrol

This primitive takes a single token:

```
\edef\testa{\scratchcounter123 \the\scratchcounter}
\edef\testc{\testa \the\scratchcounter}
\edef\testd{\localcontrol\testa \the\scratchcounter}
```

The three meanings are:

123

```
\testa  macro:\scratchcounter 123 123
\testc  macro:\scratchcounter 123 123123
\testd  macro:123
```

The `\localcontrol` makes that the following token gets expanded so we don't see the yet to be expanded assignment show up in the macro body.

495 \localcontrolled

The previously described local control feature comes with two extra helpers. The `\localcontrolled` primitive takes a token list and wraps this into a local control sidetrack. For example:

```
\edef\testa{\scratchcounter123 \the\scratchcounter}
\edef\testb{\localcontrolled{\scratchcounter123}\the\scratchcounter}
```

The two meanings are:

```
\testa  macro:\scratchcounter 123 123
\testb  macro:123
```

The assignment is applied immediately in the expanded definition.

496 \localcontrolledendless

As the name indicates this will loop forever. You need to explicitly quit the loop with `\quitloop` or `\quitloopnow`. The first quitter aborts the loop at the start of a next iteration, the second one tries to exit immediately, but is sensitive for interference with for instance nested conditionals. Of course in the next case one can just adapt the final iterator value instead. Here we step by 2:

```
\expandedloop 1 20 2 {%
    \ifnum\currentloopiterator>10
        \quitloop
    \else
        [!]
    \fi
}
```

This results in:

[!] [!] [!] [!] [!]

497 \localcontrolledloop

As with more of the primitives discussed here, there is a manual in the ‘lowlevel’ subset that goes into more detail. So, here a simple example has to do:

```
\localcontrolledloop 1 100 1 {%
    \ifnum\currentloopiterator>6\relax
        \quitloop
    \else
        [\number\currentloopnesting:\number\currentloopiterator]
        \localcontrolledloop 1 8 1 {%
            (\number\currentloopnesting:\number\currentloopiterator)
        }\par
    \fi
}
```

Here we see the main loop primitive being used nested. The code shows how we can `\quitloop` and have access to the `\currentloopiterator` as well as the nesting depth `\currentloopnesting`.

[1:1] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:2] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:3] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:4] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:5] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)
[1:6] (2:1) (2:2) (2:3) (2:4) (2:5) (2:6) (2:7) (2:8)

Be aware of the fact that `\quitloop` will end the loop at the *next* iteration so any content after it will show up. Normally this one will be issued in a condition and we want to end that properly. Also keep in mind that because we use local control (a nested TEX expansion loop) anything you feed back can be injected out of order.

The three numbers can be separated by an equal sign which is a trick to avoid look ahead issues that can result from multiple serialized numbers without spaces that indicate the end of sequence of digits.

498 \localcontrolledrepeat

This one takes one instead three arguments which looks a bit better in simple looping.

499 \localinterlinepenalty

TODO

500 \localleftbox

This sets the box that gets injected at the left of every line.

501 \localleftboxbox

This returns the box set with `\localleftbox`.

502 \localmiddlebox

This sets the box that gets injected at the left of every line but its width is ignored.

503 \localmiddleboxbox

This returns the box set with `\localmiddlebox`.

504 \localpretolerance

TODO

505 \localrightbox

This sets the box that gets injected at the right of every line.

506 \localrightboxbox

This returns the box set with `\localrightbox`.

507 \localtolerance

TODO

508 \long

This original prefix gave the macro being defined the property that it could not have `\par` (or the often equivalent empty lines) in its arguments. It was mostly a protection against a forgotten right curly brace, resulting in a so called run-away argument. That mattered on a paper terminal or slow system where such a situation should be catched early. In LuaTEX it was already optional, and in LuaMetaTEX we dropped this feature completely (so that we could introduce others).

509 \looseness

The number fo lines in the current paragraph will be increased by given number of lines. For this to succeed there need to be enough stretch in the spacing to make that happen. There is some wishful thinking involved.

510 \lower

This primitive takes two arguments, a dimension and a box. The box is moved down. The operation only succeeds in horizontal mode.

511 \lowercase

This token processor converts character tokens to their lowercase counterparts as defined per `\lccode`. In order to permit dirty tricks active characters are also processed. We don't really use this primitive in ConTEXt, but for consistency we let it respond to `\expand`:[8]

[8] Instead of providing **\lowercased** and **\uppercased** primitives that would clash with macros anyway.

```
\edef           \foo      {\lowercase{tex TeX \TEX}} \meaningless\foo
\lowercase{\edef\foo                 {tex TeX \TEX}} \meaningless\foo
\edef           \foo{\expand\lowercase{tex TeX \TEX}} \meaningless\foo
```

Watch how \lowercase is not expandable but can be forced to. Of course, as the logo macro is protected the TEX logo remains mixed case.

\lowercase {tex TeX \TEX }
tex tex \TEX
tex tex \TEX

512 \lpcode

This one can be used to set the left protrusion factor of a glyph in a font and takes three arguments: font, character code and factor. It is kind of obsolete because we can set up vectors at definition time and tweaking from TEX can have side effects because it globally adapts the font.

513 \luaboundary

This primive inserts a boundary that takes two integer values. Some mechanisms (like math constructors) can trigger a callback when preceded by such a boundary. As we go more mechanisms might do such a check but we don't want a performance hit on ConTEXt as we do so (nor unwanted interference).

514 \luabytecode

This behaves like \luafunction but here the number is a byte code register. These bytecodes are in the lua.bytecode array.

515 \luabytecodecall

This behaves like \luafunctioncall but here the number is a byte code register. These bytecodes are in the lua.bytecode array.

516 \luacopyinputnodes

When set to a positive value this will ensure that when nodes are printed from Lua to TEX copies are used.

517 \luadef

This command relates a (user) command to a Lua function registered in the lua.lualib_get_functions_table(), so after:

```
\luadef\foo123
```

the \foo command will trigger the function at index 123. Of course a macro package has to make sure that these definitions are unique.[9]

[9] Plain TEX established a norm for allocating registers, like \newdimen but there is no such convention for Lua functions.

This command is accompanied by `\luafunctioncall` and `\luafunction`. When we have funciton 123 defined as

```
function() tex.sprint("!") end
```

the following:

```
(\luafunctioncall  \foocode ?)
(\normalluafunction\foocode ?)
(\foo                       ?)
```

gives three times `(!?)`. But this:

```
\edef\oof{\foo                       } \meaning\oof  % protected
\edef\oof{\luafunctioncall  \foocode} \meaning\oof  % protected
\edef\oof{\normalluafunction\foocode} \meaning\oof  % expands
```

returns:

```
macro:!
macro:\luafunctioncall 1740
macro:!
```

Because the definition command is like any other

```
\permanent\protected\luadef\foo123
```

boils down to:

```
permanent protected luacall 123
```

518 `\luaescapestring`

This command converts the given (token) list into something that is acceptable for Lua. It is inherited from LuaTEX and not used in ConTEXt.

```
\directlua { tex.print ("\luaescapestring {{\tt This is a "test".}}") }
```

Results in: `This is a "test".` (Watch the grouping.)

519 `\luafunction`

The integer passed to this primitive is the index in the table returned by `lua.lualib_get_functions_table()`. Of course a macro package has to provide reliable management for this. This is a so called convert command so it expands in an expansion context (like an `\edef`).

520 `\luafunctioncall`

The integer passed to this primitive is the index in the table returned by `lua.lualib_get_functions_table()`. Of course a macro package has to provide reliable management for this. This primitive doesn't expand in an expansion context (like an `\edef`).

521 \luatexbanner

This gives: `This is LuaMetaTeX, Version 2.11.06`.

522 \luatexrevision

This is an integer. The current value is: `11`.

523 \luatexversion

This is an integer. The current value is: `2`.

524 \mark

The given token list is stored in a node in the current list and might become content of `\topmark`, `\botmark` or `\firstmark` when a page split off, or in the case of a box split in `\splitbotmark` or `\splitfirstmark`. In LuaMetaTEX deeply burried marks bubbly up to an outer box level.

525 \marks

This command is similar to `\mark` but first expects a number of a mark register. Multiple marks were introduced in ε-TEX.

526 \mathaccent

This takes a number and a math object to put the accent on. The four byte number has a dummy class byte, a family byte and two index bytes. It is replaced by `\Umathaccent` that handles wide fonts.

527 \mathatom

This operation wraps following content in a atom with the given class. It is part of LuaMetaTEX's extended math support. There are three class related key/values: `class`, `leftclass` and `rightclass` (or `all` for all of them). When none is given this command expects a class number before scanning the content. The `options` key expects a bitset but there are also direct option keys, like `limits`, `nolimits`, `unpack`, `unroll`, `single`, `nooverflow`, `void` and `phantom`. A `source` id can be set, one or more `attr` assigned, and for specific purposes `textfont` and `mathfont` directives are accepted. Features like this are discussed in dedicated manuals.

528 \mathatomglue

This returns the glue that will be inserted between two atoms of a given class for a specific style.

```
\the\mathatomglue \textstyle   1 1
\the\mathatomglue \textstyle   0 2
\the\mathatomglue \scriptstyle 1 1
\the\mathatomglue \scriptstyle 0 2
```

```
1.66667mu
2.22223mu plus 1.11111mu minus 1.11111mu
```

```
1.66667mu
0.55556mu minus 0.27777mu
```

529 \mathatomskip

This injects a glue with the given style and class pair specification: *xx x x x x xx xx*.

```
$x x$
$x \mathatomskip \textstyle   1 1 x$
$x \mathatomskip \textstyle   0 2 x$
$x \mathatomskip \scriptstyle 1 1 x$
$x \mathatomskip \scriptstyle 0 2 x$
```

530 \mathbackwardpenalties

See `\mathforwardpenalties` for an explanation.

531 \mathbeginclass

This variable can be set to signal the class that starts the formula (think of an imaginary leading atom).

532 \mathbin

This operation wraps following content in a atom with class 'binary'.

533 \mathboundary

This primitive is part of an experiment with granular penalties in math. When set nested fences will use the `\mathdisplaypenaltyfactor` or `\mathinlinepenaltyfactor` to increase nested penalties. A bit more control is possible with `\mathboundary`:

0	begin	factor 1000
1	end	factor 1000
2	begin	given factor
3	end	given factor

These will be used when the mentioned factors are zero. The last two variants expect factor to be given.

534 \mathchar

Replaced by `\Umathchar` this old one takes a four byte number: one byte for the class, one for the family an two for the index. The specified character is appended to to the list.

535 \mathcharclass

Returns the slot (in the font) of the given math character.

```
\the\mathcharclass\Umathchar 4 2 123
```

The first passed number is the class, so we get: 4.

536 \mathchardef

Replaced by `\Umathchardef` this primitive relates a control sequence with a four byte number: one byte for the class, one for the family an two for the index. The defined command will insert that character.

537 \mathcharfam

Returns the family number of the given math character.

```
\the\mathcharfam\Umathchar 4 2 123
```

The second passed number is the family, so we get: 2.

538 \mathcharslot

Returns the slot (or index in the font) of the given math character.

```
\the\mathcharslot\Umathchar 4 2 123
```

The third passed number is the slot, so we get: 123.

539 \mathcheckfencesmode

When set to a positive value there will be no warning if a right fence (`\right` or `\Uright`) is missing.

540 \mathchoice

This command expects four subformulas, for display, text, script and scriptscript and it will eventually use one of them depending on circumstances later on. Keep in mind that a formula is first scanned and when that is finished the analysis and typesetting happens.

541 \mathclass

There are build in classes and user classes. The first possible user class is 20 and the last one is 60. You can better not touch the special classes 'all' (61), 'begin' (62) and 'end' (63). The basic 8 classes that original TEX provides are of course also present in LuaMetaTEX. In addition we have some that relate to constructs that the engine builds.

ordinary	ord	0	the default
operator	op	1	small and large operators
binary	bin	2	
relation	rel	3	
open		4	
close		5	
punctuation	punct	6	
variable		7	adapts to the current family
active		8	character marked as such becomes active

inner	9	this class is not possible for characters
under	10	
over	11	
fraction	12	
radical	13	
middle	14	
accent	16	
fenced	17	
ghost	18	
vcenter	19	

There is no standard for user classes but ConTEXt users should be aware of quite some additional ones that are set up. The engine initialized the default properties of classes (spacing, penalties, etc.) the same as original TEX.

Normally characters have class bound to them but you can (temporarily) overload that one. The `\mathclass` primitive expects a class number and a valid character number or math character and inserts the symbol as if it were of the given class; so the original class is replaced.

```
\ruledhbox{$(x)$} and \ruledhbox{$\mathclass 1 `(x\mathclass 1 `)$}
```

Changing the class is likely to change the spacing, compare (x) and (x).

542 \mathclose

This operation wraps following content in a atom with class 'close'.

543 \mathcode

This maps a character to one in a family: the assigned value has one byte for the class, one for the family and two for the index. It has little use in an OpenType math setup.

544 \mathdictgroup

This is an experimental feature that in due time will be explored in ConTEXt. It currently has no consequences for rendering.

545 \mathdictionary

This is an experimental feature that in due time will be explored in ConTEXt. It currently has no consequences for rendering.

546 \mathdictproperties

This is an experimental feature that in due time will be explored in ConTEXt. It currently has no consequences for rendering.

547 \mathdirection

When set to 1 this will result in `r2l` typeset math formulas but of course you then also need to set up math accordingly (which is the case in ConTEXt).

548 \mathdiscretionary

The usual `\discretionary` command is supported in math mode but it has the disadvantage that one needs to make sure that the content triplet does the math right (especially the style). This command takes an optional class specification.

```
\mathdiscretionary                      {+} {+} {+}
\mathdiscretionary class \mathbinarycode {+} {+} {+}
```

It uses the same logic as `\mathchoice` but in this case we handle three snippets in the current style.

A fully automatic mechanism kicks in when a character has a `\hmcode` set:

bit	meaning	explanation
1	normal	a discretionary is created with the same components
2	italic	following italic correction is kept with the component

So we can say:

```
\hmcode `+ 3
```

When the `italic` bit is set italic correction is kept at a linebreak.

549 \mathdisplaymode

Display mode is entered with two dollars (other characters can be used but the dollars are a convention). Mid paragraph display formulas get a different treatment with respect to the width and indentation than stand alone. When `\mathdisplaymode` is larger than zero the double dollars (or equivalents) will behave as inline formulas starting out in `\displaystyle` and with `\everydisplay` expanded.

550 \mathdisplaypenaltyfactor

This one is simular to `\mathinlinepenaltyfactor` but is used when we're in display style.

551 \mathdisplayskipmode

A display formula is preceded and followed by vertical glue specified by `\abovedisplayskip` and `\belowdisplayskip` or `\abovedisplayshortskip` and `\belowdisplayshortskip`. Spacing 'above' is always inserted, even when zero, but the spacing 'below' is only inserted when it is non-zero. There's also `\baselineskip` involved. The way spacing is handled can be influenced with `\mathdisplayskipmode`, which takes the following values:

value	meaning
0	does the same as any TEX engine

1	idem
2	only insert spacing when it is not zero
3	never insert spacing

552 \mathdoublescriptmode

When this parameter has a negative value double scripts trigger an error, so with \superscript, \nosuperscript, \indexedsuperscript, \superprescript, \nosuperprescript, \indexedsuperprescript, \subscript, \nosubscript, \indexedsubscript, \subprescript, \nosubprescript, \indexedsubprescript and \primescript, as well as their (multiple) _ and ^ aliases.

A value of zero does the normal and inserts a dummy atom (basically a {}) but a positive value is more interesting. Compare these:

```
{\mathdoublescriptmode 0      $x_x_x$}
{\mathdoublescriptmode"000000 $x_x_x$}
{\mathdoublescriptmode"030303 $x_x_x$}
{$x_x_x$}
```

The three pairs of bytes indicate the main class, left side class and right side class of the inserted atom, so we get this: $x_{x\,x}$ $x_{x\,x}$ $x_{x\;\;x}$ $x_{x\,x}$. The last line gives what ConTEXt is configured for.

553 \mathendclass

This variable can be set to signal the class that ends the formula (think of an imaginary trailing atom).

554 \matheqnogapstep

The display formula number placement heuristic puts the number on the same line when there is place and then separates it by a quad. In LuaTEX we decided to keep that quantity as it can be tight into the math font metrics but introduce a multiplier \matheqnogapstep that defaults to 1000.

555 \mathfontcontrol

This bitset controls how the math engine deals with fonts, and provides a way around dealing with inconsistencies in the way they are set up. The \fontmathcontrol makes it possible to bind options ot a specific math font. In practice, we just set up the general approach which ii possible because we normalize the math fonts and 'fix' issues at runtime.

0x00000001	usefontcontrol
0x00000002	overrule
0x00000004	underrule
0x00000008	radicalrule
0x00000010	fractionrule
0x00000020	accentskewhalf
0x00000040	accentskewapply
0x00000080	applyordinarykernpair
0x00000100	applyverticalitalickern
0x00000200	applyordinaryitalickern
0x00000400	applycharitalickern

0x00000800	reboxcharitalickern
0x00001000	applyboxeditalickern
0x00002000	staircasekern
0x00004000	applytextitalickern
0x00008000	checktextitalickern
0x00010000	checkspaceitalickern
0x00020000	applyscriptitalickern
0x00040000	analyzescriptnucleuschar
0x00080000	analyzescriptnucleuslist
0x00100000	analyzescriptnucleusbox
0x00200000	accenttopskewwithoffset
0x00400000	ignorekerndimensions
0x00800000	ignoreflataccents
0x01000000	extendaccents
0x02000000	extenddelimiters

556 \mathforwardpenalties

Inline math can have multiple atoms and constructs and one can configure the penalties between then bases on classes. In addition it is possible to configure additional penalties starting from the beginning or end using `\mathforwardpenalties` and `\mathbackwardpenalties`. This is one the features that we added in the perspective of breaking paragraphs heavy on math into lines. It not that easy to come up with useable values.

These penalties are added to the regular penalties between atoms. Here is an example, as with other primitives that take more arguments the first number indicates how much follows.

```
$ a + b + c + d + e + f + g + h = x $\par
\mathforwardpenalties  3 300 200 100
\mathbackwardpenalties 3 250 150  50
$ a + b + c + d + e + f + g + h = x $\par
```

You'll notice that we apply more severe penalties at the edges:

$$a + b + c + d + e + f + g + h = x$$

$$a + b + c + d + e + f + g + h = x$$

557 \mathgluemode

We can influence the way math glue is handled. By default stretch and shrink is applied but this variable can be used to change that. The limit option ensures that the stretch and shrink doesn't go beyond their natural values.

0x01	stretch
0x02	shrink
0x04	limit

558 \mathgroupingmode

Normally a {} or \bgroup-\egroup pair in math create a math list. However, users are accustomed to using it also for grouping and then a list being created might not be what a user wants. As an alternative to the more verbose \begingroup-\endgroup or even less sensitive \beginmathgroup-\endmathgroup you can set the math grouping mode to a non zero value which makes curly braces (and the aliases) behave as expected.

559 \mathinlinepenaltyfactor

A math formula can have nested (sub)formulas and one might want to discourage a line break inside those. If this value is non zero it becomes a mulitiplier, so a value of 1000 will make an inter class penalty of 100 into 200 when at nesting level 2 and 500 when at level 5.

560 \mathinner

This operation wraps following content in a atom with class 'inner'. In LuaMetaTEX we have more classes and this general wrapper one is therefore kind of redundant.

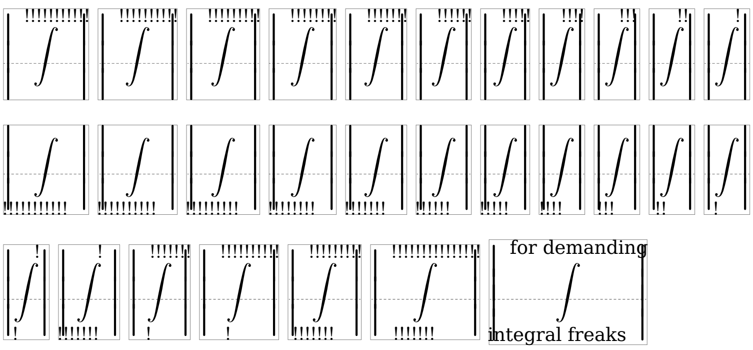
561 \mathleftclass

When set this class will be used when a formula starts.

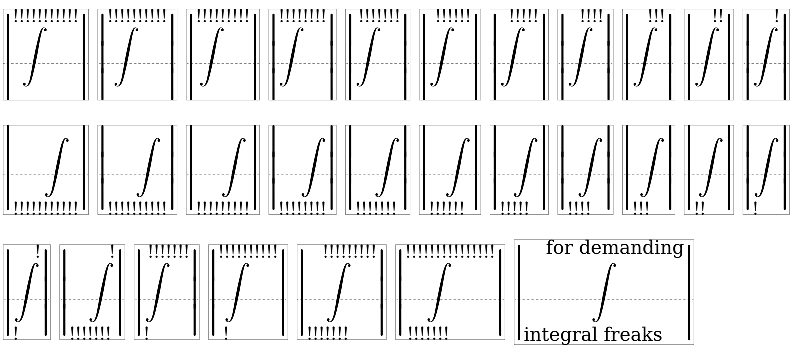
562 \mathlimitsmode

When this parameter is set to a value larger than zero real dimensions are used and longer limits will not stick out, which is a traditional TEX feature. We could have more advanced control but this will do.

Compare the zero setting:



with the positive variant:



Here we switched to Latin Modern because it's font dependent how serious this issue is. In Pagella all is fine in both modes.

563 \mathmainstyle

This inspector returns the outermost math style (contrary to `\mathstyle`), as we can see in the next examples where use these snippets:

```
\def\foo{(\the\mathmainstyle,\the\mathstyle)}
\def\oof{\sqrt[\foo]{\foo}}
\def\ofo{\frac{\foo}{\foo}}
\def\fof{\mathchoice{\foo}{\foo}{\foo}{\foo}}
```

When we use the regular math triggers we get this:

```
$\displaystyle      \foo + \oof + \ofo$
$\textstyle         \foo + \oof + \ofo$
$\displaystyle      \foo + \fof$
$\textstyle         \foo + \fof$
$\scriptstyle       \foo + \fof$
$\scriptscriptstyle\foo + \fof$
```

$(2,0) + \sqrt[(2,0)]{(2,0)} + \frac{(2,5)}{(2,5)}$
$(2,2) + \sqrt[(2,2)]{(2,2)} + \frac{(2,5)}{(2,5)}$
$(2,0) + (2,0)$
$(2,2) + (2,2)$
$(2,4)+(2,4)$
$(2,6)+(2,6)$

But we can also do this:

```
\Ustartmathmode \displaystyle      \foo + \oof + \ofo \Ustopmathmode
\Ustartmathmode \textstyle         \foo + \oof + \ofo \Ustopmathmode
\Ustartmathmode \displaystyle      \foo + \fof \Ustopmathmode
\Ustartmathmode \textstyle         \foo + \fof \Ustopmathmode
\Ustartmathmode \scriptstyle       \foo + \fof \Ustopmathmode
\Ustartmathmode \scriptscriptstyle\foo + \fof \Ustopmathmode
```

$(0,0) + \sqrt[(0,0)]{(0,0)} + \frac{(0,5)}{(0,5)}$
$(2,2) + \sqrt[(2,2)]{(2,2)} + \frac{(2,5)}{(2,5)}$
$(0,0) + (0,0)$
$(2,2) + (2,2)$
$(4,4)+(4,4)$
$(6,6)+(6,6)$

564 \mathnolimitsmode

This parameter influences the placement of scripts after an operator. The reason we have this lays in the fact that traditional TEX uses italic correction and OpenType math does the same but fonts are not consistent in how they set this up. Actually, in OpenType math it's the only reason that there is italic correction. Say that we have a shift δ determined by the italic correction:

mode	top	bottom
0	0	$-\delta$
1	$\delta \times f_t$	$\delta \times f_b$
2	0	0
3	0	$-\delta/2$
4	$\delta/2$	$-\delta/2$
>15	0	$-n \times \delta/1000$

Mode 1 uses two font parameters: f_b: `\Umathnolimitsubfactor` and f_t: `\Umathnolimitsupfactor`.

565 `\mathop`

This operation wraps following content in a atom with class 'operator'.

566 `\mathopen`

This operation wraps following content in a atom with class 'open'.

567 `\mathord`

This operation wraps following content in a atom with class 'ordinary'.

568 `\mathparentstyle`

This inspector returns the math style used in a construct, so is is either equivalent to `\mathmainstyle` or a nested `\mathstyle`. For instance in a nested fraction we get this (in ConTEXt) in display formulas:

$$\frac{\frac{(0,1,5)}{(0,1,5)}}{\frac{(0,1,5)}{(0,1,5)}} + (0,0,0)$$

but this in inline formulas:

$$\frac{\frac{(2,5,7)}{(2,5,7)}}{\frac{(2,5,7)}{(2,5,7)}} + (2,2,2)$$

where the first element in a nested fraction.

569 `\mathpenaltiesmode`

Normally the TEX math engine only inserts penalties when in textstyle. You can force penalties in displaystyle with this parameter. In inline math we always honor penalties, with mode 0 and mode 1 we get this:

$x + 2x = 0$
MP:0 MP:700 MP:500 MP:0

$x + 2x = 1$
MP:0 MP:700 MP:500 MP:0

However in ConTEXt, where all is done in inline math mode, we set this this parameter to 1, otherwise we wouldn't get these penalties, as shown next:

$$x + 2x = 0$$

$$x + 2x = 1$$

If one uses a callback it is possible to force penalties from there too.

570 \mathpretolerance

This is used instead of `\pretolerance` when a breakpoint is calculated when a math formula starts.

571 \mathpunct

This operation wraps following content in a atom with class ‘punctuation’.

572 \mathrel

This operation wraps following content in a atom with class ‘relation’.

573 \mathrightclass

When set this class will be used when a formula ends.

574 \mathrulesfam

When set, this family will be used for setting rule properties in fractions, under and over.

575 \mathrulesmode

When set to a non zero value rules (as in fractions and radicals) will be based on the font parameters in the current family.

576 \mathscale

In LuaMetaTEX we can either have a family of three (text, script and scriptscript) fonts or we can use one font that we scale and where we also pass information about alternative shapes for the smaller sizes. When we use this more compact mode this primitive reflects the scale factor used.

What gets reported depends on how math is implemented, where in ConTEXt we can have either normal or compact mode: 1000 700 550 1000 700 550. In compact mode we have the same font three times so then it doesn't matter which of the three is passed.

577 \mathscriptsmode

There are situations where you don't want TEX to be clever and optimize the position of super- and subscripts by shifting. This parameter can be used to influence this.

0: $x^2_2 + y^x_x + z_2 + w^2$	0: $x^2_2 + y^x_x + z_2 + w^2$	1: $x^2_2 + y^x_x + z_2 + w^2$
0: $x^f_f + y^x_x + z_f + w^f$	0: $x^f_f + y^x_x + z_f + w^f$	1: $x^f_f + y^x_x + z_f + w^f$
1 over 0	2 over 0	2 over 1

The next table shows what parameters kick in when:

	or (1)	and (2)	otherwise
super	sup shift up	sup shift up	sup shift up, sup bot min
sub	sub shift down	sub sup shift down	sub shift down, sub top max
both	sub shift down	sub sup shift down	sub sup shift down, sub sup vgap, sup sub bot max

578 `\mathslackmode`

When positive this parameter will make sure that script spacing is discarded when there is no reason to add it.

$x^2 + x^2$ x^2	$x^2 + x^2$ x^2	$x^2 + x^2$ x^2
disabled (0)	enabled (1)	enabled over disabled

579 `\mathspacingmode`

Zero inter-class glue is not injected but setting this parameter to a positive value bypasses that check. This can be handy when checking (tracing) how (and what) spacing is applied. Keep in mind that glue in math is special in the sense that it is not a valid breakpoint. Line breaks in (inline) math are driven by penalties.

580 `\mathstack`

There are a few commands in TEX that can behave confusing due to the way they are scanned. Compare these:

```
$ 1 \over 2 $
$ 1 + x \over 2 + x$
$ {1 + x} \over {2 + x}$
$ {{1 + x} \over {2 + x}}$
```

A single 1 is an atom as is the curly braced $1 + x$. The two arguments to `\over` eventually will get typeset in the style that this fraction constructor uses for the numerator and denominator but on might actually also like to relate that to the circumstances. It is comparable to using a `\mathchoice`. In order not to waste runtime on four variants, which itself can have side effects, for instance when counters are involved, LuaTEX introduced `\mathstack`, used like:

```
$\mathstack {1 \over 2}$
```

This `\mathstack` command will scan the next brace and opens a new math group with the correct (in this case numerator) math style. The `\mathstackstyle` primitive relates to this feature that defaults to 'smaller unless already scriptscript'.

581 `\mathstackstyle`

This returns the (normally) numerator style but the engine can be configured to default to another style. Although all these in the original TEX engines hard coded style values can be changed in Lua-MetaTEX it is unlikely to happen. So this primitive will normally return the (current) style 'smaller unless already scriptscript'.

582 \mathstyle

This returns the current math style, so `$\the\mathstyle$` gives 2.

583 \mathstylefontid

This returns the font id (a number) of a style/family combination. What you get back depends on how a macro package implements math fonts.

```
(\the\mathstylefontid\textstyle        \fam)
(\the\mathstylefontid\scriptstyle      \fam)
(\the\mathstylefontid\scriptscriptstyle\fam)
```

In ConTEXt gives: (2) (2) (2).

584 \mathsurround

The kern injected before and after an inline math formula. In practice it will be set to zero, if only because otherwise nested math will also get that space added. We also have `\mathsurroundskip` which, when set, takes precedence. Spacing is controlled by `\mathsurroundmode`.

585 \mathsurroundmode

The possible ways to control spacing around inline math formulas in other manuals and mostly serve as playground.

586 \mathsurroundskip

When set this one wins over `\mathsurround`.

587 \maththreshold

This is a glue parameter. The amount determines what happens: when it is non zero and the inline formula is less than that value it will become a special kind of box that can stretch and/ or shrink within the given specification. The par builder will use these stretch and/ or shrink components but it is up to one of the Lua callbacks to deal with the content eventually (if at all). As this is somewhat specialized, more details can be found on ConTEXt documentation.

588 \mathtolerance

This is used instead of `\tolerance` when a breakpoint is calculated when a math formula starts.

589 \maxdeadcycles

When the output routine is called this many times and no page is shipped out an error will be triggered. You therefore need to reset its companion counter `\deadcycles` if needed. Keep in mind that LuaMetaTEX has no real `\shipout` because providing a backend is up to the macro package.

590 \maxdepth

The depth of the page is limited to this value.

591 \meaning

We start with a primitive that will be used in the following sections. The reported meaning can look a bit different than the one reported by other engines which is a side effect of additional properties and more extensive argument parsing.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaning\foo
```

tolerant protected macro:[#1]#*[#2]->(#1)(#2)

592 \meaningasis

Although it is not really round trip with the original due to information being lost this primitive tries to return an equivalent definition.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningasis\foo
```

\permanent \tolerant \protected \def \foo [#1]#*[#2]{(#1)(#2)}

593 \meaningful

This one reports a bit less than `\meaningful`.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningful\foo
```

permanent tolerant protected macro

594 \meaningfull

This one reports a bit more than `\meaning`.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningfull\foo
```

permanent tolerant protected macro:[#1]#*[#2]->(#1)(#2)

595 \meaningles

This one reports a bit less than `\meaningless`.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningles\foo
```

[#1]#*[#2]

596 \meaningless

This one reports a bit less than `\meaning`.

```
\tolerant\permanent\protected\gdef\foo[#1]#*[#2]{(#1)(#2)} \meaningless\foo
```

[#1]#*[#2]->(#1)(#2)

597 \medmuskip

A predefined mu skip register that can be used in math (inter atom) spacing. The current value is `4.0mu plus 2.0mu minus 2.0mu`. In traditional TEX most inter atom spacing is hard coded using the predefined registers.

598 \message

Prints the serialization of the (tokenized) argument to the log file and/or console.

599 \middle

Inserts the given delimiter as middle fence in a math formula. In LuaMetaTEX it is a full blown fence and not (as in ε-TEX) variation of `\open`.

600 \mkern

This one injects a kern node in the current (math) list and expects a value in so called mu units.

601 \month

This internal number starts out with the month that the job started.

602 \moveleft

This primitive takes two arguments, a dimension and a box. The box is moved to the left. The operation only succeeds in vertical mode.

603 \moveright

This primitive takes two arguments, a dimension and a box. The box is moved to the right. The operation only succeeds in vertical mode.

604 \mskip

The given math glue (in `mu` units) is injected in the horizontal list. For this to succeed we need to be in math mode.

605 \muexpr

This is a companion of `\glueexpr` so it handles the optional stretch and shrink components. Here math units (`mu`) are expected.

606 \mugluespecdef

A variant of `\gluespecdef` that expects `mu` units is:

```
\mugluespecdef\MyGlue = 3mu plus 2mu minus 1mu
```

The properties are comparable to the ones described in the previous sections.

607 \multiply

The given quantity is multiplied by the given integer (that can be preceded by the keyword ‘by’, like:

```
\scratchdimen=10pt \multiply\scratchdimen by 3
```

608 \multiplyby

This is slightly more efficient variant of `\multiply` that doesn't look for `by`. See previous section.

609 \muskip

This is the accessor for an indexed muskip (muglue) register.

610 \muskipdef

This command associates a control sequence with a muskip (math skip) register (accessed by number).

611 \mutable

This prefix flags what follows can be adapted and is not subjected to overload protection.

612 \mutoglue

The sequence `\the\mutoglue 20mu plus 10mu minus 5mu` gives 20.0pt plus 10.0pt minus 5.0pt.

613 \nestedloopiterator

This is one of the accessors of loop iterators:

```
\expandedrepeat 2 {%
    \expandedrepeat 3 {%
        (n=\the\nestedloopiterator  1,
         p=\the\previousloopiterator1,
         c=\the\currentloopiterator)
    }%
}%
```

Gives:

(n=1, p=1, c=1) (n=2, p=1, c=2) (n=3, p=1, c=3) (n=1, p=2, c=1) (n=2, p=2, c=2) (n=3, p=2, c=3)

Where a nested iterator starts relative to innermost loop, the previous one is relative to the outer loop (which is less predictable because we can already be in a loop).

614 \newlinechar

When something is printed to one of the log channels the character with this code will trigger a linebreak. That also resets some counters that deal with suppressing redundant ones and possible indentation. Contrary to other engines LuaMetaTEX doesn't bother about the length of lines.

615 \noalign

The token list passed to this primitive signals that we don't enter a table row yet but for instance in a `\halign` do something between the lines: some calculation or injecting inter-row material. In LuaMetaTEX this primitive can be used nested.

616 \noaligned

The alignment mechanism is kind of special when it comes to expansion because it has to look ahead for a `\noalign`. This interferes with for instance protected macros, but using this prefix we get around that. Among the reasons to use protected macros inside an alignment is that they behave better inside for instance `\expanded`.

617 \noatomruling

Spacing in math is based on classes and this primitive inserts a signal that there is no ruling in place here. Basically we have a zero skip glue tagged as non breakable because in math mode glue is not a valid breakpoint unless we have configured inter-class penalties.

618 \noboundary

This inserts a boundary node with no specific property. It can still serve as boundary but is not interpreted in special ways, like the others.

619 \noexpand

This prefix prevents expansion in a context where expansion happens. Another way to prevent expansion is to define a macro as `\protected`.

```
          \def\foo{foo} \edef\oof{we expanded      \foo} \meaning\oof
          \def\foo{foo} \edef\oof{we keep \noexpand\foo} \meaning\oof
\protected\def\foo{foo} \edef\oof{we keep          \foo} \meaning\oof
```

macro:we expanded foo
macro:we keep \foo
macro:we keep \foo

620 \nohrule

This is a rule but flagged as empty which means that the dimensions kick in as for a normal rule but the backend can decide not to show it.

621 \noindent

This starts a paragraph. In LuaTEX (and LuaMetaTEX) a paragraph starts with a so called par node (see `\indent` on how control that. After that comes either `\parindent` glue or a horizontal box. The `\indent` makes gives them some width, while `\noindent` keeps that zero.

622 \nolimits

This is a modifier: it flags the previous math atom to have its scripts after the the atom (contrary to `\limits`. In LuaMetaTEX this can be any atom (that is: any class). In display mode the location defaults to above and below.

623 \nomathchar

This can be used when a math character is expected but not available (or needed).

624 \nonscript

This prevents TEX from adding inter-atom glue at this spot in script or scriptscript mode. It actually is a special glue itself that serves as signal.

625 \nonstopmode

This directive omits all stops.

626 \norelax

The rationale for this command can be shown by a few examples:

```
\dimen0 1pt \dimen2 1pt \dimen4 2pt
\edef\testa{\ifdim\dimen0=\dimen2\norelax N\else Y\fi}
\edef\testb{\ifdim\dimen0=\dimen2\relax   N\else Y\fi}
\edef\testc{\ifdim\dimen0=\dimen4\norelax N\else Y\fi}
\edef\testd{\ifdim\dimen0=\dimen4\relax   N\else Y\fi}
\edef\teste{\norelax}
```

The five meanings are:

```
\testa  macro:N
\testb  macro:\relax N
\testc  macro:Y
\testd  macro:Y
\teste  macro:
```

So, the `\norelax` acts like `\relax` but is not pushed back as usual (in some cases).

627 \normalizelinemode

The TEX engine was not designed to be opened up, and therefore the result of the linebreak effort can differ depending on the conditions. For instance not every line gets the left- or rightskip. The first and last lines have some unique components too. When LuaTEX made it possible too get the (intermediate) result manipulating the result also involved checking what one encountered, for instance glue and its origin. In LuaMetaTEX we can normalize lines so that they have for instance balanced skips.

`0x0001`	normalizeline	`0x0040`	clipwidth
`0x0002`	parindentskip	`0x0080`	flattendiscretionaries
`0x0004`	swaphangindent	`0x0100`	discardzerotabskips
`0x0008`	swapparshape	`0x0200`	flattenhleaders
`0x0010`	breakafterdir	`0x0400`	balanceinlinemath
`0x0020`	removemarginkerns		

The order in which the skips get inserted when we normalize is as follows:

`\lefthangskip`	the hanging indentation (or zero)
`\leftskip`	the value even when zero
`\parfillleftskip`	only on the last line
`\parinitleftskip`	only on the first line
`\indentskip`	the amount of indentation
`...`	the (optional) content
`\parinitrightskip`	only on the first line
`\parfillrightskip`	only on the last line
`\correctionskip`	the correction needed to stay within the `\hsize`
`\rightskip`	the value even when zero
`\righthangskip`	the hanging indentation (or zero)

The init and fill skips can both show up when we have a single line. The correction skip replaces the traditional juggling with the right skip and shift of the boxed line.

For now we leave the other options to your imagination. Some of these can be achieved by callbacks (as we did in older versions of ConTEXt) but having the engine do the work we get a better performance.

628 `\normalizeparmode`

For now we just mention the few options available. It is also worth mentioning that LuaMetaTEX tries to balance the direction nodes.

`0x01`	normalizepar	`0x04`	limitprevgraf
`0x02`	flattenvleaders	`0x08`	keepinterlinepenalties

629 `\noscript`

In math we can have multiple pre- and postscript. These get typeset in pairs and this primitive can be used to skip one. More about multiple scripts (and indices) can be found in the ConTEXt math manual.

630 `\nospaces`

When `\nospaces` is set to 1 no spaces are inserted, when its value is 2 a zero space is inserted. The default value is 0 which means that spaces become glue with properties depending on the font, specific parameters and/or space factors determined preceding characters. A value of 3 will inject a glyph node with code `\spacechar`.

631 \nosubprescript

This processes the given script in the current style, so:

comes out as: ${}_2x + {}_{2}x + {}_2x$.

632 \nosubscript

This processes the given script in the current style, so:

comes out as: $x_2 + x_2 + x_2$.

633 \nosuperprescript

This processes the given script in the current style, so:

comes out as: ${}^2x + {}^2x + {}^2x$.

634 \nosuperscript

This processes the given script in the current style, so:

comes out as: $x^2 + {}^2x + {}^2x$.

635 \novrule

This is a rule but flagged as empty which means that the dimensions kick in as for a normal rule but the backend can decide not to show it.

636 \nulldelimiterspace

In fenced math delimiters can be invisible in which case this parameter determines the amount of space (width) that ghost delimiter takes.

637 \nullfont

This a symbolic reference to a font with no glyphs and a minimal set of font dimensions.

638 \number

This TEX primitive serializes the next token into a number, assuming that it is indeed a number, like

```
\number`A
\number65
\number\scratchcounter
```

For counters and such the `\the` primitive does the same, but when you're not sure if what follows is a verbose number or (for instance) a counter the `\number` primitive is a safer bet, because `\the 65` will not work.

639 \numericscale

This primitive can best be explained by a few examples:

```
\the\numericscale 1323
\the\numericscale 1323.0
\the\numericscale 1.323
\the\numericscale 13.23
```

In several places TEX uses a scale but due to the lack of floats it then uses 1000 as 1.0 replacement. This primitive can be used for 'real' scales:

1323000
1323000
1323
13230

640 \numericscaled

This is a variant if `\numericscale`:

```
\scratchcounter 1000
\the\numericscaled 1323   \scratchcounter
\the\numericscaled 1323.0 \scratchcounter
\the\numericscaled 1.323  \scratchcounter
\the\numericscaled 13.23  \scratchcounter
```

The second number gets multiplied by the first fraction:

1323000
1323000
1323
13230

641 \numexpr

This primitive was introduced by ε-TEX and supports a simple expression syntax:

```
\the\numexpr 10 * (1 + 2 - 5) / 2 \relax
```

gives: -10. You can mix in symbolic integers and dimensions.

642 \numexpression

The normal `\numexpr` primitive understands the `+`, `-`, `*` and `/` operators but in LuaMetaTEX we also can use `:` for a non rounded integer division (think of Lua's `//`). if you want more than that, you can use the new expression primitive where you can use the following operators.

add	+	
subtract	-	
multiply	*	
divide	/ :	
mod	%	mod
band	&	band
bxor	^	bxor
bor	| v	bor
and	&&	and
or	||	or
setbit	<undecided>	bset
resetbit	<undecided>	breset
left	<<	
right	>>	
less	<	
lessequal	<=	
equal	= ==	
moreequal	>=	
more	>	
unequal	<> != ~=	
not	! ~	not

An example of the verbose bitwise operators is:

```
\scratchcounter = \numexpression
    "00000 bor "00001 bor "00020 bor "00400 bor "08000 bor "F0000
\relax
```

In the table you might have notices that some operators have equivalents. This makes the scanner a bit less sensitive for catcode regimes.

When `\tracingexpressions` is set to one or higher the intermediate 'reverse polish notation' stack that is used for the calculation is shown, for instance:

```
4:8: {numexpression rpn: 2 5 > 4 5 > and}
```

When you want the output on your console, you need to say:

```
\tracingexpressions 1
\tracingonline      1
```

Here are some things that `\numexpr` is not suitable for but `\numexpression` can handle:

```
\scratchcounter = \numexpression
    "00000 bor "00001 bor "00020 bor "00400 bor "08000 bor "F0000
\relax

\ifcase \numexpression
    (\scratchcounterone > 5) && (\scratchcountertwo > 5)
\relax yes\else nop\fi
```

643 \omit

This primitive cancels the template set for the upcoming cell. Often it is used in combination with `\span`.

644 \optionalboundary

This boundary is used to mark optional content. An positive `\optionalboundary` starts a range and a zero one ends it. Nesting is not supported. Optional content is considered when an additional paragraph pass enables it as part of its recipe.

645 \or

This traditional primitive is part of the condition testing mechanism and relates to an `\ifcase` test (or a similar test to be introduced in later sections). Depending on the value, TEX will do a fast scanning till the right `\or` is seen, then it will continue expanding till it sees a `\or` or `\else` or `\orelse` (to be discussed later). It will then do a fast skipping pass till it sees an `\fi`.

646 \orelse

This primitive provides a convenient way to flatten your conditional tests. So instead of

```
\ifnum\scratchcounter<-10
    too small
\else\ifnum\scratchcounter>10
    too large
\else
    just right
\fi\fi
```

You can say this:

```
\ifnum\scratchcounter<-10
    too small
\orelse\ifnum\scratchcounter>10
    too large
\else
    just right
\fi
```

You can mix tests and even the case variants will work in most cases[10]

```
\ifcase\scratchcounter          zero
\or                             one
\or                             two
\orelse\ifnum\scratchcounter<10 less than ten
\else                           ten or more
\fi
```

[10] I just play safe because there are corner cases that might not work yet.

Performance wise there are no real benefits although in principle there is a bit less housekeeping involved than with nested checks. However you might like this:

```
\ifnum\scratchcounter<-10
    \expandafter\toosmall
\orelse\ifnum\scratchcounter>10
    \expandafter\toolarge
\else
    \expandafter\justright
\fi
```

over:

```
\ifnum\scratchcounter<-10
    \expandafter\toosmall
\else\ifnum\scratchcounter>10
    \expandafter\expandafter\expandafter\toolarge
\else
    \expandafter\expandafter\expandafter\justright
\fi\fi
```

or the more ConTEXt specific:

```
\ifnum\scratchcounter<-10
    \expandafter\toosmall
\else\ifnum\scratchcounter>10
    \doubleexpandafter\toolarge
\else
    \doubleexpandafter\justright
\fi\fi
```

But then, some TEXies like complex and obscure code and throwing away working old code that took ages to perfect and get working and also showed that one masters TEX might hurt.

There is a nice side effect of this mechanism. When you define:

```
\def\quitcondition{\orelse\iffalse}
```

you can do this:

```
\ifnum\count0<10
    less
\orelse\ifnum\count0=10
    equal
    \quitcondition
    indeed
\else
    more
\fi
```

Of course it is only useful at the right level, so you might end up with cases like

```
\ifnum\count0<10
```

```
    less
\orelse\ifnum\count0=10
    equal
    \ifnum\count2=30
        \expandafter\quitcondition
    \fi
    indeed
\else
    more
\fi
```

647 \orphanpenalties

This an (single entry) array parameter: first the size is given followed by that amount of penalties. These penalties are injected before spaces, going backward from the end of a paragraph. When we see a math node with a penalty set then we take the max and jump over a (preceding) skip.

648 \orunless

This is the negated variant of `\orelse` (prefixing that one with `\unless` doesn't work well.

649 \outer

An outer macro is one that can only be used at the outer level. This property is no longer supported. Like `\long`, the `\outer` prefix is now an no-op (and we don't expect this to have unfortunate side effects).

650 \output

This token list register holds the code that will be expanded when TEX enters the output routine. That code is supposed to do something with the content in the box with number `\outputbox`. By default this is box 255 but that can be changed with `\outputbox`.

651 \outputbox

This is where the split off page contend ends up when the output routine is triggered.

652 \outputpenalty

This is the penalty that triggered the output routine.

653 \over

This math primitive is actually a bit of a spoiler for the parser as it is one of the few that looks back. The `\Uover` variant is different and takes two arguments. We leave it to the user to predicts the results of:

```
$    {1} \over {x}    $
```

```
$     1  \over  x     $
$    12  \over  x / y $
$ a + 1  \over {x}    $
```

and:

```
$  \textstyle 1  \over x  $
$ {\textstyle 1} \over x  $
$  \textstyle {1 \over x} $
```

It's one of the reasons why macro packages provide `\frac`.

654 \overfullrule

When an overfull box is encountered a rule can be shown in the margin and this parameter sets its width. For the record: ConTeXt does it different.

655 \overline

This is a math specific primitive that draws a line over the given content. It is a poor mans replacement for a delimiter. The thickness is set with `\Umathoverbarrule`, the distance between content and rule is set by `\Umathoverbarvgap` and `\Umathoverbarkern` is added above the rule. The style used for the content under the rule can be set with `\Umathoverlinevariant`.

Because ConTeXt set up math in a special way, the following example:

```
\normaloverline {
    \blackrule[color=red, height=1ex,depth=0ex,width=2cm]%
    \kern-2cm
    \blackrule[color=blue,height=0ex,depth=.5ex,width=2cm]
    x + x
}
```

gives: , while:

```
\mathfontcontrol\zerocount
\Umathoverbarkern\allmathstyles10pt
\Umathoverbarvgap\allmathstyles5pt
\Umathoverbarrule\allmathstyles2.5pt
\Umathoverlinevariant\textstyle\scriptstyle
```

gives this: . We have to disable the related `\mathfontcontrol` bits because otherwise the thickness is taken from the font. The variant is just there to overload the (in traditional TeX engines) default.

656 \overloaded

This prefix can be used to overload a frozen macro.

657 \overloadmode

The overload protection mechanism can be used to prevent users from redefining a control sequence. The mode can have several values, the higher the more strict we are:

		immutable	permanent	primitive	frozen	instance
1	warning	+	+	+		
2	error	+	+	+		
3	warning	+	+	+	+	
4	error	+	+	+	+	
5	warning	+	+	+	+	+
6	error	+	+	+	+	+

When you set a high error value, you can of course temporary lower or even zero the mode. In ConTEXt all macros and quantities are tagged so there setting the mode to 6 gives a proper protection against overloading. We need to zero the mode when we load for instance tikz, so when you use that generic package, you loose some.

658 \overshoot

This primitive is a companion to `\badness` and reports how much a box overflows.

```
\setbox0\hbox to 1em {mmm} \the\badness\quad\the\overshoot
\setbox0\hbox         {mm} \the\badness\quad\the\overshoot
\setbox0\hbox to 3em   {m} \the\badness\quad\the\overshoot
```

This reports:

1000000   18.44727pt
0   0.0pt
10000   0.0pt

And:

```
\hbox to 2cm {does it fit}               \the\overshoot
\hbox to 2cm {does it fit in here}       \the\overshoot
\hbox to 2cm {how much does fit in here} \the\overshoot
```

gives:

does it fit
0.0pt
does it fit in here
25.64333pt
how much does fit in here
69.53004pt

When traditional TEX wraps up the lines in a paragraph it uses a mix of shift (a box property) to position the content suiting the hanging indentation and/or paragraph shape, and fills up the line using right skip glue, also in order to silence complaints in packaging. In LuaMetaTEX the lines can be normalized so that they all have all possible skips to the left and right (even if they're zero). The `\overshoot` primitive fits into this picture and is present as a compensation glue. This all fits better in a situation where the internals are opened up via Lua.

659 `\overwithdelims`

This is a variant of `\over` but with delimiters. It has a more advanced upgrade in `\Uoverwithdelims`.

660 `\pageboundary`

In order to avoid side effects of triggering the page builder with a specific penalty we can use this primitive which expects a value that actually gets inserted as zero penalty before triggering the page builder callback. Think of adding a no-op to the contribution list. We fake a zero penalty so that all gets processed. The main rationale is that we get a better indication of what we do. Of course a callback can remove this node so that it is never seen. Triggering from the callback is not doable. Consider this experimental code (which is actually used in ConTEXt anyway).

661 `\pagedepth`

This page property holds the depth of the page.

662 `\pagediscards`

The left-overs after a page is split of the main vertical list when glue and penalties are normally discarded. The discards can be pushed back in (for instance) trial runs.

663 `\pageexcess`

This page property hold the amount of overflow when a page break occurs.

664 `\pageextragoal`

This (experimental) dimension will be used when the page overflows but a bit of overshoot is considered okay.

665 `\pagefilllstretch`

The accumulated amount of third order stretch on the current page.

666 `\pagefillstretch`

The accumulated amount of second order stretch on the current page.

667 `\pagefilstretch`

The accumulated amount of first order stretch on the current page.

668 `\pagefistretch`

The accumulated amount of zero order stretch on the current page.

669 \pagegoal

The target height of a page (the running text). This value will be decreased by the height of inserts something to keep into mind when messing around with this and other (pseudo) page related parameters like `\pagetotal`.

670 \pagelastdepth

The accumulated depth of the current page.

671 \pagelastfilllstretch

The accumulated amount of third order stretch on the current page. Contrary to `\pagefilllstretch` this is the really contributed amount, not the upcoming.

672 \pagelastfillstretch

The accumulated amount of second order stretch on the current page. Contrary to `\pagefillstretch` this is the really contributed amount, not the upcoming.

673 \pagelastfilstretch

The accumulated amount of first order stretch on the current page. Contrary to `\pagefilstretch` this is the really contributed amount, not the upcoming.

674 \pagelastfistretch

The accumulated amount of zero order stretch on the current page. Contrary to `\pagefistretch` this is the really contributed amount, not the upcoming.

675 \pagelastheight

The accumulated height of the current page.

676 \pagelastshrink

The accumulated amount of shrink on the current page. Contrary to `\pageshrink` this is the really contributed amount, not the upcoming.

677 \pagelaststretch

The accumulated amount of stretch on the current page. Contrary to `\pagestretch` this is the really contributed amount, not the upcoming.

678 \pageshrink

The accumulated amount of shrink on the current page.

679 `\pagestretch`

The accumulated amount of stretch on the current page.

680 `\pagetotal`

The accumulated page total (height) of the current page.

681 `\pagevsize`

This parameter, when set, is used as the target page height. This lessens the change of `\vsize` interfering.

682 `\par`

This is the explicit ‘finish paragraph’ command. Internally we distinguish a par triggered by a new line, as side effect of another primitive or this `\par` command.

683 `\parametercount`

The number of parameters passed to the current macro.

684 `\parameterdef`

Here is an example of binding a variable to a parameter. The alternative is of course to use an `\edef`.

```
\def\foo#1#2%
  {\parameterdef\MyIndexOne\plusone % 1
   \parameterdef\MyIndexTwo\plustwo % 2
   \oof{P}\oof{Q}\oof{R}\norelax}

\def\oof#1%
  {<1:\MyIndexOne><1:\MyIndexOne>%
   #1%
   <2:\MyIndexTwo><2:\MyIndexTwo>}

\foo{A}{B}
```

The outcome is:

<1:A><1:A>P<2:B><2:B><1:A><1:A>Q<2:B><2:B><1:A><1:A>R<2:B><2:B>

685 `\parameterindex`

This gives the zero based position on the parameter stack. One reason for introducing `\parameterdef` is that the position remains abstract so there we don't need to use `\parameterindex`.

686 `\parametermark`

The meaning of primitive `\parametermark` is equivalent to `#` in a macro definition, just like `\alignmark` is in an alignment. It can be used to circumvent catcode issues. The normal “duplicate them when nesting” rules apply.

```
\def\foo\parametermark1%
  {\def\oof\parametermark\parametermark1%
     {[\parametermark1:\parametermark\parametermark1]}}
```

Here `\foo{X}\oof{Y}` gives: [X:Y].

687 \parametermode

Setting this internal integer to a positive value (best use 1 because future versions might use bit set) will enable the usage of `#` for escaped in the main text and body of macros.

688 \parattribute

This primitive takes an attribute index and value and sets that attribute on the current paragraph.

689 \pardirection

This set the text direction for the whole paragraph which in the case of `r2l` (1) makes the right edge the starting point.

690 \parfillleftskip

The glue inserted at the start of the last line.

691 \parfillrightskip

The glue inserted at the end of the last line (aka `\parfillskip`).

692 \parfillskip

The glue inserted at the end of the last line.

693 \parindent

The amount of space inserted at the start of the first line. When bit 2 is set in `\normalizelinemode` a glue is inserted, otherwise an empty `\hbox` with the given width is inserted.

694 \parinitleftskip

The glue inserted at the start of the first line.

695 \parinitrightskip

The glue inserted at the end of the first line.

696 \parpasses

Specifies one or more recipes for additional second linebreak passes. Examples can be found in the ConTEXt distribution.

697 \parpassesexception

Specifies al alternative parpass to use in the upcoming paragraph, for instance one with a specific looseness that then demands for instance more emergency stretch.

698 \parshape

Stores a shape specification. The first argument is the length of the list, followed by that amount of indentation-width pairs (two dimensions).

699 \parshapedimen

This oddly named (ε-TEX) primitive returns the width component (dimension) of the given entry (an integer). Obsoleted by `\parshapewidth`.

700 \parshapeindent

Returns the indentation component (dimension) of the given entry (an integer).

701 \parshapelength

Returns the number of entries (an integer).

702 \parshapewidth

Returns the width component (dimension) of the given entry (an integer).

703 \parskip

This is the amount of glue inserted before a new paragraph starts.

704 \patterns

The argument to this primitive contains hyphenation patterns that are bound to the current language. In LuaTEX and LuaMetaTEX we can also manage this at the Lua end. In LuaMetaTEX we don't store patterns in te format file

705 \pausing

In LuaMetaTEX this variable is ignored but in other engines it can be used to single step thought the input file by setting it to a positive value.

706 \penalty

The given penalty (a number) is inserted at the current spot in the horizontal or vertical list. We also have `\vpenalty` and `\hpenalty` that first change modes.

707 \permanent

This is one of the prefixes that is part of the overload protection mechanism. It is normally used to flag a macro as being at the same level as a primitive: don't touch it. primitives are flagged as such but that property cannot be set on regular macros. The similar `\immutable` flag is normally used for variables.

708 \pettymuskip

A predefined mu skip register that can be used in math (inter atom) spacing. The current value is `1.0mu minus 0.5mu`. This one complements `\thinmuskip`, `\medmuskip`, `\thickmuskip` and the new `\tinymuskip`.

709 \positdef

The engine uses 32 bit integers for various purposes and has no (real) concept of a floating point quantity. We get around this by providing a floating point data type based on 32 bit unums (posits). These have the advantage over native floats of more precision in the lower ranges but at the cost of a software implementation.

The `\positdef` primitive is the floating point variant of `\integerdef` and `\dimensiondef`: an efficient way to implement named quantities other than registers.

```
\positdef     \MyFloatA 5.678
\positdef     \MyFloatB 567.8
[\the\MyFloatA] [\todimension\MyFloatA] [\tointeger\MyFloatA]
[\the\MyFloatB] [\todimension\MyFloatB] [\tointeger\MyFloatB]
```

For practical reasons we can map posit (or float) onto an integer or dimension:

[5.67800000309944115283] [5.678pt] [6]
[567.8000030517578125] [567.80005pt] [568]

710 \postdisplaypenalty

This is the penalty injected after a display formula.

711 \postexhyphenchar

This primitive expects a language number and a character code. A negative character code is equivalent to ignore. In case of an explicit discretionary the character is injected at the beginning of a new line.

712 \posthyphenchar

This primitive expects a language number and a character code. A negative character code is equivalent to ignore. In case of an automatic discretionary the character is injected at the beginning of a new line.

713 \postinlinepenalty

When set this penalty is inserted after an inline formula unless we have a short formula and `\postshortinlinepenalty` is set.

714 \postshortinlinepenalty

When set this penalty is inserted after a short inline formula. The criterium is set by `\shortinlinemaththreshold` but only applied when it is enabled for the class involved.

715 \prebinoppenalty

This internal quantity is a compatibility feature because normally we will use the inter atom spacing variables.

716 \predisplaydirection

This is the direction that the math sub engine will take into account when dealing with right to left typesetting.

717 \predisplaygapfactor

The heuristics related to determine if the previous line in a formula overlaps with a (display) formula are hard coded but in LuaTEX to be two times the quad of the current font. This parameter is a multiplier set to 2000 and permits you to change the overshoot in this heuristic.

718 \predisplaypenalty

This is the penalty injected before a display formula.

719 \predisplaysize

This parameter holds the length of the last line in a paragraph when a display formula is part of it.

720 \preexhyphenchar

This primitive expects a language number and a character code. A negative character code is equivalent to ignore. In case of an explicit discretionary the character is injected at the end of the line.

721 \prehyphenchar

This primitive expects a language number and a character code. A negative character code is equivalent to ignore. In case of an automatic discretionary the character is injected at the end of the line.

722 \preinlinepenalty

When set this penalty is inserted before an inline formula unless we have a short formula and `\preshortinlinepenalty` is set. These are not real penalties but properties of the math begin and end markers. Just as with spacing as such property, these penalties are not visible as nodes in the list.

723 \prerelpenalty

This internal quantity is a compatibility feature because normally we will use the inter atom spacing variables.

724 \preshortinlinepenalty

When set this penalty is inserted before a short inline formula. The criterium is set by `\shortinlinemaththreshold` but only applied when it is enabled for the class involved.

725 \pretolerance

When the badness of a line in a paragraph exceeds this value a second linebreak pass will be enabled.

726 \prevdepth

The depth of current list. It can also be set to special (signal) values in order to inhibit line corrections. It is not an internal dimension but a (current) list property.

727 \prevgraf

The number of lines in a previous paragraph.

728 \previousloopiterator

```
\edef\testA{
    \expandedrepeat 2 {%
        \expandedrepeat 3 {%
            (\the\previousloopiterator1:\the\currentloopiterator)
        }%
    }%
}
\edef\testB{
    \expandedrepeat 2 {%
        \expandedrepeat 3 {%
            (#P:#I) % #G is two levels up
        }%
    }%
}
```

These give the same result:

```
\def \testA { (1:1) (1:2) (1:3) (2:1) (2:2) (2:3) }
\def \testB { (1:1) (1:2) (1:3) (2:1) (2:2) (2:3) }
```

The number indicates the number of levels we go up the loop chain.

729 \primescript

This is a math script primitive dedicated to primes (which are somewhat troublesome on math). It complements the six script primitives (like `\subscript` and `\presuperscript`).

730 \protected

A protected macro is one that doesn't get expanded unless it is time to do so. For instance, inside an \edef it just stays what it is. It often makes sense to pass macros as-is to (multi-pass) file (for tables of contents).

In ConTEXt we use either \protected or \unexpanded because the later was the command we used to achieve the same results before ε-TEX introduced this protection primitive. Originally the \protected macro was also defined but it has been dropped.

731 \protecteddetokenize

This is a variant of \protecteddetokenize that uses some escapes encoded as body parameters, like #H for a hash.

732 \protectedexpandeddetokenize

This is a variant of \expandeddetokenize that uses some escapes encoded as body parameters, like #H for a hash.

733 \protrudechars

This variable controls protrusion (into the margin). A value 2 is comparable with other engines, while a value of 3 does a bit more checking when we're doing right-to-left typesetting.

734 \protrusionboundary

This injects a boundary with the given value:

0x00	skipnone
0x01	skipnext
0x02	skipprevious
0x03	skipboth

This signal makes the protrusion checker skip over a node.

735 \pxdimen

The current numeric value of this dimension is 65781, 1.00374pt: one bp. We kept it around because it was introduced in pdfTEX and made it into LuaTEX, where it relates to the resolution of included images. In ConTEXt it is not used.

736 \quitloop

There are several loop primitives and they can be quit with \quitloop at the next the *next* iteration. An immediate quit is possible with \quitloopnow. An example is given with \localcontrolledloop.

737 \quitloopnow

There are several loop primitives and they can be quit with \quitloopnow at the spot.

738 \quitvmode

This primitive forces horizontal mode but has no side effects when we're already in that mode.

739 \radical

This old school radical constructor is replaced by `\Uradical`. It takes a number where the first byte is the small family, the next two index of this symbol from that family, and the next three the family and index of the first larger variant.

740 \raise

This primitive takes two arguments, a dimension and a box. The box is moved up. The operation only succeeds in horizontal mode.

741 \rdivide

This is variant of `\divide` that rounds the result. For integers the result is the same as `\edivide`.

```
\the\dimexpr .4999pt                     : 2 \relax           =.24994pt
\the\dimexpr .4999pt                     / 2 \relax           =.24995pt
\the\dimexpr .4999pt                     ; 2 \relax           =.00002pt
\scratchdimen.4999pt \divide \scratchdimen 2 \the\scratchdimen =.24994pt
\scratchdimen.4999pt \edivide\scratchdimen 2 \the\scratchdimen =.24995pt
\scratchdimen 4999pt \rdivide\scratchdimen 2 \the\scratchdimen =2500.0pt
\scratchdimen 5000pt \rdivide\scratchdimen 2 \the\scratchdimen =2500.0pt

\the\numexpr   1001                        : 2 \relax            =500
\the\numexpr   1001                        / 2 \relax            =501
\the\numexpr   1001                        ; 2 \relax            =1
\scratchcounter1001 \divide \scratchcounter 2 \the\scratchcounter=500
\scratchcounter1001 \edivide\scratchcounter 2 \the\scratchcounter=501
\scratchcounter1001 \rdivide\scratchcounter 2 \the\scratchcounter=501
```

0.24994pt=.24994pt
0.24995pt=.24995pt
0.00002pt=.00002pt
0.24994pt=.24994pt
0.24995pt=.24995pt
2500.0pt=2500.0pt
2500.0pt=2500.0pt

500=500
501=501
1=1
500=500
501=501
501=501

The integer division `:` and modulo `;` are an addition to the ε-TEX compatible expressions.

742 \rdivideby

This is the by-less companion to `\rdivide`.

743 \realign

Where `\omit` suspends a preamble template, this one overloads is for the current table cell. It expects two token lists as arguments.

744 \relax

This primitive does nothing and is often used to end a verbose number or dimension in a comparison, for example:

```
\ifnum \scratchcounter = 123\relax
```

which prevents a lookahead. A variant would be:

```
\ifnum \scratchcounter = 123 %
```

assuming that spaces are not ignored. Another application is finishing an expression like `\numexpr` or `\dimexpr`. I is also used to prevent lookahead in cases like:

```
\vrule height 3pt depth 2pt width 5pt\relax
\hskip 5pt plus 3pt minus 2pt\relax
```

Because `\relax` is not expandable the following:

```
\edef\foo{\relax}   \meaningfull\foo
\edef\oof{\norelax} \meaningfull\oof
```

gives this:

macro:\relax
macro:

A `\norelax` disappears here but in the previously mentioned scenarios it has the same function as `\relax`. It will not be pushed back either in cases where a lookahead demands that.

745 \relpenalty

This internal quantity is a compatibility feature because normally we will use the inter atom spacing variables.

746 \resetlocalboxes

Its purpose should be clear from the name.

747 \resetmathspacing

This initializes all parameters to their initial values.

748 \restorecatcodetable

This is an experimental feature that should be used with care. The next example shows usage. It was added when debugging and exploring a side effect.

```
\tracingonline1

\bgroup

    \catcode`6 = 11 \catcode`7 = 11

    \bgroup

    \tracingonline1

    current: \the\catcodetable

    original: \the\catcode`6\quad \the\catcode`7

    \catcode`6 = 11 \catcode`7 = 11

    \showcodestack\catcode

    assigned: \the\catcode`6\quad \the\catcode`7

    \showcodestack\catcode

    \catcodetable\ctxcatcodes switched: \the\catcodetable

    stored: \the\catcode`6\quad \the\catcode`7

    \showcodestack\catcode

    \restorecatcodetable\ctxcatcodes

    \showcodestack\catcode

    restored: \the\catcode`6\quad \the\catcode`7

    \showcodestack\catcode

    \egroup

    \catcodetable\ctxcatcodes

    inner: \the\catcode`6\quad\the\catcode`7

\egroup

outer: \the\catcode`6\quad\the\catcode`7
```

In ConTEXt this typesets:

```
current: 9
original: 11 11
assigned: 11 11
switched: 9
```

```
stored: 11 11
restored: 12 12
inner: 11 11
outer; 12 12
```

and on the console we see:

```
3:3: [codestack 1, size 3]
3:3: [1: level 2, code 54, value 12]
3:3: [2: level 2, code 55, value 12]
3:3: [3: level 3, code 54, value 11]
3:3: [4: level 3, code 55, value 11]
3:3: [codestack 1 bottom]
3:3: [codestack 1, size 3]
3:3: [1: level 2, code 54, value 12]
3:3: [2: level 2, code 55, value 12]
3:3: [3: level 3, code 54, value 11]
3:3: [4: level 3, code 55, value 11]
3:3: [codestack 1 bottom]
3:3: [codestack 1, size 3]
3:3: [1: level 2, code 54, value 12]
3:3: [2: level 2, code 55, value 12]
3:3: [3: level 3, code 54, value 11]
3:3: [4: level 3, code 55, value 11]
3:3: [codestack 1 bottom]
3:3: [codestack 1, size 7]
3:3: [1: level 2, code 54, value 12]
3:3: [2: level 2, code 55, value 12]
3:3: [3: level 3, code 54, value 11]
3:3: [4: level 3, code 55, value 11]
3:3: [5: level 3, code 55, value 11]
3:3: [6: level 3, code 54, value 11]
3:3: [7: level 3, code 55, value 11]
3:3: [8: level 3, code 54, value 11]
3:3: [codestack 1 bottom]
3:3: [codestack 1, size 7]
3:3: [1: level 2, code 54, value 12]
3:3: [2: level 2, code 55, value 12]
3:3: [3: level 3, code 54, value 11]
3:3: [4: level 3, code 55, value 11]
3:3: [5: level 3, code 55, value 11]
3:3: [6: level 3, code 54, value 11]
3:3: [7: level 3, code 55, value 11]
3:3: [8: level 3, code 54, value 11]
3:3: [codestack 1 bottom]
```

So basically `\restorecatcodetable` brings us (temporarily) back to the global settings.

749 \retained

When a value is assigned inside a group TEX pushes the current value on the save stack in order to be able to restore the original value after the group has ended. You can reach over a group by using the `\global` prefix. A mix between local and global assignments can be achieved with the `\retained` primitive.

```
\MyDim 15pt \bgroup \the\MyDim \space
\bgroup
    \bgroup
        \bgroup \advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
    \bgroup
        \bgroup \advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
\egroup
\egroup \the\MyDim

\MyDim 15pt \bgroup \the\MyDim \space
\bgroup
    \bgroup
        \bgroup \global\advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \global\advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
    \bgroup
        \bgroup \global\advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \global\advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
\egroup
\egroup \the\MyDim

\MyDim 15pt \bgroup \the\MyDim \space
    \constrained\MyDim\zeropoint
    \bgroup
        \bgroup \retained\advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \retained\advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
    \bgroup
        \bgroup \retained\advance\MyDim10pt \the\MyDim \egroup\space
        \bgroup \retained\advance\MyDim10pt \the\MyDim \egroup\space
    \egroup
\egroup \the\MyDim
```

These lines result in:

15.0pt 25.0pt 25.0pt 25.0pt 25.0pt 15.0pt
15.0pt 25.0pt 35.0pt 45.0pt 55.0pt 55.0pt
15.0pt 10.0pt 20.0pt 30.0pt 40.0pt 15.0pt

Because LuaMetaTEX avoids redundant stack entries and reassignments this mechanism is a bit fragile but the `\constrained` prefix makes sure that we do have a stack entry. If it is needed depends on the usage pattern.

750 \retokenized

This is a companion of `\tokenized` that accepts a catcode table, so the whole repertoire is:

```
\tokenized                            {test $x$ test: current}
\tokenized   catcodetable \ctxcatcodes {test $x$ test: context}
\tokenized   catcodetable \vrbcatcodes {test $x$ test: verbatim}
\retokenized              \ctxcatcodes {test $x$ test: context}
\retokenized              \vrbcatcodes {test $x$ test: verbatim}
```

Here we pass the numbers known to ConTEXt and get:

test *x* test: current
test *x* test: context
test x test: verbatim
test *x* test: context
test x test: verbatim

751 \right

Inserts the given delimiter as right fence in a math formula.

752 \righthyphenmin

This is the minimum number of characters before the first hyphen in a hyphenated word.

753 \rightmarginkern

The dimension returned is the protrusion kern that has been added (if at all) to the left of the content in the given box.

754 \rightskip

This skip will be inserted at the right of every line.

755 \righttwindemerits

Additional demerits for a glyph sequence at the right edge when a previous line also has that sequence.

756 \romannumeral

This converts a number into a sequence of characters representing a roman numeral. Because the Romans had no zero, a zero will give no output, a fact that is sometimes used for hacks and showing off ones macro coding capabilities. A large number will for sure result in a long string because after thousand we start duplicating.

757 \rpcode

This is the companion of \lpcode (see there) and also takes three arguments: font, character code and factor.

758 \savecatcodetable

This primitive stores the currently set catcodes in the current table.

759 \savinghyphcodes

When set to non-zero, this will trigger the setting of \hjcodes from \lccodes for the current font. These codes determine what characters are taken into account when hyphenating words.

760 \savingvdiscards

When set to a positive value the page builder will store the discarded items (like glues) so that they can later be retrieved and pushed back if needed with \pagediscards or \splitdiscards.

761 \scaledemwidth

Returns the current (font specific) emwidth scaled according to \glyphscale and \glyphxscale.

762 \scaledexheight

Returns the current (font specific) exheight scaled according to \glyphscale and \glyphyscale.

763 \scaledextraspace

Returns the current (font specific) extra space value scaled according to \glyphscale and \glyphxscale.

764 \scaledfontcharba

Returns the bottom accent position of the given font-character pair scaled according to \glyphscale and \glyphyscale.

765 \scaledfontchardp

Returns the depth of the given font-character pair scaled according to \glyphscale and \glyphyscale.

766 \scaledfontcharht

Returns the height of the given font-character pair scaled according to \glyphscale and \glyphyscale.

767 `\scaledfontcharic`

Returns the italic correction of the given font-character pair scaled according to `\glyphscale` and `\glyphxscale`. This property is only real for traditional fonts.

768 `\scaledfontcharta`

Returns the top accent position of the given font-character pair scaled according to `\glyphscale` and `\glyphxscale`.

769 `\scaledfontcharwd`

Returns width of the given font-character pair scaled according to `\glyphscale` and `\glyphxscale`.

770 `\scaledfontdimen`

Returns value of a (numeric) font dimension of the given font-character pair scaled according to `\glyphscale` and `\glyphxscale` and/or `\glyphyscale`.

771 `\scaledinterwordshrink`

Returns the current (font specific) shrink of a space value scaled according to `\glyphscale` and `\glyphxscale`.

772 `\scaledinterwordspace`

Returns the current (font specific) space value scaled according to `\glyphscale` and `\glyphxscale`.

773 `\scaledinterwordstretch`

Returns the current (font specific) stretch of a space value scaled according to `\glyphscale` and `\glyphxscale`.

774 `\scaledmathaxis`

This primitive returns the math axis of the given math style. It's a dimension.

775 `\scaledmathemwidth`

Returns the emwidth of the given style scaled according to `\glyphscale` and `\glyphxscale`.

776 `\scaledmathexheight`

Returns the exheight of the given style scaled according to `\glyphscale` and `\glyphyscale`.

777 `\scaledmathstyle`

This command inserts a signal in the math list that tells how to scale the (upcoming) part of the formula.

```
$ x + {\scaledmathstyle900 x} + x$
```

We get: $x + x + x$. Of course using this properly demands integration in the macro packages font system.

778 \scaledslantperpoint

This primitive is equivalent to `\scaledfontdimen1\font` where 'scaled' means that we multiply by the glyph scales.

779 \scantextokens

This primitive scans the input as if it comes from a file. In the next examples the `\detokenize` primitive turns tokenized code into verbatim code that is similar to what is read from a file.

```
\edef\whatever{\detokenize{This is {\bf bold} and this is not.}}
\detokenize   {This is {\bf bold} and this is not.}\crlf
\scantextokens{This is {\bf bold} and this is not.}\crlf
\scantextokens{\whatever}\crlf
\scantextokens\expandafter{\whatever}\par
```

This primitive does not have the end-of-file side effects of its precursor `\scantokens`.

This is {\bf bold} and this is not.
This is **bold** and this is not.
This is {\bf bold} and this is not.
This is **bold** and this is not.

780 \scantokens

Just forget about this ε-TEX primitive, just take the one in the next section.

781 \scriptfont

This primitive is like `\font` but with a family number as (first) argument so it is specific for math. It is the middle one of the three family members; its relatives are `\textfont` and `\scriptscriptfont`.

782 \scriptscriptfont

This primitive is like `\font` but with a family number as (first) argument so it is specific for math. It is the smallest of the three family members; its relatives are `\textfont` and `\scriptfont`.

783 \scriptscriptstyle

One of the main math styles, normally one size smaller than `\scriptstyle`: integer representation: 6.

784 \scriptspace

The math engine will add this amount of space after subscripts and superscripts. It can be seen as compensation for the often too small widths of characters (in the traditional engine italic correction is used too). It prevents scripts from running into what follows.

785 \scriptspaceafterfactor

This is a (1000 based) multiplier for `\Umathspaceafterscript`.

786 \scriptspacebeforefactor

This is a (1000 based) multiplier for `\Umathspacebeforescript`.

787 \scriptspacebetweenfactor

This is a (1000 based) multiplier for `\Umathspacebetweenscript`.

788 \scriptstyle

One of the main math styles, normally one size smaller than `\displaystyle` and `\textstyle`; integer representation: 4.

789 \scrollmode

This directive omits error stops.

790 \semiexpand

This command expands the next macro when it is protected with `\semprotected`. See that primitive there for an example.

791 \semiexpanded

This command expands the tokens in the given list including the macros protected by with `\semprotected`. See that primitive there for an example.

792 \semiprotected

The working of this prefix can best be explained with an example. We define a few macros first:

```
              \def\TestA{A}
\semiprotected\def\TestB{B}
    \protected\def\TestC{C}

\edef\TestD{\TestA           \TestB           \TestC}
\edef\TestE{\TestA\semiexpand\TestB\semiexpand\TestC}
\edef\TestF{\TestA\expand    \TestB\expand    \TestC}
```

```
\edef\TestG{\normalexpanded    {\TestA\TestB\TestC}}
\edef\TestH{\normalsemiexpanded{\TestA\TestB\TestC}}
```

The meaning of the macros that are made from the other three are:

Here we use the `\normal..` variants because (currently) we still have the macro with the `\expanded` in the ConTEXt core.

```
A\TestB \TestC
AB\TestC
ABC
A\TestB \TestC
AB\TestC
```

793 \setbox

This important primitive is used to set a box register. It expects a number and a box, like `\hbox` or `\box`. There is no `\boxdef` primitive (analogue to other registers) because it makes no sense but numeric registers or equivalents are okay as register value.

794 \setdefaultmathcodes

This sets the math codes of upper- and lowercase alphabet and digits and the delimiter code of the period. It's not so much a useful feature but more just an accessor to the internal initializer.

795 \setfontid

Internally a font instance has a number and this number is what gets assigned to a glyph node. You can get the number with `\fontid` an set it with `\setfontid`.

```
\setfontid\fontid\font
```

The code above shows both primitives and effectively does nothing useful but shows the idea.

796 \setlanguage

In LuaTEX and LuaMetaTEX this is equivalent to `\language` because we carry the language in glyph nodes instead of putting triggers in the list.

797 \setmathatomrule

The math engine has some built in logic with respect to neighboring atoms that change the class. The following combinations are intercepted and remapped:

old first	old second	new first	new second
begin	binary	ordinary	ordinary
operator	binary	operator	ordinary
open	binary	open	ordinary
punctuation	binary	punctuation	ordinary

binary	end	ordinary	ordinary
binary	binary	binary	ordinary
binary	close	ordinary	close
binary	punctuation	ordinary	punctuation
binary	relation	ordinary	relation
relation	binary	relation	ordinary
relation	close	ordinary	close
relation	punctuation	ordinary	punctuation

You can change this logic if needed, for instance:

```
\setmathatomrule 1 2 \allmathstyles 1 1
```

Keep in mind that the defaults are what users expect. You might set them up for additional classes that you define but even then you probably clone an existing class and patch its properties. Most extra classes behave like ordinary anyway.

798 \setmathdisplaypostpenalty

This penalty is inserted after an item of a given class but only in inline math when display style is used, for instance:

```
\setmathdisplayprepenalty 2 750
```

799 \setmathdisplayprepenalty

This penalty is inserted before an item of a given class but only in inline math when display style is used, for instance:

```
\setmathdisplayprepenalty 2 750
```

800 \setmathignore

You can flag a math parameter to be ignored, like:

```
\setmathignore \Umathxscale             2
\setmathignore \Umathyscale             2
\setmathignore \Umathspacebeforescript  1
\setmathignore \Umathspacebetweenscript 1
\setmathignore \Umathspaceafterscript   1
```

A value of two will not initialize the variable, so its old value (when set) is kept. This is somewhat experimental and more options might show up.

801 \setmathoptions

This primitive expects a class (number) and a bitset.

0x00000001	nopreslack	0x00004000	raiseprime
0x00000002	nopostslack	0x00008000	carryoverlefttopkern
0x00000004	lefttopkern	0x00010000	carryoverrighttopkern
0x00000008	righttopkern	0x00020000	carryoverleftbottomkern
0x00000010	leftbottomkern	0x00040000	carryoverrightbottomkern
0x00000020	rightbottomkern	0x00080000	preferdelimiterdimensions
0x00000040	lookaheadforend	0x00100000	autoinject
0x00000080	noitaliccorrection	0x00200000	removeitaliccorrection
0x00000100	checkligature	0x00400000	operatoritaliccorrection
0x00000200	checkitaliccorrection	0x00800000	shortinline
0x00000400	checkkernpair	0x01000000	pushnesting
0x00000800	flatten	0x02000000	popnesting
0x00001000	omitpenalty	0x04000000	obeynesting
0x00002000	unpack		

802 \setmathpostpenalty

This penalty is inserted after an item of a given class but only in inline math when text, script or scriptscript style is used, for instance:

```
\setmathpostpenalty 2 250
```

803 \setmathprepenalty

This penalty is inserted before an item of a given class but only in inline math when text, script or scriptscript style is used, for instance:

```
\setmathprepenalty 2 250
```

804 \setmathspacing

More details about this feature can be found in ConTEXt but it boils down to registering what spacing gets inserted between a pair of classes. It can be defined per style or for a set of styles, like:

```
\inherited\setmathspacing
  \mathimplicationcode \mathbinarycode
  \alldisplaystyles \thickermuskip
\inherited\setmathspacing
  \mathradicalcode \mathmiddlecode
  \allunsplitstyles \pettymuskip
```

Here the `\inherited` prefix signals that a change in for instance `\pettymuskip` is reflected in this spacing pair. In ConTEXt there is a lot of granularity with respect to spacing and it took years of experimenting (and playing with examples) to get at the current stage. In general users are not invited to mess around too much with these values, although changing the bound registers (here `\pettymuskip` and `thickermuskip`) is no problem as it consistently makes related spacing pairs follow.

805 `\sfcode`

You can set a space factor on a character. That factor is used when a space factor is applied (as part of spacing). It is (mostly) used for adding a different space (glue) after punctuation. In some languages different punctuation has different factors.

806 `\shapingpenaltiesmode`

Shaping penalties are inserted after the lines of a `\parshape` and accumulate according to this mode, a bitset of:

`0x01`	interlinepenalty
`0x02`	widowpenalty
`0x04`	clubpenalty
`0x08`	brokenpenalty

807 `\shapingpenalty`

In order to prevent a `\parshape` to break in unexpected ways we can add a dedicated penalty, specified by this parameter.

808 `\shipout`

Because there is no backend, this is not supposed to be used. As in traditional TeX a box is grabbed but instead of it being processed it gets shown and then wiped. There is no real benefit of turning it into a callback.

809 `\shortinlinemaththreshold`

This parameter determines when an inline formula is considered to be short. This criterium is used for for `\preshortinlinepenalty` and `\postshortinlinepenalty`.

810 `\shortinlineorphanpenalty`

Short formulas at the end of a line are normally not followed by something other than punctuation. This penalty will discourage a break before a short inline formula. In practice one can set this penalty to e.g. a relatively low 200 to get the desired effect.

811 `\show`

Prints to the console (and/or log) what the token after it represents.

812 `\showbox`

The given box register is shown in the log and on te console (depending on `\tracingonline`. How much is shown depends on `\showboxdepth` and `\showboxbreadth`. In LuaMetaTeX we show more detailed information than in the other engines; some specific information is provided via callbacks.

813 \showboxbreadth

This primitives determine how much of a box is shown when asked for or when tracing demands it.

814 \showboxdepth

This primitives determine how deep tracing a box goes into the box. Some boxes, like the ones that has the assembled page.

815 \showcodestack

This inspector is only useful for low level debugging and reports the current state of for instance the current catcode table: **\showcodestack\catcode**. See \restorecatcodes for an example.

816 \showgroups

This primitive reports the group nesting. At this spot we have a not so impressive nesting:

```
2:3: simple group entered at line 9375:
1:3: semisimple group: \begingroup
0:3: bottomlevel
```

817 \showifs

This primitive will show the conditional stack in the log file or on the console (assuming \tracingonline being non-zero). The shown data is different from other engines because we have more conditionals and also support a more flat nesting model

818 \showlists

This shows the currently built list.

819 \shownodedetails

When set to a positive value more details will be shown of nodes when applicable. Values larger than one will also report attributes. What gets shown depends on related callbacks being set.

820 \showstack

This tracer is only useful for low level debugging of macros, for instance when you run out of save space or when you encounter a performance hit.

```
  test\scratchcounter0 \showstack
 {test\scratchcounter1 \showstack}
{{test\scratchcounter1 \showstack}}
```

reports

```
1:3: [savestack size 0]
1:3: [savestack bottom]
```

```
2:3: [savestack size 2]
2:3: [1: restore, level 1, cs \scratchcounter=integer 1]
2:3: [0: boundary, group 'bottomlevel', boundary 0, attrlist 3600, line 0]
2:3: [savestack bottom]

3:3: [savestack size 3]
3:3: [2: restore, level 1, cs \scratchcounter=integer 1]
3:3: [1: boundary, group 'simple', boundary 0, attrlist 3600, line 12]
3:3: [0: boundary, group 'bottomlevel', boundary 0, attrlist 3600, line 0]
3:3: [savestack bottom]
```

while

```
  test\scratchcounter1 \showstack
 {test\scratchcounter1 \showstack}
{{test\scratchcounter1 \showstack}}
```

shows this:

```
1:3: [savestack size 0]
1:3: [savestack bottom]

2:3: [savestack size 1]
2:3: [0: boundary, group 'bottomlevel', boundary 0, attrlist 3600, line 0]
2:3: [savestack bottom]

3:3: [savestack size 2]
3:3: [1: boundary, group 'simple', boundary 0, attrlist 3600, line 16]
3:3: [0: boundary, group 'bottomlevel', boundary 0, attrlist 3600, line 0]
3:3: [savestack bottom]
```

Because in the second example the value of `\scratchcounter` doesn't really change inside the group there is no need for a restore entry on the stack. In LuaMetaTEX there are checks for that so that we consume less stack space. We also store some states (like the line number and current attribute list pointer) in a stack boundary.

821 `\showthe`

Prints to the console (and/or log) the value of token after it.

822 `\showtokens`

This command expects a (balanced) token list, like

```
\showtokens{a few tokens}
```

Depending on what you want to see you need to expand:

```
\showtokens\expandafter{\the\everypar}
```

which is equivalent to `\showthe\everypar`. It is an ε-TEX extension.

823 \singlelinepenalty

This is a penalty that gets injected before a paragraph that has only one line. It is a one-shot parameter, so like `\looseness` it only applies to the upcoming (or current) paragraph.

824 \skewchar

This is an (imaginary) character that is used in math fonts. The kerning pair between this character and the current one determines the top anchor of a possible accent. In OpenType there is a dedicated character property for this (but for some reason not for the bottom anchor).

825 \skip

This is the accessor for an indexed skip (glue) register.

826 \skipdef

This command associates a control sequence with a skip register (accessed by number).

827 \snapshotpar

There are many parameters involved in typesetting a paragraph. One complication is that parameters set in the middle might have unpredictable consequences due to grouping, think of:

```
text  text <some setting> text   text \par
text {text <some setting> text } text \par
```

This makes in traditional TEX because there is no state related to the current paragraph. But in LuaTEX we have the initial so called par node that remembers the direction as well as local boxes. In LuaMetaTEX we store way more when this node is created. That means that later settings no longer replace the stored ones.

The `\snapshotpar` takes a bitset that determine what stored parameters get updated to the current values.

`0x00000001`	hsize	`0x00000400`	lastline	`0x00100000`	shapingpenalty
`0x00000002`	skip	`0x00000800`	linepenalty	`0x00200000`	orphanpenalty
`0x00000004`	hang	`0x00001000`	clubpenalty	`0x00400000`	toddlerpenalty
`0x00000008`	indent	`0x00002000`	widowpenalty	`0x00800000`	emergency
`0x00000010`	parfill	`0x00004000`	displaypenalty	`0x01000000`	parpasses
`0x00000020`	adjust	`0x00008000`	brokenpenalty	`0x02000000`	singlelinepenalty
`0x00000040`	protrude	`0x00010000`	demerits	`0x04000000`	hyphenpenalty
`0x00000080`	tolerance	`0x00020000`	shape	`0x08000000`	exhyphenpenalty
`0x00000100`	stretch	`0x00040000`	line	`0x10000000`	linebreakchecks
`0x00000200`	looseness	`0x00080000`	hyphenation	`0x20000000`	twindemerits

One such value covers multiple values, so for instance `skip` is good for storing the current `\leftskip` and `\rightskip` values. More about this feature can be found in the ConTEXt documentation.

The list of parameters that gets reset after a paragraph is longer than for pdfTEX and LuaMetaTEX: \emergencyleftskip, \emergencyrightskip, \hangafter, \hangindent, \interlinepenalties, \localbrokenpenalty, \localinterlinepenalty, \localpretolerance, \localtolerance, \looseness, \parshape and \singlelinepenalty.

828 \spacechar

When \nospaces is set to 3 a glyph node with the character value of this parameter is injected.

829 \spacefactor

The space factor is a somewhat complex feature. When during scanning a character is appended that has a \sfcode other than 1000, that value is saved. When the time comes to insert a space triggered glue, and that factor is 2000 or more, and when \xspaceskip is nonzero, that value is used and we're done.

If these criteria are not met, and \spaceskip is nonzero, that value is used, otherwise the space value from the font is used. Now, it if the space factor is larger than 2000 the extra space value from the font is added to the set value. Next the engine is going to tweak the stretch and shrink if that value and in LuaMetaTEX that can be done in different ways, depending on \spacefactormode, \spacefactorstretchlimit and \spacefactorshrinklimit.

First the stretch. When the set limit is 1000 or more and the saved space factor is also 1000 or more, we multiply the stretch by the limit, otherwise the saved space factor is used.

Shrink is done differently. When the shrink limit and space factor are both 1000 or more, we will scale the shrink component by the limit, otherwise we multiply by the saved space factor but here we have three variants, determined by the value of \spacefactormode.

In the first case, when the limit kicks in, a mode value 1 will multiply by limit and divides by 1000. A value of 2 multiplies by 2000 and divides by the limit. Other mode values multiply by 1000 and divide by the limit. When the limit is not used, the same happens but with the saved space factor.

If this sounds complicated, here is what regular TEX does: stretch is multiplied by the factor and divided by 1000 while shrink is multiplied by 1000 and divided by the saved factor. The (new) mode driven alternatives are the result of extensive experiments done in the perspective of enhancing the rendering of inline math as well as additional par builder passes. For sure alternative strategies are possible and we can always add more modes.

A better explanation of the default strategy around spaces can be found in (of course) The TEXbook and TEX by Topic.

830 \spacefactormode

Its setting determines the way the glue components (currently only shrink) adapts itself to the current space factor (determined by by the character preceding a space).

831 \spacefactoroverload

When set to value between zero and thousand, this value will be used when TEX encounters a below thousand space factor situation (usually used to suppress additional space after a period following an uppercase character which then gets (often) a 999 space factor. This feature only kicks in when the overload flag is set in the glyph options, so it can be applied selectively.

832 \spacefactorshrinklimit

This limit is used when \spacefactormode is set. See \spacefactor for a bit more explanation.

833 \spacefactorstretchlimit

This limit is used when \spacefactormode is set. See \spacefactor for a bit more explanation.

834 \spaceskip

Normally the glue inserted when a space is encountered is taken from the font but this parameter can overrule that.

835 \span

This primitive combined two upcoming cells into one. Often it is used in combination with \omit. However, in the preamble it forces the next token to be expanded, which means that nested \tabskips and align content markers are seen.

836 \specificationdef

There are some datastructures that are like arrays: \adjacentdemerits, \brokenpenalties, \clubpenalties, \displaywidowpenalties, \fitnessclasses, \interlinepenalties, \mathbackwardpenalties, \mathforwardpenalties, \orphanpenalties, \parpasses, \parshape and \widowpenalties. They accept a counter than tells how many entries follow and depending in the specification options, keywords and/or just values are expected.

With \specificationdef you can define a command that holds such an array and that can be used afterwards as a fast way to enable that specification. The way it work is as follows:

```
\specificationdef\MyWidowPenalties
    \widowpenalties 4 2000 1000 500 250
\relax
```

where the relax is optional but a reasonabel way to make sure we end the definition (when keywords are used, as in \parpasses it prevents running into unexpected keywords.

837 \splitbotmark

This is a reference to the last mark on the currently split off box, it gives back tokens.

838 \splitbotmarks

This is a reference to the last mark with the given id (a number) on the currently split off box, it gives back tokens.

839 \splitdiscards

When a box is split off, items like glue are discarded. This internal register keeps the that list so that it can be pushed back if needed.

840 \splitfirstmark

This is a reference to the first mark on the currently split off box, it gives back tokens.

841 \splitfirstmarks

This is a reference to the first mark with the given id (a number) on the currently split off box, it gives back tokens.

842 \splitmaxdepth

The depth of the box that results from a `\vsplit`.

843 \splittopskip

This is the amount of glue that is added to the top of a (new) split of part of a box when `\vsplit` is applied.

844 \srule

This inserts a rule with no width. When a `font` and a `char` are given the height and depth of that character are taken. Instead of a font `fam` is also accepted so that we can use it in math mode.

845 \string

We mention this original primitive because of the one in the next section. It expands the next token or control sequence as if it was just entered, so normally a control sequence becomes a backslash followed by characters and a space.

846 \subprescript

Instead of three or four characters with catcode 8 (`___` or `____`) this primitive can be used. It will add the following argument as lower left script to the nucleus.

847 \subscript

Instead of one or two characters with catcode 7 (`_` or `__`) this primitive can be used. It will add the following argument as upper left script to the nucleus.

848 \superprescript

Instead of three or four characters with catcode 7 (`^^^` or `^^^^`) this primitive can be used. It will add the following argument as upper left script to the nucleus.

849 \superscript

Instead of one or two character with catcode 7 (^ or ^^)t his primitive can be used. It will add the following argument as upper right script to the nucleus.

850 \supmarkmode

As in other languages, TEX has ways to escape characters and get whatever character needed into the input. By default multiple ^ are used for this. The dual ^^ variant is a bit weird as it is not continuous but ^^^^ and ^^^^^^ provide four or six byte hexadecimal references ot characters. The single ^ is also used for superscripts but because we support prescripts and indices we get into conflicts with the escapes.

When this internal quantity is set to zero, multiple ^'s are interpreted in the input and produce characters. Other values disable the multiple parsing in text and/or math mode:

```
\normalsupmarkmode0 $ X^58 \quad X^^58 $ ^^58
\normalsupmarkmode1 $ X^58 \quad X^^58 $ ^^58
\normalsupmarkmode2 $ X^58 \quad X^^58 $ % ^^58 : error
```

In ConTEXt we default to one but also have the `\catcode` set to 12and the `\amcode` to 7.

$X^5 8 \quad X X$ X
$X^5 8 \quad X^5 8$ X
$X^5 8 \quad X^5 8$

851 \swapcsvalues

Because we mention some `def` and `let` primitives here, it makes sense to also mention a primitive that will swap two values (meanings). This one has to be used with care. Of course that what gets swapped has to be of the same type (or at least similar enough not to cause issues). Registers for instance store their values in the token, but as soon as we are dealing with token lists we also need to keep an eye on reference counting. So, to some extend this is an experimental feature.

```
\scratchcounterone 1 \scratchcountertwo 2
(\the\scratchcounterone,\the\scratchcountertwo)
\swapcsvalues \scratchcounterone \scratchcountertwo
(\the\scratchcounterone,\the\scratchcountertwo)
\swapcsvalues \scratchcounterone \scratchcountertwo
(\the\scratchcounterone,\the\scratchcountertwo)
```

```
\scratchcounterone 3 \scratchcountertwo 4
(\the\scratchcounterone,\the\scratchcountertwo)
\bgroup
\swapcsvalues \scratchcounterone \scratchcountertwo
(\the\scratchcounterone,\the\scratchcountertwo)
\egroup
(\the\scratchcounterone,\the\scratchcountertwo)
```

We get similar results:

(1,2)

(2,1)
(1,2)

(3,4)
(4,3)
(3,4)

852 \tabsize

This primitive can be used in the preamble of an alignment and sets the size of a column, as in:

```
\halign{%
    \aligncontent              \aligntab
    \aligncontent\tabsize 3cm \aligntab
    \aligncontent              \aligntab
    \aligncontent\tabsize 0cm \cr
    1  \aligntab 111\aligntab 1111\aligntab 11\cr
    222\aligntab 2  \aligntab 2222\aligntab 22\cr
}
```

As with `\tabskip` you need to reset the value explicitly, so that is why we get two wide columns:

1	111	1111	11
222	2	2222	22

853 \tabskip

This traditional primitive can be used in the preamble of an alignment and sets the space added between columns, for example:

```
\halign{%
    \aligncontent              \aligntab
    \aligncontent\tabskip 3cm \aligntab
    \aligncontent              \aligntab
    \aligncontent\tabskip 0cm \cr
    1  \aligntab 111\aligntab 1111\aligntab 11\cr
    222\aligntab 2  \aligntab 2222\aligntab 22\cr
}
```

You need to reset the skip explicitly, which is why we get it applied twice here:

1	111	1111	11
222	2	2222	22

854 \textdirection

This set the text direction to `l2r` (0) or `r2l` (1). It also triggers additional checking for balanced flipping in node lists.

855 `\textfont`

This primitive is like `\font` but with a family number as (first) argument so it is specific for math. It is the largest one of the three family members; its relatives are `\scriptfont` and `\scriptscriptfont`.

856 `\textstyle`

One of the main math styles; integer representation: 2.

857 `\the`

The `\the` primitive serializes the following token, when applicable: integers, dimensions, token registers, special quantities, etc. The catcodes of the result will be according to the current settings, so in `\the\dimen0`, the `pt` will have catcode 'letter' and the number and period will become 'other'.

858 `\thewithoutunit`

The `\the` primitive, when applied to a dimension variable, adds a `pt` unit. because dimensions are the only traditional unit with a fractional part they are sometimes used as pseudo floats in which case `\thewithoutunit` can be used to avoid the unit. This is more convenient than stripping it off afterwards (via an expandable macro).

859 `\thickmuskip`

A predefined mu skip register that can be used in math (inter atom) spacing. The current value is `5.0mu plus 3.0mu minus 1.0mu`. In traditional TEX most inter atom spacing is hard coded using the predefined registers.

860 `\thinmuskip`

A predefined mu skip register that can be used in math (inter atom) spacing. The current value is `3.0mu`. In traditional TEX most inter atom spacing is hard coded using the predefined registers.

861 `\time`

This internal number starts out with minute (starting at midnight) that the job started.

862 `\tinymuskip`

A predefined mu skip register that can be used in math (inter atom) spacing. The current value is `2.0mu minus 1.0mu`. This one complements `\thinmuskip`, `\medmuskip`, `\thickmuskip` and the new `\pettymuskip`

863 `\tocharacter`

The given number is converted into an utf-8 sequence. In LuaTEX this one is named **`\Uchar`**.

864 \toddlerpenalties

This an (possible double entry) array parameter: first the size is given followed by that amount of penalties (can be pairs). These penalties are injected after (and before) single glyphs bounded by spaces, going backward from the end of a sequence of them.

865 \todimension

The following code gives this: 1234.0pt and like its numeric counterparts accepts anything that resembles a number this one goes beyond (user, internal or pseudo) registers values too.

```
\scratchdimen = 1234pt \todimension\scratchdimen
```

866 \tohexadecimal

The following code gives this: 4D2 with uppercase letters.

```
\scratchcounter = 1234 \tohexadecimal\scratchcounter
```

867 \tointeger

The following code gives this: 1234 and is equivalent to `\number`.

```
\scratchcounter = 1234 \tointeger\scratchcounter
```

868 \tokenized

Just as `\expanded` has a counterpart `\unexpanded`, it makes sense to give `\detokenize` a companion:

```
\edef\foo{\detokenize{\inframed{foo}}}
\edef\oof{\detokenize{\inframed{oof}}}

\meaning\foo \crlf \dontleavehmode\foo

\edef\foo{\tokenized{\foo\foo}}

\meaning\foo \crlf \dontleavehmode\foo

\dontleavehmode\tokenized{\foo\oof}
```

```
macro:\inframed {foo}
\inframed {foo}

macro:\inframed {foo}\inframed {foo}
```

$\boxed{\texttt{foo}}\boxed{\texttt{foo}}$

$\boxed{\texttt{foo}}\boxed{\texttt{foo}}\boxed{\texttt{oof}}$

This primitive is similar to:

```
\def\tokenized#1{\scantextokens\expandafter{\normalexpanded{#1}}}
```

and should be more efficient, not that it matters much as we don't use it that much (if at all).

869 \toks

This is the accessor of a token register so it expects a number or `\toksdef`'d macro.

870 \toksapp

One way to append something to a token list is the following:

```
\scratchtoks\expandafter{\the\scratchtoks more stuff}
```

This works all right, but it involves a copy of what is already in `\scratchtoks`. This is seldom a real issue unless we have large token lists and many appends. This is why LuaTEX introduced:

```
\toksapp\scratchtoks{more stuff}
\toksapp\scratchtoksone\scratchtokstwo
```

At some point, when working on LuaMetaTEX, I realized that primitives like this one and the next appenders and prependers to be discussed were always on the radar of Taco and me. Some were even implemented in what we called `eetex`: extended ε-TEX, and we even found back the prototypes, dating from pre-pdfTEX times.

871 \toksdef

The given name (control sequence) will be bound to the given token register (a number). Often this primitive is hidden in a high level macro that manages allocation.

872 \tokspre

Where appending something is easy because of the possible `\expandafter` trickery a prepend would involve more work, either using temporary token registers and/or using a mixture of the (no)expansion added by ε-TEX, but all are kind of inefficient and cumbersome.

```
\tokspre\scratchtoks{less stuff}
\tokspre\scratchtoksone\scratchtokstwo
```

This prepends the token list that is provided.

873 \tolerance

When the par builder runs into a line with a badness larger than this value and when `\emergencystretch` is set a third pass is enabled. In LuaMetaTEX we can have more than one second pass and there are more parameters that influence the process.

874 \tolerant

This prefix tags the following macro as being tolerant with respect to the expected arguments. It only makes sense when delimited arguments are used or when braces are mandate.

```
\tolerant\def\foo[#1]#*[#2]{(#1)(#2)}
```

This definition makes `\foo` tolerant for various calls:

```
\foo \foo[1] \foo [1] \foo[1] [2] \foo [1] [2]
```

these give: ()()(1)()(1)()(1)(2) (1)(2). The spaces after the first call disappear because the macro name parser gobbles it, while in the second case the `#*` gobbles them. Here is a variant:

```
\tolerant\def\foo[#1]#,[#2]{!#1!#2!}
```

```
\foo[?] x
\foo[?] [?] x
```

```
\tolerant\def\foo[#1]#*[#2]{!#1!#2!}
```

```
\foo[?] x
\foo[?] [?] x
```

We now get the following:

!?!! x !?!?! x

!?!!x !?!?! x

Here the `#,` remembers that spaces were gobbles and they will be put back when there is no further match. These are just a few examples of this tolerant feature. More details can be found in the lowlevel manuals.

875 \tomathstyle

Internally math styles are numbers, where `\displaystyle` is 0 and `\crampedscriptscriptstyle` is 7. You can convert the verbose style to a number with `\tomathstyle`.

876 \topmark

This is a reference to the last mark on the previous (split off) page, it gives back tokens.

877 \topmarks

This is a reference to the last mark with the given id (a number) on the previous page, it gives back tokens.

878 \topskip

This is the amount of glue that is added to the top of a (new) page.

879 \toscaled

The following code gives this: 1234.0 is similar to `\todimension` but omits the `pt` so that we don't need to revert to some nasty stripping code.

```
\scratchdimen = 1234pt \toscaled\scratchdimen
```

880 \tosparsedimension

The following code gives this: 1234pt where ‘sparse’ indicates that redundant trailing zeros are not shown.

```
\scratchdimen = 1234pt \tosparsedimension\scratchdimen
```

881 \tosparsescaled

The following code gives this: 1234 where ‘sparse’ means that redundant trailing zeros are omitted.

```
\scratchdimen = 1234pt \tosparsescaled\scratchdimen
```

882 \tpack

This primitive is like `\vtop` but without the callback overhead.

883 \tracingadjusts

In LuaMetaTEX the adjust feature has more functionality and also is carried over. When set to a positive values `\vadjust` processing reports details. The higher the number, the more you'll get.

884 \tracingalignments

When set to a positive value the alignment mechanism will keep you informed about what is done in various stages. Higher values unleash more information, including what callbacks kick in.

885 \tracingassigns

When set to a positive values assignments to parameters and variables are reported on the console and/or in the log file. Because LuaMetaTEX avoids redundant assignments these don't get reported.

886 \tracingcommands

When set to a positive values the commands (primitives) are reported on the console and/or in the log file.

887 \tracingexpressions

The extended expression commands like `\numexpression` and `\dimexpression` can be traced by setting this parameter to a positive value.

888 \tracingfitness

Because we have more fitness classes we also have (need) a (bit) more detailed tracing.

889 \tracingfullboxes

When set to a positive value the box will be shown in case of an overfull box. When a quality callback is set this will not happen as all reporting is then delegated.

890 \tracinggroups

When set to a positive values grouping is reported on the console and/or in the log file.

891 \tracinghyphenation

When set to a positive values the hyphenation process is reported on the console and/or in the log file.

892 \tracingifs

When set some details of what gets tested and what results are seen is reported.

893 \tracinginserts

A positive value enables tracing where values larger than 1 will report more details.

894 \tracinglevels

The lines in a log file can be prefixed with some details, depending on the bits set:

0x1	current group
0x2	current input
0x4	catcode table

895 \tracinglists

At various stages the lists being processed can be shown. This is mostly an option for developers.

896 \tracingloners

With loners we mean 'widow' and 'club' lines. This tracer can be handy when `\doublepenaltymode` is set and facing pages have different penalty values.

897 \tracinglooseness

This tracer reports some details about the decision made towards a possible loose result.

898 \tracinglostchars

When set to one characters not present in a font will be reported in the log file, a value of two will also report this on the console.

899 \tracingmacros

This parameter controls reporting of what macros are seen and expanded.

900 \tracingmarks

Marks are information blobs that track states that can be queried when a page is handled over to the shipout routine. They travel through the system in a bit different than traditionally: like like adjusts and inserts deeply buried ones bubble up to outer level boxes. This parameters controls what progress gets reported.

901 `\tracingmath`

The higher the value, the more information you will get about the various stages in rendering math. Because tracing of nodes is rather verbose you need to know a bit what this engine does. Conceptually there are differences between the LuaMetaTEX and traditional engine, like more passes, inter-atom spacing, different low level mechanisms. This feature is mostly meant for developers who tweak the many available parameters.

902 `\tracingnesting`

A positive value triggers log messages about the current level.

903 `\tracingnodes`

When set to a positive value more details about nodes (in boxes) will be reported. Because this is also controlled by callbacks what gets reported is macro package dependent.

904 `\tracingonline`

The engine has two output channels: the log file and the console and by default most tracing (when enabled) goes to the log file. When this parameter is set to a positive value tracing will also happen in the console. Messages from the Lua end can be channeled independently.

905 `\tracingoutput`

Values larger than one result in some information about what gets passed to the output routine.

906 `\tracingpages`

Values larger than one result in some information about the page building process. In LuaMetaTEX there is more info for higher values.

907 `\tracingparagraphs`

Values larger than one result in some information about the par building process. In LuaMetaTEX there is more info for higher values.

908 `\tracingpasses`

In LuaMetaTEX you can configure additional second stage par builder passes and this parameter controls what gets reported on the console and/or in the log file.

909 `\tracingpenalties`

This setting triggers reporting of actions due to special penalties in the page builder.

910 \tracingrestores

When set to a positive values (re)assignments after grouping to parameters and variables are reported on the console and/or in the log file. Because LuaMetaTEX avoids redundant assignments these don't get reported.

911 \tracingstats

This parameter is a dummy in LuaMetaTEX. There are anyway some statistic reported when the format is made but for a regular run it is up to the macro package to come up with useful information.

912 \tsplit

This splits like `\vsplit` but it returns a `\vtop` box instead.

913 \uccode

When the `\uppercase` operation is applied the uppercase code of a character is used for the replacement. This primitive is used to set that code, so it expects two character number.

914 \uchyph

When set to a positive number words that start with a capital will be hyphenated.

915 \uleaders

This leader adapts itself after a paragraph has been typeset. Here are a few examples:

```
test \leaders  \hbox      {x}\hfill\               test
test \uleaders \hbox{x x x x}\hfill\               test
test           \hbox{x x x x}\hskip 3cm plus 1cm\ test
test \uleaders \hbox{x x x x}\hskip 3cm plus 1cm\ test
```

When an `\uleaders` is used the glue in the given box will be adapted to the available space.

test xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx test
test x x x x test
test x x x x test
test x x x x test

Optionally the `callback` followed by a number can be given, in which case a callback kicks in that gets that the node, a group identifier, and the number passed. It permits (for instance) adaptive graphics: 1=i 6=vi 11=xi 16=xvi 21=xxi 26=xxvi 31=xxxi 36=xxxvi 41=xli 46=xlvi 51=li 56=lvi 61=lxi 66=lxvi 71=lxxi 76=lxxvi 81=lxxxi 86=lxxxvi 91=xci 96=xcvi .

These `\uleaders` can be used in horizontal and vertical mode so we give a few more examples.

```
\unexpandedloop 1 30 1 {x             \hbox{1 2 3}
  }
```

```
\unexpandedloop 1 30 1 {x {\uleaders \hbox{1 2 3}\hskip 0pt plus 10pt           minus
  10pt\relax}        x }
\unexpandedloop 1 30 1 {x {\uleaders \hbox{1 2 3}\hskip 0pt plus  \interwordstretch
  minus \interwordshrink}  x }
\unexpandedloop 1 30 1 {x {\uleaders \hbox{1 2 3}\hskip 0pt plus 2\interwordstretch
  minus 2\interwordshrink} x }
```

This renders as:

x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x
1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x
1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x
x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x
x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x
x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x
x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x
x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x
x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x
x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x
x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x
x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x x 1 2 3 x

It is clear that the flexibility of the box plays a role in the line break calculations. But in the end the backend has to do the work which is why it's a ‘user’ leader. Here is an example of a vertical one. Compare:

```
{\green \hrule width \hsize} \par \vskip2pt
\vbox to 40pt {
    {\red\hrule width \hsize} \par \vskip2pt
    \vbox {
        \vskip2pt {\blue\hrule width \hsize} \par
        \vskip 10pt plus 10pt minus 10pt
        {\blue\hrule width \hsize} \par \vskip2pt
    }
    \vskip2pt {\red\hrule width \hsize} \par
}
\vskip2pt {\green \hrule width \hsize} \par
```

with:

```
{\green \hrule width \hsize} \par \vskip2pt
\vbox to 40pt {
    {\red\hrule width \hsize} \par \vskip2pt
    \uleaders\vbox {
        \vskip2pt {\blue\hrule width \hsize} \par
        \vskip 10pt plus 10pt minus 10pt
        {\blue\hrule width \hsize} \par \vskip2pt
    }\vskip 0pt plus 10pt minus 10pt
    \vskip2pt {\red\hrule width \hsize} \par
}
\vskip2pt {\green \hrule width \hsize} \par
```

In the first case we get the this:

but with `\uleaders` we get:

or this:

In the second case we flatten the leaders in the engine by setting the second bit in the `\normalizeparmode` parameter (`0x2`). We actually do the same with `\normalizelinemode` where bit 10 is set (`0x200`). The `delay` keyword can be passed with a box to prevent flattening. If we don't do this in the engine, the backend has to take care of it. In principle this permits implementing variants in a macro package. Eventually there will be plenty examples in the ConTEXt code base and documentation. Till then, consider this experimental.

916 \unboundary

When possible a preceding boundary node will be removed.

917 \undent

When possible the already added indentation will be removed.

918 \underline

This is a math specific primitive that draws a line under the given content. It is a poor mans replacement for a delimiter. The thickness is set with `\Umathunderbarrule`, the distance between content and rule is set by `\Umathunderbarvgap` and `\Umathunderbarkern` is added above the rule. The style used for the content under the rule can be set with `\Umathunderlinevariant`. See `\overline` for what these parameters do.

919 \unexpanded

This is an ε-TEX enhancement. The content will not be expanded in a context where expansion is happening, like in an `\edef`. In ConTEXt you need to use `\normalunexpanded` because we already had a macro with that name.

```
\def \A{!}                          \meaning\A
\def \B{?}                          \meaning\B
\edef\C{\A\B}                       \meaning\C
```

```
\edef\C{\normalunexpanded{\A}\B} \meaning\C
```

```
macro:!
macro:?
macro:!?
macro:\A ?
```

920 \unexpandedendless

This one loops forever so you need to quit explicitly.

921 \unexpandedloop

As follow up on `\expandedloop` we now show its counterpart:

```
\edef\whatever
  {\unexpandedloop 1 10 1
     {\scratchcounter=\the\currentloopiterator\relax}}
```

```
\meaningasis\whatever
```

```
\def \whatever {\scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter
=0\relax \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax \scratchcounter =0\relax
\scratchcounter =0\relax \scratchcounter =0\relax }
```

The difference between the (un)expanded loops and a local controlled one is shown here. Watch the out of order injection of A's.

```
\edef\TestA{\localcontrolledloop 1 5 1 {A}} % out of order
\edef\TestB{\expandedloop        1 5 1 {B}}
\edef\TestC{\unexpandedloop      1 5 1 {C\relax}}
```

AAAAA

We show the effective definition as well as the outcome of using them

```
\meaningasis\TestA
\meaningasis\TestB
\meaningasis\TestC

A: \TestA
B: \TestB
C: \TestC
```

```
\def \TestA {}
\def \TestB {BBBBB}
\def \TestC {C\relax C\relax C\relax C\relax C\relax }
```

```
A:
B: BBBBB
C: CCCCC
```

Watch how because it is empty \TestA has become a constant macro because that's what deep down empty boils down to.

922 \unexpandedrepeat

This one takes one instead of three arguments which looks better in simple loops.

923 \unhbox

A box is a packaged list and once packed travels through the system as a single object with properties, like dimensions. This primitive injects the original list and discards the wrapper.

924 \unhcopy

This is like \unhbox but keeps the original. It is one of the more costly operations.

925 \unhpack

This primitive is like \unhbox but without the callback overhead.

926 \unkern

This removes the last kern, if possible.

927 \unless

This ε-TEX prefix will negate the test (when applicable).

```
       \ifx\one\two YES\else NO\fi
\unless\ifx\one\two NO\else YES\fi
```

This primitive is hardly used in ConTEXt and we probably could get rid of these few cases.

928 \unletfrozen

A frozen macro cannot be redefined: you get an error. But as nothing in TEX is set in stone, you can do this:

```
\frozen\def\MyMacro{...}
\unletfrozen\MyMacro
```

and \MyMacro is no longer protected from overloading. It is still undecided to what extend ConTEXt will use this feature.

929 \unletprotected

The complementary operation of \letprotected can be used to unprotect a macro, so that it gets expandable.

```
                \def  \MyMacroA{alpha}
\protected      \def  \MyMacroB{beta}
                \edef \MyMacroC{\MyMacroA\MyMacroB}
\unletprotected       \MyMacroB
                \edef \MyMacroD{\MyMacroA\MyMacroB}
\meaning              \MyMacroC\crlf
\meaning              \MyMacroD\par
```

Compare this with the example in the previous section:

```
macro:alpha\MyMacroB
macro:alphabeta
```

930 \unpenalty

This removes the last penalty, if possible.

931 \unskip

This removes the last glue, if possible.

932 \untraced

Related to the meaning providers is the `\untraced` prefix. It marks a macro as to be reported by name only. It makes the macro look like a primitive.

```
         \def\foo{}
\untraced\def\oof{}

\scratchtoks{\foo\foo\oof\oof}

\tracingall \the\scratchtoks \tracingnone
```

This will show up in the log as follows:

```
1:4: {\the}
1:5: \foo ->
1:5: \foo ->
1:5: \oof
1:5: \oof
```

This is again a trick to avoid too much clutter in a log. Often it doesn't matter to users what the meaning of a macro is (if they trace at all).[11]

933 \unvbox

A box is a packaged list and once packed travels through the system as a single object with properties, like dimensions. This primitive injects the original list and discards the wrapper.

[11] An earlier variant could also hide the expansion completely but that was just confusing.

934 \unvcopy

This is like `\unvbox` but keeps the original. It is one of the more costly operations.

935 \unvpack

This primitive is like `\unvbox` but without the callback overhead.

936 \uppercase

See its counterpart `\lowercase` for an explanation.

937 \vadjust

This injects a node that stores material that will injected before or after the line where it has become part of. In LuaMetaTEX there are more features, driven by keywords.

938 \valign

This command starts vertically aligned material. Its counterpart `\halign` is used more frequently. Most macro packages provide wrappers around these commands. First one specifies a preamble which is then followed by entries (rows and columns).

939 \variablefam

In traditional TEX sets the family of what are considered variables (class 7) to the current family (which often means that they adapt to the current alphabet) and then injects a math character of class ordinary. This parameter can be used to obey the given class when the family set for a character is the same as this parameter. So we then use the given class with the current family. It is mostly there for compatibility with LuaTEX and experimenting (outside ConTEXt).

940 \vbadness

This sets the threshold for reporting a (vertical) badness value, its current value is 0.

941 \vbox

This creates a vertical box. In the process callbacks can be triggered that can preprocess the content, influence line breaking as well as assembling the resulting paragraph. More can be found in dedicated manuals. The baseline is at the bottom.

942 \vcenter

In traditional TEX this box packer is only permitted in math mode but in LuaMetaTEX it also works in text mode. The content is centered in the vertical box.

943 \vfil

This is a shortcut for **`\vskip`** `plus 1 fil` (first order filler).

944 \vfill

This is a shortcut for **\vskip** `plus 1 fill` (second order filler).

945 \vfilneg

This is a shortcut for **\vskip** `plus - 1 fil` so it can compensate `\vfil`.

946 \vfuzz

This dimension sets the threshold for reporting vertical boxes that are under- or overfull. The current value is 0.1pt.

947 \virtualhrule

This is a horizontal rule with zero dimensions from the perspective of the frontend but the backend can access them as set.

948 \virtualvrule

This is a vertical rule with zero dimensions from the perspective of the frontend but the backend can access them as set.

949 \vkern

This primitive is like `\kern` but will force the engine into vertical mode if it isn't yet.

950 \vpack

This primitive is like `\vbox` but without the callback overhead.

951 \vpenalty

This primitive is like `\penalty` but will force the engine into vertical mode if it isn't yet.

952 \vrule

This creates a vertical rule. Unless the height and depth are set they will stretch to fix the available space. In addition to the traditional `width`, `height` and `depth` specifiers some more are accepted. These are discussed in other manuals. See `\hrule` for a simple example.

953 \vsize

This sets (or gets) the current vertical size. While setting the `\hsize` inside a `\vbox` has consequences, setting the `\vsize` mostly makes sense at the outer level (the page).

954 \vskip

The given glue is injected in the vertical list. If possible vertical mode is entered.

955 \vsplit

This operator splits a given amount from a vertical box. In LuaMetaTEX we can split `to` but also `upto`, so that we don't have to repack the result in order to see how much is actually in there.

956 \vss

This is the vertical variant of `\hss`. See there for what it means.

957 \vtop

This creates a vertical box. In the process callbacks can be triggered that can preprocess the content, influence line breaking as well as assembling the resulting paragraph. More can be found in dedicated manuals. The baseline is at the top.

958 \wd

Returns the width of the given box.

959 \widowpenalties

This is an array of penalty put before the last lines in a paragraph. High values discourage (or even prevent) a lone line at the beginning of a next page. This command expects a count value indicating the number of entries that will follow. The first entry is ends up before the last line.

960 \widowpenalty

This is the penalty put before a widow line in a paragraph. High values discourage (or even prevent) a lone line at the beginning of a next page.

961 \wordboundary

The hypenation routine has to decide where a word begins and ends. If you want to make sure that there is a proper begin or end of a word you can inject this boundary.

962 \wrapuppar

What this primitive does can best be shown with an example:

```
some text\wrapuppar{one} and some\wrapuppar{two} more
```

We get:

some text and some more twoone

So, it is a complementary command to `\everypar`. It can only be issued inside a paragraph.

963 \xdef

This is an alternative for **`\global\edef`**:

```
\xdef\MyMacro{...}
```

964 \xdefcsname

This is the companion of `\xdef`:

```
\expandafter\xdef\csname MyMacro:1\endcsname{...}
             \xdefcsname MyMacro:1\endcsname{...}
```

965 \xleaders

See `\gleaders` for an explanation.

966 \xspaceskip

Normally the glue inserted when a space is encountered after a character with a space factor other than 1000 is taken from the font (fontdimen 7) unless this parameter is set in which case its value is added.

967 \xtoks

This is the global variant of `\etoks`.

968 \xtoksapp

This is the global variant of `\etoksapp`.

969 \xtokspre

This is the global variant of `\etokspre`.

970 \year

This internal number starts out with the year that the job started.

5.4 Obsolete

The LuaMetaTEX engine has more than its LuaTEX ancestor but it also has less. Because in the end the local control mechanism performed quite okay I decided to drop the `\immediateassignment` and `\immediateassigned` variants. They sort of used the same trick so there isn't much to gain and it was less generic (read: error prone).

5.5 Syntax

5.5.1 accent

t \accent
[xoffset *dimension*] [yoffset *dimension*] *integer character*

5.5.2 aftersomething

l \afterassigned
{ *tokens* }
t \afterassignment
token
t \aftergroup
token
l \aftergrouped
{ *tokens* }
l \atendoffile
token
l \atendoffiled
[reverse] { *tokens* }
l \atendofgroup
token
l \atendofgrouped
{ *tokens* }

5.5.3 alignmenttab

l \aligntab

5.5.4 arithmic

t \advance
quantity [by] *quantity*
l \advanceby
quantity quantity
t \divide
quantity [by] *quantity*
l \divideby
quantity quantity
l \edivide
quantity quantity
l \edivideby
quantity quantity
t \multiply
quantity [by] *quantity*
l \multiplyby
quantity quantity
l \rdivide
quantity quantity
l \rdivideby
quantity quantity

5.5.5 association

l \associateunit
\cs [=] *integer*
> \cs : *integer*

5.5.6 auxiliary

l \insertmode
integer
: *integer*
e \interactionmode
integer
: *integer*
t \prevdepth
dimension
: *dimension*
t \prevgraf
integer
: *integer*
t \spacefactor
integer
: *integer*

5.5.7 begingroup

t \begingroup
l \beginmathgroup
l \beginsimplegroup

5.5.8 beginlocal

l \beginlocalcontrol
l \expandedendless
{ *tokens* }
l \expandedloop
integer integer integer { *tokens* }

l **\expandedrepeat**
integer { *tokens* }
l **\localcontrol**
tokens\endlocalcontrol
l **\localcontrolled**
{ *tokens* }
l **\localcontrolledendless**
{ *tokens* }
l **\localcontrolledloop**
see \expandedloop
l **\localcontrolledrepeat**
integer { *tokens* }
l **\unexpandedendless**
{ *tokens* }
l **\unexpandedloop**
see \expandedloop
l **\unexpandedrepeat**
integer { *tokens* }

5.5.9 beginparagraph

t **\indent**
t **\noindent**
l **\parattribute**
integer [=] *integer*
l **\quitvmode**
l **\snapshotpar**
cardinal
: *integer*
l **\undent**
l **\wrapuppar**
[reverse] { *tokens* }

5.5.10 boundary

l **\boundary**
[=] *integer*
l **\luaboundary**
[=] *integer integer*
l **\mathboundary**
[=] *integer* [*integer*]
l **\noboundary**
l **\optionalboundary**
[=] *integer*
l **\pageboundary**
[=] *integer*
l **\protrusionboundary**
[=] *integer*
l **\wordboundary**

5.5.11 boxproperty

l **\boxadapt**
(index | *box*) [=] *integer*
> (index | *box*) : *dimension*
l **\boxanchor**
see \boxadapt
l **\boxanchors**
(index | *box*) [=] *integer integer*
> (index | *box*) : *integer*
l **\boxattribute**
(index | *box*) *integer* [=] *integer*
> (index | *box*) *integer* : *integer*
l **\boxdirection**
see \boxadapt
l **\boxfinalize**
see \boxadapt
l **\boxfreeze**
see \boxadapt
l **\boxgeometry**
see \boxadapt
l **\boxlimit**
(index | *box*)
l **\boxlimitate**
see \boxadapt
l **\boxorientation**
see \boxadapt
l **\boxrepack**
see \boxlimit
l **\boxshift**
(index | *box*) [=] *dimension*
> (index | *box*) : *dimension*
l **\boxshrink**
see \boxlimit
l **\boxsource**
see \boxadapt
l **\boxstretch**
see \boxlimit
l **\boxtarget**
see \boxadapt
l **\boxtotal**
see \boxlimit
l **\boxvadjust**
(index | *box*) { *tokens* }
> (index | *box*) : cardinal

l \boxxmove
see \boxshift
l \boxxoffset
see \boxshift
l \boxymove
see \boxshift
l \boxyoffset
see \boxshift
t \dp
see \boxshift
t \ht
see \boxshift
t \wd
see \boxshift

5.5.12 caseshift

t \lowercase
{ *tokens* }
t \uppercase
{ *tokens* }

5.5.13 catcodetable

l \initcatcodetable
integer
l \restorecatcodetable
integer
l \savecatcodetable
integer

5.5.14 charnumber

t \char
integer
l \glyph
[xoffset *dimension*] [yoffset *dimension*] [scale *integer*] [xscale *integer*] [yscale *integer*] [left *dimension*] [right *dimension*] [raise *dimension*] [options *integer*] [*font integer*] [id *integer*] *integer*

5.5.15 combinetoks

l \etoks
toks { *tokens* }
l \etoksapp
toks { *tokens* }
l \etokspre
toks { *tokens* }
l \gtoksapp
toks { *tokens* }
l \gtokspre
toks { *tokens* }
l \toksapp
toks { *tokens* }
l \tokspre
toks { *tokens* }
l \xtoks
toks { *tokens* }
l \xtoksapp
toks { *tokens* }
l \xtokspre
toks { *tokens* }

5.5.16 convert

l \csactive
> *token* : *tokens*
l \csnamestring
: *tokens*
l \csstring
> *token* : *tokens*
l \detokened
> (\cs | { *tokens* } | *toks*) : *tokens*
l \detokenized
> { *tokens* } : *tokens*
l \directlua
> { *tokens* } : *tokens*
l \expanded
> { *tokens* } : *tokens*
l \fontidentifier
TODO
t \fontname
> (*font* | *integer*) : *tokens*
l \fontspecifiedname
> (*font* | *integer*) : *tokens*
l \formatname
: *tokens*
t \jobname
: *tokens*
l \luabytecode
> *integer* : *tokens*
l \luaescapestring

> { *tokens* } : *tokens*

l \luafunction

> *integer* : *tokens*

l \luatexbanner

: *tokens*

t \meaning

> *token* : *tokens*

l \meaningasis

> *token* : *tokens*

l \meaningful

> *token* : *tokens*

l \meaningfull

> *token* : *tokens*

l \meaningles

> *token* : *tokens*

l \meaningless

> *token* : *tokens*

t \number

> *integer* : *tokens*

t \romannumeral

> *integer* : *tokens*

l \semiexpanded

> { *tokens* } : *tokens*

t \string

> *token* : *tokens*

l \tocharacter

> *integer* : *tokens*

l \todimension

> *dimension* : *tokens*

l \tohexadecimal

> *integer* : *tokens*

l \tointeger

> *integer* : *tokens*

l \tomathstyle

> *mathstyle* : *tokens*

l \toscaled

> *dimension* : *tokens*

l \tosparsedimension

> *dimension* : *tokens*

l \tosparsescaled

> *dimension* : *tokens*

5.5.17 csname

l \begincsname

tokens\endcsname

t \csname

tokens\endcsname

l \futurecsname

tokens\endcsname

l \lastnamedcs

5.5.18 def

l \cdef

\cs [*preamble*] { *tokens* }

l \cdefcsname

tokens\endcsname [*preamble*] { *tokens* }

t \def

\cs [*preamble*] { *tokens* }

l \defcsname

tokens\endcsname [*preamble*] { *tokens* }

t \edef

\cs [*preamble*] { *tokens* }

l \edefcsname

tokens\endcsname [*preamble*] { *tokens* }

t \gdef

\cs [*preamble*] { *tokens* }

l \gdefcsname

tokens\endcsname [*preamble*] { *tokens* }

t \xdef

\cs [*preamble*] { *tokens* }

l \xdefcsname

tokens\endcsname [*preamble*] { *tokens* }

5.5.19 definecharcode

l \Udelcode

integer [=] *integer*

> *integer* : *integer*

l \Umathcode

integer [=] *integer*

> *integer* : *integer*

l \amcode

integer [=] *integer*

> *integer* : *integer*

t \catcode

integer [=] *integer*

> *integer* : *integer*

l \cccode

integer [=] *integer*

> *integer* : *integer*

t \delcode

integer [=] *integer*

> *integer* : *integer*

l \hccode

integer [=] *integer*

> *integer* : *integer*

l \hmcode

integer [=] *integer*

> *integer* : *integer*

t \lccode

integer [=] *integer*

> *integer* : *integer*

t \mathcode

integer [=] *integer*

> *integer* : *integer*

t \sfcode

integer [=] *integer*

> *integer* : *integer*

t \uccode

integer [=] *integer*

> *integer* : *integer*

5.5.20 definefamily

t \scriptfont

family (*font* | *integer*)

> family : *integer*

t \scriptscriptfont

see \scriptfont

t \textfont

see \scriptfont

5.5.21 definefont

t \font

\cs ({ filename } | filename) [(at *dimension* | scaled *integer*)]

: *tokens*

5.5.22 delimiternumber

l \Udelimiter

integer integer integer

t \delimiter

integer

5.5.23 discretionary

t \-

l \automaticdiscretionary

t \discretionary

[penalty] [postword] [preword] [break] [nobreak] [options] [class] { *tokens* } { *tokens* } { *tokens* }

l \explicitdiscretionary

5.5.24 endcsname

t \endcsname

5.5.25 endgroup

t \endgroup

l \endmathgroup

l \endsimplegroup

5.5.26 endjob

t \dump

t \end

5.5.27 endlocal

l \endlocalcontrol

5.5.28 endparagraph

l \localbreakpar

t \par

5.5.29 endtemplate

l \aligncontent

t \cr

t \crcr

t \noalign

{ *tokens* }

t \omit

l \realign

{ *tokens* } { *tokens* }

t \span

5.5.30 equationnumber

t \eqno
{ *tokens* }
t \leqno
{ *tokens* }

5.5.31 expandafter

l \expand
token
l \expandactive
token
t \expandafter
token token
l \expandafterpars
token
l \expandafterspaces
token
l \expandcstoken
token
l \expandedafter
token { *tokens* }
l \expandparameter
integer
l \expandtoken
token
l \expandtoks
{ *tokens* }
l \futureexpand
token token token
l \futureexpandis
TODO
l \futureexpandisap
TODO
l \semiexpand
token
e \unless

5.5.32 explicitspace

**t **
l \explicitspace

5.5.33 fontproperty

l \cfcode
(*font* | *integer*) *integer* [=] *integer*
> (*font* | *integer*) *integer* : *integer*
l \efcode
see \cfcode
t \fontdimen
(*font* | *integer*) *integer* [=] *dimension*
> (*font* | *integer*) *integer* : *dimension*
t \hyphenchar
(*font* | *integer*) [=] *integer*
> (*font* | *integer*) : *integer*
l \lpcode
see \fontdimen
l \rpcode
see \fontdimen
l \scaledfontdimen
see \hyphenchar
t \skewchar
see \hyphenchar

5.5.34 getmark

t \botmark
e \botmarks
integer
l \currentmarks
integer
t \firstmark
e \firstmarks
integer
t \splitbotmark
e \splitbotmarks
integer
t \splitfirstmark
e \splitfirstmarks
integer
t \topmark
e \topmarks
integer

5.5.35 halign

t \halign
[attr *integer integer*] [callback *integer*] [discard] [noskips] [reverse] [to *dimension*] [spread *dimension*] { *tokens* }

5.5.36 hmove

t \moveleft
dimension box
t \moveright
dimension box

5.5.37 hrule

t \hrule
[attr *integer* [=] *integer*] [width *dimension*] [height *dimension*] [depth *dimension*] [xoffset *dimension*] [yoffset *dimension*] [left *dimension*] [right *dimension*] [top *dimension*] [bottom *dimension*]
l \nohrule
see \hrule
l \virtualhrule
see \hrule

5.5.38 hskip

t \hfil
t \hfill
t \hfilneg
t \hskip
dimension [plus (*dimension* | fi [n*l])] [minus (*dimension* | fi [n*l])]
t \hss

5.5.39 hyphenation

l \hjcode
integer [=] *integer*
t \hyphenation
{ *tokens* }
l \hyphenationmin
[=] *integer*
t \patterns
{ *tokens* }
l \postexhyphenchar
[=] *integer*
l \posthyphenchar
[=] *integer*
l \preexhyphenchar
[=] *integer*
l \prehyphenchar
[=] *integer*

5.5.40 iftest

t \else
t \fi
t \if
l \ifabsdim
dimension
(! | < | = | > | ∈ | ∉ | ≠ | ≤ | ≥ | ≰ | ≱)
dimension
l \ifabsfloat
float (! | < | = | > | ∈ | ∉ | ≠ | ≤ | ≥ | ≰ | ≱)
float
l \ifabsnum
integer
(! | < | = | > | ∈ | ∉ | ≠ | ≤ | ≥ | ≰ | ≱)
integer
l \ifarguments
l \ifboolean
integer
t \ifcase
integer
t \ifcat
token
l \ifchkdim
tokens\or
l \ifchkdimension
tokens\or
l \ifchknum
tokens\or
l \ifchknumber
tokens\or
l \ifcmpdim
dimension dimension
l \ifcmpnum
integer integer
l \ifcondition
\if...
l \ifcramped
e \ifcsname
tokens\endcsname
l \ifcstok
tokens\relax
e \ifdefined
token
t \ifdim
see \ifabsdim

l \ifdimexpression
tokens\relax
l \ifdimval
tokens\or
l \ifempty
(*token* | { *tokens* })
t \iffalse
l \ifflags
\cs
l \iffloat
see \ifabsfloat
e \iffontchar
integer integer
l \ifhaschar
token { *tokens* }
l \ifhastok
token { *tokens* }
l \ifhastoks
tokens\relax
l \ifhasxtoks
tokens\relax
t \ifhbox
(index | *box*)
t \ifhmode
l \ifinalignment
l \ifincsname
tokens\endcsname
t \ifinner
l \ifinsert
integer
l \ifintervaldim
dimension dimension dimension
l \ifintervalfloat
integer integer integer
l \ifintervalnum
float float float
l \iflastnamedcs
l \ifmathparameter
integer
l \ifmathstyle
mathstyle
t \ifmmode
t \ifnum
see \ifabsnum
l \ifnumexpression
tokens\relax
l \ifnumval
tokens\or
t \ifodd
integer
l \ifparameter
parameter\or
l \ifparameters
l \ifrelax
token
l \iftok
tokens\relax
t \iftrue
t \ifvbox
see \ifhbox
t \ifvmode
t \ifvoid
see \ifhbox
t \ifx
token
l \ifzerodim
dimension
l \ifzerofloat
float
l \ifzeronum
integer
t \or
l \orelse
l \orunless

5.5.41 ignoresomething

l \ignorearguments
l \ignorenestedupto
token
l \ignorepars
l \ignorerest
t \ignorespaces
l \ignoreupto
token

5.5.42 input

t \endinput
t \eofinput
{ *tokens* } ({ filename } | filename)
t \input
({ filename } | filename)
l \quitloop
l \quitloopnow

l \retokenized
[catcodetable] { *tokens* }
l \scantextokens
{ *tokens* }
e \scantokens
{ *tokens* }
l \tokenized
{ *tokens* }

5.5.43 insert

t \insert
integer

5.5.44 interaction

t \batchmode
t \errorstopmode
t \nonstopmode
t \scrollmode

5.5.45 internaldimension

t \boxmaxdepth
[=] *dimension*
: *dimension*
t \delimitershortfall
[=] *dimension*
: *dimension*
t \displayindent
[=] *dimension*
: *dimension*
t \displaywidth
[=] *dimension*
: *dimension*
t \emergencyextrastretch
[=] *dimension*
: *dimension*
t \emergencystretch
[=] *dimension*
: *dimension*
l \glyphxoffset
[=] *dimension*
: *dimension*
l \glyphyoffset
[=] *dimension*
: *dimension*
t \hangindent
[=] *dimension*
: *dimension*
t \hfuzz
[=] *dimension*
: *dimension*
t \hsize
[=] *dimension*
: *dimension*
l \ignoredepthcriterion
[=] *dimension*
: *dimension*
t \lineskiplimit
[=] *dimension*
: *dimension*
t \mathsurround
[=] *dimension*
: *dimension*
t \maxdepth
[=] *dimension*
: *dimension*
t \nulldelimiterspace
[=] *dimension*
: *dimension*
t \overfullrule
[=] *dimension*
: *dimension*
l \pageextragoal
[=] *dimension*
: *dimension*
t \parindent
[=] *dimension*
: *dimension*
t \predisplaysize
[=] *dimension*
: *dimension*
l \pxdimen
[=] *dimension*
: *dimension*
t \scriptspace
[=] *dimension*
: *dimension*
l \shortinlinemaththreshold
[=] *dimension*
: *dimension*
t \splitmaxdepth
[=] *dimension*
: *dimension*

l \tabsize
[=] *dimension*
: *dimension*
t \vfuzz
[=] *dimension*
: *dimension*
t \vsize
[=] *dimension*
: *dimension*

5.5.46 internalglue

t \abovedisplayshortskip
[=] *glue*
: *glue*
t \abovedisplayskip
[=] *glue*
: *glue*
l \additionalpageskip
[=] *glue*
: *glue*
t \baselineskip
[=] *glue*
: *glue*
t \belowdisplayshortskip
[=] *glue*
: *glue*
t \belowdisplayskip
[=] *glue*
: *glue*
l \emergencyleftskip
[=] *glue*
: *glue*
l \emergencyrightskip
[=] *glue*
: *glue*
l \initialpageskip
[=] *glue*
: *glue*
l \initialtopskip
[=] *glue*
: *glue*
t \leftskip
[=] *glue*
: *glue*
t \lineskip
[=] *glue*
: *glue*
l \mathsurroundskip
[=] *glue*
: *glue*
l \maththreshold
[=] *glue*
: *glue*
l \parfillleftskip
[=] *glue*
: *glue*
l \parfillrightskip
[=] *glue*
: *glue*
t \parfillskip
[=] *glue*
: *glue*
l \parinitleftskip
[=] *glue*
: *glue*
l \parinitrightskip
[=] *glue*
: *glue*
t \parskip
[=] *glue*
: *glue*
t \rightskip
[=] *glue*
: *glue*
t \spaceskip
[=] *glue*
: *glue*
t \splittopskip
[=] *glue*
: *glue*
t \tabskip
[=] *glue*
: *glue*
t \topskip
[=] *glue*
: *glue*
t \xspaceskip
[=] *glue*
: *glue*

5.5.47 internalinteger

t \adjdemerits
[=] *integer*
: *integer*

l \adjustspacing
[=] *integer*
: *integer*

l \adjustspacingshrink
[=] *integer*
: *integer*

l \adjustspacingstep
[=] *integer*
: *integer*

l \adjustspacingstretch
[=] *integer*
: *integer*

l \alignmentcellsource
[=] *integer*
: *integer*

l \alignmentwrapsource
[=] *integer*
: *integer*

l \automatichyphenpenalty
[=] *integer*
: *integer*

l \automigrationmode
[=] *integer*
: *integer*

l \autoparagraphmode
[=] *integer*
: *integer*

t \binoppenalty
[=] *integer*
: *integer*

l \boxlimitmode
[=] *integer*
: *integer*

t \brokenpenalty
[=] *integer*
: *integer*

l \catcodetable
[=] *integer*
: *integer*

t \clubpenalty
[=] *integer*
: *integer*

t \day
[=] *integer*
: *integer*

t \defaulthyphenchar
[=] *integer*
: *integer*

t \defaultskewchar
[=] *integer*
: *integer*

t \delimiterfactor
[=] *integer*
: *integer*

l \discretionaryoptions
[=] *integer*
: *integer*

t \displaywidowpenalty
[=] *integer*
: *integer*

t \doublehyphendemerits
[=] *integer*
: *integer*

l \doublepenaltymode
[=] *integer*
: *integer*

t \endlinechar
[=] *integer*
: *integer*

t \errorcontextlines
[=] *integer*
: *integer*

t \escapechar
[=] *integer*
: *integer*

l \eufactor
[=] *integer*
: *integer*

l \exceptionpenalty
[=] *integer*
: *integer*

t \exhyphenchar
[=] *integer*
: *integer*

t \exhyphenpenalty
[=] *integer*
: *integer*

l \explicithyphenpenalty
[=] *integer*
: *integer*

t \fam
[=] *integer*
: *integer*

t \finalhyphendemerits
[=] *integer*
: *integer*

l \firstvalidlanguage
[=] *integer*
: *integer*

t \floatingpenalty
[=] *integer*
: *integer*

t \globaldefs
[=] *integer*
: *integer*

l \glyphdatafield
[=] *integer*
: *integer*

l \glyphoptions
[=] *integer*
: *integer*

l \glyphscale
[=] *integer*
: *integer*

l \glyphscriptfield
[=] *integer*
: *integer*

l \glyphscriptscale
[=] *integer*
: *integer*

l \glyphscriptscriptscale
[=] *integer*
: *integer*

l \glyphslant
[=] *integer*
: *integer*

l \glyphstatefield
[=] *integer*
: *integer*

l \glyphtextscale
[=] *integer*
: *integer*

l \glyphweight
[=] *integer*
: *integer*

l \glyphxscale
[=] *integer*
: *integer*

l \glyphyscale
[=] *integer*
: *integer*

t \hangafter
[=] *integer*
: *integer*

t \hbadness
[=] *integer*
: *integer*

t \holdinginserts
[=] *integer*
: *integer*

l \holdingmigrations
[=] *integer*
: *integer*

l \hyphenationmode
[=] *integer*
: *integer*

t \hyphenpenalty
[=] *integer*
: *integer*

t \interlinepenalty
[=] *integer*
: *integer*

t \language
[=] *integer*
: *integer*

e \lastlinefit
[=] *integer*
: *integer*

t \lefthyphenmin
[=] *integer*
: *integer*

l \lefttwindemerits
[=] *integer*
: *integer*

l \linebreakchecks
[=] *integer*
: *integer*

l \linebreakoptional
[=] *integer*
: *integer*

l \linebreakpasses
[=] *integer*
: *integer*

l \linedirection
[=] *integer*
: *integer*

t \linepenalty
[=] *integer*
: *integer*

l \localbrokenpenalty
[=] *integer*
: *integer*

l \localinterlinepenalty
[=] *integer*
: *integer*

l \localpretolerance
[=] *integer*
: *integer*

l \localtolerance
[=] *integer*
: *integer*

t \looseness
[=] *integer*
: *integer*

l \luacopyinputnodes
[=] *integer*
: *integer*

l \mathbeginclass
[=] *integer*
: *integer*

l \mathcheckfencesmode
[=] *integer*
: *integer*

l \mathdictgroup
[=] *integer*
: *integer*

l \mathdictproperties
[=] *integer*
: *integer*

l \mathdirection
[=] *integer*
: *integer*

l \mathdisplaymode
[=] *integer*
: *integer*

l \mathdisplaypenaltyfactor
[=] *integer*
: *integer*

l \mathdisplayskipmode
[=] *integer*
: *integer*

l \mathdoublescriptmode
[=] *integer*
: *integer*

l \mathendclass
[=] *integer*
: *integer*

l \matheqnogapstep
[=] *integer*
: *integer*

l \mathfontcontrol
[=] *integer*
: *integer*

l \mathgluemode
[=] *integer*
: *integer*

l \mathgroupingmode
[=] *integer*
: *integer*

l \mathinlinepenaltyfactor
[=] *integer*
: *integer*

l \mathleftclass
[=] *integer*
: *integer*

l \mathlimitsmode
[=] *integer*
: *integer*

l \mathnolimitsmode
[=] *integer*
: *integer*

l \mathpenaltiesmode
[=] *integer*
: *integer*

l \mathpretolerance
[=] *integer*
: *integer*

l \mathrightclass
[=] *integer*
: *integer*

l \mathrulesfam
[=] *integer*
: *integer*

l \mathrulesmode
[=] *integer*
: *integer*

l \mathscriptsmode
[=] *integer*
: *integer*

l \mathslackmode
[=] *integer*
: *integer*

l \mathspacingmode
[=] *integer*
: *integer*

l \mathsurroundmode
[=] *integer*
: *integer*

l \mathtolerance
[=] *integer*
: *integer*

t \maxdeadcycles
[=] *integer*
: *integer*

t \month
[=] *integer*
: *integer*

t \newlinechar
[=] *integer*
: *integer*

l \normalizelinemode
[=] *integer*
: *integer*

l \normalizeparmode
[=] *integer*
: *integer*

l \nospaces
[=] *integer*
: *integer*

l \outputbox
[=] *integer*
: *integer*

t \outputpenalty
[=] *integer*
: *integer*

l \overloadmode
[=] *integer*
: *integer*

l \parametermode
[=] *integer*
: *integer*

l \pardirection
[=] *integer*
: *integer*

t \pausing
[=] *integer*
: *integer*

t \postdisplaypenalty
[=] *integer*
: *integer*

l \postinlinepenalty
[=] *integer*
: *integer*

l \postshortinlinepenalty
[=] *integer*
: *integer*

l \prebinoppenalty
[=] *integer*
: *integer*

e \predisplaydirection
[=] *integer*
: *integer*

l \predisplaygapfactor
[=] *integer*
: *integer*

t \predisplaypenalty
[=] *integer*
: *integer*

l \preinlinepenalty
[=] *integer*
: *integer*

l \prerelpenalty
[=] *integer*
: *integer*

l \preshortinlinepenalty
[=] *integer*
: *integer*

t \pretolerance
[=] *integer*
: *integer*

l \protrudechars
[=] *integer*
: *integer*

t \relpenalty
[=] *integer*
: *integer*

t \righthyphenmin
[=] *integer*
: *integer*

l \righttwindemerits
[=] *integer*
: *integer*

e \savinghyphcodes
[=] *integer*
: *integer*

e \savingvdiscards
[=] *integer*
: *integer*

l \scriptspaceafterfactor
[=] *integer*
: *integer*

l \scriptspacebeforefactor
[=] *integer*
: *integer*

l \scriptspacebetweenfactor
[=] *integer*
: *integer*

l \setfontid
[=] *integer*
: *integer*

t \setlanguage
[=] *integer*
: *integer*

l \shapingpenaltiesmode
[=] *integer*
: *integer*

l \shapingpenalty
[=] *integer*
: *integer*

l \shortinlineorphanpenalty
[=] *integer*
: *integer*

t \showboxbreadth
[=] *integer*
: *integer*

t \showboxdepth
[=] *integer*
: *integer*

t \shownodedetails
[=] *integer*
: *integer*

l \singlelinepenalty
[=] *integer*
: *integer*

l \spacechar
[=] *integer*
: *integer*

l \spacefactormode
[=] *integer*
: *integer*

l \spacefactoroverload
[=] *integer*
: *integer*

l \spacefactorshrinklimit
[=] *integer*
: *integer*

l \spacefactorstretchlimit
[=] *integer*
: *integer*

l \supmarkmode
[=] *integer*
: *integer*

l \textdirection
[=] *integer*
: *integer*

t \time
[=] *integer*
: *integer*

t \tolerance
[=] *integer*
: *integer*

l \tracingadjusts
[=] *integer*
: *integer*

l \tracingalignments
[=] *integer*
: *integer*

e \tracingassigns
[=] *integer*
: *integer*

t \tracingcommands
[=] *integer*
: *integer*

l \tracingexpressions
[=] *integer*
: *integer*

l \tracingfitness
[=] *integer*
: *integer*

l \tracingfullboxes
[=] *integer*
: *integer*

e \tracinggroups
[=] *integer*
: *integer*

l \tracinghyphenation
[=] *integer*
: *integer*

e \tracingifs
[=] *integer*
: *integer*

l \tracinginserts
[=] *integer*
: *integer*

l \tracinglevels
[=] *integer*
: *integer*

l \tracinglists
[=] *integer*
: *integer*

t \tracingloners
[=] *integer*
: *integer*
l \tracinglooseness
[=] *integer*
: *integer*
t \tracinglostchars
[=] *integer*
: *integer*
t \tracingmacros
[=] *integer*
: *integer*
l \tracingmarks
[=] *integer*
: *integer*
l \tracingmath
[=] *integer*
: *integer*
e \tracingnesting
[=] *integer*
: *integer*
l \tracingnodes
[=] *integer*
: *integer*
t \tracingonline
[=] *integer*
: *integer*
l \tracingorphans
TODO
t \tracingoutput
[=] *integer*
: *integer*
t \tracingpages
[=] *integer*
: *integer*
t \tracingparagraphs
[=] *integer*
: *integer*
l \tracingpasses
[=] *integer*
: *integer*
l \tracingpenalties
[=] *integer*
: *integer*
t \tracingrestores
[=] *integer*
: *integer*
t \tracingstats
[=] *integer*
: *integer*
l \tracingtoddlers
TODO
t \uchyph
[=] *integer*
: *integer*
l \variablefam
[=] *integer*
: *integer*
t \vbadness
[=] *integer*
: *integer*
t \widowpenalty
[=] *integer*
: *integer*
t \year
[=] *integer*
: *integer*

5.5.48 internalmuglue

t \medmuskip
[=] *muglue*
: *muglue*
l \pettymuskip
[=] *muglue*
: *muglue*
t \thickmuskip
[=] *muglue*
: *muglue*
t \thinmuskip
[=] *muglue*
: *muglue*
l \tinymuskip
[=] *muglue*
: *muglue*

5.5.49 internaltoks

t \errhelp
[=] *toks*
: *toks*
l \everybeforepar
[=] *toks*
: *toks*
t \everycr
[=] *toks*
: *toks*

t \everydisplay
[=] *toks*
: *toks*
e \everyeof
[=] *toks*
: *toks*
t \everyhbox
[=] *toks*
: *toks*
t \everyjob
[=] *toks*
: *toks*
t \everymath
[=] *toks*
: *toks*
l \everymathatom
[=] *toks*
: *toks*
t \everypar
[=] *toks*
: *toks*
l \everytab
[=] *toks*
: *toks*
t \everyvbox
[=] *toks*
: *toks*
t \output
[=] *toks*
: *toks*

5.5.50 italiccorrection

t \/
l \explicititalicorrection
l \forcedleftcorrection
l \forcedrightcorrection

5.5.51 kern

t \hkern
dimension
t \kern
dimension
t \vkern
dimension

5.5.52 leader

t \cleaders
(*box* | *rule* | glyph) *glue*
l \gleaders
see \cleaders
t \leaders
see \cleaders
l \uleaders
[callback *integer*] [line] [nobreak]
(*box* | *rule* | glyph) *glue*
t \xleaders
see \cleaders

5.5.53 legacy

t \shipout
{ *tokens* }

5.5.54 let

l \futuredef
\cs \cs
t \futurelet
\cs [=] \cs
l \glet
\cs
l \gletcsname
tokens\endcsname
l \glettonothing
\cs
t \let
\cs
l \letcharcode
\cs
l \letcsname
tokens\endcsname
l \letfrozen
\cs
l \letprotected
\cs
l \lettolastnamedcs
\cs
l \lettonothing
\cs
l \swapcsvalues
\cs \cs

l \unletfrozen
\cs
l \unletprotected
\cs

5.5.55 localbox

l \localleftbox
box
l \localmiddlebox
box
l \localrightbox
box
l \resetlocalboxes

5.5.56 luafunctioncall

l \luabytecodecall
integer
l \luafunctioncall
integer

5.5.57 makebox

t \box
(index | *box*)
t \copy
see \box
l \dbox
[target *integer*] [to *dimension*] [adapt] [attr *integer integer*] [anchor *integer*] [axis *integer*] [shift *dimension*] [spread *dimension*] [source *integer*] [direction *integer*] [delay] [orientation *integer*] [xoffset *dimension*] [xmove *dimension*] [yoffset *dimension*] [ymove *dimension*] [reverse] [retain] [container] [mathtext] [class *integer*] { *tokens* }
l \dpack
see \dbox
l \dsplit
[attr] [to] [upto] { *tokens* }
t \hbox
see \dbox
l \hpack
see \dbox
l \insertbox
integer
l \insertcopy
integer
t \lastbox
l \localleftboxbox
l \localmiddleboxbox
l \localrightboxbox
l \tpack
see \dbox
l \tsplit
see \dsplit
t \vbox
see \dbox
l \vpack
see \dbox
t \vsplit
see \dsplit
t \vtop
see \dbox

5.5.58 mark

l \clearmarks
integer
l \flushmarks
t \mark
{ *tokens* }
e \marks
integer { *tokens* }

5.5.59 mathaccent

l \Umathaccent
[attr *integer integer*] [center] [class *integer*] [exact] [source *integer*] [stretch] [shrink] [fraction *integer*] [fixed] [keepbase] [nooverflow] [base] (both [fixed] *character* [fixed] *character* | bottom [fixed] *character* | top [fixed] *character* | overlay *character* | *character*)
t \mathaccent
{ *tokens* }

5.5.60 mathcharnumber

l \Umathchar
integer
t \mathchar
integer
l \mathclass
integer
l \mathdictionary
integer mathchar
l \nomathchar

5.5.61 mathchoice

t \mathchoice
{ *tokens* } { *tokens* } { *tokens* } { *tokens* }
l \mathdiscretionary
[class *integer*] { *tokens* } { *tokens* } { *tokens* }
l \mathstack
{ *tokens* }

5.5.62 mathcomponent

l \mathatom
[attr *integer integer*] [all *integer*] [leftclass *integer*] [limits] [rightclass *integer*] [class *integer*] [unpack] [unroll] [single] [source *integer*] [text*font*] [math*font*] [options *integer*] [nolimits] [nooverflow] [void] [phantom] [continuation] [*integer*]
t \mathbin
{ *tokens* }
t \mathclose
{ *tokens* }
t \mathinner
{ *tokens* }
t \mathop
{ *tokens* }
t \mathopen
{ *tokens* }
t \mathord
{ *tokens* }
t \mathpunct
{ *tokens* }
t \mathrel
{ *tokens* }
t \overline
{ *tokens* }
t \underline
{ *tokens* }

5.5.63 mathfence

l \Uleft
[auto] [attr *integer integer*] [axis] [bottom *dimension*] [depth *dimension*] [factor *integer*] [height *dimension*] [noaxis] [nocheck] [nolimits] [nooverflow] [leftclass *integer*] [limits] [exact] [void] [phantom] [class *integer*] [rightclass *integer*] [scale] [source *integer*] [top] delimiter
l \Umiddle
see \Uleft
l \Uoperator
see \Uleft
l \Uright
see \Uleft
l \Uvextensible
see \Uleft
t \left
see \Uleft
t \middle
see \Uleft
t \right
see \Uleft

5.5.64 mathfraction

l \Uabove
dimension [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*] [usecallback]
l \Uabovewithdelims
delimiter delimiter *dimension* [attr *integer integer*] [class *integer*] [center] [exact] [proportional]

[noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*] [usecallback]

l \Uatop

see \Uabove

l \Uatopwithdelims

see \Uabovewithdelims

l \Uover

[attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*] [usecallback]

l \Uoverwithdelims

delimiter delimiter [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*] [usecallback]

l \Uskewed

delimiter [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*] [usecallback]

l \Uskewedwithdelims

delimiter delimiter delimiter [attr *integer integer*] [class *integer*] [center] [exact] [proportional] [noaxis] [nooverflow] [style *mathstyle*] [source *integer*] [hfactor *integer*] [vfactor *integer*] [*font*] [thickness *dimension*] [usecallback]

l \Ustretched

see \Uskewed

l \Ustretchedwithdelims

see \Uskewedwithdelims

t \above

dimension

t \abovewithdelims

delimiter delimiter *dimension*

t \atop

dimension

t \atopwithdelims

delimiter delimiter *dimension*

t \over

t \overwithdelims

delimiter delimiter

5.5.65 mathmodifier

l \Umathadapttoleft

l \Umathadapttoright

l \Umathlimits

l \Umathnoaxis

l \Umathnolimits

l \Umathopenupdepth

dimension

l \Umathopenupheight

dimension

l \Umathphantom

l \Umathsource

[nucleus] *integer*

l \Umathuseaxis

l \Umathvoid

t \displaylimits

t \limits

t \nolimits

5.5.66 mathparameter

l \Umathaccentbasedepth

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \Umathaccentbaseheight

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \Umathaccentbottomovershoot

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \Umathaccentbottomshiftdown

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \Umathaccentextendmargin

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \Umathaccentsuperscriptdrop

mathstyle [=] *dimension*

> *mathstyle* : *dimension*

l \Umathaccentsuperscriptpercent
mathstyle [=] *integer*
> *mathstyle* : *integer*

l \Umathaccenttopovershoot
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathaccenttopshiftup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathaccentvariant
[=] *mathstyle*
: *mathstyle*

l \Umathaxis
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathbottomaccentvariant
[=] *mathstyle*
: *mathstyle*

l \Umathconnectoroverlapmin
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathdegreevariant
[=] *mathstyle*
: *mathstyle*

l \Umathdelimiterextendmargin
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathdelimiterovervariant
[=] *mathstyle*
: *mathstyle*

l \Umathdelimiterpercent
mathstyle [=] *integer*
> *mathstyle* : *integer*

l \Umathdelimitershortfall
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathdelimiterundervariant
[=] *mathstyle*
: *mathstyle*

l \Umathdenominatorvariant
[=] *mathstyle*
: *mathstyle*

l \Umathexheight
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathextrasubpreshift
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathextrasubprespace
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathextrasubshift
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathextrasubspace
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathextrasuppreshift
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathextrasupprespace
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathextrasupshift
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathextrasupspace
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathflattenedaccentbasedepth
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathflattenedaccentbaseheight
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathflattenedaccentbottomshiftdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathflattenedaccenttopshiftup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractiondelsize
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractiondenomdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractiondenomvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractionnumup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathfractionnumvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathfractionrule**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathfractionvariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathhextensiblevariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathlimitabovebgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathlimitabovekern**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathlimitabovevgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathlimitbelowbgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathlimitbelowkern**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathlimitbelowvgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathnolimitsubfactor**
mathstyle [=] *integer*
> *mathstyle* : *integer*

l **\Umathnolimitsupfactor**
mathstyle [=] *integer*
> *mathstyle* : *integer*

l **\Umathnumeratorvariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathoperatorsize**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathoverbarkern**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathoverbarrule**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathoverbarvgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathoverdelimiterbgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathoverdelimitervariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathoverdelimitervgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathoverlayaccentvariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathoverlinevariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathprimeraise**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathprimeraisecomposed**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathprimeshiftdrop**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathprimeshiftup**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathprimespaceafter**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathprimevariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathquad**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathradicaldegreeafter**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathradicaldegreebefore**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathradicaldegreeraise**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathradicalextensibleafter**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalextensiblebefore
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalkern
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalrule
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathradicalvariant
[=] *mathstyle*
: *mathstyle*

l \Umathradicalvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathruledepth
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathruleheight
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathskeweddelimitertolerance
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathskewedfractionhgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathskewedfractionvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathspaceafterscript
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathspacebeforescript
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathspacebetweenscript
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathstackdenomdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathstacknumup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathstackvariant
[=] *mathstyle*
: *mathstyle*

l \Umathstackvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubscriptsnap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubscriptvariant
[=] *mathstyle*
: *mathstyle*

l \Umathsubshiftdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubshiftdrop
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubsupshiftdown
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubsupvgap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsubtopmax
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupbottommin
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsuperscriptsnap
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsuperscriptvariant
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupshiftdrop
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupshiftup
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathsupsubbottommax
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \Umathtopaccentvariant
[=] *mathstyle*
: *mathstyle*

l \Umathunderbarkern
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathunderbarrule**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathunderbarvgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathunderdelimiterbgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathunderdelimitervariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathunderdelimitervgap**
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l **\Umathunderlinevariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathvextensiblevariant**
[=] *mathstyle*
: *mathstyle*

l **\Umathxscale**
mathstyle [=] *integer*
> *mathstyle* : *integer*

l **\Umathyscale**
mathstyle [=] *integer*
> *mathstyle* : *integer*

l **\copymathatomrule**
integer integer

l **\copymathparent**
integer integer

l **\copymathspacing**
integer integer

l **\letmathatomrule**
integer integer integer integer integer

l **\letmathparent**
integer integer

l **\letmathspacing**
see \letmathatomrule

l **\resetmathspacing**

l **\setdefaultmathcodes**

l **\setmathatomrule**
integer integer mathstyle integer integer

l **\setmathdisplaypostpenalty**
integer [=] *integer*

l **\setmathdisplayprepenalty**
integer [=] *integer*

l **\setmathignore**
mathparameter *integer*

l **\setmathoptions**
integer [=] *integer*

l **\setmathpostpenalty**
integer [=] *integer*

l **\setmathprepenalty**
integer [=] *integer*

l **\setmathspacing**
integer integer mathstyle glue

5.5.67 mathradical

l **\Udelimited**
[attr *integer integer*] [bottom] [exact] [top] [style *mathstyle*] [source *integer*] [stretch] [shrink] [width *dimension*] [height *dimension*] [depth *dimension*] [left] [middle] [right] [nooverflow] [usecallback] delimiter delimiter [delimiter] [delimiter] (mathatom | { *tokens* })

l **\Udelimiterover**
[attr *integer integer*] [bottom] [exact] [top] [style *mathstyle*] [source *integer*] [stretch] [shrink] [width *dimension*] [height *dimension*] [depth *dimension*] [left] [middle] [right] [nooverflow] [usecallback] delimiter [delimiter] [delimiter] (mathatom | { *tokens* })

l **\Udelimiterunder**
see \Udelimiterover

l **\Uhextensible**
see \Udelimiterover

l **\Uoverdelimiter**
see \Udelimiterover

l **\Uradical**
see \Udelimiterover

l **\Uroot**
[attr *integer integer*] [bottom] [exact] [top] [style *mathstyle*] [source *integer*] [stretch] [shrink] [width *dimension*] [height *dimension*] [depth *dimension*] [left] [middle] [right] [nooverflow] [usecallback] delimiter [delimiter] [delimiter] (mathatom | { *tokens* }) (mathatom | { *tokens* })

l \Urooted
[attr *integer integer*] [bottom]
[exact] [top] [style *mathstyle*]
[source *integer*] [stretch] [shrink]
[width *dimension*] [height *dimension*]
[depth *dimension*] [left] [middle]
[right] [nooverflow] [usecallback]
delimiter delimiter [delimiter]
[delimiter] (mathatom | {*tokens*})
(mathatom | {*tokens*})

l \Uunderdelimiter
see \Udelimiterover

t \radical
see \Uroot

5.5.68 mathscript

l \indexedsubprescript
(mathatom | {*tokens*})

l \indexedsubscript
see \indexedsubprescript

l \indexedsuperprescript
see \indexedsubprescript

l \indexedsuperscript
see \indexedsubprescript

l \noatomruling

t \nonscript

l \noscript

l \nosubprescript

l \nosubscript

l \nosuperprescript

l \nosuperscript

l \primescript
see \indexedsubprescript

l \subprescript
see \indexedsubprescript

l \subscript
see \indexedsubprescript

l \superprescript
see \indexedsubprescript

l \superscript
see \indexedsubprescript

5.5.69 mathshiftcs

l \Ustartdisplaymath

l \Ustartmath

l \Ustartmathmode

l \Ustopdisplaymath

l \Ustopmath

l \Ustopmathmode

5.5.70 mathstyle

l \allcrampedstyles

l \alldisplaystyles

l \allmainstyles

l \allmathstyles

l \allscriptscriptstyles

l \allscriptstyles

l \allsplitstyles

l \alltextstyles

l \alluncrampedstyles

l \allunsplitstyles

l \crampeddisplaystyle

l \crampedscriptscriptstyle

l \crampedscriptstyle

l \crampedtextstyle

l \currentlysetmathstyle

t \displaystyle

l \givenmathstyle
mathstyle

l \scaledmathstyle
integer
> *mathstyle* : *integer*

t \scriptscriptstyle

t \scriptstyle

t \textstyle

5.5.71 message

t \errmessage
{*tokens*}

t \message
{*tokens*}

5.5.72 mkern

t \mkern
dimension

5.5.73 mskip

l \mathatomskip
muglue

t \mskip
muglue

5.5.74 noexpand

t \noexpand
token

5.5.75 pageproperty

t \deadcycles
[=] *integer*
: *integer*
l \insertdepth
integer [=] *dimension*
> *integer* : *dimension*
l \insertdistance
integer [=] *dimension*
> *integer* : *dimension*
l \insertheight
integer [=] *dimension*
> *integer* : *dimension*
l \insertheights
[=] *dimension*
: *dimension*
l \insertlimit
integer [=] *dimension*
> *integer* : *dimension*
l \insertmaxdepth
integer [=] *dimension*
> *integer* : *dimension*
l \insertmultiplier
integer [=] *integer*
> *integer* : *integer*
t \insertpenalties
[=] *integer*
: *integer*
l \insertpenalty
integer [=] *integer*
> *integer* : *integer*
l \insertstorage
integer [=] *integer*
> *integer* : *integer*
l \insertstoring
[=] *integer*
: *integer*
l \insertwidth
integer [=] *dimension*
> *integer* : *dimension*
l \pagedepth
[=] *dimension*
: *dimension*
l \pageexcess
[=] *dimension*
: *dimension*
t \pagefilllstretch
[=] *dimension*
: *dimension*
t \pagefillstretch
[=] *dimension*
: *dimension*
t \pagefilstretch
[=] *dimension*
: *dimension*
l \pagefistretch
[=] *dimension*
: *dimension*
t \pagegoal
[=] *dimension*
: *dimension*
l \pagelastdepth
[=] *dimension*
: *dimension*
l \pagelastfilllstretch
[=] *dimension*
: *dimension*
l \pagelastfillstretch
[=] *dimension*
: *dimension*
l \pagelastfilstretch
[=] *dimension*
: *dimension*
l \pagelastfistretch
[=] *dimension*
: *dimension*
l \pagelastheight
[=] *dimension*
: *dimension*
l \pagelastshrink
[=] *dimension*
: *dimension*
l \pagelaststretch
[=] *dimension*
: *dimension*
t \pageshrink
[=] *dimension*
: *dimension*

t \pagestretch

[=] *dimension*

: *dimension*

t \pagetotal

[=] *dimension*

: *dimension*

l \pagevsize

[=] *dimension*

: *dimension*

5.5.76 parameter

l \alignmark
l \parametermark

5.5.77 penalty

l \hpenalty

integer

t \penalty

integer

l \vpenalty

integer

5.5.78 prefix

l \aliased
l \constant
l \constrained
l \deferred
l \enforced
l \frozen
t \global
l \immediate
l \immutable
l \inherited
l \instance
t \long
l \mutable
l \noaligned
t \outer
l \overloaded
l \permanent
e \protected
l \retained
l \semiprotected
l \tolerant
l \untraced

5.5.79 register

l \attribute

(index | *box*) [=] *integer*

> (index | *box*) : *integer*

t \count

see \attribute

t \dimen

(index | *box*) [=] *dimension*

> (index | *box*) : *dimension*

l \float

(index | *box*) [=] *float*

> (index | *box*) : *float*

t \muskip

(index | *box*) [=] *muglue*

> (index | *box*) : *muglue*

t \skip

(index | *box*) [=] *glue*

> (index | *box*) : *glue*

t \toks

(index | *box*) [=] { *tokens* }

> (index | *box*) : { *tokens* }

5.5.80 relax

l \norelax
t \relax

5.5.81 removeitem

t \unboundary
t \unkern
t \unpenalty
t \unskip

5.5.82 setbox

t \setbox

(index | *box*) [=]

5.5.83 setfont

t \nullfont

5.5.84 shorthanddef

l \Umathchardef
\cs *integer*
l \Umathdictdef
\cs *integer integer*
l \attributedef
\cs *integer*
t \chardef
\cs *integer*
t \countdef
\cs *integer*
t \dimendef
\cs *integer*
l \dimensiondef
\cs *integer*
l \floatdef
\cs *integer*
l \fontspecdef
\cs (*font* | *integer*)
l \gluespecdef
\cs *integer*
l \integerdef
\cs *integer*
l \luadef
\cs *integer*
t \mathchardef
\cs *integer*
l \mugluespecdef
\cs *integer*
t \muskipdef
\cs *integer*
l \parameterdef
\cs *integer*
l \positdef
\cs *integer*
t \skipdef
\cs *integer*
l \specificationdef
\cs *tokens*\relax
t \toksdef
\cs *integer*

5.5.85 someitem

t \badness
[=] *integer*
: *integer*
e \currentgrouplevel
[=] *integer*
: *integer*
e \currentgrouptype
[=] *integer*
: *integer*
e \currentifbranch
[=] *integer*
: *integer*
e \currentiflevel
[=] *integer*
: *integer*
e \currentiftype
[=] *integer*
: *integer*
l \currentloopiterator
[=] *integer*
: *integer*
l \currentloopnesting
[=] *integer*
: *integer*
e \currentstacksize
[=] *integer*
: *integer*
e \dimexpr
tokens\relax [=] *dimension*
> *tokens*\relax : *dimension*
l \dimexpression
tokens\relax [=] *dimension*
> *tokens*\relax : *dimension*
l \floatexpr
tokens\relax [=] *float*
> *tokens*\relax : *float*
l \fontcharba
integer [=] *dimension*
> *integer* : *dimension*
e \fontchardp
integer [=] *dimension*
> *integer* : *dimension*
e \fontcharht
integer [=] *dimension*
> *integer* : *dimension*
e \fontcharic
integer [=] *dimension*
> *integer* : *dimension*
l \fontcharta
integer [=] *dimension*
> *integer* : *dimension*

e \fontcharwd
integer [=] *dimension*
> *integer* : *dimension*

l \fontid
(*font* | *integer*) [=] *integer*
> (*font* | *integer*) : *integer*

l \fontmathcontrol
see \fontid

l \fontspecid
see \fontid

l \fontspecifiedsize
see \fontid

l \fontspecscale
see \fontid

l \fontspecslant
see \fontid

l \fontspecweight
see \fontid

l \fontspecxscale
see \fontid

l \fontspecyscale
see \fontid

l \fonttextcontrol
see \fontid

e \glueexpr
tokens\relax [=] *glue*
> *tokens*\relax : *glue*

e \glueshrink
glue [=] *dimension*
> *glue* : *dimension*

e \glueshrinkorder
glue [=] *dimension*
> *glue* : *dimension*

e \gluestretch
glue [=] *integer*
> *glue* : *integer*

e \gluestretchorder
glue [=] *integer*
> *glue* : *integer*

e \gluetomu
glue [=] *glue*
> *glue* : *glue*

l \glyphxscaled
[=] *integer*
: *integer*

l \glyphyscaled
[=] *integer*
: *integer*

l \indexofcharacter
integer [=] *integer*
> *integer* : *integer*

l \indexofregister
integer [=] *integer*
> *integer* : *integer*

t \inputlineno
[=] *integer*
: *integer*

l \insertprogress
integer [=] *dimension*
> *integer* : *dimension*

l \lastarguments
[=] *integer*
: *integer*

l \lastatomclass
[=] *integer*
: *integer*

l \lastboundary
[=] *integer*
: *integer*

l \lastchkdimension
[=] *dimension*
: *dimension*

l \lastchknumber
[=] *integer*
: *integer*

t \lastkern
[=] *dimension*
: *dimension*

l \lastleftclass
[=] *integer*
: *integer*

l \lastloopiterator
[=] *integer*
: *integer*

l \lastnodesubtype
[=] *integer*
: *integer*

e \lastnodetype
[=] *integer*
: *integer*

l \lastpageextra
[=] *dimension*
: *dimension*

l \lastparcontext
[=] *integer*
: *integer*

l \lastpartrigger
[=] *integer*
: *integer*

t \lastpenalty
[=] *integer*
: *integer*

l \lastrightclass
[=] *integer*
: *integer*

t \lastskip
[=] *glue*
: *glue*

l \leftmarginkern
[=] *dimension*
: *dimension*

l \luatexrevision
[=] { *tokens* }
: { *tokens* }

l \luatexversion
[=] { *tokens* }
: { *tokens* }

l \mathatomglue
[=] *glue*
: *glue*

l \mathcharclass
integer [=] *integer*
> *integer* : *integer*

l \mathcharfam
integer [=] *integer*
> *integer* : *integer*

l \mathcharslot
integer [=] *integer*
> *integer* : *integer*

l \mathmainstyle
[=] *integer*
: *integer*

l \mathparentstyle
[=] *integer*
: *integer*

l \mathscale
[=] *integer*
: *integer*

l \mathstackstyle
[=] *integer*
: *integer*

l \mathstyle
[=] *integer*
: *integer*

l \mathstylefontid
[=] *integer*
: *integer*

e \muexpr
tokens\relax [=] *muglue*
> *tokens*\relax : *muglue*

e \mutoglue
muglue [=] *glue*
> *muglue* : *glue*

l \nestedloopiterator
[=] *integer*
: *integer*

l \numericscale
(*integer* | *float*) [=] *integer*
> (*integer* | *float*) : *integer*

l \numericscaled
see \numericscale

e \numexpr
tokens\relax [=] *integer*
> *tokens*\relax : *integer*

l \numexpression
tokens\relax [=] *integer*
> *tokens*\relax : *integer*

l \overshoot
[=] *dimension*
: *dimension*

l \parametercount
[=] *integer*
: *integer*

l \parameterindex
[=] *integer*
: *integer*

e \parshapedimen
integer [=] *dimension*
> *integer* : *dimension*

e \parshapeindent
integer [=] *dimension*
> *integer* : *dimension*

e \parshapelength
[=] *dimension*
: *dimension*

l \parshapewidth
[=] *dimension*
: *dimension*

l \previousloopiterator
[=] *integer*
: *integer*

l \rightmarginkern
[=] *dimension*

: *dimension*

l \scaledemwidth
(*font* | *integer*) [=] *dimension*
> (*font* | *integer*) : *dimension*

l \scaledexheight
see \scaledemwidth

l \scaledextraspace
see \scaledemwidth

l \scaledfontcharba
integer [=] *dimension*
> *integer* : *dimension*

l \scaledfontchardp
integer [=] *dimension*
> *integer* : *dimension*

l \scaledfontcharht
integer [=] *dimension*
> *integer* : *dimension*

l \scaledfontcharic
integer [=] *dimension*
> *integer* : *dimension*

l \scaledfontcharta
integer [=] *dimension*
> *integer* : *dimension*

l \scaledfontcharwd
integer [=] *dimension*
> *integer* : *dimension*

l \scaledinterwordshrink
see \scaledemwidth

l \scaledinterwordspace
see \scaledemwidth

l \scaledinterwordstretch
see \scaledemwidth

l \scaledmathaxis
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \scaledmathemwidth
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \scaledmathexheight
mathstyle [=] *dimension*
> *mathstyle* : *dimension*

l \scaledslantperpoint
see \scaledemwidth

5.5.86 specification

l \adjacentdemerits
[options] *integer* n * (*integer*)
: *integer*

l \brokenpenalties
see \adjacentdemerits

e \clubpenalties
see \adjacentdemerits

e \displaywidowpenalties
see \adjacentdemerits

l \fitnessclasses
see \adjacentdemerits

e \interlinepenalties
see \adjacentdemerits

l \mathbackwardpenalties
see \adjacentdemerits

l \mathforwardpenalties
see \adjacentdemerits

l \orphanlinefactors
TODO

l \orphanpenalties
see \adjacentdemerits

l \parpasses
[options] n * ([next] [quit] [skip] [adjdemerits *integer*] [adjacentdemerits <adjacentdemerits>] [adjustspacing *integer*] [adjustspacingshrink *integer*] [adjustspacingstep *integer*] [adjustspacingstretch *integer*] [callback *integer*] [classes *integer*] [demerits *integer*] [doubleadjdemerits *integer*] [doublehyphendemerits *integer*] [emergencyleftextra *integer*] [emergencypercentage *dimension*] [emergencyrightextra *integer*] [emergencystretch *dimension*] [emergencywidthextra *integer*] [extrahyphenpenalty *integer*] [finalhyphendemerits *integer*] [fitnessclasses <fitnessclasses>] [hyphenation *integer*] [identifier *integer*] [ifadjustspacing *integer*] [ifemergencystretch *integer*] [ifglue *integer*] [iflooseness *integer*] [ifmath *integer*] [iftext *integer*] [lefttwindemerits *integer*] [linebreakchecks *integer*] [linebreakcriterium *integer*] [linebreakoptional *integer*] [linepenalty *integer*] [looseness *integer*] [mathpenaltyfactor *integer*]

[orphanpenalties] [toddlerpenalties <toddlerpenalties>] [righttwindemerits *integer*] [threshold *dimension*] [tolerance *integer*] [unlessmath *integer*])
: *integer*

l \parpassesexception
see \type{ \parpasses }
: *integer*

t \parshape
[options] *integer* n * (*dimension* *dimension*)
: *integer*

l \toddlerpenalties
see \adjacentdemerits

e \widowpenalties
see \adjacentdemerits

5.5.87 the

e \detokenize
{ *tokens* }

l \expandeddetokenize
{ *tokens* }

l \protecteddetokenize
{ *tokens* }

l \protectedexpandeddetokenize
{ *tokens* }

t \the
dimension

l \thewithoutunit
quantity

e \unexpanded
{ *tokens* }

5.5.88 unhbox

t \unhbox
integer

t \unhcopy
integer

l \unhpack
integer

5.5.89 unvbox

l \insertunbox
integer

l \insertuncopy
integer

e \pagediscards

e \splitdiscards

t \unvbox
integer

t \unvcopy
integer

l \unvpack
integer

5.5.90 vadjust

t \vadjust
[pre] [post] [baseline] [before] [index *integer*] [after] [attr *integer* *integer*] [depth (after | before | check | last)] { *tokens* }

5.5.91 valign

t \valign
[attr *integer* *integer*] [callback *integer*] [discard] [noskips] [reverse] [to *dimension*] [spread *dimension*] { *tokens* }

5.5.92 vcenter

t \vcenter
[target *integer*] [to *dimension*] [adapt] [attr *integer* *integer*] [anchor *integer*] [axis *integer*] [shift *dimension*] [spread *dimension*] [source *integer*] [direction *integer*] [delay] [orientation *integer*] [xoffset *dimension*] [xmove *dimension*] [yoffset *dimension*] [ymove *dimension*] [reverse] [retain] [container] [mathtext] [class *integer*] { *tokens* }

5.5.93 vmove

t \lower
dimension *box*

t \raise
dimension box

e \showtokens
{ *tokens* }

5.5.94 vrule

l \novrule
[attr *integer* [=] *integer*] [width *dimension*] [height *dimension*] [depth *dimension*] [xoffset *dimension*] [yoffset *dimension*] [left *dimension*] [right *dimension*] [top *dimension*] [bottom *dimension*]

l \srule
[attr *integer* [=] *integer*] [width *dimension*] [height *dimension*] [depth *dimension*] [xoffset *dimension*] [yoffset *dimension*] [*font integer*] [fam *integer*] [char *integer*]

l \virtualvrule
see \novrule

t \vrule
see \novrule

5.5.95 vskip

t \vfil
t \vfill
t \vfilneg
t \vskip
dimension [plus (*dimension* | fi [n*l])] [minus (*dimension* | fi [n*l])]

t \vss

5.5.96 xray

t \show
token

t \showbox
(index | *box*)

l \showcodestack
e \showgroups
e \showifs
t \showlists
l \showstack
t \showthe
quantity

5.6 To be checked primitives (new)

fontidentifier
ifcmpdim
ifcmpnum
orphanlinefactors
tracingorphans
tracingtoddlers

5.7 To be checked primitives (math)

Uabove
Udelcode
Udelimited
Udelimiter
Udelimiterover
Udelimiterunder
Uhextensible
Uleft
Umathaccentbasedepth
Umathaccentbaseheight
Umathaccentbottomovershoot
Umathaccentbottomshiftdown
Umathaccentextendmargin
Umathaccentsuperscriptdrop
Umathaccentsuperscriptpercent
Umathaccenttopovershoot
Umathaccenttopshiftup
Umathaccentvariant
Umathadapttoleft
Umathadapttoright
Umathaxis
Umathbottomaccentvariant
Umathcode
Umathconnectoroverlapmin
Umathdegreevariant
Umathdelimiterextendmargin
Umathdelimiterovervariant
Umathdelimiterpercent
Umathdelimitershortfall
Umathdelimiterundervariant
Umathdenominatorvariant
Umathdictdef
Umathexheight
Umathextrasubpreshift
Umathextrasubprespace
Umathextrasubshift
Umathextrasubspace
Umathextrasuppreshift
Umathextrasupprespace
Umathextrasupshift
Umathextrasupspace
Umathflattenedaccentbasedepth
Umathflattenedaccentbaseheight
Umathflattenedaccentbottomshiftdown
Umathflattenedaccenttopshiftup
Umathfractiondelsize
Umathfractiondenomdown
Umathfractiondenomvgap
Umathfractionnumup
Umathfractionnumvgap
Umathfractionrule
Umathfractionvariant
Umathhextensiblevariant
Umathlimitabovebgap
Umathlimitabovekern
Umathlimitabovevgap
Umathlimitbelowbgap
Umathlimitbelowkern
Umathlimitbelowvgap
Umathlimits
Umathnoaxis
Umathnolimits
Umathnumeratorvariant
Umathopenupdepth
Umathopenupheight
Umathoperatorsize
Umathoverdelimiterbgap
Umathoverdelimitervariant
Umathoverdelimitervgap
Umathoverlayaccentvariant
Umathphantom
Umathprimeraise
Umathprimeraisecomposed
Umathprimeshiftdrop
Umathprimeshiftup
Umathprimespaceafter
Umathprimevariant
Umathquad
Umathradicaldegreeafter
Umathradicaldegreebefore
Umathradicaldegreeraise
Umathradicalextensibleafter
Umathradicalextensiblebefore
Umathradicalkern
Umathradicalrule
Umathradicalvariant
Umathradicalvgap
Umathruledepth
Umathruleheight
Umathskeweddelimitertolerance
Umathskewedfractionhgap
Umathskewedfractionvgap
Umathsource
Umathstackdenomdown

`Umathstacknumup`
`Umathstackvariant`
`Umathstackvgap`
`Umathsubscriptsnap`
`Umathsubscriptvariant`
`Umathsubshiftdown`
`Umathsubshiftdrop`
`Umathsubsupshiftdown`
`Umathsubsupvgap`
`Umathsubtopmax`
`Umathsupbottommin`
`Umathsuperscriptsnap`
`Umathsuperscriptvariant`
`Umathsupshiftdrop`
`Umathsupshiftup`
`Umathsupsubbottommax`
`Umathtopaccentvariant`
`Umathunderdelimiterbgap`
`Umathunderdelimitervariant`
`Umathunderdelimitervgap`
`Umathuseaxis`
`Umathvextensiblevariant`
`Umathvoid`
`Umiddle`
`Uoperator`
`Uoverdelimiter`
`Uroot`
`Urooted`
`Uskewed`
`Uskewedwithdelims`
`Ustartdisplaymath`
`Ustartmath`
`Ustartmathmode`
`Ustopdisplaymath`
`Ustopmath`
`Ustopmathmode`
`Ustretched`
`Ustretchedwithdelims`
`Uunderdelimiter`
`Uvextensible`

Many primitives starting with `Umath` are math parameters that are discussed elsewhere, if at all.

5.8 To be checked primitives (old)

5.9 Indexed primitives

-
/
<space>
Uabovewithdelims
Uatop
Uatopwithdelims
Umathaccent
Umathchar
Umathchardef
Umathnolimitsubfactor
Umathnolimitsupfactor
Umathoverbarkern
Umathoverbarrule
Umathoverbarvgap
Umathoverlinevariant
Umathspaceafterscript
Umathspacebeforescript
Umathspacebetweenscript
Umathunderbarkern
Umathunderbarrule
Umathunderbarvgap
Umathunderlinevariant
Umathxscale
Umathyscale
Uover
Uoverwithdelims
Uradical
Uright

above
abovedisplayshortskip
abovedisplayskip
abovewithdelims
accent
additionalpageskip
adjacentdemerits
adjdemerits
adjustspacing
adjustspacingshrink
adjustspacingstep
adjustspacingstretch
advance
advanceby
afterassigned
afterassignment
aftergroup
aftergrouped
aliased
aligncontent
alignmark
alignmentcellsource
alignmentwrapsource
aligntab
allcrampedstyles
alldisplaystyles
allmainstyles
allmathstyles
allscriptscriptstyles
allscriptstyles
allsplitstyles
alltextstyles
alluncrampedstyles
allunsplitstyles
amcode
associateunit
atendoffile
atendoffiled
atendofgroup
atendofgrouped
atop
atopwithdelims
attribute
attributedef
automaticdiscretionary
automatichyphenpenalty
automigrationmode
autoparagraphmode
badness
baselineskip
batchmode
begincsname
begingroup
beginlocalcontrol
beginmathgroup
beginsimplegroup
belowdisplayshortskip
belowdisplayskip
binoppenalty
botmark
botmarks
boundary
box
boxadapt

boxanchor
boxanchors
boxattribute
boxdirection
boxfinalize
boxfreeze
boxgeometry
boxlimit
boxlimitate
boxlimitmode
boxmaxdepth
boxorientation
boxrepack
boxshift
boxshrink
boxsource
boxstretch
boxtarget
boxtotal
boxvadjust
boxxmove
boxxoffset
boxymove
boxyoffset
brokenpenalties
brokenpenalty
catcode
catcodetable
cccode
cdef
cdefcsname
cf
cfcode
char
chardef
cleaders
clearmarks
clubpenalties
clubpenalty
constant
constrained
copy
copymathatomrule
copymathparent
copymathspacing
correctionskip
count
countdef
cr
crampeddisplaystyle
crampedscriptscriptstyle
crampedscriptstyle
crampedtextstyle
crcr
csactive
csname
csnamestring
csstring
currentgrouplevel
currentgrouptype
currentifbranch
currentiflevel
currentiftype
currentloopiterator
currentloopnesting
currentlysetmathstyle
currentmarks
currentstacksize
day
dbox
deadcycles
def
defaulthyphenchar
defaultskewchar
defcsname
deferred
delcode
delimiter
delimiterfactor
delimitershortfall
detokened
detokenize
detokenized
dimen
dimendef
dimensiondef
dimexpr
dimexpression
directlua
discretionary
discretionaryoptions
displayindent
displaylimits
displayskipmode
displaystyle
displaywidowpenalties
displaywidowpenalty
displaywidth

divide
divideby
doublehyphendemerits
doublepenaltymode
dp
dpack
dsplit
dump
edef
edefcsame
edefcsname
edivide
edivideby
efcode
else
emergencyextrastretch
emergencyleftskip
emergencyrightskip
emergencystretch
end
endcsname
endgroup
endinput
endlinechar
endlocalcontrol
endmathgroup
endsimplegroup
enforced
eofinput
eqno
errhelp
errmessage
errorcontextlines
errorstopmode
escapechar
etoks
etoksapp
etokspre
eufactor
everybeforepar
everycr
everydisplay
everyeof
everyhbox
everyjob
everymath
everymathatom
everypar
everytab
everyvbox
exceptionpenalty
exhyphenchar
exhyphenpenalty
expand
expandactive
expandafter
expandafterpars
expandafterspaces
expandcstoken
expanded
expandedafter
expandeddetokenize
expandedendless
expandedloop
expandedrepeat
expandparameter
expandtoken
expandtoks
explicitdiscretionary
explicithyphenpenalty
explicititaliccorrection
explicitspace
fam
fi
finalhyphendemerits
firstmark
firstmarks
firstvalidlanguage
fitnessclasses
float
floatdef
floatexpr
floatingpenalty
flushmarks
font
fontcharba
fontchardp
fontcharht
fontcharic
fontcharta
fontcharwd
fontdimen
fontid
fontmathcontrol
fontname
fontspecdef
fontspecid
fontspecifiedname

fontspecifiedsize
fontspecscale
fontspecslant
fontspecweight
fontspecxscale
fontspecyscale
fonttextcontrol
forcedleftcorrection
forcedrightcorrection
formatname
frozen
futurecsname
futuredef
futureexpand
futureexpandis
futureexpandisap
futurelet
gdef
gdefcsname
givenmathstyle
gleaders
glet
gletcsname
glettonothing
global
globaldefs
glue
glueexpr
glueshrink
glueshrinkorder
gluespecdef
gluestretch
gluestretchorder
gluetomu
glyph
glyphdatafield
glyphoptions
glyphscale
glyphscriptfield
glyphscriptscale
glyphscriptscriptscale
glyphslant
glyphstatefield
glyphtextscale
glyphweight
glyphxoffset
glyphxscale
glyphxscaled
glyphyoffset
glyphyscale
glyphyscaled
gtoksapp
gtokspre
halign
hangafter
hangindent
hbadness
hbox
hccode
hfil
hfill
hfilneg
hfuzz
hj
hjcode
hkern
hmcode
holdinginserts
holdingmigrations
hpack
hpenalty
hrule
hsize
hskip
hss
ht
hyphenation
hyphenationmin
hyphenationmode
hyphenchar
hyphenpenalty
if
ifabsdim
ifabsfloat
ifabsnum
ifarguments
ifboolean
ifcase
ifcat
ifchkdim
ifchkdimension
ifchknum
ifchknumber
ifcondition
ifcramped
ifcsname
ifcstok
ifdefined

ifdim
ifdimexpression
ifdimval
ifempty
iffalse
ifflags
iffloat
iffontchar
ifhaschar
ifhastok
ifhastoks
ifhasxtoks
ifhbox
ifhmode
ifinalignment
ifincsname
ifinner
ifinsert
ifintervaldim
ifintervalfloat
ifintervalnum
iflastnamedcs
ifmathparameter
ifmathstyle
ifmmode
ifnum
ifnumexpression
ifnumval
ifodd
ifparameter
ifparameters
ifrelax
iftok
iftrue
ifvbox
ifvmode
ifvoid
ifx
ifzerodim
ifzerofloat
ifzeronum
ignorearguments
ignoredepthcriterion
ignorenestedupto
ignorepars
ignorerest
ignorespaces
ignoreupto
immediate
immediateassigned
immediateassignment
immutable
indent
indentskip
indexedsubprescript
indexedsubscript
indexedsuperprescript
indexedsuperscript
indexofcharacter
indexofregister
inherited
initcatcodetable
initialpageskip
initialtopskip
input
inputlineno
insert
insertbox
insertcopy
insertdepth
insertdistance
insertheight
insertheights
insertlimit
insertmaxdepth
insertmode
insertmultiplier
insertpenalties
insertpenalty
insertprogress
insertstorage
insertstoring
insertunbox
insertuncopy
insertwidth
instance
integerdef
interactionmode
interlinepenalties
interlinepenalty
jobname
kern
language
lastarguments
lastatomclass
lastboundary
lastbox
lastchkdimension

lastchknumber
lastkern
lastleftclass
lastlinefit
lastloopiterator
lastnamedcs
lastnodesubtype
lastnodetype
lastpageextra
lastparcontext
lastpartrigger
lastpenalty
lastrightclass
lastskip
lccode
leaders
left
lefthangskip
lefthyphenmin
leftmarginkern
leftskip
lefttwindemerits
leqno
let
letcharcode
letcsname
letfrozen
letmathatomrule
letmathparent
letmathspacing
letprotected
lettolastnamedcs
lettonothing
limits
linebreakchecks
linebreakoptional
linebreakpasses
linedirection
linepenalty
lineskip
lineskiplimit
localbreakpar
localbrokenpenalty
localcontrol
localcontrolled
localcontrolledendless
localcontrolledloop
localcontrolledrepeat
localinterlinepenalty
localleftbox
localleftboxbox
localmiddlebox
localmiddleboxbox
localpretolerance
localrightbox
localrightboxbox
localtolerance
long
looseness
lower
lowercase
lpcode
luaboundary
luabytecode
luabytecodecall
luacopyinputnodes
luadef
luaescapestring
luafunction
luafunctioncall
luatexbanner
luatexrevision
luatexversion
mark
marks
mathaccent
mathatom
mathatomglue
mathatomskip
mathbackwardpenalties
mathbeginclass
mathbin
mathboundary
mathchar
mathcharclass
mathchardef
mathcharfam
mathcharslot
mathcheckfencesmode
mathchoice
mathclass
mathclose
mathcode
mathdictgroup
mathdictionary
mathdictproperties
mathdirection
mathdiscretionary

mathdisplaymode
mathdisplaypenaltyfactor
mathdisplayskipmode
mathdoublescriptmode
mathendclass
matheqnogapstep
mathfontcontrol
mathforwardpenalties
mathgluemode
mathgroupingmode
mathinlinepenaltyfactor
mathinner
mathleftclass
mathlimitsmode
mathmainstyle
mathnolimitsmode
mathop
mathopen
mathord
mathparentstyle
mathpenaltiesmode
mathpretolerance
mathpunct
mathrel
mathrightclass
mathrulesfam
mathrulesmode
mathscale
mathscriptsmode
mathslackmode
mathspacingmode
mathstack
mathstackstyle
mathstyle
mathstylefontid
mathsurround
mathsurroundmode
mathsurroundskip
maththreshold
mathtolerance
maxdeadcycles
maxdepth
meaning
meaningasis
meaningful
meaningfull
meaningles
meaningless
medmuskip
message
middle
mkern
month
moveleft
moveright
mskip
muexpr
mugluespecdef
multiply
multiplyby
muskip
muskipdef
mutable
mutoglue
nestedloopiterator
newlinechar
noalign
noaligned
noatomruling
noboundary
noexpand
nohrule
noindent
nolimits
nomathchar
nonscript
nonstopmode
norelax
normalizelinemode
normalizeparmode
normalunexpanded
noscript
nospaces
nosubprescript
nosubscript
nosuperprescript
nosuperscript
novrule
nulldelimiterspace
nullfont
number
numericscale
numericscaled
numexpr
numexpression
omit
open
optionalboundary

rdivide
rdivideby
realign
relax
relpenalty
resetlocalboxes
resetmathspacing
restorecatcodes
restorecatcodetable
retained
retokenized
right
righthangskip
righthyphenmin
rightmarginkern
rightskip
righttwindemerits
romannumeral
rpcode
savecatcodetable
savinghyphcodes
savingvdiscards
scaledemwidth
scaledexheight
scaledextraspace
scaledfontcharba
scaledfontchardp
scaledfontcharht
scaledfontcharic
scaledfontcharta
scaledfontcharwd
scaledfontdimen
scaledinterwordshrink
scaledinterwordspace
scaledinterwordstretch
scaledmathaxis
scaledmathemwidth
scaledmathexheight
scaledmathstyle
scaledslantperpoint
scantextokens
scantokens
scriptfont
scriptscriptfont
scriptscriptstyle
scriptspace
scriptspaceafterfactor
scriptspacebeforefactor
scriptspacebetweenfactor
scriptstyle
scrollmode
semiexpand
semiexpanded
semiprotected
semprotected
setbox
setdefaultmathcodes
setfontid
setlanguage
setmathatomrule
setmathdisplaypostpenalty
setmathdisplayprepenalty
setmathignore
setmathoptions
setmathpostpenalty
setmathprepenalty
setmathspacing
sfcode
shapingpenaltiesmode
shapingpenalty
shipout
shortinlinemaththreshold
shortinlineorphanpenalty
show
showbox
showboxbreadth
showboxdepth
showcodestack
showgroups
showifs
showlists
shownodedetails
showstack
showthe
showtokens
singlelinepenalty
skewchar
skip
skipdef
snapshotpar
spacechar
spacefactor
spacefactormode
spacefactoroverload
spacefactorshrinklimit
spacefactorstretchlimit
spaceskip
span

special
specificationdef
splitbotmark
splitbotmarks
splitdiscards
splitfirstmark
splitfirstmarks
splitmaxdepth
splittopskip
srule
string
subprescript
subscript
superprescript
superscript
supmarkmode
swapcsvalues
tabsize
tabskip
tabskips
textdirection
textfont
textstyle
the
thewithoutunit
thickmuskip
thinmuskip
time
tinymuskip
tocharacter
toddlerpenalties
todimension
tohexadecimal
tointeger
tokenized
toks
toksapp
toksdef
tokspre
tolerance
tolerant
tomathstyle
topmark
topmarks
topskip
toscaled
tosparsedimension
tosparsescaled
tpack
tracingadjusts
tracingalignments
tracingassigns
tracingcommands
tracingexpressions
tracingfitness
tracingfullboxes
tracinggroups
tracinghyphenation
tracingifs
tracinginserts
tracinglevels
tracinglists
tracingloners
tracinglooseness
tracinglostchars
tracingmacros
tracingmarks
tracingmath
tracingnesting
tracingnodes
tracingonline
tracingoutput
tracingpages
tracingparagraphs
tracingpasses
tracingpenalties
tracingrestores
tracingstats
tsplit
uccode
uchyph
uleaders
unboundary
undent
underline
unexpanded
unexpandedendless
unexpandedloop
unexpandedrepeat
unhbox
unhcopy
unhpack
unkern
unless
unletfrozen
unletprotected
unpenalty
unskip

untraced
unvbox
unvcopy
unvpack
uppercase
vadjust
valign
variablefam
vbadness
vbox
vcenter
vfil
vfill
vfilneg
vfuzz
virtualhrule
virtualvrule
vkern
vpack
vpenalty
vrule
vsize
vskip
vsplit
vss
vtop
wd
widowpenalties
widowpenalty
wordboundary
wrapuppar
write
xdef
xdefcsname
xleaders
xspaceskip
xtoks
xtoksapp
xtokspre
year

callbacks

6 Callbacks

6.1 Introduction

Right from the start of the LuaTEX project callbacks were the way to extend the engine. At various places in processing the document source and typesetting the text the engine checks if there is a callback set and if so, calls out to Lua. Here we collect the various callbacks. For examples you can consult the ConTEXt code base.

The `callback` library has functions that register, find and list callbacks. Callbacks are Lua functions that are called in well defined places. There are two kinds of callbacks: those that mix with existing functionality, and those that (when enabled) replace functionality. In most 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 kern routine. Doing something else is possible but in the end might not give the user the expected outcome.

In order for a callback to kick in you need register it. This can be permanent or temporarily.

```
id = callback.register(<t:string> callback_name, <function> func)
id = callback.register(<t:string> callback_name, nil)
id = callback.register(<t:string> callback_name, false)
```

Here the `callback_name` is a predefined callback name as discusses in following sections. The function returns the internal `id` of the callback or `nil`, if the callback could not be registered. LuaMetaTEX 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). *This needs checking.*

```
<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, `find` returns `nil`. The `known` function can be used to check if a callback is supported.

```
if callback.known("foo") then
    -- do what is needed
end
```

6.2 Files

6.2.1 find_log_file

This is one of the callbacks that has to be set in order for the engine to work at all.

```
function (
    <t:string> askedname
)
    return <t:string> foundname
end
```

6.2.2 find_format_file

A format file is an efficient memory dump of the (in our case ConTEXt) macro package. In LuaTEX it can have a mix of TEXand Lua code but one should be aware that storing the Lua state is not up to the engine.

```
function (<t:string> askedname)
    return <t:string> foundname
end
```

A format file can be read from any valid location but is always written in the current directory. When written the number of bytes for each section is reported. When read all kind of checks take place ijn order to intercept corruption or incompatibilities. Contrary to LuaTEX, the LuaMetaTEX is not (zip) compressed so, in spite of more aggressive compression of data otherwise the file is a bit larger.

6.2.3 open_data_file

This callback function gets a filename passed. The return value is either the boolean value false or a table with two functions. A mandate `reader` function fill be called once for each new line to be read, the optional `close` function will be called once LuaTEX is done with the file.

```
function (
    <t:string> filename
)
    return <table> {
        <function> reader(<table> environment) end,
        <function> close (<table> environment) end,
    }
end
```

LuaMetaTEX never looks at the rest of the table, so we can use it to store additional per-file data. Both the callback functions will receive the table as their only argument.

6.2.4 start_file

This callback replaces the code that LuaMetaTEX prints when a file is opened like `(filename` for regular files. The category is a number:

```
function (
    <t:integer> category,
    <t:string>  filename
)
    -- no return values
end
```

The following categories can occur:

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

6.2.5 stop_file

This callback replaces the code that LuaMetaTEX prints when a file is closed like the `)` for regular files.

```
function (
    <t:integer> category
)
    -- no return values
end
```

6.3 Running

6.3.1 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 (
    <t:string> jobname
)
    return <t: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`, LuaMetaTEX will pretend your callback never happened. This callback does not replace any internal code.

6.3.2 pre_dump

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. It can be used to do some cleanup and other housekeeping.

```
function (
```

```
    -- no arguments
)
    -- no return values
end
```

6.3.3 start_run

```
function(
    -- no arguments
)
    -- no return values
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.

6.3.4 stop_run

```
function(
    -- no arguments
)
    -- no return values
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.

6.3.5 intercept_tex_error

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. The TEX related callback gets two arguments: the current processing mode and a boolean indicating if there was a runaway.

```
function (
    -- no arguments
)
    -- no return values
end
```

6.3.6 intercept_lua_error

This callback is similar to the one discussed in the previous section but for Lua. Of course we should in a recoverable state for this to work well.

```
function (
    -- no arguments
```

```
)
    -- no return values
end
```

6.3.7 show_error_message

This callback replaces the code that prints the error message. The usual interaction after the message is not affected but it is best to quit the run after reporting.

```
function (
    -- no arguments
)
    -- no return values
end
```

6.3.8 show_warning_message

This callback replaces the code that prints a (non fatal) warning message. The usual interaction after the message is not affected.

```
function (
    -- no arguments
)
    -- no return values
end
```

6.3.9 wrapup_run

This callback is called after the pdf and log files are closed. Use it at your own risk. efine_f risk.

```
function (
    -- no arguments
)
    -- no return values
end
```

6.3.10 handle_overload

One characteristic of TEX is that you have quite some control over what a control sequence triggers. For instance, `\hbox` normally starts a horizontal box but a user can redefine this primitive as macro to do whatever is required. This means that when other macros use this primitive their behavior will change. One way out of this is using aliases, for instance:

```
\normalsetbox0\normalhbox{test}
\normalifdim\normalwd0>10pt \normalbox0 \normalfi
```

But even these normal aliases can be redefined. Of course you can use special characters like _ in names but once you start doing this:

```
\p_setbox0\p_hbox{test}
```

```
\p_ifdim\p_wd0>10pt \p_box0 \p_fi
```

you should wonder if you still offer the user TEX as a programming language. It's not the route that ConTEXt takes.

In LuaMetaTEX every macro (including primitives) can be flagged and that happens with so called prefixes. Traditional TEX offers:

```
\global\def\foo{...}
\long  \def\foo{...} % no-op
\outer \def\foo{...} % no-op
```

The `\long` and `\outer` made sense at that time but are no-ops in LuaMetaTEX: every macro can take `\par` equivalents as arguments and can be defined at every level. The ε-TEX extensions introduced this prefix:

```
\protected\def\foo{...}
```

which prevents expansion unless the value is really expected (needed). The LuaMetaTEX engine added:

```
\semiprotected\def\foo{...}
```

but when eventually I see no reason to use it in ConTEXt it might be dropped. A special prefix is:

```
\constant\def\foo{...}
```

This effectively is equivalent to `\edef` but signals that in some scenarios (like an `\csname` equivalent situation) no expansion and checking has to happen which improves performance.

These two prefixes are just signals to Lua driven functionality:

```
\deferred  \foo
\immediate \foo
```

The prefixes do nothing except when `\foo` are Lua calls that can use this information to adapt behavior. Because we have no backend the macro package has to come up with equivalents for e.g. `\write` than can be immediate or deferred (default) operations.

Another prefix relates to alignments:

```
\noaligned\protected\def\foo{...}
```

Which makes a macro accepted between alignment rows where otherwise protected macros will trigger an error due to look ahead.

A definition with `\def` or `\gdef` can take arguments and these can be made optional with:

```
\def\tolerant[#1]{...}
```

but there are more features related to tolerant:

```
\def\tolerant[#1]#*[#2]{...}
```

that are discusses in low level manuals. Users can define macros that are reported (in tracing) as if they were primitives:

```
\untraced\protected\def\foo{...}
```

The prefixes `\constrained` and `\retained` relate to register values being saved and restored in groups. The `\inherited` is used in for instance math spacing assignments where we need dynamic binding to for instance `\muskip` registers (instead of values).

Although not related to the callback discussed here we mentioned these prefixes because they belong to the `prefixed_cmd` operator/operand pair. So to come back to users being able to use primitives instead of funny unreadable aliases. It's good to keep in mind that one can combine prefixes like the following:

```
\frozen    \foo{...}
\immutable \foo{...}
\instance  \foo{...}
\mutable   \foo{...}
\overloaded\foo{...}
\permanent \foo{...}
```

so this is valid too:

```
\global\permanent\untraced\tolerant\protected\def\foo[#1]#*[#2]{...}
```

So what do these prefixes do? It depends on the value of an internal integer `\overloadmode` where the following values have meaning:

		immutable	permanent	primitive	frozen	instance
1	warning	⋆	⋆	⋆		
2	error	⋆	⋆	⋆		
3	warning	⋆	⋆	⋆	⋆	
4	error	⋆	⋆	⋆	⋆	
5	warning	⋆	⋆	⋆	⋆	⋆
6	error	⋆	⋆	⋆	⋆	⋆

The `\enforced` prefix can be used to bypass this mechanism:

```
\permanent\protected\def\foo{...}
```

```
\protected\def\oof{\enforced\def\foo{...}}
```

But only in so called quote ini mode, that is when the format file is created. In order to save work we also have:

```
\aliased\let\foo\relax
```

This makes `\foo` a copy (or more precise, a reference) including all flags, so in this case it will be flagged a a primitive which is `\permanent` too. You cannot define primitives yourself but when reported in a trace you see it being a primitive indeed.

Of course this all means that one has to define basically all relevant macros with a combination of prefixes and that happens to be the case in ConTEXt, which in the end makes this callback a rather ConTEXt specific one.

```
function (
```

```
    <t:boolean> error,
    <t:integer> overload,
    <t:string>  csname,
    <t:integer> flags
)
    -- no return values
end
```

6.4 Fonts

6.4.1 define_font

The engine has no font loader but it does need some information about the glyphs that are uses like width, height and depth, possibly italic correction, kerns, and ligatures. And for math some more information is needed. Keep in mind that for instance italic correction is something specific for TEX and that kerns and ligatures only are needed when you leave them to the engine. For modern OpenType fonts we let Lua deal with this.

```
function (
    <t:string>  name,
    <t:integer> size
)
    return <t:integer> id
end
```

The string `name` is the filename part of the font specification, as given by the user, for instance when `\font` is used for defining an instance. 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 design size of the font.

The font can be defined with `font.define` which returns a font identifier that can be returned in the callback. Contrary to LuaTEX, in LuaMetaTEX we only accept a number.

The internal structure of the `font` table that is passed to `font.define` is explained elsewhere but there can be much more in that table. Likely the macro package will keep the passes table around for other usage, for instance for usage in the backend.

Setting this callback to `false` is pointless because it will prevent font loading completely because without fonts there is little to do for the engine.

6.4.2 quality_font

When you use font expansion you will normally pass the glyph specific expansion and compression values along with the dimensions. However, this can be delayed. When we use par passes (or otherwise set one of the adjust parameters) and a font has not yet bee setup for expansion this callback will kick in but only once per font.

```
function (
```

```
    <t:integer> id
)
    -- no return values
end
```

The function can set additional parameters in the font and pass them to TEX using helpers from the font library.

6.5 Typesetting

6.5.1 pre_output_filter

This callback is called when TEX is ready to start boxing the box 255 for `\output`. The callback does not replace any internal code.

```
function (
    <t:node>    head,
    <t:string>  groupcode,
    <t:integer> size,
    <t:string>  packtype,
    <t:integer> maxdepth,
    <t:integer> direction
)
    return <t:node> newhead
end
```

6.5.2 buildpage_filter

This callback is called whenever LuaMetaTEX 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 (
    <t:string> extrainfo
)
    -- no return values
end
```

The string `extrainfo` gives some additional information about what TEX's state is with respect to the 'current page'. The possible values for this 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`	LuaMetaTeX is terminating (it's all over)

6.5.3 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. The callback does not replace any internal code.

```
function (
    <t:node>    head,
    <t:string>  groupcode,
    <t:integer> size,
    <t:string>  packtype
    <t:integer> direction,
    <t:node>    attributelist
)
    return <t: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.

6.5.4 vpack_filter

This callback is called when TeX is ready to start boxing some vertical mode material. Math displays are ignored at the moment. The callback does not replace any internal code.

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

```
function (
    <t:node>    head,
    <t:string>  groupcode,
    <t:integer> size,
    <t:string>  packtype,
    <t:integer> maxdepth,
    <t:integer> direction,
    <t:node>    attributelist
}
    return <t:node> newhead
end
```

6.5.5 hyphenate

This callback is supposed to insert discretionary nodes in the node list it receives.

```
function (
    <t:node> head,
```

```
    <t:node> tail
)
    -- no return values
end
```

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

6.5.6 ligaturing

This callback, which expects no return values, has to apply ligaturing to the node list it receives.

```
function (
    <t:node> head,
    <t:node> tail
)
    -- no return values
end
```

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.

6.5.7 kerning

This callback has to apply kerning between the nodes in the node list it receives. See `ligaturing` for calling conventions.

```
function (
    <t:node> head,
    <t:node> tail
)
    -- no return values
end
```

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.

6.5.8 glyph_run

When set this callback is triggered when TEX normally handles the ligaturing and kerning. In LuaTEX you use the `hpack_filter` and `per_linebreak_filter` callbacks for that (where each passes different arguments). This callback doesn't get triggered when there are no glyphs (in LuaTEX this optimization is controlled by a a variable).

```
function (
    <t:node>    head,
    <t:string>  groupcode,
    <t:integer> direction
)
    return <t:node> newhead
end
```

The traditional TEX font processing is bypassed so you need to take care of that with the helpers. (For the moment we keep the ligaturing and kerning callbacks but they are kind of obsolete.)

6.5.9 pre_linebreak_filter

This callback is called just before LuaTEX starts converting a list of nodes into a stack of **\hbox**es, after the addition of **\parfillskip**. The callback does not replace any internal code.

```
function (
    <t:node>   head,
    <t:string> groupcode
)
    return <t: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 here, some are for the `hpack_filter` and `vpack_filter` callbacks.

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
- `<t: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

6.5.10 linebreak_filter

This callback replaces LuaTEX's line breaking algorithm. The callback does not replace any internal code.

```
function (
    <t:node>    head,
    <t:boolean> is_display
)
    return <t: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 `<t:node>`, LuaTEX will apply the internal linebreak algorithm on the list that starts at `<head>`. Otherwise, the `<t: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 an **\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’.

6.5.11 post_linebreak_filter

This callback is called just after LuaTEX has converted a list of nodes into a stack of **\hbox**es.

```
function (
    <t:node>   head,
    <t:string> groupcode
)
    return <t:node> newhead
end
```

6.5.12 append_to_vlist_filter

This callback is called whenever LuaTEX adds a box to a vertical list (the `mirrored` argument is obsolete):

```
function (
    <t:node>    box,
    <t:string>  locationcode,
    <t:integer> prevdepth
)
```

```
    return <t:node> list [, <t:integer> prevdepth [, <t:boolean> checkdepth ] ]
end
```

It is ok to return nothing or `nil` 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`. When the third argument returned is `true` the normal prevdepth correction will be applied, based on the first node.

6.5.13 alignment_filter

This is an experimental callback that when set is called several times during the construction of an alignment. The context values are available in `tex.getalignmentcontextvalues()`.

```
function (
    <t:node>   head,
    <t:string> context,
    <t:node>   attributes,
    <t:node>   preamble
)
    -- no return values
end
```

There are no sanity checks so if a user messes up the passed node lists the results can be unpredictable and, as with other node related callbacks, crash the engine.

6.5.14 local_box_filter

Local boxes are a somewhat tricky and error prone feature so use this callback with care because the paragraph is easily messed up. A line can have a left, right and middle box where the middle one has no width. This callback does not replace any internal code. The callback gets quite some parameters passed:

```
function (
    <t:node>    linebox,
    <t:node>    leftbox,
    <t:node>    rightbox,
    <t:node>    middlebox,
    <t:integer> linenumber,
    <t:integer> leftskip,
    <t:integer> rightskip,
    <t:integer> lefthang,
    <t:integer> righthang,
    <t:integer> indentation,
    <t:integer> parinitleftskip,
    <t:integer> parinitrightskip,
    <t:integer> parfillleftskip,
    <t:integer> parfillrightskip,
    <t:integer> overshoot
)
    -- no return values
```

```
end
```

This is an experimental callback that will be tested in different ConTEXt mechanisms before it will be declared stable.

6.5.15 packed_vbox_filter

After the `vpack_filter` callback (see previous section) is triggered the box get packed and after that this callback can be configured to kick in.

```
function (
    <t:node>   head,
    <t:string> groupcode
)
    return <t:node> newhead
end
```

6.5.16 handle_uleader

The `\uleaders` command inserts a user leader into the list. When a list get packed and has such leaders, a run over the list happens after packing so that it can be finalized.

```
function (
    <t:node>    head,
    <t:string>  context,
    <t:integer> index,
    <t:node>    box,
    <t:integer> location
)
    return <t:node> head
end
```

6.5.17 italic_correction

The concept of italic correction is very much related to traditional TEX fonts. At least in 2024 it is absent from OpenType although it has some meaning in OpenType math. In TEX this correction is normally inserted by `\/` although in LuaMetaTEX we also have `\explicititaliccorrection` as well as `\forcedleftcorrection` and `\forcedrightcorrection`.

When this callback is enabled it gets triggered when one of left or right correction commands is given and the returned kern is then used as correction.

```
function (
    <t:node>    glyph,
    <t:integer> kern,
    <t:integer> subtype,
)
    return <t:integer> kern
end
```

6.5.18 insert_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 (
    <t:node>   par,
    <t:string> location
)
    -- no return values
end
```

There is no return value and you should make sure that the node stays valid as otherwise TEX can get confused.

6.5.19 append_line_filter

Every time a line is added this callback is triggered, when set. migrated material and adjusts also qualify as such and the detail relates to the adjust index.

```
function (
    <t:node>    head,
    <t:node>    tail,
    <t:string>  context,
    <t:integer> detail
)
    return <t:node> newhead
end
```

A list of possible context values can be queried with `tex.getappendlinecontextvalues()`.

6.5.20 insert_distance

This callback is called when the page builder 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 (
    <t:integer> class,
    <t:integer> order
)
    return <t:integer> register
end
```

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 `class` equals one) and intermediate space (when `class` is larger than one). Of course you can mess with the insert box but you need to make sure that LuaTEX is happy afterwards.

6.5.21 begin_paragraph

Every time a paragraph starts this callback, when configured, will kick in:

```
function (
    <t:boolean> invmode,
    <t:boolean> indented,
    <t:string>  context
)
    return <t:boolean> indented
end
```

There are many places where a new paragraph can be triggered:

0x00	normal	0x04	dbox	0x08	output	0x0C	math
0x01	vmode	0x05	vcenter	0x09	align	0x0D	lua
0x02	vbox	0x06	vadjust	0x0A	noalign	0x0E	reset
0x03	vtop	0x07	insert	0x0B	span		

6.5.22 paragraph_context

When the return value of this callback is `false` the paragraph related settings, when they have been updated, will not be updated.

```
function (
    <t:string> context
)
    return <t:boolean> ignore
end
```

6.5.23 missing_character

This callback is triggered when a character node is created and the font doesn't have the requested character.

```
function (
    <t:node>    glyph,
    <t:integer> font,
    <t:integer> character
)
    -- no return value
end
```

6.5.24 process_character

This callback is experimental and gets called when a glyph node is created and the callback field in a character is set.

```
function (
    <t:integer> font,
    <t:integer> character
)
```

```
    -- no return value
end
```

6.5.25 tail_append

6.6 Tracing

6.6.1 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 (
    <t:string>  incident,
    <t:integer> detail,
    <t:node>    head,
    <t:integer> first,
    <t:integer> last
 )
    return <t: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).

6.6.2 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 (
    <t:string>  incident,
    <t:integer> detail,
    <t:node>    head,
    <t:integer> first,
    <t:integer> last
)
    -- no return values
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.

6.6.3 line_break

This callback is actually a set of callbacks that has to be deals with as a whole. The main reason why we have this callback is that we wanted to be able to see what the par builder is doing, especially when we implement multiple paragraph building passes. This makes the callback pretty much a rather ConTEXt specific one.

We can also consider fetching the passive and active lists because we now keep much more info around.

```
function(
    <t:integer> context,
    <t:integer> checks,
    ...
)
    -- no return values
end

function initialize (
    <t:integer> context,
    <t:integer> checks,
    <t:integer> subpasses
)
    -- no return values
end

function start (
    <t:integer> context,
    <t:integer> checks,
    <t:integer> pass,
    <t:integer> subpass,
    <t:integer> classes,
    <t:integer> decent
)
    -- no return values
end

function stop (
    <t:integer> context,
    <t:integer> checks,
    <t:integer> demerits
)
    -- no return values
end

function collect (
    <t:integer> context,
    <t:integer> checks
)
    -- no return values
end

function line (
    <t:integer> context,
    <t:integer> checks,
    <t:integer> box,
```

```
    <t:integer> badness,
    <t:integer> overshoot,
    <t:integer> shrink,
    <t:integer> stretch,
    <t:integer> line,
    <t:integer> serial
)
    -- no return values
end

function delete (
    <t:integer> context,
    <t:integer> checks,
    <t:integer> serial
)
    -- no return values
end

function wrapup (
    <t:integer> context,
    <t:integer> checks,
    <t:integer> demerits,
    <t:integer> looseness
)
    -- no return values
end

function check (
    <t:integer> context,
    <t:integer> checks,
    <t:integer> pass,
    <t:integer> subpass,
    <t:integer> serial,
    <t:integer> prevserial,
    <t:integer> linenumber,
    <t:integer> nodetype,
    <t:integer> fitness,,
    <t:integer> demerits,
    <t:integer> classes,
    <t:integer> badness,
    <t:integer> demerits,
    <t:node>    breakpoint,
    <t:integer> short,
    <t:integer> glue,
    <t:integer> linewidth
)
    return <t:integer> demerits  -- optional
end

function list (
```

```
    <t:integer> context,
    <t:integer> checks,
    <t:integer> serial
)
    -- no return values
end
```

Every one of these gets a `context` and `checks` passes. Possible contexts are:

`0x00`	initialize	`0x03`	stop	`0x06`	delete
`0x01`	start	`0x04`	collect	`0x07`	report
`0x02`	list	`0x05`	line	`0x08`	wrapup

The `checks` parameters is the value of **`\linebreakchecks`** which makes it possible to plug in actions depending on that number. To give an idea if what gets called, this is what you get when typesetting `tufte.tex`: initialize, start, report, delete, delete, stop, start, report, report, delete, report, report, report, delete, delete, report, report, report, delete, report, delete, delete, report, report, report, delete, report, delete, delete, report, report, delete, report, report, delete, delete, delete, report, delete, report, delete, delete, report, report, report, delete, delete, report, delete, report, report, delete, report, delete, delete, delete, report, report, delete, report, report, delete, report, delete, report, delete, report, delete, delete, report, report, report, report, delete, delete, delete, delete, delete, delete, delete, delete, delete, report, stop, collect, list, list, list, list, list, list, list, list, list, line, line, line, line, line, line, line, line, line, wrapup.

6.6.4 show_build

You can trace (and even influence) the page builder with this callback. It comes in several variants that are called during the process. Callbacks like these assume that one knows what is going on in the engine.

```
function initialize (
    <t:integer> context
)
    -- no return values
end

function step (
    <t:integer> context,
    <t:node>    current,
    <t:integer> pagegoal,
    <t:integer> pagetotal
)
    -- no return values
end

function check (
    <t:integer> context,
    <t:node>    current,
    <t:boolean> moveon,
    <t:boolean> fireup,
```

```
    <t:integer> badness,
    <t:integer> costs,
    <t:integer> penalty
)
    return <t:boolean> moveon, <t:boolean> fireup
end

function skip (
    <t:integer> context,
    <t:node>    current,
)
    -- no return values
end

function move (
    <t:integer> context,
        <t:node>    current,
        <t:integer> lastheight,
        <t:integer> lastdepth,
        <t:integer> laststretch,
        <t:integer> lastshrink,
        <t:boolean> hasstretch
)
    -- no return values
end

function fireup (
    <t:integer> context,
    <t:node> current
)
    -- no return values
end

function wrapup (
    <t:integer> context
)
    -- no return values
end
```

6.6.5 show_whatsit

Because we only have a generic whatsit it is up to the macro package to provide details when tracing them.

```
function (
    <t:node>    whatsit,
    <t:integer> indentation,
    <t:integer> tracinglevel,
    <t:integer> currentlevel,
    <t:integer> inputlevel
```

```
)
    -- no return value
end
```

Here `indentation` tells how many periods are to be typeset if you want to be compatible with the rest of tracing. The `tracinglevel` indicates if the current level and/or input level are shown cf. **`\tracinglevels`**. Of course one is free to show whatever in whatever way suits the whatsit best.

6.6.6 linebreak_quality

```
function (
    <t:node>    par,
    <t:integer> id,
    <t:integer> pass,
    <t:integer> subpass,
    <t:integer> subpasses,
    <t:integer> state,
    <t:integer> overfull,
    <t:integer> underfull,
    <t:integer> verdict,
    <t:integer> classified,
    <t:integer> line
)
    return <t:node> result
end
```

6.6.7 show_loners

In spite of widow, club, broken and shaping penalties we can have single lines in the result. When set, this callback replaces the output that normally **`\tracingloners`** produces.

```
function (
    <t:integer> options,
    <t:integer> penalty
)
    return <t:node> result
end
```

The options are those set on the encountered penalty:

`0x0000`	normal	`0x0010`	clubbed	`0x0200`	shaping
`0x0001`	mathforward	`0x0020`	toddlered	`0x0400`	double
`0x0002`	mathbackward	`0x0040`	widow	`0x0800`	doubleused
`0x0004`	orphaned	`0x0080`	club	`0x1000`	factorused
`0x0008`	widowed	`0x0100`	broken	`0x2000`	endofpar

6.6.8 get_attribute

Because attributes are abstract pairs of indices and values the reported properties makes not much sense and are very macro package (and user) dependent. This callback permits more verbose reporting by the engine when tracing is enabled.

```
function (
    <t:integer> index,
    <t:integer> value
)
    return <t:string>, <t:string>
end
```

6.6.9 get_noad_class

We have built-in math classes but there can also be user defined ones. This callback can be used to report more meaningful strings instead of numbers when tracing.

```
function (
    <t:integer> class
)
    return <t:string>
end
```

6.6.10 get_math_dictionary

todo

6.6.11 show_lua_call

When the engine traces something that involves a Lua call it makes sense to report something more meaningful than just that. This callback can be used provide a meaningful string (like the name of a function).

```
function (
    <t:string>  name,
    <t:integer> index
)
    return <t:string>
end
```

6.6.12 trace_memory

When the engine starts all kind of memory is pre-allocated> depending on the configuration more gets allocated when a category runs out of memory. The LuaMetaTEX engine is more dynamic than LuaTEX. If this callback is set it will get called as follows:

```
function (
    <t:string>  category,
```

```
    <t:boolean> success
)
    -- no return value
end
```

The boolean indicates if the allocation has been successful. One can best quit the run when this one is `false` which the engine is likely to do that anyway, be in in a less graceful way that you might like.

6.6.13 paragraph_pass

This callback is not yet stable.

6.7 Math

6.7.1 mlist_to_hlist

This callback replaces LuaTEX's math list to node list conversion algorithm.

```
function (
    <t:node>    head,
    <t:string>  display_type,
    <t:boolean> need_penalties
)
    return <t: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.

6.7.2 math_rule

In math rules are used for fractions, radicals and accents. In the case of radicals rules mix with glyphs to build the symbol. In ConTEXt we can enable an alternate approach that uses glyphs instead of rules so that we can have more consistent shapes, for instance with slopes or non square endings. This callback takes care of that.

```
function (
    <t:integer> subtype,
    <t:integer> font,
    <t:integer> width,
    <t:integer> height,
    <t:node>    attributes
)
    return <t:node> rule
end
```

6.7.3 make_extensible

Like `math_rule` this callback is used to construct nicer extensibles in ConTEXt math support. It can optionally be followed by `register_extensible`.

```
function (
    <t:node>    extensible,
    <t:integer> fnt,
    <t:integer> chr,
    <t:integer> size,
    <t:integer> width,
    <t:integer> height,
    <t:integer> depth,
    <t:integer> linewidth,
    <t:integer> axis,
    <t:integer> exheight,
    <t:integer> emwidth
)
    return <t:node> boxed extensible
end
```

6.7.4 register_extensible

This callback is a possible follow up on `make_extensible` and it can be used to share pre-build extensibles or package them otherwise (for instance as Type3 glyph).

```
function (
    <t:integer> fnt,
    <t:integer> chr,
    <t:integer> size,
    <t:node>    attributes,
    <t:node>    extensible
)
    return <t:node> boxed
end
```

fonts

7 Fonts

7.1 Introduction

The LuaTEX engine changed the approach to loading fonts and processing kerns and ligatures by introducing a Lua loader and callbacks for processing replacement and positioning features. In LuaMetaTEX we go a step further and no longer load fonts otherwise than with Lua. In the end, all that TEX needs are a few dimensions and optionally ligature and kerning tables. Of course for math a bit more is needed but even there we can safely delegate all loading to Lua. In LuaMetaTEX we still have the traditional kerning and ligature built in because after all that method is the reference for traditional fonts and the amount of code needed is relatively small.

The backend is gone, so here the final font inclusion is also done by Lua. This means that in the engine the amount of code involved in that is zero. In the engine we have glyphs and glyphs traditionally carry a font identifier (an number) and a glyph reference (also a number). Both are used to fetch the width, height, depth, italic correction ans some more from the fonts registered in the engine. For TEX a font is more of an abstraction that from Lua, where we can manipulate details and deal with the real shapes.

In LuaMetaTEX the situation is simplified on the one hand, read: no font loader, but complicated on the other, for instance because we have dynamic scaling. In this chapter we discuss what data is stored in glyphs, what primitives are involved, and how loading takes place. Because a lot can be done in Lua and because there are no standards involved, we don't need to discuss how a macro package is supposed to deal with all this; one can consider ConTEXt as a reference implementation if needed.

Removing the font loader and backend had relatively little impact on ConTEXt because we already did most in Lua, but as we developed LuaMetaTEX both subsystems evolved further. Especially moving more backend processing to Lua had some impact on performance but in the end the engine is much faster so we gained that back. Additions to the font system, like dynamic scaling of course have impact too but we could also limit the amount of fonts that get loaded which compensates for any loss in performance. The most complicated and demanding part of the backend code is that what deals with fonts: sharing, subsetting, devirtualizing, scaling, effects like weight, slanting, expansion, accuracy, accessibility, ..., all of that has to be dealt with.

In this chapter we discuss a few aspects like primitives, defining fonts, Lua helpers, and virtual fonts, but for a more complete picture one really has to read the documents that describe how all evolved, how fonts are used in ConTEXt as well as look at how we apply all this. There is no reason to repeat everything here, especially because for most users this is not something they need to know. There are dedicated manuals and articles that cover different aspects.

7.2 Primitives

7.2.1 Basic properties

Although primitives are discusses in their own chapter we repeat some here because it impacts following sections. Let's start with the commands that change the look and feel of a font:

```
\begingroup                  glyphs represent characters \endgroup
\begingroup \glyphscale  1200 glyphs represent characters \endgroup
```

```
\begingroup \glyphxscale 1200 glyphs represent characters \endgroup
\begingroup \glyphyscale  800 glyphs represent characters \endgroup
\begingroup \glyphslant   200 glyphs represent characters \endgroup
\begingroup \glyphweight  200 glyphs represent characters \endgroup
```

This results in:

glyphs represent characters
glyphs represent characters
glyphs represent characters
glyphs represent characters
glyphs represent characters
glyphs represent characters

These parameters are applied to glyphs that get added to the current list of nodes. Whenever the engine (or the Lua end) needs a dimension, two scales have to be applied, depending on the dimension being horizontal or vertical. Sometimes the slant and weight also have to be taken into account. Later we will see that we have additional math scaling so you can imagine that applying a handful of scales has a bit of impact on the code and also performance. However, the later will not be noticed because computers are fast enough.

Here is how we can apply the scaling factors to dimensions:

```
{\glyphxscale 1500                  \the\glyphxscaled 100pt} and
{\glyphyscale  750                  \the\glyphyscaled 100pt} and
{\glyphscale  1500 \glyphxscale 500 \the\glyphxscaled 100pt}
```

We get: 150.0pt and 75.0pt and 75.0pt. In scenarios like these you need to keep in mind that the currently set scales also apply. The main reason why we use these 1000 based factor is that it is the way TEX does things. We could have used posits instead but those were added later so for now it's factors that dominate.

7.2.2 Specifications

A font is loaded at a specific size, so these properties start from that: the design size and the requested size which results in a scaling factor. Every font has a number so here we have:

```
\tf \the \fontid \font \hskip1cm
\bf \the \fontid \font \hskip1cm
\sl \the \fontid \font
```

1 **4** *10*

A set of settings can be combined in specification, here **\font** is the current font, from which the specification takes the identifier.

```
\fontspecdef \MyFontA \font xscale 2000 yscale 800 weight 200 slant 200 \relax
\fontspecdef \MyFontB \font all 1000 1500 800 250 150 \relax
```

```
\begingroup \MyFontA Is this neat or not? \endgroup
\begingroup \MyFontB Is this neat or not? \endgroup
```

Is this neat or not?

Is this neat or not?

Instead of an id an already defined specification can be given in which case we start from a copy:

```
\fontspecdef \MyFontA 2 all 1000
\fontspecdef \MyFontB \MyFontA xscale 1200
```

Say that we have:

```
\fontspecdef\MyFoo\font xscale 1200 \relax
```

The four properties of such a specification can then be queried as follows:

```
[\the\fontspecid       \MyFoo]
[\the\fontspecscale    \MyFoo]
[\the\fontspecxscale   \MyFoo]
[\the\fontspecyscale   \MyFoo]
[\the\fontspecifiedsize\MyFoo]
[\fontspecifiedname    \MyFoo]
```

[1] [1000] [1200] [1000] [10.0pt] [Serif sa 1]

A font specification obeys grouping but is not a register. Like `\integerdef` and `\dimendef` it is just a control sequence with a special meaning.

If you read about compact font mode in ConTEXt, this is what we're using there. It started out by more aggressive sharing and scaling but eventually all five properties were integrated in a fast font switch. However, setting these five properties, even with one command has some overhead because they are saved on the save stack. Okay, that was a bit if a lie: no one will notice that overhead:

```
\fontspecdef \MyFontA \font
    scale 1100 xscale 2000 yscale 800 weight 200 slant 200
\relax
\fontspecdef \MyFontB \font
    scale 1200 xscale 1000 yscale 200 weight 100 slant 100
\relax
```

A 100.000 times `{\MyFontA\MyFontB}` grouped expansion takes 0.02 seconds runtime on my 2018 laptop, which is just noise once we start processing text: 100.000 times `{\MyFontA efficient \MyFontB efficient}` takes 1.4 seconds and 100.000 times `{\MyFontA test \MyFontB test}` takes 0.4 seconds. Guess why.

7.2.3 Offsets

These two parameters control the horizontal and vertical shift of glyphs with, when applied to a stretch of them, the horizontal offset probably being the least useful. The values default to the currently set values. Here is a ConTEXt example:

```
\ruledhbox {%
    \ruledhbox {\glyph yoffset 1ex xoffset -.5em 123}
    \ruledhbox {\glyph yoffset 1ex               125}
    \ruledhbox \bgroup
```

```
        baseline
        \glyphyoffset 1ex \glyphxscale 800 \glyphyscale \glyphxscale
        raised%
    \egroup
\egroup
```

Visualized:

{ } baseline raised

7.2.4 Math scales and identifiers

More details about fonts in math mode can be found in the chapters about math and primitives so here we just mention a few of these primitives. The internal `\glyphtextscale`, `\glyphscriptscale` and `\glyphscriptscriptscale` registers can be set to enforce additional scaling of math, like this:

```
$                             a = b^2 = c^{d^2}$
$\glyphtextscale          800 a = b^2 = c^{d^2}$
$\glyphscriptscale        800 a = b^2 = c^{d^2}$
$\glyphscriptscriptscale 800 a = b^2 = c^{d^2}$
```

You can of course set them all in any mix as long as the value is larger than zero and doesn't exceed 1000. In ConTEXt we use this for special purposes so don't mess with it there. as there can be unexpected (but otherwise valid) side effects.

$a = b^2 = c^{d^2}$
$a = b^2 = c^{d^2}$
$a = b^2 = c^{d^2}$
$a = b^2 = c^{d^2}$

The next few reported values depend on the font setup. A math font can be loaded at a certain scale and further scaled on the fly. An open type math font comes with recommended script and script script scales and gets passed to the engine scaled. The values reported by `\mathscale` are *additional* scales.

```
$\the\mathscale\textfont         \zerocount$
$\the\mathscale\scriptfont       \zerocount$
$\the\mathscale\scriptscriptfont\zerocount$
```

gives: 1000 1000 1000

In math mode the font id depends on the style because there we have a family of three related fonts or the same font with different scales. In this document we get the following identifiers:

```
$\the\mathstylefontid\scriptscriptstyle \fam$
$\the\mathstylefontid\scriptstyle       \fam$
$\the\mathstylefontid\textstyle         \fam$
```

Gives: 2 2 2, which is no surprise because we use the same font for all sizes combined with the `smaller` field options discusses later. In ConTEXt math uses compact font mode with in-place scaling by default.

7.2.5 Scaled fontdimensions

When you use `\glyphscale`, `\glyphxscale` and/or `\glyphyscale` the font dimensions also scale. The values that are currently used can be queried:

dimension	scale	xscale	yscale
\scaledemwidth	⋆	⋆	
\scaledexheight	⋆		⋆
\scaledextraspace	⋆	⋆	
\scaledinterwordshrink	⋆	⋆	
\scaledinterwordspace	⋆	⋆	
\scaledinterwordstretch	⋆	⋆	
\scaledslantperpoint	⋆	⋆	

The next table shows the effective sized when we scale by 2000. The last two columns scale twice: the shared scale and the x or y scale.

\scaledemwidth	20.0	20.0	10.0	40.0	20.0
\scaledexheight	10.38086	5.19043	10.38086	10.38086	20.76172
\scaledextraspace	2.11914	2.11914	1.05957	4.23828	2.11914
\scaledinterwordshrink	2.11914	2.11914	1.05957	4.23828	2.11914
\scaledinterwordspace	6.35742	6.35742	3.17871	12.71484	6.35742
\scaledinterwordstretch	3.17871	3.17871	1.58936	6.35742	3.17871
\scaledslantperpoint	0.0	0.0	0.0	0.0	0.0

7.2.6 Character properties

The **\fontcharwd**, **\fontcharht**, **\fontchardp** and **\fontcharic** give access to character properties. To this repertoire LuaMetaTEX adds the top and bottom accent accessors **\fontcharta** and **\fontcharba** that came in handy for tracing. You pass a font reference and character code. Normally only OpenType math fonts have this property.

7.2.7 Glyph options

In LuaTEX the \noligs and \nokerns primitives suppress these features but in LuaMetaTEX these primitives are gone. They are replace by a more generic control primitive **\glyphoptions**. This numerical parameter is a bitset with the following fields:

0x00000000	normal
0x00000001	noleftligature
0x00000002	norightligature
0x00000004	noleftkern
0x00000008	norightkern
0x00000010	noexpansion
0x00000020	noprotrusion
0x00000040	noitaliccorrection
0x00000080	nozeroitaliccorrection
0x00000100	applyxoffset
0x00000200	applyyoffset
0x00000400	mathdiscretionary
0x00000800	mathsitalicstoo
0x00001000	mathartifact
0x00002000	weightless
0x00004000	spacefactoroverload
0x00008000	checktoddler
0x00010000	checktwin
0x00020000	istoddler
0x00100000	userfirst
0x40000000	userlast

The effects speak for themselves. They provide detailed control over individual glyph, this because the current value of this option is stored with glyphs. In ConTEXt we have commands that set flags like that and also make sure that there is no interference in setting them. it's good to know that some of these options are there so that we can properly demonstrate, discuss and document LuaMetaTEX behavior. The current value of this parameter is `0x18080` but that can of course change because we experiment with options and bit positions might change over time, which is why we can query the engine.

7.3 Nodes

This chapter is not about nodes so we keep this section short. A glyph node is an important one and a page easily has a few thousand of them. When a list that has glyphs nodes is processed, depending on the font quite some passes over that list are made in order to sort out substitutions, alternatives and ligatures as well as font kerning and anchoring. When the paragraph is constructed these glyphs are consulted and dimensions and expansion properties are accessed and scaling can happen. These glyph nodes are among the largest and have many fields. To what extend you can use these fields depends on the macro package and the reason is that some of these fields also affect the backend and the backend is provided by the macro package. When the script/language combination that you use supports hyphenation, there can be discretionary nodes that have a pre, post and/or replace component set that are node lists that can contain glyph nodes and whenever we mess around with glyphs we also need to check these.

The most important fields are `font` and `character`, as these uniquely point to what shape is used. That also means that at the Lua end we can have more information than TEX needs and can do things that TEX in its role as constructor is unaware of. The par builder doesn't really care what it deals with, it only needs dimensions and maybe some properties.

The `data`, `state`, `script` and `protected` fields are used for instance by ConTEXt and in particular the font handler. There are primitives that can query and set these fields, like **`\glyphdatafield`**, **`\glyphscriptfield`** and **`\glyphstatefield`**.

These primitives can be used to set an additional glyph properties. Of course it's very macro package dependent what is done with that. It started with just the first one as experiment, simply because we had some room left in the glyph data structure. It's basically an single attribute. Then, when we got rid of the ligature pointer we could either drop it or use that extra field for some more, and because ConTEXt already used the data field, that is what happened. The script and state fields are shorts, that is, they run from zero to `0xFFFF` where we assume that zero means 'unset'. Although they can be used for whatever purpose their use in ConTEXt is fixed. So far for a historical note.

The `language` field is used by the hyphenator but can also be used by the macro package. The `lhmin` and `rhmin` are only useful for the hyphenator and these values are set by the language mechanisms and primitives. The `discpart` bitset registers what the engine did which can be handy for tracing.

We already mentioned scales, slant and weight and these go to fields `scale`, `xscale`, `yscale`, `slant` and `weight`. The `expansion`, `raise`, `left`, `right`, `xoffset` and `yoffset` can be set by TEX but also by the font handler. Messing with any of these fields at the TEX end is easy but one really should take into account what the macro packages needs them for and does with them at the Lua end and in the backend. In that respect LuaMetaTEX lets the user free but it also means that you cannot expect macro packages (assuming that ConTEXt is not the only user) to behave the same.

The various math subsystems use `properties`, `group` and `index` and again this also macro package specific. The `options` bitset controls all kind of processes in the engine when it comes to using glyphs (user level **`\glyphoptions`**) as do `control` and `hyphenate`.

It would take many pages to explain all this so again we just refer to how ConTEXt uses these fields, the way they can be set from TEX and accessed in Lua. In the end, all the users see of this is shapes anyway, while macro packages integrate and present these as features.

7.4 Loading

A font is normally defined by **\font** which in LuaMetaTEX is just a trigger for a callback. You can even do without that primitive because you can load a font and then use **\setfontid** or the previously mentioned specification to switch to a font. The callback, discussed in the callbacks chapter, gets a name and size, and is supposed to return a font identifier. You can use the name to locate and load a font, register the font using the following function, which gives you an identifier that satisfies the callback.

```
function font.define ( <t:table> font, <t:integer> id )
    return <t:integer> id
end
```

with respect to **\font** it's good to know that the engine accept a braced argument as a font name:

```
\font\myfont = {My Fancy Font}
```

This allows for embedded spaces, without the need for double quotes. Macro expansion takes place inside the argument. Although in ConTEXt LMTX we don't use the **\font** for defining fonts, it still can be uses.

The font table is mandate but the identifier is optional. The table has the following fields, most of which concern math. The `name` field is mandate because it is needed in various feedback scenarios.

key	type	description
`name`	string	metric (file) name
`original`	string	the name used in logging and feedback
`designsize`	number	expected size (default: 655360 == 10pt)
`size`	number	the required scaling (by default the same as `designsize`)
`compactmath`	boolean	use the `smaller` fields in lookups
`mathcontrol`	bitset	this controls various options in the math engine
`textcontrol`	bitset	this controls various options in the text engine
`nomath`	boolean	don't check for math parameters and properties
`characters`	table	the defined glyphs of this font
`fonts`	table	locally used fonts
`parameters`	table	parameters by index and/or key
`MathConstants`	table	OpenType math parameter
`hyphenchar`	number	default: TEX's **\hyphenchar**
`skewchar`	number	default: TEX's **\skewchar**
`textscale`	number	scale applied to math text
`scriptscale`	number	scale applied to math script
`scriptscriptscale`	number	scale applied to math script script
`textxscale`	number	horizontal scale applied to math text
`scriptxscale`	number	horizontal scale applied to math script

`scriptxscriptscale`	number	horizontal scale applied to math script script
`textyscale`	number	vertical scale applied to math text
`scriptyscale`	number	vertical scale applied to math script
`scriptyscriptscale`	number	vertical scale applied to math script script
`textweight`	number	weight applied to math text
`scriptweight`	number	weight applied to math script
`scriptscriptweight`	number	weight applied to math script script

There are three tables that need their own explanation. The `parameters` table is a hash with mixed key types. There are seven possible string keys, as well as a number of integer indices. The seven strings are actually used instead of the bottom seven indices, because that gives a nicer user interface. There are additional indexed entries possible for math fonts but nowadays one will use OpenType math fonts so these no longer make sense.

name	index
`slant`	1
`space`	2
`spacestretch`	3
`spaceshrink`	4
`xheight`	5
`quad`	6
`extraspace`	7

The `characters` table can be pretty large when we have OpenType fonts. In ConTEXt we use Unicode as encoding which means that glyphs are organized as such. This also means that we have a hash and not an indexed array due to gaps. There can be more data in the glyph sub tables than the engine needs because the engine only picks up those that it needs. You can also later decide to pass additional properties and even glyphs to the engine, but changes can of course have consequences because at some point the backend will pick up data and use that. Additions are fine but changes have to be consistent. Of course it all depends on how you implement a backend.

When a character in the input is turned into a glyph node, it gets a character code that normally refers to an entry in that table. For proper paragraph building and math rendering the fields in the tables below can best be present in an entry in the `characters` table. As said, you can of course add all kind of extra fields. The engine only uses those that it needs for typesetting a paragraph or formula. The sub tables that define ligatures and kerns are also hashes with integer keys, and these indices should point to entries in the main characters table. The fields common to text and math chartacters are: `callback`, `compression`, `depth`, `expansion`, `height`, `italic`, `kerns`, `leftprotrusion`, `ligatures`, `rightprotrusion`, `tag`, `width`.

Providing ligatures and kerns via this table permits TEX to construct ligatures and add inter-character kerning. However, normally you will use an OpenType font in combination with Lua code that does this. In ConTEXt we have base mode that uses the engine, and node mode that uses Lua. A mono spaced font normally has no ligatures and inter character kerns and is normally not processed at all.

We can group the parameters. All characters have the following base set. It must be noted here that OpenType doesn't have a italic property and that the height and depth are also not part of the design: one can choose to derive them from the bounding box.

key	type	description

`width`	number	width in sp (default 0)
`height`	number	height in sp (default 0)
`depth`	number	depth in sp (default 0)
`italic`	number	italic correction in sp (default 0)

There are four parameters that are more optional and relate to advanced optical paragraph optimization:

key	type	description
`leftprotruding`	number	left protruding factor (**`\lpcode`**)
`rightprotruding`	number	right protruding factor (**`\rpcode`**)
`expansion`	number	expansion factor (**`\efcode`**)
`compression`	number	compression factor (**`\cfcode`**)

The left and right protrusion factors as well as the expansion factor are comparable to the ones introduced by pdfTEX, but compression is new and complements expansion. In LuaMetaTEX the expansion mechanism is also available in math. You might have noticed that we don't have expansion related parameters in the main font table. This is because we have a more dynamic model. These values are anyway only used when **`\protrudechars`** and/or **`\adjustspacing`** are set. The later can also be controlled by so called par passes and thereby applied more selectively. Because setting these fields using specific glyph properties can take time, it is also possible to delay these settings till a dedicated callback is triggered when they are needed.

From TEX we inherit the following tables. Ligatures are only used in so call base mode, when the engine does the font magic. Kerns are used in base mode text and optionally in math.

key	type	description
`ligatures`	table	ligaturing information
`kerns`	table	kerning information

The next fields control the engine and are a variant on TEX's tfm tag property. In a future we might provide a bit more (local) control although currently we see no need. Originally the `tag` and `next` field were combined into a packed integer but in current LuaMetaTEX we have a 32 bit tag and the next field moved to the math blob as it only is used as variant selector.

key	type	description
`tag`	number	a bitset, currently not really exposed

In a math font characters have many more fields: `bottomanchor`, `bottomleft`, `bottommargin`, `bottomovershoot`, `bottomright`, `extensible`, `flataccent`, `innerlocation`, `innerxoffset`, `inneryoffset`, `keepbase`, `leftmargin`, `mathkerns`, `mirror`, `parts`, `rightmargin`, `smaller`, `topanchor`, `topleft`, `topmargin`, `topovershoot`, `topright`.

key	type	description
`smaller`	number	the next smaller math size character
`mirror`	number	a right to left alternative
`flataccent`	number	an accent alternative with less height (OpenType)
`next`	number	‘next larger’ character index

`topleft`	number	alternative script kern
`topright`	number	alternative script kern
`bottomleft`	number	alternative script kern
`bottomright`	number	alternative script kern
`topmargin`	number	alternative accent calculation margin
`bottomargin`	number	alternative accent calculation margin
`leftmargin`	number	alternative accent calculation margin
`rightmargin`	number	alternative accent calculation margin
`topovershoot`	number	accent width tolerance
`bottomovershoot`	number	accent width tolerance
`topanchor`	number	horizontal top accent alignment position
`bottomanchor`	number	horizontal bottom accent alignment position
`innerlocation`	string	`left` or `right`
`innerxoffset`	number	radical degree horizontal position
`inneryoffset`	number	radical degree vertical position
`parts`	table	constituent parts of an extensible
`partsitalic`	number	the italic correction applied with the extensible
`partsorientation`	number	`horizontal` or `vertical`
`mathkerns`	table	math cut-in specifications
`extensible`	table	stretch a fixed width accent to fit

In LuaMetaTEX combined with ConTEXt MkXL we go beyond OpenType math and have more fields here than in LuaTEX. In ConTEXt those values are set with so called tweaks and defined in so called font goody files. This relates to the extended math rendering engine in LuaMetaTEX.

Bidirectional math is also supported and driven by (in ConTEXt speak) tweaks which means that it has to be set up explicitly as it uses a combination of fonts. The `mirror` field points to an alternative glyph. The `smaller` field points to a script glyph alternative and that glyph can then point to a script script one (in OpenType speak `ssty` alternates respectively one 1 and 2). In ConTEXt is also uses specific features of the font subsystems that hook into the backend where we have a more advanced virtual font subsystem than in LuaTEX. Because this is macro package dependent it will not be discussed here.

Here is the character ‘f’ (decimal 102) in the font `cmr10 at 10pt`. The numbers that represent dimensions are in scaled points. Of course you will use Latin Modern OpenType instead but the principles are the same:

```
[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 }
    }
}
```

In ConTEXt, when they are really needed, we normally turn these traditional eight bit fonts into emulated OpenType (Unicode) fonts so there you will only encounter tables like this when we process a font in base mode.

Two very special string indexes can be used also: `leftboundary` is a virtual character whose ligatures and kerns are used to handle word boundary processing. `rightboundary` is similar but not actually used for anything (yet).

The values of `topanchor`, `bottomanchor` and `mathkern` are used only for math accent and superscript placement, see page ?? in this manual for details. The italic corrections are a story in themselves and discussed in detail in other manuals. The additional parameters that deal with kerns, margins, overshoots, inner anchoring, etc. are engine specific and not part of OpenType. More information can be found in the ConTEXt distribution; they relate the upgraded math engine project by Mikael and Hans.

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 `parts`, 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 `parts` entry is an arrays of components. Each of those components is itself a hash of up to five keys:

key	type	description
`glyph`	number	character index
`extender`	number	(1) if this part is repeatable, (0) otherwise
`start`	number	maximum overlap at the starting side (scaled points)
`end`	number	maximum overlap at the ending side (scaled points)
`advance`	number	advance width of this item (width is default)

The traditional (text and math) `kerns` table is a hash indexed by character index (and 'character index' is defined as either a non-negative integer or the string value `rightboundary`), with the values of the kerning to be applied, in scaled points.

The traditional (text) `ligatures` table is a hash indexed by character index (and 'character index' is defined as either a non-negative integer or the string value `rightboundary`), 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'.

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

The default value is 0, and can be left out. That signifies a 'normal' ligature where the ligature replaces both original glyphs. In this table the | indicates the final insertion point.

The third table has the `MathConstants` as the camel case name suggests. These are not discussed here. The `fonts` table relates to virtual fonts that are discussed later.

7.5 Helpers

Without argument this function returns the current font identifier and when an identifier is passed that one is made current.

```
function font.current ( <t:nil> | <t:integer> )
    -- no return value
end
```

This returns the maximum font identifier in use:

```
function font.max ( )
    return <t:integer> -- identifier
end
```

This one defines a font but needs an identifier, for instance reserved by `font.nextid`. The table is the same as with `font.define`.

```
function font.setfont ( <t:integer> identifier, <t:table> data )
    -- no return value
end
```

The next function can be used to add characters to a font. The table is the same as the table used when defining the characters in a font. The identifier must be known.

```
function font.addcharacters ( <t:integer> identifier, <t:table> characters )
    -- no return value
end
```

When protrusion or expansion data is needed for a character in a font and the relevant values are not yet known, a callback can be triggered and the next function can then be used to assign these.

```
function font.addquality (
    <t:integer> identifier,
    <t:table>   characters
)
    -- no return value
end
```

The table looks like this:

```
{
    [index] = {
        leftprotrusion  = <t:integer>,
        rightprotrusion = <t:integer>,
        expansion       = <t:integer>,
        compression     = <t:integer>,
    },
    ...
}
```

Sometimes it can be handy to check what the next identifier will be. The optional boolean, when true, makes that the font is allocated.

```
function font.nextid ( <t:nil> | <t:boolean> )
    return <t:integer> -- identifier
end
```

This function does a lookup by name and returns the font identifier when it's known:

```
function font.id ( <t:string> name )
    return <t:integer> -- identifier
end
```

The value that gets returned or is assigned is always an integer because that is what these parameters are: scaled dimensions, percentages, factors.

```
function font.getfontdimen (
    <t:integer> identifier,
    <t:integer> parameter
)
    return <t:integer> -- value
end
```

```
function font.setfontdimen (
```

```
    <t:integer> identifier,
    <t:integer> parameter,
    <t:integer> value
)
    -- no return value
end
```

This one returns the properties that relate to a `\fontspecdef`:

```
function font.getfontspec ( <t:string> name )
    return
        <t:integer>, -- identifier
        <t:integer>, -- scale
        <t:integer>, -- xscale
        <t:integer>, -- yscale
        <t:integer>, -- slant
        <t:integer>  -- weight
end
```

Math characters are not really defined along with a font but their family can bind them to one. However, in ConTEXt we have them decoupled and families are assigned fonts when the need is there.

```
function font.getmathspec ( )
    return
        <t:integer>, -- class
        <t:integer>, -- family
        <t:integer>  -- character

end
```

Internally a math font parameter has a number. This function returns that number plus a boolean indicating if we have an variable that is not officially in OpenType math but an addition to the LuaMetaTEX engine.

```
function font.getmathindex ( <t:string> | <t:number> )
    return
        <t:number>  -- index
        <t:boolean> -- engine
end
```

These two don't operate on a font but multiply the given value by the `\glyphscale` and `\glyphxscale` respectively `\glyphyscale`.

```
function font.xscaled ( <t:number> value)
    return <t:number> -- scaled value
end
```

```
function font.yscaled ( <t:number> value)
    return <t:number> -- scaled value
end
```

Like in other places the engine can report what fields we have, which is handy when we want to check manuals like this one.

```
function font.getparameterfields     ( ) return <t:table> end
function font.getfontfields          ( ) return <t:table> end
function font.gettextcharacterfields ( ) return <t:table> end
function font.getmathcharacterfields ( ) return <t:table> end
```

7.6 Virtual fonts

Virtual fonts have been introduced in TEX because they permit combining fonts and constructing for instance accented characters from several glyphs and they are what one nowadays tags as a ‘cool’ feature, especially because in LuaTEX we can use this mechanism runtime. The nice thing is that because all that TEX needs is dimensions, the hard work is delegated to the backend which means that the front end can be agnostic when it comes to virtual fonts.

So, in the beginning they were mostly used for providing a direct mapping from for instance accented characters onto a glyph btu we use it for a lot of other situations, like math. But keep in mind that because we basically define the backend ourselves and because we also control everything fonts, we can go way further in ConTEXt than in other engines and macro packages.

A character is virtual when it has a `commands` array as part of the data. 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).

At the font level there can be a an (indexed) `fonts` 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 in for instance a virtual font, you can use the `slot` command with a zero font reference, which indicates that the font itself is used. So, a table looks like this:

```
fonts = {
  { name = "ptmr8a", size = 655360 }, -- referenced as font 1
  { name = "psyr", size = 600000 },   -- referenced as font 2
  { id = 38 }                         -- referenced as font 3
}
```

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 LuaMetaTEX as font id 38. The array index numbers are used by the character command definitions that are part of each character.

However, the only place in ConTEXt where we really need this `fonts` table is in some math fonts where we, also as illustration and as recognition of past work, assemble a Unicode math font from sort of obsolete Type1 fonts. In most cases the virtual glyphs use glyphs that are also in the font. In that case we can use `id` zero which is resolved to the font identifiers of the font itself.

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 frontend is only interested in the dimensions, ligatures and kerns of a font, which is the reason why the TEX engine didn't have to be extended when virtual fonts showed up: dealing with it is up to the driver that comes after the backend. The first block in the next table is what the standard mentions. These two engines also support the `special` and LuaTEX brings the `pdf` and `pdfmode` commands but in LuaMetaTEX we dropped all three and also LuaTEX's `image`.

But ... in LuaMetaTEX there is no backend built in but we might assume that the one provided deals with the standard entries. However, a provided backend can provide more and that is indeed what happens in ConTEXt. Because we no longer have compacting (of passed tables) and unpacking (when embedding) of these tables going on we stay in the Lua domain. None of the virtual specification is ever seen in the engine.

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
`slot`	2	2 numbers	a shortcut for the combination of a font and char command
`push`	0		save current position
`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
`nop`	0		do nothing
`node`	1	node	output this node (list), and move right by the width of this list
`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
`comment`	any	any	the arguments of this command are ignored

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.

In addition to the above in ConTEXt we have `use`, `left`, `up`, `offset`, `stay`, `compose`, `frame`, `line`, `inspect`, `trace` and a plugin feature so that we can add more commands (which we do). These not only provide more advanced trickery but also make for smaller command tables. For some features we don't even need virtual magic but have additional parameters in the glyph tables. But all that is not part of the engine and its specification so it will be discussed elsewhere.

7.7 Callbacks

The traditional TEX ligature and kerning routines are build into the engine but anything more (like OpenType rendering) has to be implemented in Lua. The same is true for math: the engine has some expectations, for instance with respect to script and script script sizes, larger sizes and extensibles and needs to know at least dimensions and slots in fonts in order to assemble the math. Actually there are additional scaling factors in play here because math has its own scaling demands.

7.8 Protrusion

This is more an implementation note. Compared to pdfTEX and LuaTEX 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?»
```

7.9 Spaces

There are officially no spaces in TEX, there is only glue. This is not problem, on the contrary, it is what makes the rendering so good. In ConTEXt the backend can convert glue to spaces in a font but that's not an engine feature.

The `\nospaces` 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 7.1 we see the results for four characters separated by a space.

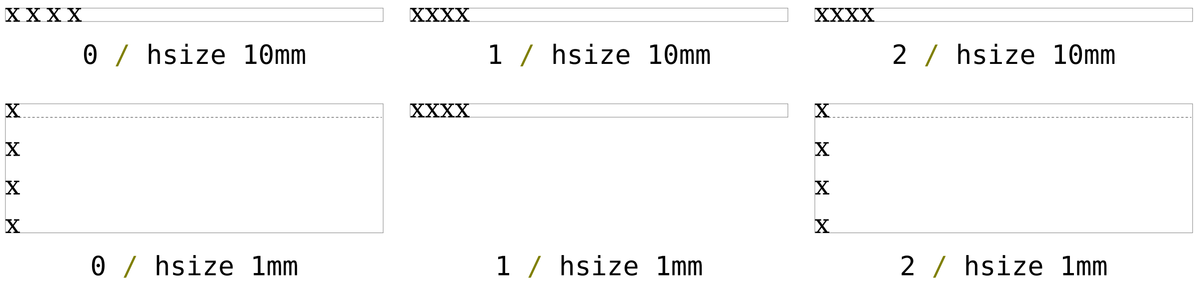


Figure 7.1 The nospaces options.

You can, in ConTEXt see where spaces are added by enabling a visualizer: `\showmakeup[space]` does the trick, as in this paragraph. We see regular spaces as well as spaces that have a space factor applied (after punctuation).

languages

8 Languages

8.1 Introduction

Although languages play an important role in a macro package that doesn't mean that TEX is busy with it. The engine only needs to know how to hyphenate and for that a number that identifies what patterns to use is sufficient. All the action happens in the hyphenator: what characters make words, how many characters are kept at the left and right, which symbols end up at the end or beginning of a line, what input combine into (normally) dashes, how do we penalize a hyphenation point, etc.

Where in regular TEX we have special nodes that signal a language switch, and some shared variables that determine mentioned details, in LuaTEX every glyph carries the language information, including those minima. In LuaMetaTEX we put even more in a glyph by using a bitset of options. We also have some more character code bound properties. The LuaTEX engines store the current state in the glyph and discretionary nodes.

You can find more practical information about languages in ConTEXt manuals than in this document because users seldom go low level. Before we discuss these low level aspect anyway, we discuss how we came thus far; for that we borrow from the LuaTEX and LuaMetaTEX manuals.

8.2 Evolution

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. No hyphenation is needed in that case.

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.

Lets stress this: before LuaTEX ligaturing and kerning took place during input, and hyphenation, combined with temporarily juggling ligatures and kerns, took place while building the paragraph. It's a selective process where hyphenation only takes place where it is expected to influence the line breaks.

Those `char` node records are somewhat misnamed, as they are glyph positions in specific fonts, and therefore not really ‘characters’ in the linguistic sense. In TEX82 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 and thereby LuaMetaTEX 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. Actually in LuaMetaTEX the subtype is no longer used to store the state but that is not relevant here.

In LuaMetaTEX we also have this separation but there is more control over when hyphenation is applied, what becomes en- and em-dashes, hoe penalties kick in, etc. There are some additional callbacks that can manipulate words as they are encountered and exceptions can be handled differently.

8.3 Characters, glyphs and discretionaries

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 LuaMetaTEX we no longer keep the list of components with the glyph node because we have to deal with more advanced scenarios in 'node mode', for instance in attaching vowels to stepwise constructed ligatures. Also, in OpenType ligatures are just a many to one mapping and the kind of ligatures that we see TEX fonts in OpenType often are achieved by kerning substituted single glyphs.

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', 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.

But while TEX86 has this construct, deconstruct and reconstruct model in LuaTEX we don't do that so in the end this made little sense do we dropped it. We still have a (small) protection field that fulfills the job of signaling that we're done with processing glyphs.

We now arrive at languages. 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). In LuaMetaTEX we just use small dedicated fields instead.

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. In LuaMetaTEX we have a more reasonable allowance because we don't expect that many languages in one document, but we do permits longer words.

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. But this is true for more properties: for instance we store a penalty in a discretionary node and freeze glue in spaces, of course all at the price of using more memory.

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. Also, every list is hyphenated so that the font handler can do it's job taking that into account.

In traditional TEX the process of hyphenation is driven by `\lccode`s. In LuaTEX we made this dependency less strong. There are several strategies possible. When you do nothing, the currently used `\lccode`s 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 `\lccode`s 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 the value 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 two characters. 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.

We also have **\hccode** or hyphen code. A character can be set to non zero to indicate that it should be regarded as value visible hyphenation point. These examples show how that works (it is the second bit in **\hyphenationmode** that does the magic but we set them all here):

```
{\hsize 1mm \hccode"2014 \zerocount  \hyphenationmode "0000000 xxx\emdash xxx \par}
{\hsize 1mm \hccode"2014 "2014\relax \hyphenationmode "0000000 xxx\emdash xxx \par}

{\hsize 1mm \hccode"2014 \zerocount  \hyphenationmode "FFFFFFF xxx\emdash xxx \par}
{\hsize 1mm \hccode"2014 "2014\relax \hyphenationmode "FFFFFFF xxx\emdash xxx \par}

{\hyphenationmode "0000000 xxx--xxx---xxx \par}
{\hyphenationmode "FFFFFFF xxx--xxx---xxx \par}
```

Here we assign the code point because who knows what future extensions will bring. As with the other codes you can also set them from Lua. The feature is experimental and might evolve when ConTEXt users come up with reasonable demands.

xxx—xxx
xxx—
xxx
xxx—xxx
xxx—
xxx
xxx--xxx---xxx
xxx–xxx—xxx

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
```

dis-
crete—
dis-
crete

```
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`, `insert`, and `adjust` node indicate a start or end. You can omit the last set from the test by setting flags in **`\hyphenationmode`**:

`0x000001`	normal
`0x000002`	automatic
`0x000004`	explicit
`0x000008`	syllable
`0x000010`	uppercase
`0x000020`	compound
`0x000040`	strictstart
`0x000080`	strictend
`0x000100`	automaticpenalty
`0x000200`	explicitpenalty
`0x000400`	permitglue
`0x000800`	permitall
`0x001000`	permitmathreplace
`0x002000`	forcecheck
`0x004000`	lazyligatures
`0x008000`	forcehandler
`0x010000`	feedbackcompound
`0x020000`	ignorebounds
`0x040000`	collapse

The word start is determined as follows:

node	behaviour
boundary	yes when wordboundary
hlist	when the start bit is set
vlist	when the start bit is set
rule	when the start bit is set
dir	when the start bit is set
whatsit	when the start bit is set
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 the end bit is set

vlist	when the end bit is set
rule	when the end bit is set
dir	when the end bit is set
whatsit	when the end bit is set
ins	when the end bit is set
adjust	when the end bit is set

Figures 8.1 upto 8.5 show some examples. In all cases we set the min values to 1 and make sure that the words hyphenate at each character.

0	64	128	192
o- n- e	o- n- e	o- n- e	o- n- e

Figure 8.1 `one`

0	64	128	192
o- n- et- w- o	o- n- et- w- o	onet- w- o	onet- w- o

Figure 8.2 `one\null two`

0	64	128	192
o- n- et- w- o	o- n- et- w- o	onet- w- o	onet- w- o

Figure 8.3 `\null one\null two`

0	64	128	192
o- n- et- w- o	o- n- et- w- o	onetwo	onetwo

Figure 8.4 `one\null two\null`

0	64	128	192
o- n- et- w- o	o- n- et- w- o	onetwo	onetwo

Figure 8.5 `\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).

Hyphenation and discretionary injection is driven by a mode parameter which is a bitset made from the following values, some of which we saw in the previous examples.

1	honour (normal) `\discretionary`'s
2	turn `-` into (automatic) discretionaries
4	turn `\-` into (explicit) discretionaries
8	hyphenate (syllable) according to language
10	hyphenate uppercase characters too (replaces `\uchyph`
20	permit break at an explicit hyphen (border cases)
40	traditional TEX compatibility wrt the start of a word
80	traditional TEX compatibility wrt the end of a word
100	use `\automatichyphenpenalty`
200	use `\explicithyphenpenalty`
400	turn glue in discretionaries into kerns
800	okay, let's be even more tolerant in discretionaries
1000	and again we're more permissive
4000	controls how successive explicit discretionaries are handled in base mode
2000	treat all discretionaries equal when breaking lines (in all three passes)
8000	kick in the handler (experiment)
10000	feedback compound snippets

Some of these options are still experimental, simply because not all aspects and side effects have been explored. You can find some experimental use cases in ConTEXt.

There are also `\discretionaryoptions`. Some are set by the engine:

0x00000000	normalword
0x00000001	preword
0x00000002	postword
0x00000010	preferbreak
0x00000020	prefernobreak
0x00000040	noitaliccorrection
0x00000080	nozeroitaliccorrection
0x00010000	userfirst
0x40000000	userlast

8.4 Controlling hyphenation

The `\hyphenationmin` parameter 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.

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.

8.5 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 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. 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.[12]

   The **`\defaulthyphenchar`** parameter is used as fallback when defining a font where that one is not explicitly set.

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

[12] 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.

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.

4. 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**.

5. 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.

6. 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 behavior configurable a few new primitives have been added:

```
\automatichyphenpenalty
\explicithyphenpenalty
```

These relate to:

```
\automaticdiscretionary % -
\explicitdiscretionary  % \-
```

The usage of these penalties is controlled by the **\hyphenationmode** flags `0x100` and `0x200` and when these are not set **\exhyphenpenalty** is used.

You can use the **\tracinghyphenation** variable to get a bit more information about what happens.

value	effect
1	report redundant pattern (happens by default in LuaTEX)
2	report words that reach the hyphenator and got treated
3	show the result of a hyphenated word (a node list)

8.6 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 **\chardef**d 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 `language` 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 related 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 zza- -ba- -bzz 123	123 za- -ba- -ba- -ba- -bz 123	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- -bzza- -bz 123	123 za- -bzza- -bz 123	123 zzzza- -bz zzzzzz zzzzzz za- -bzzzz 123

8.7 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 `language.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 `language.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 `language.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.

8.8 Applying ligatures and kerning

We discuss this base mode aspect here because in traditional TEX the process is interwoven with hyphenation. 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 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.

In so called base mode, where TEX does the work, the ligature construction (normally) goes in small steps. An `f` followed by an `f` becomes an `ff` ligatures and that one followed by an `i` can become a `ffi` ligature. The situation can be complicated by hyphenation points between these characters. When there are several in a ligature collapsing happens. Flag `0x4000` in the `\hyphenationmode` variable determines if this happens lazy or greedy, i.e. the first hyphen wins or the last one does. In practice a ConTEXt user won't have to deal with this because most fonts are processed in node mode.

8.9 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 there can be no 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. In practice that doesn't matter much because the par builder has enough solution space due to spaces; it's not like out of a sudden we wonder why paragraphs look worse.

The **\doublehyphendemerits** and **\finalhyphendemerits** parameters play a role in the par builder: they discourage a page break when there are two or more hyphens in a row and if there's one in the pre-last line. These are not bound to a language.

8.10 The `language` library

This library provides the interface to the internal structure representing a language, and the associated functions.

```
function language.new ( <t:nil> | <t:integer> identifier )
    return <t:userdata> -- language
end
```

This function creates a new userdata object. An object of type `<language>` is the first argument to most of the other functions in the `language` 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. The number returned is the internal **\language** id number this object refers to.

```
function language.id ( <t:userdata> language )
    return <t:integer> -- identifier
end
```

You can load exceptions with:

```
function language.hyphenation( <t:userdata> language, <t:string> list)
    -- no return value
end
```

When no string is given (the first example) a string with all exceptions is returned.

```
function language.hyphenation ( <t:userdata> language )
    return <t:string> list
end
```

This either returns the current hyphenation exceptions for this language, or adds new ones. The syntax of the string is explained in section 8.6.

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

```
function language.clearhyphenation( <t:userdata> language )
    --- no return value
end
```

This function creates a hyphenation key from the supplied hyphenation value. The syntax of the argument string is explained in section 8.6. The function is useful if you want to do something else based on the words in a dictionary file, like spell-checking.

```
function language.clean(<t:userdata> language, <t:string> str)
    return <t:string> cln
end
```

```
function language.clean(<t:string> str)
    return <t:string> cln
end
```

This adds additional patterns for this language object, or returns the current set. The syntax of this string is explained in section 8.6.

```
function language.patterns( <t:userdata> language, <string> list )
    -- no return value
end
```

The registered list can be fetched with:

```
function language.patterns( <t:userdata> language )
    return <t:string> -- list
end
```

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

```
function language.clearpatterns ( <t:userdata> language )
    -- no return value
end
```

This function sets (or gets) the value of the TeX parameter **\hyphenationmin**.

```
function language.hyphenationmin ( <t:userdata> language, <t:number> n )
    -- no return value
end
```

```
function language.hyphenationmin ( <t:userdata> language )
    return <t:integer> n
end
```

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

```
function language.prehyphenchar  ( <t:userdata> language, <t:integer> n) end
function language.posthyphenchar ( <t:userdata> language, <t:integer> n) end
```

```
function language.prehyphenchar  ( <t:userdata> language) return <t:integer> n end
function language.posthyphenchar ( <t:userdata> language) return <t:integer> n end
```

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).

```
function language.preexhyphenchar  ( <t:userdata> language, <t:integer> n) end
function language.postexhyphenchar ( <t:userdata> language, <t:integer> n) end
```

```
function language.preexhyphenchar  ( <t:userdata> language) return <t:integer> n end
function language.postexhyphenchar ( <t:userdata> language) return <t:integer> n end
```

The next 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 optionally `tail`) are proper nodes, regardless of possible other errors.

```
function language.hyphenate( <t:node> head, <t:node> tail)
    return <t:boolean> success
end
```

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 8.3 for more details.

The following two commands can be used to set or query a **\hjcode**:

```
function language.sethjcode (
    <t:userdata> language,
    <t:number>   character,
    <t:number>   usedchar
)
    -- no return value
end
```

```
function language.gethjcode (
    <t:userdata> language,
    <t:number>   character
)
    return <t:number> -- usedchar
end
```

There are similar function for **\hccode**:

```
function language.sethccode (
    <t:userdata> language,
    <t:number>   character,
    <t:number>   usedchar
)
    -- no return value
end
```

```
function language.gethccode (
    <t:userdata> language,
    <t:number>   character
)
    return <t:number> -- usedchar
end
```

8.11 Math

For the record we mention that in math you can also have discretionaries:

```
$ 2x \mathdiscretionary{+}{+}{+} 1 = 3y $
```

these actually do relate to languages but are not stored in the language data but have to be handled by the macro package. It will be clear that there is a bit involved because we have spacing and penalties driven by math classes.

8.12 Tracing

There are several trackers in ConTEXt that can show where hyphenation was considered and where it got applied, but this is really macro package dependent. There is also a built in tracing command: `\tracinghyphenation`. When you say:

```
\tracinghyphenation2
\tracingonline     2
```

You get something like this:

```
1:3: [language: not hyphenated There]
1:3: [language: hyphenated several at 1 positions]
1:3: [language: hyphenated trackers at 1 positions]
1:3: [language: not hyphenated where]
1:3: [language: hyphenated hyphenation at 2 positions]
1:3: [language: hyphenated considered at 2 positions]
1:3: [language: not hyphenated where]
1:3: [language: hyphenated applied at 1 positions]
1:3: [language: hyphenated really at 1 positions]
1:3: [language: not hyphenated macro]
1:3: [language: hyphenated package at 1 positions]
1:3: [language: hyphenated dependent at 2 positions]
1:3: [language: not hyphenated There]
1:3: [language: not hyphenated built]
1:3: [language: hyphenated tracing at 1 positions]
1:3: [language: hyphenated command at 1 positions]
1:3: [language: hyphenated tracinghyphenation at 3 positions]
```

```
Higher values give more details, like the pre, post and replace lists so that
output is rather noisy. Contrary to \type {\tracinghyphenation} is verbatim we do
permit it \type {\tracinghyphenation} to be hyphenated.
```

renders as:

Higher values give more details, like the pre, post and replace lists so that output is rather noisy. Contrary to `\tracinghyphenation` is verbatim we do permit it `\tracinghyphenation` to be hyphenated.

and traces as:

```
1:3: [language: hyphenated renders at 1 positions]
1:4: [language: not hyphenated Higher]
1:4: [language: hyphenated values at 1 positions]
1:4: [language: hyphenated details at 1 positions]
1:4: [language: hyphenated replace at 1 positions]
1:4: [language: not hyphenated lists]
1:4: [language: hyphenated output at 1 positions]
1:4: [language: not hyphenated rather]
1:4: [language: not hyphenated noisy]
1:4: [language: hyphenated Contrary at 1 positions]
1:4: [language: hyphenated verbatim at 2 positions]
1:4: [language: hyphenated permit at 1 positions]
1:4: [language: hyphenated hyphenated at 2 positions]
1:3: [language: not hyphenated traces]
```

lua

9 Lua

9.1 Introduction

In this chapter aspects of the Lua interfaces will be discusses. The `lua` module described here is rather low level and probably not of much interest to the average user as its functions are meant to be used in higher level interfaces.

9.2 Initialization

9.2.1 A bare bone engine

When the LuaMetaTEX program launches it will not do much useful. You can compare it to computer hardware without (high level) operating system with a TEX kernel being the bios. It can interpret TEX code but for typesetting you need a reasonable setup. You also need to load fonts, and for output you need a backend, and both can be implemented in Lua. If you don't like that and want to get up and running immediately, you will be more happy with LuaTEX, pdfTEX or XƎTEX, combined with your favorite macro package.

If you just want to play around you can install the ConTEXt distribution which (including manuals and some fonts) is tiny compared to a full TEXLive installation and can be run alongside it without problems. If there are issues you can go to the usual ConTEXt support platforms and seek help where you can find the people who made LuaTEX and LuaMetaTEX.

If you use the engine as TEX interpreter you need to set up a few characters. Of course one can wonder why this is the case, but let's consider this to be educational of nature as it right from the start forces one to wonder what category codes are.

```
\catcode`\{=1 \catcode`\}=2 \catcode`\#=6
```

After that you can start defining macros. Contrary to LuaTEX the LuaMetaTEX engine initializes all the primitives but it will quit when the minimal set of callback is not initialized, like a logger. The lack of font loader and backend makes that it is not usable for loading an arbitrary macro package that doesn't set up these components. There is simply no argument for starting in in original TEX mode without ε-TEX extensions and such.

9.2.2 LuaMetaTEX as a Lua interpreter

Although LuaMetaTEX is primarily meant as a TEX engine, it can also serve as a stand alone Lua interpreter and there are two ways to make LuaMetaTEX behave like one. The first method uses the command line option `--luaonly` followed by a filename. The second is more automatic: if the only non-option argument (file) on the command line has the extension `lmt` or `lua`. The `luc` extension has been dropped because bytecode compiled files are not portable and one can always load them indirectly. The `lmt` suffix is more ConTEXt specific and makes it possible to have files for LuaTEX and LuaMetaTEX alongside.

In interpreter mode, the program 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 does. The program will exit immediately after executing the specified Lua script and is thereby effectively just a somewhat bulky stand alone Lua interpreter with a bunch of extra preloaded libraries. But we still wanted and managed to keep the binary small, somewhere around 3MB, which is okay for a script engine.

When no argument is given, LuaMetaTEX will look for a Lua file with the same name as the binary and run that one when present. This makes it possible to use the engine as a stub. For instance, in ConTEXt a symlink from `mtxrun` to type luametatex will run the `mtxrun.lmt` or `mtxrun.lua` script when present in the same path as the binary itself. As mentioned before first checking for (ConTEXt) `lmt` files permits different files for different engines in the same path.

9.2.3 Other commandline processing

When the LuaMetaTEX executable starts, it looks for the `--lua` command line option. If there is no such option, the command line is interpreted in a similar fashion as the other TEX engines. All options are accepted but only some are understood by LuaMetaTEX itself:

commandline argument	explanation
`--credits`	display credits and exit
`--fmt=FORMAT`	load the format file `FORMAT`
`--help`	display help and exit
`--ini`	be `iniluatex`, for dumping formats
`--jobname=STRING`	set the job name to `STRING`
`--lua=FILE`	load and execute a Lua initialization script
`--version`	display version and exit
`--permitloadlib`	permits loading of external libraries

There are less options than with LuaTEX, because one has to deal with them in Lua anyway. So for instance there are no options to enter a safer mode or control executing programs because this can easily be achieved with a startup Lua script, which can interpret whatever options got passed.

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. The Lua libraries that don't deal with TEX are initialized rather soon so you have these available.

LuaMetaTEX 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). 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.

So let's summarize this. The handling of what is called job name is a bit complex. There can be explicit names set on the command line but when not set they can be taken from the `texconfig` table.

startup filename	`--lua`	`a Lua file`	
startup jobname	`--jobname`	`a TEX tex`	`texconfig.jobname`
startup dumpname	`--fmt`	`a format file`	`texconfig.formatname`

These names are initialized according to `--luaonly` or the first filename seen in the list of options. Special treatment of `&` and `*` as well as interactive startup is gone but we still enter TEX via an forced **`\input`** into the input buffer.[13]

When we are in TEX mode at some point the engine needs a filename, for instance for opening a log file. At that moment the set jobname becomes the internal one and when it has not been set which internalized to jobname but when not set becomes `texput`. When you see a `texput.log` file someplace on your system it normally indicates a bad run.

The command line option `--permitloadlib` has to be given when you load external libraries via Lua. Although you could manage this via Lua itself in a startup script, the reason for having this as option is the wish for security (at some point that became a demand for LuaTEX), so this might give an extra feeling of protection.

9.3 Lua behaviour

9.3.1 The Lua version

We currently use Lua version 5.5 and will follow developments of the language but normally with some delay. Therefore the user needs to keep an eye on (subtle) differences in successive versions of the language. Here are a few examples.

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. It is best not to depend the automatic cast from string to number and vise versa as this can change in future versions.

When Lua introduced bitwise operators, instead of providing functions in the `bit32` library, we wanted to use these. The solution in ConTEXt was to implement a macro subsystem (kind of like what C does) and replace the function calls by native bitwise operations. However, because LuajitTEX didn't evolve we dropped that and when we split the code base between MkIV and MkXL we went native bitwise. The `bit32` library is still there but implemented in Lua instead.

9.3.2 Locales

In stock Lua, many things depend on the current locale. In LuaMetaTEX, we can't do that, because it makes documents non-portable. While LuaMetaTEX is running if forces the following locale settings:

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

[13] This might change at some point into an explicit loading triggered via Lua.

```
LC_NUMERIC=C
```

There is no way to change that as it would interfere badly with the often language specific conversions needed at the TEX end.

9.4 Lua modules

Of course the regular Lua modules are present. In addition we provide the `lpeg` library by Roberto Ierusalimschy, 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 that need more than one byte, 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 multi-byte 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.

There are some more libraries present. For instance we embed `luasocket` but contrary to LuaTEX don't embed the related Lua code but some patched and extended variant. The `luafilesystem` module has been replaced by a more efficient one that also deals with the MS Windows file and environment properties better (Unicode support in MS Windows dates from before utf8 became dominant so we need to deal with wide Unicode16). We don't have a Unicode library because we always did conversions in Lua, but there are some helpers in the `string` library, which makes sense because Lua itself is now also becoming Unicode aware.

There are more extensive math libraries and there are libraries that deal with encryption and compression. There are also some optional libraries that we do interface but that are loaded on demand. The interfaces are as minimal as can be because we so much in Lua, which also means that one can tune behavior to usage better.

9.5 Files

9.5.1 File syntax

LuaMetaTEX 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. Keep in mind that as side effect of delegating io to Lua the `\openin` primitive is not provided by the engine and has to be implemented by the macro package. This also means that the limit on the number of open files is not enforced by the engine.

9.5.2 Writing to file

Writing to a file in TEX has two forms: delayed and immediate. Delayed writing means that the to be written text is anchored in the node list and flushed by the backend. As all io is delegated to Lua, this also means that it has to deal with distinction. In LuaTEX the number of open files was already bumped to 127, but in LuaMetaTEX it depends on the macro package. The special meaning of channel 18 was already dropped in LuaTEX because we have `os.execute`.

9.6 Testing

For development reasons you can influence the used startup date and time. By setting the `start_time` variable in the `texconfig` table; as with other variables we use the internal name there. When Universal Time is needed, set the entry `use_utc_time` in the `texconfig` table.

In ConTEXt we provide the command line argument `--nodates` that does a bit more than disabling dates; it avoids time dependent information in the output file for instance.

9.7 Helpers

9.7.1 Basics

The `lua` library is relatively small and only provides a few functions. There are many more helpers but these are organized in specific modules for file i/o, handling strings, and manipulating table.

The Lua interpreter is stack bases and when you put a lot of values on the stack it can overflow. However, if that is the case you're probably doing something wrong. The next function returns the current top and is mainly there for development reasons.

```
function lua.getstacktop ( )
    return <t:integer>
end
```

The next example:

```
\startluacode
context(lua.getstacktop())
context(lua.getstacktop(1,2,3))
context(lua.getstacktop(1,2,3,4,5,6))
\stopluacode
```

typesets: 036, so we're fine.

```
\startluacode
context(lua.getstacktop(unpack(token.getprimitives())))
\stopluacode
```

But even this one os okay: 1134, because some thousand plus entries is not bothering the engine. Of course it makes little sense because now one has to loop over the arguments.

The engines exit code can be set with:

```
function lua.setexitcode ( <t:integer> )
    -- no return values
end
```

and queried with:

```
function lua.getexitcode ( )
    return <t:integer>
end
```

The name of the startup file, in our case ‘`cont-en.lui`’ with the path part stripped, can be fetched with:

```
function lua.getstartupfile ( )
    return <t:string>
end
```

The current Lua version, as reported by the next helper, is Lua 5.5.

```
function lua.getversion ( )
    -- return todo
end
```

We provide high resolution timers so that we can more reliable do performance tests when needed and for that we have time related helpers. The `getruntime` function returns the time passed since startup. The `getcurrenttime` does what its name says. Just play with them to see how it pays off. The `getpreciseticks` returns a number that can be used later, after a similar call, to get a difference. The `getpreciseseconds` function gets such a tick (delta) as argument and returns the number of seconds. Ticks can differ per operating system, but one always creates a reference first and then use deltas to this reference.

9.7.2 Timers

```
function lua.getruntime ( )
    return <t:number> -- actually an integer
end

function lua.getcurrenttime ( )
    return <t:number> -- actually an integer
end

function lua.getpreciseticks ( )
    return <t:number> -- actually an integer
end

function lua.getpreciseseconds ( <t:number> ticks )
    return <t:number>
end
```

There is a little bit of duplication in the timers; here is what they report at this stage of the current run:

library	function	result
lua	`getruntime`	9.8034820556640625
	`getcurrenttime`	1730370968.9371567
	`getpreciseticks`	4405057915613.0
	`getpreciseseconds`	440505.791649

os	clock	9.803
	time	1730370968
	gettimeofday	1730370968.9371581

9.7.3 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.

```
function lua.setbytecode (
    <t:integer>  register,
    <t:function> loader,
    <t:boolean>  strip
)
    -- no return values
end
```

An example of a valid call is lua.setbytecode(5,loadfile("foo.lua")). The complement of this helper is:

```
function lua.getbytecode ( <t:integer> register )
    retrurn <t:bytecode>
end
```

The codes are stored in the virtual table lua.bytecode. The contents of this 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 lua.callbytecode ( <t:integer> register )
    -- <t:boolean> -- success
end
```

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.

9.7.4 Tables

You can preallocate tables with these two helpers. The first one preallocates the given amount of hash entries and index entries. The newindex function create an indexed table with default values:

```
function lua.newtable ( <t:integer> hashsize, <t:integer> indexsize )
    return <t:table>
end

function lua.newindex ( <t:integer> size, <t:whatever> default )
    return <t:table>
end
```

9.7.5 Nibbles

Nibbles are half bytes so they run from `0x0` upto `0xF`. When we needed this for math state fields, the helpers made it here.

```
function lua.setnibble ( <t:integer> original, <t:integer> position, <t:integer>
  value )
    return <t:integer>
end
```

```
function lua.getnibble ( <t:integer> original, <t:integer> position )
    return <t:integer>
end
```

```
function lua.addnibble ( <t:integer> original, <t:integer> position, <t:integer>
  value )
    return <t:integer>
end
```

```
function lua.subnibble ( <t:integer> original, <t:integer> position, <t:integer>
  value )
    return <t:integer>
end
```

Here a a few examples (positions go from right to left and start at one):

`lua.setnibble(0x0000,2,0x1)`	0x0010
`lua.setnibble(0x0000,4,0x7)`	0x7000
`lua.getnibble(0x1234,2)`	0x3
`lua.getnibble(0x1234,4)`	0x1
`lua.addnibble(0x0000,2)`	0x0010
`lua.addnibble(0x0030,2)`	0x0040
`lua.subnibble(0x00F0,2)`	0x00E0
`lua.subnibble(0x0080,2)`	0x0070

9.7.6 Functions

The functions table stores functions by index. The index can be used with the primitives that call functions by index. In order to prevent interferences a macro package should provide some interface to the function call mechanisms, just like it does with registers.

```
function lua.getfunctionstable ( )
    return <t:table>
end
```

9.7.7 Tracing

The engine also includes the serializer from the `luac` program, just because it can be interesting to see what Lua does with your code.

```
function luac.print ( <t:string> bytecode, <t:boolean> detailed )
    -- nothing to return
end
```

metapost

10 Metapost

10.1 Introduction

Four letters in the name LuaMetaTEX refer to the graphical subsystem MetaPost, originally writen by John Hobby as follow up on MetaFont. This library was introduced in LuaTEX in order to generate graphics runtime instead of via a separate system call. The library in LuaTEX is also used for the standalone program so it has a PostScript backend as well as font related frontend. The version used in LuaMetaTEX has neither. The lack of a backend can be explained from the fact that we have to provide one anyway: the pdf output is generated by Lua, which at that time was derived from the converter that I wrote for pdfTEX, although there the starting point is the PostScript output. Removing the font related code also makes sense, because in MkIV we never used it: we need to support OpenType and also want to use properly typeset text so we used a different approach (**textext** and friends).

Another difference with the LuaTEX library is that we don't support the binary number model, which removes a dependency. We kept decimal number support and also opened that up to the TEX end via Lua. In addition we support the posit number model, mostly because we also have that at the TEX end to suit the 32 bit model. The repertoire of scanners and injectors has been enlarged which makes it easier and more efficient to interface between the LuaMetaTEX subsystems. We also added functionality to MetaPost, the language and processor. From the users perspective the library is downward compatible but at the same time it offers more.

Just as LuaTEX is frozen, the MetaPost library that it uses can be considered frozen. In LuaMetaTEX we have plans for some more extensions. We don't discuss the already present new functionality here in detail, for that we have separate manuals, organized under the LuaMetaFun umbrella. After all, most of what we did was done in the perspective of using ConTEXt. Users don't use the functions discussed below because they only make sense in a more integrated approach as with LuaMetaFun.

10.2 Instances

Before you can process MetaPost code an instance needs to be created. There can be multiple instances active at the same time. They are isolated from each other so they can use different number models and macro sets. Although you can do without files, normally you will load (for instance) macros from a file. This means that we need to interface the library to the file system. If we want to run Lua, we need to be able to load Lua code. All this is achieved via callbacks that have to be set up when an instance is created.

```
function mplib.new (
    {
        random_seed    = <t:integer>,
        interaction    = <t:string>,
        job_name       = <t:string>,
        find_file      = <t:function>,
        open_file      = <t:function>,
        run_script     = <t:function>,
        run_internal   = <t:function>,
        make_text      = <t:function>,
        math_mode      = <t:string>,
```

```
        utf8_mode      = <t:boolean>,
        text_mode      = <t:boolean>,
        show_mode      = <t:boolean>,
        halt_on_error  = <t:boolean>,
        run_logger     = <t:function>,
        run_overload   = <t:function>,
        run_error      = <t:function>,
        run_warning    = <t:function>,
        bend_tolerance = <t:number>,
        move_tolerance = <t:number>,
    }
)
    return <t:mp>
end
```

The library is fed with MetaPost snippets via an `execute` function. We will discuss this in a while.

```
function mplib.execute ( <t:mp> instance )
    return <t:table> -- results
end
```

An instance can be released with:

```
function mplib.finish ( <t:mp> instance )
    return <t:table> -- results
end
```

Keeping an instance open is more efficient than creating one per graphic especially when a format has to be loaded. When you execute code, there can be results that for instance can be converted into pdf and included in the currently made document. If one closes an instance it can be that there are pending results to take care of, although in practice that is unlikely to happen.

When the `utf8_mode` parameter is set to `true` characters with codes 128 upto 255 can be part of identifiers. There is no checking if we have valid utf but it permits to use that encoding. In ConTEXt, of course, we enable this. When `text_mode` is `true` you can use the characters with ascii `STX` (2) and `ETC` (3) to fence string literals so that we can use double quotes in strings without the need to escape them. The `math_mode` parameter controls the number model that this instance will use. Valid values are `scaled` (default), `double` (default in ConTEXt), `binary` (not supported), `decimal` (less performing but maybe useful) and `posit` (so that we can complements the TEX end).

Valid `interaction` values are `batch`, `nonstop`, `scroll`, `errorstop` (default) and `silent` but in ConTEXt only the last one makes sense. Setting the `random_seed` parameter makes it possible to reproduce graphics and prevent documents to be different each run when the size of graphics are different due to randomization. The `job_name` parameter is used in reporting and therefore it is mandate.

Both tolerance parameters default to 131/65536 = 0.001998901 and help to make the output smaller: ‘bend’ relate to straight lines and ‘move’ to effectively similar points. You can adapt the tolerance any time with:

```
function mplib.settolerance (
    <t:mp>     instance,
    <t:number> bendtolerance,
```

```
    <t:number> movetolerance
)
    -- no return values
end

function mplib.gettolerance ( <t:mp> instance )
    return
        <t:number>, -- bendtolerance
        <t:number>  -- movetolerance
end
```

Next we detail the functions that are hooked into the instance because without them being passed to the engine not that much will happen. We start with the finder. Here `mode` is `w` or `r`. Normally a lookup of a file that is to be read from is done by a dedicated lookup mechanism that knows about the ecosystem the library operates in (like the TEX Directory Structure).

```
function find_file (
    <t:string> filename,
    <t:string> mode,
    <t:string> filetype | <t:integer> index
)
    return <t:string> -- foundname
end
```

A (located) file is opened with the `open_file` callback that has to return a table with a `close` method and a `reader` or a `writer` dependent of the `mode`.

```
function open_file (
    <t:string> filename,
    <t:string> mode,
    <t:string> filetype | <t:integer> index
)
    return {
        close  = function()
            -- return nothing
        end,
        reader = function()
            return <t:string>
        end,
        writer = function(<t:string>)
            -- return nothing
        end
    }
end
```

This approach is not that different from the way we do this at the TEX so like there a reader normally returns lines. The way MetaPost writes to and read from files using primitives is somewhat curious which is why the file type can be a number indicating what handle is used. However, apart from reading files that have code using **`input`** one hardly needs the more low level read and write related primitives.

The runner is what makes it possible to communicate between MetaPost and Lua and thereby TEX. There are two possible calls:

```
function run_script ( <t:string> code | <t:integer> reference )
    return <t:string> metapost
end
```

The second approach makes it possible to implement efficient interfaces where code is turned into functions that are kept around. At the MetaPost end we therefore have, as in LuaTEX:

```
runscript "some code that will be loaded and run"
% more code
runscript "some code that will be loaded and run"
```

which can of course be optimized by caching, but more interesting is:

```
newinternal my_function ; my_function := 123 ;
runscript my_function ;
% more code
runscript my_function ;
```

which of course has to be dealt with in Lua. The return value can be a string but also a direct object:

```
function run_script (
    <t:string> code | <t:integer> reference,
    <t:boolean> direct
)
    return
        <t:boolean> | <t:number> | <t:string> | <t:table>, -- result
        <t:boolean>                                        -- success
end
```

When the second argument is true, the results are injected directly and tables become pairs, colors, paths, transforms, depending on how many elements there are.

In MetaPost internal variables are quantities that are stored a bit differently and are accessed without using the expression scanner. The `run_internal` function triggers when internal MetaPost variables flagged with `runscript` are initialized, saved or restored. The first argument is an action, the second the value of internal. When we initialize an internal a third and fourth argument are passed.

```
function run_internal (
    <t:integer> action,
    <t:integer> internal,
    <t:integer> category,
    <t:string>  name
)
    -- no return values
end
```

The category is one of the types that MetaPost also uses elsewhere: integer, boolean, numeric or known. From this you can deduce that internals in LuaMetaTEX can not only be numbers but also strings or booleans. The possible actions are:

0	initialize
1	save
2	restore

There is of course bit extra overhead involved than normal but that can be neglected especially because we can have more efficient calls to Lua using references stored in internals. It also has the benefit that one can implement additional tracing.

MetaPost is a graphic language and system and typesetting text is not what it is meant for so that gets delegated to (for instance) TEX using **btex** which grabs text upto **etex** and passes it to this callback:

```
function make_text ( <t:string> str, <t:integer> mode )
    return <t:string> -- metapost
end
```

Here mode is only relevant if you want to intercept verbatimtex which is something that we don't recommend doing in ConTEXt, just like we don't recommend using **btex**. But, if you use these, keep in mind that spaces matter. The parameter **texscriptmode** controls how spaces and newlines get honored. The default value is 1. Possible values are:

value	meaning
0	no newlines
1	newlines in **verbatimtex**
2	newlines in **verbatimtex** and **etex**
3	no leading and trailing strip in **verbatimtex**
4	no leading and trailing strip in **verbatimtex** and **btex**

That way the Lua handler (assigned to make_text) can do what it likes. An **etex** has to be followed by a space or ; or be at the end of a line and preceded by a space or at the beginning of a line. But let's repeat: these commands are kind of old school and not to be used in LuaMetaFun.

Logging, which includes the output of message and show, is also handled by a callback:

```
function run_logger ( <t:integer> target, <t:string> str )
    -- no return values
end
```

The possible log targets are:

0	void
1	terminal
2	file
3	both
4	error

An overload handler will take care of potentially dangerous overloading of for instance primitives, macro package definitions and special variables.

```
function run_overload ( <t:integer> property, <t:string> name, <t:integer> mode )
    return <t:boolean> -- resetproperty
end
```

The mode value is the currently set **overloadmode** internal. The MetaPost command **setproperty** can be used to relate an integer value to a quantity and when that value is positive a callback is triggered when that quantity gets redefined. Primitives get a property value 1 by the engine.

-3	mutable
1	primitive
2	permanent
3	immutable
4	frozen

Overload protect is something very ConTEXt and also present at the TEX end. All TEX and MetaPost quantities have such properties assigned.

When an error is issued it is often best to just quit the run and fix the issue, just because the instance can now be in a confused state,

```
function run_error (
    <t:string>  message,
    <t:string>  helpinfo,
    <t:integer> interactionmode
)
    -- no return values
end
```

You can get some statistics concerning an instance but in practice that is not so relevant for users. In ConTEXt these go to the log file.

```
function mplib.getstatistics ( <t:mp> instance )
    return <t:table>
end
```

The next set of numbers reflect for the current state of the `metafun:1` instance that is active for this specific run.

characters	18688	knots	0	parameters	42
hash	3065	memory	7983520	strings	1162
input	17	nodes	33	symbols	1000
internals	521	pairs	23	tokens	1106

In this 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. There is no need to configure memory.

The scanner in an instance can be in a specific state:

```
function mplib.getstatus ( <t:mp> instance )
    return <t:integer>
end
```

where possible states are:

0	normal	1	skipping

2	flushing	4	var_defining	5	op_defining	6	loop_defining
3	absorbing						

Macro names and variable names are stored in a hash table. You can get a list with entries with `gethashentries`, which takes an instance as first argument. When the second argument is `true` more details will be provided. With `gethashentry` you get info about the given macro or variable.

```
function mplib.gethashentries ( <t:mp> instance, <t:boolean> details )
    <t:table> hashentries
end
```

```
function mplib.gethashentry ( <t:mp> instance, <t:string> name )
    return
        <t:integer> -- command
        <t:integer> -- property
        <t:integer> -- subcommand
end
```

Say that we have defined:

```
numeric a ; numeric b ; numeric c ; a = b ; c := b ;
```

We get values like:

a	44	0	22
b	44	0	20
c	44	0	20
d	44	0	
def	19	1	1
vardef	19	1	2
fullcircle	44	3	10

These numbers represent commands, properties and subcommands, and thereby also assume some knowledge about how MetaPost works internally. As this kind of information is only useful when doing low level development we leave it at that.

10.2.1 Processing

It is up to the user to decide what to pass to the `execute` function as long as it is valid code. Think of each chunk being a syntactically correct file. Statements cannot be split over chunks.

```
function mplib.execute ( <t:mp> instance, <t:string> code )
    return {
        status = <t:integer>,
        fig    = <t:table>,
    }
end
```

In contrast with the normal stand alone `mpost` command, there is no implied ‘input’ at the start of the first chunk. When no string is passed to the execute function, there will still be one triggered because it then expects input from the terminal and you can emulate that channel with the callback you provide. In practice this is not something you need to be worry about.

When code is fed into the library at some point it will shipout a picture. The result always has a `status` field and an indexed `fig` table that has the graphics produced, although that is not mandate, for instance macro definitions can happen or variables can be set in which case graphics will be constructed later.

```
<t:userdata> o = <t:mpobj>:objects     ( )
<t:table>    b = <t:mpobj>:boundingbox ( )
<t:number>   w = <t:mpobj>:width       ( )
<t:number>   h = <t:mpobj>:height      ( )
<t:number>   d = <t:mpobj>:depth       ( )
<t:number>   i = <t:mpobj>:italic      ( )
<t:integer>  c = <t:mpobj>:charcode    ( )
<t:number>   t = <t:mpobj>:tolerance   ( )
<t:boolean>  s = <t:mpobj>:stacking    ( )
```

When you access a object that object gets processed before its properties are returned and in the process we loose the original. This means that some information concerning the whole graphic is also no longer reliably available. For instance, you can check if a figure uses stacking with the `stacking` function but because objects gets freed after being accessed, no information about stacking is available then.

The `charcode`, `width`, `height`, `depth` and `italic` are a left-over from MetaFont. They are values of the MetaPost variables **charcode**, `fontcharwd`, `fontcharht`, `fontchardp` and `fontcharit` at the time the graphic is shipped out.

You can call `fig:objects()` only once for any one `fig` object! In the end the graphic is a list of such userdata objects with accessors that depends on what specific data we have at hand. You can check out what fields with the following helper:

```
function mplib.getfields ( <t:integer> object | <t:mpobj> object | <t:nil> )
    return <t:table>
end
```

You get a simple table with one list of fields, or a table with all possible fields, organized per object type. In practice this helper is only used for documentation.

1	`fill`	`type path htap pen color linejoin miterlimit prescript postscript stacking curvature`
2	`outline`	`type path pen color linejoin miterlimit linecap dash prescript postscript stacking curvature`
3	`start_clip`	`type path prescript postscript stacking`
4	`start_group`	`type path prescript postscript stacking`
5	`start_bounds`	`type path prescript postscript stacking`
6	`stop_clip`	`type stacking`
7	`stop_group`	`type stacking`
8	`stop_bounds`	`type stacking`

All graphical objects have a field `type` (the second column in the table above) that gives the object type as a string value. When you have a non circular pen an envelope is uses defined by `path` as well as htap and the backend has to make sure that this gets translated into the proper pdf operators. Discussing this is beyond this manual. A color table has one, three or four values depending on the color space used. The prescript and postscript strings are the accumulated values of these operators, separated by newline characters. The stacking number is just that: a number, which can be used to put shapes in front or other shapes, some order, but it depends on the macro package as well as the backend to deal with that; it's basically just a numeric tag.

Each dash is a hash with two items. We use the same model as PostScript for the representation of the dash list; dashes is an array of 'on' and 'off', values, and offset is the phase of the pattern.

There is helper function peninfo that returns a table containing a bunch of vital characteristics of the used pen:

```
function mplib.peninfo ( <t:mpobj> object )
    return {
        width = <t:number>,
        rx    = <t:number>,
        ry    = <t:number>,
        sx    = <t:number>,
        sy    = <t:number>,
        tx    = <t:number>,
        ty    = <t:number>,
    }
end
```

10.2.2 Internals

There are a couple of helpers that can be used to query the meaning of specific codes and states.

```
function mplib.gettype ( <mopobj> object )
    return <t:integer> -- typenumber
end
```

```
function mplib.gettypes ( )
    return <t:table> -- types
end
```

0	undefined	7	unknownpen	14	transform	21	protodependent
1	vacuous	8	nep	15	color	22	independent
2	boolean	9	unknownnep	16	cmykcolor	23	tokenlist
3	unknownboolean	10	path	17	pair	24	structured
4	string	11	unknownpath	18	numeric	25	unsuffixedmacro
5	unknownstring	12	picture	19	known	26	suffixedmacro
6	pen	13	unknownpicture	20	dependent		

```
function mplib.getcolormodels ( )
    return <t:table> -- colormodels
end
```

0	no	1	grey

2 rgb	3 cmyk		

```
function mplib.getcodes ( )
    return <t:table> -- codes
end
```

0 undefined	21 addto	42 string	63 atleast
1 btex	22 setbounds	43 internal	64 curl
2 etex	23 protection	44 tag	65 macrospecial
3 if	24 property	45 numeric	66 rightdelimiter
4 fiorelse	25 show	46 plusorminus	67 leftbracket
5 input	26 mode	47 secondarydef	68 rightbracket
6 iteration	27 onlyset	48 tertiarybinary	69 rightbrace
7 repeatloop	28 message	49 leftbrace	70 with
8 exittest	29 everyjob	50 pathjoin	71 thingstoadd
9 relax	30 delimiters	51 pathconnect	72 of
10 scantokens	31 write	52 ampersand	73 to
11 runscript	32 typename	53 tertiarydef	74 step
12 maketext	33 leftdelimiter	54 primarybinary	75 until
13 expandafter	34 begingroup	55 equals	76 within
14 definedmacro	35 nullary	56 and	77 assignment
15 save	36 unary	57 primarydef	78 colon
16 interim	37 str	58 slash	79 comma
17 let	38 void	59 secondarybinary	80 semicolon
18 newinternal	39 cycle	60 parametertype	81 endgroup
19 macrodef	40 ofbinary	61 controls	82 stop
20 shipout	41 capsule	62 tension	83 undefinedcs

```
function mplib.getstates ( )
    return <t:table> -- states
end
```

0 normal	3 absorbing	5 op_defining
1 skipping	4 var_defining	6 loop_defining
2 flushing		

Knots is how the ‘points’ in a curve are called internally and in paths we can find these:

```
function mplib.getknotstates ( )
    return <t:table> -- knotstates
end
```

0 regular	2 end
1 begin	3 single

```
function mplib.getscantypes ( )
    return <t:table> -- scantypes
end
```

0 expression	2 secondary
1 primary	3 tertiary

As with TEX we can log to the console, a log file or both. But one will normally intercept log message anyway.

```
function mplib.getlogtargets ( )
    return <t:table> -- logtargets
end
```

0	void	2	file	3	both	4	error
1	terminal						

```
function mplib.getinternalactions ( )
    return <t:table> -- internalactions
end
```

0	initialize	2	restore
1	save		

```
function mplib.getobjecttypes ( )
    return <t:table> -- objecttypes
end
```

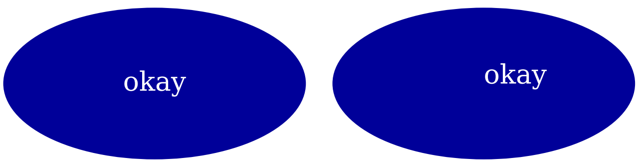
0		3	start_clip	6	stop_clip
1	fill	4	start_group	7	stop_group
2	outline	5	start_bounds	8	stop_bounds

The next one is of course dependent on what one runs. These statistics are for all instances:

```
function mplib.getcallbackstate ( )
    return <t:table> -- callbackstate
end
```

count	48275	overloaded	0
error	0	script	33650
file	14508	text	0
log	117	warning	0

The text counter is only counting what gets intercepted by MetaPost and as you can see below, the recommended `textext` is handled differently and not counted at all.



intercepted by ConTEXt

intercepted by MetaPost

So we get this now. The file count goes up because from the perspective of MetaPost code that gets executed and passed as string is just like reading from file. The relative high number that we see here reflects that we load quite some MetaFun macros when an instance is initialized.

count	48310	file	14522	overloaded	0
error	0	log	129	script	33658

text	1	warning	0

10.2.3 Information

The MetaPost library in LuaTEX starts with version 2 so in LuaMetaTEX we start with version 3, assuming that there will me no major update to the older library.

```
function mplib.version ( )
    return <t:string>
end
```

When there is an error you can ask for some more context:

```
function mplib.showcontext ( <t:mp> instance )
    return <t:string>
end
```

10.2.4 Methods

For historical reasons we provide a few functions as methods to an instance: `execute`, `finish`, `getstatistics`, `getstatus` and `solvepath`, just in case someone goes low level.

10.2.5 Scanners

There are quite some scanners available and they all take an instance as first argument. Some have optional arguments that give some control. A very basic one is the following. Scanning for a next token in MetaPost is different from TEX because while TEX just gets the token, MetaPost can delay in cases where an expression is seen. This means that you can inspect what is coming but do some further scanning based on that. Examples of usage can be found in ConTEXt as it permits to come up with extensions that behave like new primitives or implement interfaces that are otherwise hard to do in pure MetaPost.

```
function mplib.scannext ( <t:mp> instance, <t:boolean> keep )
    return <t:integer> token, <t:integer> mode, <t:integer> kind
end
```

here the optional `keep` boolean argument default to false but when true we basically have a look ahead scan. Contrary to TEX a next token is not expanded. If we want to pick up the result from an expression we use the next one where again we can push back the result:

0	expression	2	secondary
1	primary	3	tertiary

```
function mplib.scanexpression ( <t:mp> instance, <t:boolean> keep )
    return <t:integer> -- kind
end
```

The difference between `scantoken` and `scannext` is that the first one scans for a token and the later for a value and yes, one has to play a bit with this to see when one gets what.

```
function mplib.scantoken ( <t:mp> instance, <t:boolean> keep )
    return
        <t:integer>, -- token
        <t:integer>, -- mode
        <t:integer>  -- kind
end

function mplib.scansymbol ( <t:mp> instance, <t:boolean> expand, <t:boolean> keep )
    return <t:string>
end

function mplib.scanproperty ( <t:mp> instance )
    return <t:integer>
end
```

These are scanners for the simple data types:

```
function mplib.scannumeric ( <t:mp> instance ) return <t:number>  end -- scannumber
function mplib.scaninteger ( <t:mp> instance ) return <t:integer> end
function mplib.scanboolean ( <t:mp> instance ) return <t:boolean> end
function mplib.scanstring  ( <t:mp> instance ) return <t:string>  end
```

The scanners that return data types with more than one value can will return a table when the second argument is `true`:

```
function mplib.scanpair ( <t:mp> instance, <t:boolean astable )
    return
        <t:number>, -- x
        t:number>   -- y
end

function mplib.scancolor (
    <t:mp>      instance,
    <t:boolean astable
)
    return
        <t:number>, -- r
        <t:number>, -- g
        <t:number>  -- b
end

function mplib.scancmykcolor ( <t:mp> instance, <t:boolean astable )
    return
        <t:number>, -- c
        <t:number>, -- m
        <t:number>, -- y
        <t:number>  -- k
end

function mplib.scantransform ( <t:mp> instance, <t:boolean astable )
    return
        <t:number>, -- x
```

```
        <t:number>, -- y
        <t:number>, -- xx
        <t:number>, -- yx
        <t:number>, -- xy
        <t:number>  -- yy
end
```

The path scanned is more complex. First an expression is scanned and when okay it is converted to a table. The compact option gives:

```
{
    cycle = <t:boolean>, -- close
    pen   = <t:boolean>,
    {
        <t:number>, -- x_coordinate
        <t:number>, -- y_coordinate
    },
    ...
}
```

otherwise we get the more detailed:

```
{
    curved = <t:boolean>,
    pen    = <t:boolean>,
    {
        [1]        = <t:number>,  -- x_coordinate
        [2]        = <t:number>,  -- y_coordinate
        [3]        = <t:number>,  -- x_left
        [4]        = <t:number>,  -- y_left
        [5]        = <t:number>,  -- x_right
        [6]        = <t:number>,  -- y_right
        left_type  = <t:integer>,
        right_type = <t:integer>,
        curved     = <t:boolean>,
        state      = <t:integer>,
    },
    ...
}
```

Possible (knot, the internal name for a point) states are:

0	regular	2	end
1	begin	3	single

The path scanner function that produces such tables is:

```
function mplib.scanpath (
    <t:mp>      instance,
    <t:boolean> compact,
```

```
    <t:integer> kind,
    <t:boolean> check
)
    return <t:table>
end
```

This pen scanner returns similar tables:

```
function mplib.scanpen (
    <t:mp>      instance,
    <t:boolean> compact,
    <t:integer> kind,
    <t:boolean> check
)
    return <t:table>
end
```

The next is not really a scanner. It skips a token that matches the given command and returns a boolean telling if that succeeded.

```
function mplib.skiptoken ( <t:mp> instance, <t:integer> command )
    return <t:boolean>
end
```

10.2.6 Injectors

The scanners are complemented by injectors. Instead of strings that have to be parsed by MetaPost they inject the right data structures directly.

```
function mplib.injectnumeric ( <t:mp> instance, <t:number>  value ) end
function mplib.injectinteger ( <t:mp> instance, <t:integer> value ) end
function mplib.injectboolean ( <t:mp> instance, <t:boolean> value ) end
function mplib.injectstring  ( <t:mp> instance, <t:string>  value ) end
```

In following injectors accept a table as well as just the values. which can more efficient:

```
function mplib.injectpair      ( <t:mp> instance, <t:table> value ) end
function mplib.injectcolor     ( <t:mp> instance, <t:table> value ) end
function mplib.injectcmykcolor ( <t:mp> instance, <t:table> value ) end
function mplib.injecttransform ( <t:mp> instance, <t:table> value ) end
```

Injecting a path is not always trivial because we have to connect the points emulating `..`, `...`, `---` and even `&&` and `cycle`. A path is passed as table. The table can be nested and has entries like these:

```
{
    {
        x_coord       = <t:number>,
        y_coord       = <t:number>,
        x_left        = <t:number>,
        y_left        = <t:number>,
        x_right       = <t:number>,
```

```
        y_right       = <t:number>,
        left_curl     = <t:number>,
        right_curl    = <t:number>,
        left_tension  = <t:number>,
        right_tension = <t:number>,
        direction_x   = <t:number>,
        direction_y   = <t:number>,
    },
    {
        [1] = <t:number>, -- x_coordinate
        [2] = <t:number>, -- x_coordinate
        [3] = <t:number>, -- x_left
        [4] = <t:number>, -- y_left
        [5] = <t:number>, -- x_right
        [6] = <t:number>, -- y_right
    }
    "append",
    "cycle",
}
```

Here append is like **&&** which picks up the pen, and **cycle**, not surprisingly, behaves like the **cycle** operator.

```
function mplib.injectpath ( <t:mp> instance,  <t:table> value )
    -- return nothing
end
```

```
function mplib.injectwhatever ( <t:mp> instance, <t:hybrid> value )
    -- return nothing
end
```

When a path is entered and has to be injected some preparation takes place out of the users sight. A special variant of the path processor is the following, where the path is adapted and the boolean indicates success.

```
function mplib.solvepath ( <t:mp> instance, <t:table> value )
    return <t:boolean>
end
```

A still somewhat experimental injectors is the following one, that can be used to fetch information from the TEX end. Valid values for `expected` are 1 (integer), 2 (cardinal, 3 (dimension), 5 (boolean) and 7 (string).

```
function mplib.expandtex (
    <t:mp>       instance,
    <t:integer>  expected,
    <t:string>   macro,
    <t:whatever> arguments
)
    return <t:whatever>
end
```

tex

11 TEX

11.1 Introduction

Here we don't explain TEX itself but the interface between TEX and Lua. We don't need to talk nodes and tokens because they have their own chapters.

11.2 Status information

The `status` library provides information not only about the current run and system setup but also about all kind of variables and constants used in the engine. A difference between LuaTEX and LuaMetaTEX is that every quantity that is hard coded is available as a constant to be used. The same is true for various bit sets for instance those use in setting options, as we will see in the `tex` library.

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.

```
function status.list ( )
    return <t:table>
end
```

The keys in the returned table are the known items, the value is the current value. There are top level items and items that are tables with sub entries. The current list gives:

toplevel statistics	
banner	This is LuaMetaTeX, Version 2.11.06
copyright	Taco Hoekwater, Hans Hagen, Wolfgang Schuster & Mikael Sundqvist
development_id	20241031
filename	luametatex-tex.tex
format_id	711
logfilename	luametatex.log
lua_format	2
lua_version	5.5
lua_version_major	5
lua_version_minor	5
lua_version_release	0
luatex_engine	luametatex
luatex_release	6
luatex_revision	11
luatex_verbose	2.11.06
luatex_version	2
permit_loadlib	false
run_state	2
used_compiler	gcc
version	211.6

bufferstate.*	
all	1000000

ini	-1
max	100000000
mem	1000000
min	1000000
ptr	0
set	10000000
stp	1000000
top	3182

callbackstate.*

bytecode	631
count	500105
direct	3895
file	29592
function	129469
local	0
message	0
saved	321089
value	15429

enginestate.*

banner	This is LuaMetaTeX, Version 2.11.06
copyright	Taco Hoekwater, Hans Hagen, Wolfgang Schuster & Mikael Sundqvist
development_id	20241031
format_id	711
logfilename	luametatex.log
luatex_engine	luametatex
luatex_release	6
luatex_revision	11
luatex_verbose	2.11.06
luatex_version	2
permit_loadlib	false
run_state	2
tex_hash_size	262144
used_compiler	gcc
version	211.6

errorlinestate.*

max	255
min	132
set	250
top	0

errorstate.*

error	unset
errorcontext	unset
luaerror	unset

expandstate.*	
max	1000000
min	10000
set	10000
top	10

extrastate.*	
all	280
ini	-1
max	-1
mem	280
min	-1
ptr	280
set	-1
stp	-1
top	6040

filestate.*	
all	16000
ini	-1
max	2000
mem	500
min	500
ptr	6
set	2000
stp	250
top	11

fontstate.*	
all	13906936
ini	-1
max	100000
mem	13906936
min	250
ptr	58
set	100000
stp	250
top	250

halferrorlinestate.*	
max	255
min	80
set	234
top	0

hashstate.*	
all	2400000
ini	0
max	2097152
mem	150000
min	150000
ptr	8246
set	250000
stp	100000
top	938189

hyphenationstate.*	
checked	28174
exceptions	299
hyphenated	26512
lists	28174
nothing	18301
words	36214

inputstate.*	
all	320000
ini	-1
max	100000
mem	10000
min	10000
ptr	7
set	100000
stp	10000
top	45

insertstate.*	
all	10240
ini	-1
max	500
mem	320
min	10
ptr	9
set	250
stp	25
top	10

languagestate.*	
all	96
ini	0
max	10000
mem	96

min	250
ptr	0
set	250
stp	250
top	250

linebreakstate.*	
align	table: 000002455aa55d10
dbox	table: 000002455aa55c50
doubletwins	0
insert	table: 000002455aa55d40
lefttwins	0
lua	table: 000002455aa55e60
math	table: 000002455aa55e00
noalign	table: 000002455aa55d70
normal	table: 000002455aa55ad0
output	table: 000002455aa55ce0
reset	table: 000002455aa55e90
righttwins	0
span	table: 000002455aa55da0
vadjust	table: 000002455aa55cb0
vbox	table: 000002455aa55bc0
vcenter	table: 000002455aa55c80
vmode	table: 000002455aa55b30
vtop	table: 000002455aa55c20

lookupstate.*	
all	-1
ini	50497
max	2097152
mem	-1
min	150000
ptr	54960
set	250000
stp	100000
top	262146

luastate.*	
bytecodebytes	16416
bytecodes	1025
functionsize	32768
propertiessize	10000
statebytes	114006805
statebytesmax	224720030

markstate.*	
all	28800

ini	-1
max	10000
mem	1200
min	50
ptr	28
set	250
stp	50
top	50

neststate.*	
all	80000
ini	-1
max	10000
mem	1000
min	1000
ptr	0
set	10000
stp	1000
top	19

nodestate.*	
all	18000000
ini	0
max	100000000
mem	2000000
min	2000000
ptr	-336162
set	100000000
stp	500000
top	360498

parameterstate.*	
all	80000
ini	-1
max	100000
mem	20000
min	20000
ptr	1
set	100000
stp	10000
top	55

poolstate.*	
all	1059601
ini	963323
max	100000000
mem	1059601

min	10000000
ptr	-1
set	10000000
stp	1000000
top	-1

readstate.*	
filename	luametatex-tex.tex
iocode	5
linenumber	71
skiplinenumber	33

savestate.*	
all	160000
ini	-1
max	500000
mem	10000
min	100000
ptr	331
set	500000
stp	10000
top	1402

sparsestate.*	
all	3724592
ini	-1
max	-1
mem	3724592
min	-1
ptr	-1
set	-1
stp	-1
top	-1

stringstate.*	
all	2400000
ini	2147666
max	2097152
mem	150000
min	150000
ptr	54977
set	500000
stp	100000
top	54977

texstate.*	
approximate	60186545

tokenstate.*	
all	16000000
ini	557268
max	10000000
mem	2000000
min	2000000
ptr	-1830050
set	10000000
stp	500000
top	648800

warningstate.*	
warning	unset
warningtag	unset

The `getconstants` query gives back a table with all kind of internal quantities and again these are only relevant for diagnostic and development purposes. Many are good old TEX constants that are describes in the original documentation of the source but some are definitely LuaMetaTEX specific.

```
function status.getconstants ( )
    return <t:table>
end
```

The returned table contains:

constants.*	
all_fitness_values	255
assumed_math_control	4125694
awful_bad	1073741823
decent_criterion	12
default_catcode_table	-1
default_character_control	0
default_deadcycles	25
default_eqno_gap_step	1000
default_hangafter	1
default_output_box	255
default_pre_display_gap	2000
default_rule	26214
default_space_factor	1000
default_tolerance	10000
deplorable	100000
eject_penalty	-10000
ignore_depth	-65536000
infinite_bad	10000
infinite_penalty	10000
infinity	2147483647
large_width_excess	7230584
loose_criterion	99

math_all_class	61
math_begin_class	62
math_default_penalty	10001
math_end_class	63
math_first_user_class	20
math_last_user_class	60
max_attribute_register_index	65535
max_box_register_index	65535
max_bytecode_index	65535
max_calculated_badness	8189
max_cardinal	4294967295
max_category_code	15
max_character_code	1114111
max_data_value	2097151
max_dimen	1073741823
max_dimen_register_index	65535
max_dimension	1073741823
max_dimension_register_index	65535
max_endline_character	127
max_float_register_index	65535
max_font_adjust_shrink_factor	500
max_font_adjust_step	100
max_font_adjust_stretch_factor	1000
max_function_reference	2097151
max_glue_register_index	65535
max_half_value	32767
max_halfword	1073741823
max_int_register_index	65535
max_integer	2147483647
max_integer_register_index	65535
max_limited_scale	1000
max_mark_index	9999
max_math_class_code	63
max_math_family_index	63
max_math_scaling_factor	5000
max_math_style_scale	2000
max_muglue_register_index	65535
max_n_of_bytecodes	65536
max_n_of_catcode_tables	256
max_n_of_fitness_values	15
max_n_of_fonts	100000
max_n_of_languages	10000
max_n_of_marks	10000
max_n_of_math_families	64
max_newline_character	127
max_quarterword	65535
max_scale_factor	100000
max_size_of_word	1000
max_space_factor	32767

max_toks_register_index	65535
max_twin_length	16
min_cardinal	0
min_data_value	0
min_dimen	-1073741823
min_dimension	-1073741823
min_halfword	-1073741823
min_infinity	-2147483647
min_integer	-2147483647
min_n_of_fitness_values	5
min_quarterword	0
min_scale_factor	0
min_space_factor	0
no_catcode_table	-2
null	0
null_flag	-1073741824
null_font	0
one_bp	65781
preset_rule_thickness	1073741824
running_rule	-1073741824
small_stretchability	1663497
special_space_factor	999
tex_eqtb_size	788189
tex_hash_prime	262103
tex_hash_size	262144
two	131072
undefined_math_parameter	1073741823
unity	65536
unused_attribute_value	-2147483647
unused_math_family	255
unused_math_style	255
unused_script_value	0
unused_state_value	0
zero_glue	0

Most variables speak for themselves, some are more obscure. For instance the `runstate` variable indicates what the engine is doing:

0x00	initializing
0x01	updating
0x02	production

These overviews can get asked for, for instance with `getrunstatevalues` in the `tex` library. Most of these constants are stable but especially for those that relate to evolving engine functionality there can be changes, so keep an eye on these mappings!

The individual states can be fetched with the following helpers:

```
function status.getbufferstate      ( ) return <t:table> end
function status.getcallbackstate    ( ) return <t:table> end
```

```
function status.geterrorlinestate     ( ) return <t:table> end
function status.geterrorstate         ( ) return <t:table> end
function status.getexpandstate        ( ) return <t:table> end
function status.getextrastate         ( ) return <t:table> end
function status.getfilestate          ( ) return <t:table> end
function status.getfontstate          ( ) return <t:table> end
function status.gethalferrorlinestate ( ) return <t:table> end
function status.gethashstate          ( ) return <t:table> end
function status.gethyphenationstate   ( ) return <t:table> end
function status.getinputstate         ( ) return <t:table> end
function status.getinsertstate        ( ) return <t:table> end
function status.getlanguagestate      ( ) return <t:table> end
function status.getlinebreakstate     ( ) return <t:table> end
function status.getlookupstate        ( ) return <t:table> end
function status.getluastate           ( ) return <t:table> end
function status.getmarkstate          ( ) return <t:table> end
function status.getneststate          ( ) return <t:table> end
function status.getnodestate          ( ) return <t:table> end
function status.getparameterstate     ( ) return <t:table> end
function status.getpoolstate          ( ) return <t:table> end
function status.getreadstate          ( ) return <t:table> end
function status.getsavestate          ( ) return <t:table> end
function status.getsparsestate        ( ) return <t:table> end
function status.getstringstate        ( ) return <t:table> end
function status.gettexstate           ( ) return <t:table> end
function status.gettokenstate         ( ) return <t:table> end
function status.getwarningstate       ( ) return <t:table> end
```

The error and warning messages can be wiped with:

```
function status.resetmessages ( )
    -- no return values
end
```

11.3 Everything TEX

11.3.1 Introduction

The `tex` library contains a large list of (possibly virtual) internal TEX parameters that are partially writable. The designation 'virtual' means that these items are not properly defined in Lua, but are only front-ends 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. In addition to this kind of access we have getters and setters, which are the preferred way, but e keep the field like accessors around for compatibility reasons.

At the moment, it is possible to access almost every parameter that you can use after **`\the`**, is a single token 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`**).

11.3.2 Registers

Among of the oldest accessors to internals are `tex.dimen` and `tex.count`. This permits calls like this:

```
\setbox0\hbox{test}
\directlua{tex.sprint(tex.box[0].width)}
```

to give us (in this case typeset): 1250880 scaled points. Here we access a box register, get back a userdata node, and access one of its fields. The skip registers also are stored on userdata. The register are accessed in the following way; watch the different value types that you get:

```
<t:integer> value = tex.attribute [index]
<t:node>    value = tex.skip      [index]
<t:integer> value = tex.glue      [index]
<t:node>    value = tex.muskip    [index]
<t:integer> value = tex.muglue    [index]
<t:integer> value = tex.dimen     [index]
<t:integer> value = tex.count     [index]
<t:number>  value = tex.posit     [index]
<t:string>  value = tex.toks      [index]
<t:node>    value = tex.box       [index]
```

You can also assign values:

```
tex.attribute [index] = value -- <t:integer>
tex.skip      [index] = value -- <t:node>
tex.glue      [index] = value -- <t:integer>
tex.muskip    [index] = value -- <t:node>
tex.muglue    [index] = value -- <t:integer>
tex.dimen     [index] = value -- <t:integer>
tex.count     [index] = value -- <t:integer>
tex.posit     [index] = value -- <t:number>
tex.toks      [index] = value -- <t:string>
tex.box       [index] = value -- <t:node>
```

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.copylist` unless you are assigning to a temporary variable:

```
tex.box[0] = node.copylist(tex.box[2])
```

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.

Although these are actually not stored in arrays but in hashes, the various ‘codes’ can also be accessed this way:

```
<t:integer> value = tex.sfcode   = [index]
```

```
<t:integer> value = tex.lccode   = [index]
<t:integer> value = tex.uccode   = [index]
<t:integer> value = tex.hccode   = [index]
<t:integer> value = tex.hmcode   = [index]
<t:integer> value = tex.amcode   = [index]
<t:integer> value = tex.cccode   = [index]
<t:integer> value = tex.catcode  = [index]
<t:integer> value = tex.mathcode = [index]
<t:integer> value = tex.delcode  = [index]
```

and

```
tex.sfcode   = [index] = value -- <t:integer>
tex.lccode   = [index] = value -- <t:integer>
tex.uccode   = [index] = value -- <t:integer>
tex.hccode   = [index] = value -- <t:integer>
tex.hmcode   = [index] = value -- <t:integer>
tex.amcode   = [index] = value -- <t:integer>
tex.cccode   = [index] = value -- <t:integer>
tex.catcode  = [index] = value -- <t:integer>
tex.mathcode = [index] = value -- <t:integer>
tex.delcode  = [index] = value -- <t:integer>
```

The getters are

```
function tex.getamcode  ( <t:integer> character ) return <t:integer> end
function tex.getcatcode ( <t:integer> character ) return <t:integer> end
function tex.getcccode  ( <t:integer> character ) return <t:integer> end
function tex.gethccode  ( <t:integer> character ) return <t:integer> end
function tex.gethmcode  ( <t:integer> character ) return <t:integer> end
function tex.getlccode  ( <t:integer> character ) return <t:integer> end
function tex.getsfcode  ( <t:integer> character ) return <t:integer> end
function tex.getuccode  ( <t:integer> character ) return <t:integer> end
```

and the setters:

```
function tex.setamcode  ( <t:integer> character, <t:integer> value ) end
function tex.setcatcode ( <t:integer> character, <t:integer> value ) end
function tex.setcccode  ( <t:integer> character, <t:integer> value ) end
function tex.sethccode  ( <t:integer> character, <t:integer> value ) end
function tex.sethmcode  ( <t:integer> character, <t:integer> value ) end
function tex.setlccode  ( <t:integer> character, <t:integer> value ) end
function tex.setsfcode  ( <t:integer> character, <t:integer> value ) end
function tex.setuccode  ( <t:integer> character, <t:integer> value ) end
```

The `setlccode` and `setuccode` additionally allow you to set the associated sibling at the same time by passing an extra argument.

```
function tex.setlccode  ( <t:integer> character, <t:integer> lcvalue, <t:integer>
  ucvalue ) end
function tex.setuccode  ( <t:integer> character, <t:integer> ucvalue, <t:integer>
  lcvalue ) end
```

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

```
function tex.setcatcode (
    <t:integer> catcodetable,
    <t:integer> character,
    <t:integer> value
)
    -- no return values
end
```

All these setters accept an initial global string.

11.3.3 Setters and getters

Most of TEX's parameters can be accessed directly by 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. In most cases we have integers but especially glue have more properties than just the amount. For the parameters that *can* be set, it is possible to use global as the first argument to tex.set. Them being more complete is an argument for using setters instead of assignments.

The set function is meant for what we call internal parameter. These can be registers but without a known number (one can actually figure out the internal number via the token library).

```
function tex.set ( <t:string> name, <t:whatever> value )
    -- no return values
end

function tex.set ( "global", <t:string> name, <t:whatever> value )
    -- no return values
end
```

You can get back a value with:

```
function tex.get ( <t:string> name )
    return <t:whatever>
end
```

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.

Traditional TEX has 256 registers per type, ε-TEX bumps that to 32K and LuaMetaTEX doubles that. But how many are enough? Do we really need that many different attributes and glue specifiers?

In LuaMetaTEX on the one hand can go lower on registers and at the same time go beyond with alternatives when using named quantities.

It is possible to define named registers with t**\attributedef**, **\countdef**, **\dimendef**, **\skipdef**, **\floatdef** or **\toksdef** control sequences as indices to these tables and these can be accessed by name at the Lua end. Here are some examples:

```
tex.count.scratchcounter = 123
tex.dimen.scratchdimen   = "20pt"

tex.setcount(          "scratchcounter", 123)
tex.setdimen(          "scratchdimen",   10 *65536)
tex.setdimen("global", "scratchdimen",   "10pt")

enormous = tex.dimen.maxdimen
enormous = tex.getdimen("maxdimen")

unknown = tex.dimen[3]
unknown = tex.getdimen(3)
```

Of course this assumes that these registers are defined. What you can do depends on the type:

- The count registers accept and return Lua numbers (integers in this case).
- The dimension registers accept Lua numbers (in scaled points) or strings with a dimension.
- 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).
- The glue registers are just skip registers but instead of userdata accept verbose (integers).
- Like the counts, the attribute registers accept and return integers.
- Float (aka posit) registers accept and return floating point numbers.

The `setglue` function accepts upto five arguments:

```
function tex.setskip (
    <t:string> register, -- can also be an index
    <t:node>   value     -- glue_spec
)
    -- no return values
end

function tex.setglue (
    <t:string>  register, -- can also be an index
    <t:integer> amount,
    <t:integer> stretch,
    <t:integer> shrink,
    <t:integer> stretchorder,
    <t:integer> shrinkorder
)
    -- no return values
```

```
end
```

Actually there can be one more argument here because as first argument we can pass "`global`". The whole repertoire is:

```
function tex.getattribute ( <t:string> name ) return <t:integer> end
function tex.getcount     ( <t:string> name ) return <t:integer> end
function tex.getdimen     ( <t:string> name ) return <t:integer> end
function tex.getfloat     ( <t:string> name ) return <t:number>  end
function tex.getskip      ( <t:string> name ) return <t:node>    end
function tex.getmuskip    ( <t:string> name ) return <t:node>    end
function tex.gettoks      ( <t:string> name ) return <t:string>  end

function tex.getglue   ( <t:string> name        ) return <t:integer>, ... end
function tex.getmuglue ( <t:string> name        ) return <t:integer>, ... end
function tex.getglue   ( <t:string> name, false ) return <t:integer> end
function tex.getmuglue ( <t:string> name, false ) return <t:integer> end
```

and

```
function tex.setattribute (<t:string> name, <t:integer> value) end
function tex.setcount     (<t:string> name, <t:integer> value) end
function tex.setdimen     (<t:string> name, <t:integer> value) end
function tex.setfloat     (<t:string> name, <t:number>  value) end
function tex.setmuskip    (<t:string> name, <t:node> value) end
function tex.setskip      (<t:string> name, <t:node>    value) end
function tex.settoks      (<t:string> name, <t:string>  value) end

function tex.setglue      (<t:string> name, <t:integer> value, ...) end
function tex.setmuglue    (<t:string> name, <t:integer> value, ...) end
```

Just to be clear, getting a glue has two variants, the third one is just a reduced variant:

```
function tex.getskip (
    <t:string> register -- can also be an index
)
    return <t:node> -- a glue_spec
end

function tex.getglue (
    <t:string>  register, -- can also be an index
)
    return
        <t:integer> -- amount,
        <t:integer> -- stretch,
        <t:integer> -- shrink,
        <t:integer> -- stretchorder,
        <t:integer> -- shrinkorder
end

function tex.getglue (
    <t:string> register, -- can also be an index
```

```
    false
)
    return <t:integer> amount,
)
end
```

When `tex.gettoks` gets an extra argument `true` it will return a table with userdata tokens. For tokens registers we have an alternative where a catcode table is specified:

```
function tex.scantoks (
    <t:integer> catcodetable,
    <t:integer> registerindex, -- or just a name
    <t:string>  data
)
    -- no return values
end
```

Again there is the option to pass `"global"` as first argument. Here is an example that used the default ConTEXt catcode table index `tex.ctxcatcodes`.

```
local t = tex.scantoks("global",tex.ctxcatcodes,3,"$e=mc^2$")
```

This is a bit different getter that was introduced to accommodate interfacing between TEX and MetaPost. We specify what kind of parsing takes place:

```
function tex.expandasvalue (
    <t:integer> kind, -- how interpreted
    <t:string>  name  -- macro name
)
    return <t:integer> | <t:boolean> | <t:string>
end
```

`0x00`	none	`0x03`	dimension	`0x06`	float	`0x09`	direct
`0x01`	integer	`0x04`	skip	`0x07`	string	`0x0A`	conditional
`0x02`	cardinal	`0x05`	boolean	`0x08`	node		

11.3.4 Fonts

There are a few functions that deal with fonts. The next function relates a control sequence to a font identifier. This is not to be confused with registering font data in the engine which happens with the functions in the `font` library. This is basically a setter that as one some in the `token` library also accepts prefixes (like `global`):

```
function tex.definefont (
    <t:string>  name,
    <t:integer> fontid,
    <t:string>  prefix
 -- there can be more prefixes
)
    -- no return values
```

```
end
```

In LuaTEX and other engines the file names are stored in the table of equivalents but not so in LuaMetaTEX. But for old times sake we keep some getters in the `tex` library, as they are basically 'convert' commands. The next two are like **`\fontid`** and **`\fontname`**:

```
function tex.fontidentifier ( <t:integer> id ) return <t:integer> end
function tex.fontname       ( <t:integer> id ) return <t:string>  end
```

When no `id` is given the current font is assumed, as if **`\font`** was the argument to the mentioned equivalent macros, so here we have: `<5: DejaVuSansMono @ 10.0pt>` and `DejaVuSansMono at 10.0pt`.

We can query the font id bound to a family (and optionally style):

```
function tex.getfontoffamily (
    <t:integer> family,
    <t:integer> style   -- 0, 1, 2
)
     return <t:integer> -- id
end
```

This is a good place to mention a pitfall when it comes to accessing some internals. Many variables are just that, variables, but there are also some that need an argument. This means that we get the following:

Lua call	result (if any)
`tex.fontname`	
`tex.fontidentifier`	
`tex.fontname()`	`DejaVuSerif at 10.0pt`
`tex.fontidentifier()`	`<1: DejaVuSerif @ 10.0pt>`
`tex.get("fontname",-1)`	`DejaVuSerif at 10.0pt`
`tex.get("fontidentifier",-1)`	`<1: DejaVuSerif @ 10.0pt>`

When called as 'field' we get nothing. When called as a function we get the font info of the id passes as argument. When no argument is given the current font is used. When we use a getter the id is mandate but a negative value will again make that the current font is used. Making the first two use the current font and the last two accept no second argument is technically possible but complicating the code for these few cases makes no sense. We already handle more than in LuaTEX anyway.

11.3.5 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. In the setters you can pass as first argument `global` if needed. Alternatively you can use the `tex.box` array interface.

```
function tex.setbox (
    <t:integer> index,
    <t:node>    packedlist
)
    -- no return values
end
```

```
function tex.setbox (
    <t:string> name,
    <t:node>   packedlist
)
    -- no return values
end
```

The getters return a packed list or `nil` when the register is void.

```
function tex.getbox (
    <t:integer> index
)
    return <t:node>
end
```

```
function tex.getbox (
    <t:string> name
)
    return <t:node>
end
```

You can split a box:

```
local vlist =
function tex.splitbox (
    <t:integer> index,
    <t:integer> height,
    <t:integer> 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.

11.3.6 Marks

There is a dedicated getter for marks:

```
function tex.getmark (
    <t:string>  name,
    <t:integer> markindex
)
    -- no return values
end
```

```
function tex.getmark ( )
    return <t:integer> -- max mark class
end
```

The first argument can also be an integer, actually the subtype of a mark node:

`0x00`	current	`0x01`	top

The largest used mark class is returned by:

```
function tex.getlargestusedmark ( )
    return <t:integer> -- max mark class
end
```

11.3.7 Inserts

Access to inserts is kind of special and often only makes sense when we are constructing the final page. Where in traditional TEX inserts use a **\dimen**, **\count**, **\skip** and **\box**, registers, in LuaMetaTEX we can use dedicted storage instead. This is why we need setters and getters.

```
function tex.getinsertdistance   ( <t:integer> class ) return <t:integer> end
function tex.getinsertmultiplier ( <t:integer> class ) return <t:integer> end
function tex.getinsertlimit      ( <t:integer> class ) return <t:integer> end
function tex.getinsertcontent    ( <t:integer> class ) return <t:node>    end
function tex.getinsertheight     ( <t:integer> class ) return <t:integer> end
function tex.getinsertdepth      ( <t:integer> class ) return <t:integer> end
function tex.getinsertwidth      ( <t:integer> class ) return <t:integer> end
```

Only some properties can be set:

```
function tex.setinsertdistance   ( <t:integer> class, <t:integer> distance   ) end
function tex.setinsertmultiplier ( <t:integer> class, <t:integer> multiplier ) end
function tex.setinsertlimit      ( <t:integer> class, <t:integer> limit      ) end
function tex.setinsertcontent    ( <t:integer> class, <t:node>    list       ) end
```

11.3.8 Local boxes

Local boxes, **\localleftbox**, **\localrightbox** and specific for LuaMetaTEX, **\localmiddlebox**, are not regular box registers so they have dedicated accessors:

```
function tex.getlocalbox ( <t:integer> location )
    return <t:node>
end
```

```
function tex.setlocalbox ( <t:integer> location, <t:node> list )
    -- no return values
end
```

Instead of integers you can also use the name. Valid local box locations are:

`0x00`	left
`0x01`	right
`0x02`	middle

11.3.9 Constants

The name of this section is a bit misleading but reflects history. At some point LuaMetaTEX got a way to store values differently than in registers because it felt a bit weird to use registers for what actually are constant values. However, it was not that hard to make them behave like registers which opens up the possibility to reduce the number of registers at some point.

At the TEX end we have **\integerdef**, **\dimensiondef**, **\floatdef**, **\gluespecdef** and **\mugluespecdef** but at the Lua end we (currently) only handle the first three.

```
function tex.dimensiondef ( <t:string> name ) end
function tex.integerdef   ( <t:string> name ) end
function tex.positdef     ( <t:string> name ) end
```

These are the setters:

```
function tex.setdimensionvalue ( <t:string> name, <t:integer> value ) end
function tex.setintegervalue   ( <t:string> name, <t:integer> value ) end
function tex.setcardinalvalue  ( <t:string> name, <t:integer> value ) end
function tex.setpositvalue     ( <t:string> name, <t:number>  value ) end
```

and these the getters:

```
function tex.getdimensionvalue ( <t:string> name ) return <t:integer> end
function tex.getintegervalue   ( <t:string> name ) return <t:integer> end
function tex.getcardinalvalue  ( <t:string> name ) return <t:integer> end
function tex.getpositvalue     ( <t:string> name ) return <t:number>  end
```

Now, in order to make access more convenient, the getters and setters that deal with these quantities that we discussed in a previous section also handle these ‘constants’.

The following helper is a bit tricky:

```
function tex.getregisterindex ( <t:string> name )
    return <t:integer>
end
```

The integer that is returned can be used instead of a name when accessing a register,

```
\newcount \MyCount  \newinteger   \MyInteger
\newdimen \MyDimen  \newdimension \MyDimension

\startluacode
    context("[%s] [%s] [%s] [%s]",
        tex.getregisterindex("MyCount"),
        tex.getregisterindex("MyInteger"),
        tex.getregisterindex("MyDimen"),
        tex.getregisterindex("MyDimension")
    )
\stopluacode
```

This will only show something for the registers:

[272] [] [269] []

This is why we have a more complete completed solution:

```
tex.isattribute ( <t:string> name ) return <t:integer> end
```

```
tex.iscount     ( <t:string> name ) return <t:integer> end
tex.isdimen     ( <t:string> name ) return <t:integer> end
tex.isfloat     ( <t:string> name ) return <t:integer> end
tex.isglue      ( <t:string> name ) return <t:integer> end
tex.ismuglue    ( <t:string> name ) return <t:integer> end
tex.ismuskip    ( <t:string> name ) return <t:integer> end
tex.isskip      ( <t:string> name ) return <t:integer> end
tex.istoks      ( <t:string> name ) return <t:integer> end
tex.isbox       ( <t:string> name ) return <t:integer> end
```

We now use this:

```
\startluacode
    context("[%s] [%s] [%s] [%s]",
        tex.iscount("MyCount"),
        tex.iscount("MyInteger"),
        tex.isdimen("MyDimen"),
        tex.isdimen("MyDimension")
    )
\stopluacode
```

This time all four names are resolved:

[272] [112321] [269] [250825]

The larger numbers are references to these ‘constants’. Using these instead of names in the getters (like `getcount` and `getdimen` can be more efficient when the number times we need access is very large because we bypass a hash lookup. Of course these numbers are to be seen as abstract references, so these larger numbers are unpredictable.

11.3.10 Nesting

The virtual table `nest` contains the currently active semantic nesting state (think building boxes). It has two main parts: a zero-based array of userdata for the semantic nest itself, and the numerical value `ptr`, which gives the highest available index. Neither the array items in `nest[]` nor `ptr` can be assigned to, because this would confuse the typesetting engine beyond repair, but you can assign to the individual values inside the array items.

The zero entry `nest[0]` is the outermost (main vertical list) level while `tex.nest [tex.nest.ptr]` is the current nest state. The next example shows all of this:

```
\setbox\scratchbox\vbox\bgroup
    \vbox\bgroup
        \startluacode
            for i=0,tex.nest.ptr do
                context(tostring(tex.nest[i]))
                context.space()
                context(tostring(tex.nest[i].prevdepth))
                context.par()
            end
        \stopluacode
```

```
    \egroup
\egroup
```

tex.nest.instance: 0000024556c7cc90 266685
tex.nest.instance: 0000024556c7d1d0 -65536000
tex.nest.instance: 0000024556c7d380 -65536000

The current nest level (`tex.nest.ptr` is also available with:

```
function tex.getnestlevel ( )
    return <t:integer>
end
```

The getter function is `tex.getnest`. You can pass a number (which gives you a list), nothing or `top`, which returns the topmost list, or the string `ptr` which gives you the index of the topmost list. The complete list of fields is: `delimiter`, `direction`, `head`, `mathbegin`, `mathdir`, `mathend`, `mathflatten`, `mathmainstyle`, `mathmode`, `mathparentstyle`, `mathscale`, `mathstyle`, `modeline`, `noad`, `prevdepth`, `prevgraf`, `spacefactor`, `tail`.

Possible modes are:

`0x00`	unset	`0x02`	horizontal
`0x01`	vertical	`0x03`	math

Valid directions are:

`0x00`	lefttoright
`0x01`	righttoleft

Math styles conforms to:

`0x00`	display	`0x04`	script
`0x01`	crampeddisplay	`0x05`	crampedscript
`0x02`	text	`0x06`	scriptscript
`0x03`	crampedtext	`0x07`	crampedscriptscript

The math begin and end classes can be built-in or ConTEXt specific:

`0x00`	ordinary	`0x0B`	over	`0x17`	exponential	`0x22`	textpunctuation
`0x01`	operator	`0x0C`	fraction	`0x18`	integral	`0x23`	unspaced
`0x02`	binary	`0x0D`	radical	`0x19`	ellipsis	`0x24`	experimental
`0x03`	relation	`0x0E`	middle	`0x1A`	function	`0x25`	fake
`0x04`	open	`0x10`	accent	`0x1B`	digit	`0x26`	numbergroup
`0x05`	close	`0x11`	fenced	`0x1C`	division	`0x27`	maybeordinary
`0x06`	punctuation	`0x12`	ghost	`0x1D`	factorial	`0x28`	mayberelation
`0x07`	variable	`0x13`	vcenter	`0x1E`	wrapped	`0x29`	maybebinary
`0x08`	active	`0x14`	explicit	`0x1F`	construct	`0x2A`	chemicalbond
`0x09`	inner	`0x15`	imaginary	`0x20`	dimension	`0x2B`	implication
`0x0A`	under	`0x16`	differential	`0x21`	unary	`0x2C`	continuation

The helpers are:

```
function tex.getnest ( <t:integer> level )
    return <t:userdata> -- nest
end

function tex.getnest ( <t:integer> level, <t:string> name )
    return <t:whatever> -- value
end

function tex.setnest ( <t:integer> level, <t:string> name ), <t:whatever> value )
    -- no return values
end
```

Instead of an integer level you can use the keywords `ptr` and `top` instead of the current level or zero.

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   = <t:integer> value
tex.nest[tex.nest.ptr].spacefactor = <t:integer> value
```

However, the following also works for the current level:

```
tex.prevdepth   = <t:integer> value
tex.spacefactor = <t:integer> value
```

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.

11.3.11 Directions

In LuaMetaTEX we only have left-to-right (l2r) and right-to-left (r2l) directions, contrary to LuaTEX that has few more. In the end those made no sense because the typesetter is not geared for that and demands can be met by a combination of TEX macros and Lua code.

There are two sets of helpers:

```
function tex.gettextdir ( ) return <t:integer end
function tex.getlinedir ( ) return <t:integer end
function tex.getmathdir ( ) return <t:integer end
function tex.getpardir  ( ) return <t:integer end
function tex.getboxdir  ( ) return <t:integer end
```

and:

```
function tex.settextdir ( <t:integer> direction ) end -- no return values
function tex.setlinedir ( <t:integer> direction ) end -- no return values
function tex.setmathdir ( <t:integer> direction ) end -- no return values
function tex.setpardir  ( <t:integer> direction ) end -- no return values
function tex.setboxdir  ( <t:integer> direction ) end -- no return values
```

For old times sake you can also set them using the virtual interfaces, like

```
tex.textdirection = 1
```

but in ConTEXt we consider this obsolete. In LuaMetaTEX we dropped the direction related keywords and only use numbers:

`0x00`	lefttoright
`0x01`	righttoleft

11.3.12 Special lists

The virtual table `tex.lists` contains the set of internal registers that keep track of building page lists. We have the following lists plus some extras: `alignhead`, `bestpagebreak`, `bestsize`, `contributehead`, `holdhead`, `insertheights`, `insertpenalties`, `leastpagecost`, `pagediscardshead`, `pagehead`, `pageinserthead`, `postadjusthead`, `postmigratehead`, `preadjusthead`, `premigratehead`, `splitdiscardshead`, `temphead`. Using these assumes that you know what TEX is doing.

The getter and setter functions are `getlist` and `setlist`. You have to be careful with what you set as TEX can have expectations with regards to how a list is constructed or in what state it is.

```
function tex.getlist ( <t:string> name )
    return <t:whatever> -- value
end
```

```
function tex.setlist ( <t:string> name ), <t:whatever> value )
    -- no return values
end
```

You can mess up I ways that make the engine fail, for instance due to wrongly linked lists, for instance maybe circular, or invalid nodes.

11.3.13 Printing

The engine reads tokens from file, token lists and Lua. When we print from Lua it ends up in a special data structure that efficiently handle strings, tokens and nodes because we can push all three back to TEX. It is important to notice that when we have a call to Lua, that new input is collected and only pushed onto the input stack when we are done. The total amount of returnable text from a **`\directlua`** command or primitive driven function call 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. First we look at `tex.print` and `tex.sprint`.

```
function tex.print ( -- also tex.sprint
    <t:string> data,
    -- more strings
)
    -- nothing to return
end
```

```
function tex.print ( -- also tex.sprint
    <t:integer> catcodetable,
    <t:string>  data,
    -- more strings
```

```
)
    -- nothing to return
end

function tex.print ( -- also tex.sprint
    <t:table> data
)
    -- nothing to return
end

function tex.print ( -- also tex.sprint
    <t:integer> catcodetable,
    <t:table>   data
)
    -- nothing to return
end
```

With `tex.print` 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 first integer parameter can be used to print the strings using the catcode regime defined by **`\catcodetable`**. A value of -1 means that the currently active catcode regime is used while -2 gives a result similar to **`\the\toks`**: all category codes are 12 (other) except for the space character, that has category code 10 (space). An invalid catcode table index is silently ignored, and the currently active catcode regime is used instead. The very last string of the very last `tex.print` command in a **`\directlua`** call will not have the **`\endlinechar`** appended, all others do.

In the case if `tex.sprint` 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(" in between")}after
  ```

  the space before `in between` will be gobbled as a result of the 'normal' scanning of `\relax`.

Although this needs to be used with care, in both function 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 `<t:token>` already is tokenized and effectively is a token list now. A `<t:node>` 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.

The `tex.cprint` function is similar to `tex,sprint` but instead of am optional first catcodetable it takes a catcode value, like:

```
function tex.cprint (
     <t:integer> catcode,
     <string>    data
    -- more strings
)
    -- no return values
end
```

Of course the other three ways to call it are also supported. This might explain better:

```
\startluacode
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,"14: $&{\\foo}") tex.print("\\par") -- comment triggers
\stopluacode
```

We get two lines separate by one with only spaces:

11: $&{\foo}

12: $&{\foo}

A variant on `tex.sprint` is the next one:

```
function tex.tprint (
     { <t:integer> catcodetable, <string> data},
    -- more tables
)
    -- no return values
end
```

The `tex.write` function is a quick way to dump information. Each string argument is treated as a special kind of input line that only has spaces and letters.

```
function tex.write ( <t:string> data, ... )
    -- no return values
end

function tex.write ( <t:table> data)
    -- no return values
end
```

Often you can mix strings, nodes and tokens in a print but you might want to check beforehand what you pass:

```
function tex.isprintable ( <t:whatever> object )
    return <t:boolean>
end
```

11.3.14 Numbers and dimensions

We can rounds a Lua number to an integer 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.

```
function tex.round ( <t:number> n )
    return <t:integer>
end
```

In many places the engine multiplies and divides integers and ensures proper rounding. In LuaMetaTEX some (new) mechanisms use doubles and round, especially when multiple scale value accumulate beyond the available integer range. The next function multiplies two Lua numbers 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.

```
function tex.scale ( <t:number> original, <t:number> factor )
    return <t:integer> -- result
end

function tex.scale ( <t:table> originals, <t:number> factor )
    return <t:table> -- results
end
```

Here are companions to the primitives **\number** and **\romannumeral**. Both take the long route: the string goes to TEX, gets tokenized, then converted to what is wanted and finally ends up in Lua. They can be used like:

```
function tex.number       ( <t:integer> original ) return <t:string> end
function tex.romannumeral ( <t:integer> original ) return <t:string> end
```

The dimension converter takes a string and returns an integer that represents an dimension in scaled points. When a number is passed it gets rounded.

```
function tex.toscaled ( <t:string> original ) return <t:integer> end
function tex.toscaled ( <t:number> original ) return <t:integer> end
```

For completeness the engine also provides `tex.tonumber`:

```
function tex.tonumber ( <t:string> original ) return <t:integer> end
function tex.tonumber ( <t:number> original ) return <t:integer> end
```

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)

11.3.15 Primitives

Where in LuaTeX we explicitly need to enable the core set of primitives, LuaMetaTeX does that for you. The only reason that we still have a way to enable them is that it's a convenient way to create prefixed copies.

```
function tex.enableprimitives (
    <t:string> prefix,
    <t:table>  names
)
    -- no return values
end
```

Only valid primitive names are processed. Because it is no fun to enter the names, there is this one. It has two variants, where the boolean variant returns a table with all primitives.

```
function tex.extraprimitives ( <t:string> subset, ... )
    return <t:table> -- names
end

function tex.extraprimitives ( <t:true> )
    return <t:table> -- names
end
```

Possible values for `subset` are:

`0x01`	tex
`0x02`	etex
`0x04`	luatex

You can feed the result of the last one in `tex.enableprimitives`. If there is already a macro with that name if will not be overloaded.

```
tex.enableprimitives('normal',tex.extraprimitives(true))}
```

A complete list of primitives can be requested by:

```
function tex.primitives ( )
    return <t:table> -- names
end
```

of course the fact that the name is there doesn't mean that it has the same meaning.

A complete list of all hash entries can be asked for by the following function, but in ConTEXt it will be a big one, a bit more that 50.000 names, many of which are kind of weird because they use some namespace.

```
function tex.hashtokens ( )
    return <t:table> -- names
end
```

11.3.16 Values (constants)

The engine uses lots of very specific values (constants) for control. These can be status values (where are we currently), options (in nodes), control parameters (typesetting), etc. and all are available in lists that relate numbers to strings. Here is the complete list. We show the results in various places in the documentation. The advantage is that the engine is partly self documenting.

```
function tex.getadjustoptionvalues        ( ) return <t:table> end
function tex.getalignmentcontextvalues    ( ) return <t:table> end
function tex.getappendlinecontextvalues   ( ) return <t:table> end
function tex.getautomigrationvalues       ( ) return <t:table> end
function tex.getautoparagraphvalues       ( ) return <t:table> end
function tex.getbreakcontextvalues        ( ) return <t:table> end
function tex.getbuildcontextvalues        ( ) return <t:table> end
function tex.getcharactercontrolvalues    ( ) return <t:table> end
function tex.getcharactertagvalues        ( ) return <t:table> end
function tex.getdirectionvalues           ( ) return <t:table> end
function tex.getdiscoptionvalues          ( ) return <t:table> end
function tex.getdiscpartvalues            ( ) return <t:table> end
function tex.getdoublescriptoptionvalues  ( ) return <t:table> end
function tex.geterrorvalues               ( ) return <t:table> end
function tex.getfillvalues                ( ) return <t:table> end
function tex.getflagvalues                ( ) return <t:table> end
function tex.getfrozenparvalues           ( ) return <t:table> end
function tex.getglueoptionvalues          ( ) return <t:table> end
function tex.getglyphdiscvalues           ( ) return <t:table> end
function tex.getglyphoptionvalues         ( ) return <t:table> end
function tex.getglyphprotectionvalues     ( ) return <t:table> end
function tex.getgroupvalues               ( ) return <t:table> end
function tex.gethyphenationvalues         ( ) return <t:table> end
function tex.getiftypes                   ( ) return <t:table> end
function tex.getinteractionmodes          ( ) return <t:table> end
function tex.getiovalues                  ( ) return <t:table> end
function tex.getkerneloptionvalues        ( ) return <t:table> end
function tex.getlinebreakparameterfields  ( ) return <t:table> end
function tex.getlinebreakresultfields     ( ) return <t:table> end
function tex.getlinebreakstatevalues      ( ) return <t:table> end
function tex.getlistanchorvalues          ( ) return <t:table> end
function tex.getlistfields                ( ) return <t:table> end
function tex.getlistgeometryvalues        ( ) return <t:table> end
```

```
function tex.getlistsignvalues             ( ) return <t:table> end
function tex.getlocalboxlocations          ( ) return <t:table> end
function tex.getmathclassoptionvalues      ( ) return <t:table> end
function tex.getmathcontrolvalues          ( ) return <t:table> end
function tex.getmathgluevalues             ( ) return <t:table> end
function tex.getmathoptionvalues           ( ) return <t:table> end
function tex.getmathparametervalues        ( ) return <t:table> end
function tex.getmathscriptordervalues      ( ) return <t:table> end
function tex.getmathscriptsmodevalues      ( ) return <t:table> end
function tex.getmathstylenamevalues        ( ) return <t:table> end
function tex.getmathstylevalues            ( ) return <t:table> end
function tex.getmathsurroundvalues         ( ) return <t:table> end
function tex.getmathvariantpresets         ( ) return <t:table> end
function tex.getmathvariantvalues          ( ) return <t:table> end
function tex.getmarknames                  ( ) return <t:table> end
function tex.getmodevalues                 ( ) return <t:table> end
function tex.getnestfields                 ( ) return <t:table> end
function tex.getnoadoptionvalues           ( ) return <t:table> end
function tex.getnormalizelinevalues        ( ) return <t:table> end
function tex.getnormalizeparvalues         ( ) return <t:table> end
function tex.getpacktypevalues             ( ) return <t:table> end
function tex.getpagecontextvalues          ( ) return <t:table> end
function tex.getpagestatevalues            ( ) return <t:table> end
function tex.getparametermodevalues        ( ) return <t:table> end
function tex.getparcontextvalues           ( ) return <t:table> end
function tex.getparmodevalues              ( ) return <t:table> end
function tex.getpartriggervalues           ( ) return <t:table> end
function tex.getpenaltyoptionvalues        ( ) return <t:table> end
function tex.getprepoststatevalues         ( ) return <t:table> end
function tex.getprimitiveorigins           ( ) return <t:table> end
function tex.getprotrusionboundaryvalues   ( ) return <t:table> end
function tex.getruleoptionvalues           ( ) return <t:table> end
function tex.getrunstatevalues             ( ) return <t:table> end
function tex.getshapingpenaltiesvalues     ( ) return <t:table> end
function tex.getspecialmathclassvalues     ( ) return <t:table> end
function tex.getspecificationoptionvalues ( ) return <t:table> end
function tex.gettextcontrolvalues          ( ) return <t:table> end
function tex.getuleaderlocationvalues      ( ) return <t:table> end
function tex.getunitclassvalues            ( ) return <t:table> end
```

11.3.17 Glyphs

There are a few (internal) integer parameters that relate to glyphs, **\glyphdatafield**, **\glyphstatefield**, **\glyphscriptfield** as well as the three scales **\glyphscale**, **\glyphxscale** and **\glyphyscale**, and for these we have fast accessors:

```
function tex.setglyphdata   ( <t:integer> ) end
function tex.setglyphstate  ( <t:integer> ) end
function tex.setglyphscript ( <t:integer> ) end
```

and

```
function tex.getglyphdata   ( ) return <t:integer> end
function tex.getglyphstate  ( ) return <t:integer> end
function tex.getglyphscript ( ) return <t:integer> end
```

The scale getter returns more:

```
function tex.getglyphscales ( )
    return
        <t:integer>, -- scale
        <t:integer>, -- xscale
        <t:integer>, -- yscale
        <t:integer>  -- data
end
```

11.3.18 Whatever

We have no backend so all that the next does is wiping the box:

```
function tex.shipout ( <t:integer> index )
    -- no return values
end
```

This helper function is useful during line break calculations. The arguments `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 $100(t/s)^3$.

```
function tex.badness (
    <t:integer> t,
    <t:integer> s
)
    return <t:integer>
end
```

The page builder can be in different states, so here is how you get the current state:

```
function tex.getpagestate ( )
    return <t:integer>
end
```

possible states are:

`0x00`	none	`0x02`	box
`0x01`	insert	`0x03`	rule

You can also check if we're in the output routine:

```
function tex.getoutputactive ( )
    return <t:boolean>
end
```

An example of a (possible error triggering) complication is that TEX expects to be in some state, say horizontal mode, and you have to make sure it is when you start feeding back something from Lua into TEX. Normally a user will not run into issues but when you start writing tokens or nodes or have a nested run there can be situations that you need to enforce horizontal mode. There is no recipe for this and intercepting possible cases would weaken LuaTEX's flexibility. Therefore we provide `forcehmode` which is similar to **\quitvmode** at the TEX end, although in ConTEXt, that had it already, we always use **\dontleavehmode** as name.

```
function tex.forcehmode ( )
    -- no return values
end
```

The last node in the current list is queried with the following helper. If there is no node you get `nil`'s back.

```
function tex.lastnodetype ( )
    return
        <t:integer>, -- type
        <t:integer>  -- subtype
end
```

The current mode is available with:

```
function tex.getmode ( )
    return <t:integer>
end
```

Currently we're in mode `0x2`, a number that you can give meaning with `tex.getmodevalues()`:

0x00	unset
0x01	vertical
0x02	horizontal
0x03	math

The run state can be fetched with:

```
function tex.getrunstate ( )
    return <t:integer>
end
```

which returns one of:

0x00	initializing
0x01	updating
0x02	production

When we load create the format file we're initializing and when we then do a regular run we are in production. The updating state is just there so that we can deal with overload protection. In that case we need to honor the **\enforced** prefix, that can only be used when not in production mode. When a runtime module nevertheless wants to use that prefix it can (from Lua) set the mode to updating. This is all kind of ConTEXt specific because there we use the overload protection mechanism.

11.3.19 Files and lines

You can register a file id and line number in a `glyph`, `hlist` and `vlist` nodes, for instance for implementing a SyncTEX emulator. There are some helpers that relate to this. When the mode is zero, no registering will done, when set to one, lists will be tagged and larger values make that glyphs will be tagged too.

```
function tex.setinputstatemode ( <t:integer> mode )
    -- no return values
end
```

The file is registered as a number and the engine is agnostic about what it refers too. The same is true for lines. In fact, you can use these fields for whatever purpose you like.

```
function tex.setinputstatefile ( <t:integer> fileid )
    -- no return values
end
```

```
function tex.setinputstateline ( <t:integer> linenumber )
    -- no return values
end
```

The getters just return the currently set values:

```
function tex.getinputstatemode ( ) return <t:integer> end
function tex.getinputstatefile ( ) return <t:integer> end
function tex.getinputstateline ( ) return <t:integer> end
```

The file and line number are bound to the current input which can be nested. So, nesting is handled by the engine. However, you can overload that with the following two helpers. The values set will win over the ones bound to the current input file.

```
function tex.forceinputstatefile ( <t:integer> fileid )
    -- no return values
end
```

```
function tex.forceinputstateline ( <t:integer> linenumber )
    -- no return values
end
```

11.3.20 Interacting

In LuaMetaTEX valid interaction modes are:

`0x00`	batch	`0x02`	scroll
`0x01`	nonstop	`0x03`	errorstop

You can get and set the mode with:

```
function tex.getinteraction ( )
    return <t:integer> -- mode
```

```
end
function tex.setinteraction ( <t:integer> mode )
    -- no return values
end
```

When an error occurs it can be intercepted by a callback in which case you have to handle the feedback yourself. For this we have two helpers:

```
function tex.showcontext ( ) end
function tex.gethelptext ( ) end
```

An error can be triggered with:

```
function tex.error (
    <t:string> error,
    <t:string> help
)
    -- no return values
end
```

Of course these are also intercepted by the callback, when set, in which case the help text can be fetched. There can arise a situation where the engine is in a state where properly dealing with errors has become a problem. In that case you can use:

```
function tex.fatalerror ( <t:string> error) end
```

In this case the run will be aborted. For the record: in ConTEXt any error will quit the run, just because it makes no sense to try to recover from unpredictable situations and a fix is needed anyway.

11.3.21 Save levels

When you start a group or any construct that behaves like one, for instance boxing, the save stack is 'pushed' which means that a boundary is set. When the group ends the values that were saved in the current region (bounded) are restored. You can also do this in Lua:

```
function tex.pushsavelevel ( ) end
function tex.popsavelevel  ( ) end
```

This is a way to create grouping when in Lua so that when you set some register the engine will handle the restore.

11.3.22 Local control

When we talk about local control we mean expanding TEX code in a nested main loop. We start with explaining `tex.runlocal`. The first argument can be a number (of a token register), a macro name, the name of a token list or some (userdata) token made at the Lua end. The second argument is optional and when true forces expansion inside a definition. The optional third argument can be used to force grouping. The return value indicates an error: 0 means no error, 1 means that a bad register number has been passed, a value of 2 indicated an unknown register or macro name, while 3 reports that the macro is not suitable for local control because it takes arguments.

```
\scratchtoks{This is {\bf an example} indeed.}%
```

```
\startluacode
    tex.runlocal("scratchtoks")
\stopluacode
```

This typesets: This is **an example** indeed.

However, the neat thing about local control is that it happens immediately, so not after the Lua blob ended as with `tex.print ("\\the\scratchtoks")`.

```
\scratchtoks{\setbox\scratchbox\hbox{This is {\bf an example} indeed.}}%
\startluacode
    tex.runlocal("scratchtoks")
    context("The width is: %p",tex.box.scratchbox.width)
\stopluacode
```

This typesets: The width is: 140.53223pt

```
function tex.runlocal (
    <t:string>  name,
    <t:boolean> expand,
    <t:boolean> group
)
    return <t:integer> -- state
end
```

You can quit a local controlled expansion with the following, but if it works depends on the situation.

```
function tex.quitlocal ( )
    -- no return values
end
```

There might be situations that you push something from Lua to TEX in a local call and don't want interference. In that case wrapping might help but it is not that well tested yet:

```
function tex.pushlocal ( )
    -- no return values
end

function tex.poplocal  ( )
    -- no return values
end
```

The current level of local calls is available with:

```
function tex.getlocallevel ( )
    return <t:integer>
end
```

You can also run a string through TEX; the last three booleans are optional.

```
function tex.runlocal (
    <t:string>  str,
    <t:boolean> expand_in_definitions,
```

```
    <t:boolean> group,
    <t:boolean> ignore_undefind_cs
)
    -- no return values
end

function tex.runlocal (
    <t:integer> catcodetable,
    <t:string>  str,
    <t:boolean> expand_in_definitions,
    <t:boolean> group,
    <t:boolean> ignore_undefind_cs
)
    -- no return values
end
```

11.3.23 Math

There are some setters and getters that relate to the math sub engine. The setter has two variants:

```
function tex.setmathcode (
    <t:integer> target,
    <t:integer> class,
    <t:integer> family,
    <t:integer> character
)
    -- no return values
end

function tex.setmathcode (
    <t:integer> target,
    <t:table>   {
        <t:integer>, -- class
        <t:integer>, -- family
        <t:integer>  -- character
    }
)
    -- no return values
end
```

But there are two getters:

```
function tex.getmathcode (
    <t:integer> target
)
    return <t:table> {
        <t:integer>, -- class
        <t:integer>, -- family
        <t:integer>  -- character
    }
```

```
end

function tex.getmathcodes (
    <t:integer> target
)
    return
        <t:integer>, -- class
        <t:integer>, -- family
        <t:integer>  -- character
end
```

Delcodes have different properties:

```
function tex.setdelcode (
    <t:integer> target,
    <t:integer> smallfamily,
    <t:integer> smallcharacter,
    <t:integer> largefamily,
    <t:integer> largecharacter
)
    -- no return values
end

function tex.setdelcode (
    <t:integer> target,
    <t:table>   {
        <t:integer>, -- smallfamily,
        <t:integer>, -- smallcharacter,
        <t:integer>, -- largefamily,
        <t:integer>  -- largecharacter
    }
)
    -- no return values
end
```

Again there two getters:

```
function tex.getdelcode (
    <t:integer> target
)
    return <t:table> {
        <t:integer>, -- smallfamily,
        <t:integer>, -- smallcharacter,
        <t:integer>, -- largefamily,
        <t:integer>  -- largecharacter
    }
end

function tex.getdelcodes (
    <t:integer> target
)
```

```
    return
        <t:integer>, -- smallfamily,
        <t:integer>, -- smallcharacter,
        <t:integer>, -- largefamily,
        <t:integer>  -- largecharacter
 end
```

In LuaMetaTEX the engine can do without these delimiter specifications so they might eventually go way. The reason is that when a delimiter is needed we also accent a math character. When we use an OpenType model it's likely that the large character comes from the same font as the small character. And because the font is loaded under Lua control one can always use a virtual character to refer to an other font, something that we do in ConTEXt when we load a Type1 based math font.

A named math character is defined with `mathchardef` but contrary to its TEX counterpart **`\mathchardef`** it accepts three four extra parameters. The `properties`, `group` and `index` are data fields that the (for instance) the backend can use. We make no assumptions about their use because it is macro package dependent. There can be flags before the three optional parameters.

```
function tex.mathchardef (
    <t:string>  name
    <t:integer> class,
    <t:integer> family,
    <t:integer> character,
    <t:integer> flags,  -- zero or more
    <t:integer> properties,
    <t:integer> group,
    <t:integer> index
)
    -- no return values
end
```

The **`\chardef`** equivalent is:

```
function tex.chardef (
    <t:string>  name
    <t:integer> character,
    <t:integer> flags,  -- zero or more
)
    -- no return values
end
```

Math parameters have their own setter and getter. The first string is the parameter name minus the leading `Umath`, and the second string is the style name minus the trailing `style`. A value is either an integer (representing a dimension or number) or a list of glue components.

```
function tex.setmath (
    <t:string>  prefix, -- zero or more
    <t:integer> parameter,
    <t:integer> style,
    <t:integer> value,  -- one or more
)
```

```
    -- no return values
end

function tex.setmath (
    <t:integer> parameter,
    <t:integer> style
)
    return <t:integer> -- one or more value
end
```

For the next one you need to know what style variants which we will not discuss here:

```
function tex.getmathstylevariant (
    <t:integer> style,
    <t:integer> parameter
)
    <t:integer>, -- value
    <t:integer>  -- variant
end
```

11.3.24 Processing

You should not expect to much from the `triggerbuildpage` helpers because often TeX doesn't do much if it thinks nothing has to be done, but it might be useful for some applications. It just does as it says it calls the internal function that build a page, given that there is something to build.

```
function tex.triggerbuildpage ( )
    -- no return values
end
```

This function resets the parameters that TeX normally resets when a new paragraph is seen.

```
function tex.resetparagraph ( )
    -- no return values
end
```

The linebreak algorithm can also be applied explicitly to a node list that better be right. There is some checking done with respect to the beginning and paragraph and interfering glue.

```
function tex.linebreak (
    <t:direct> listhead,
    <t:table>  parameters
)
    return
        <t:direct>, -- nodelist
        <t:table>   -- info
end
```

There are a lot of parameters that drive the process and many can be set. The interface might be extended in the future. Valid parameter fields are: `adjacentdemerits`, `adjdemerits`, `adjustspacing`, `adjustspacing`, `adjustspacingshrink`, `adjustspacingstep`, `adjustspacingstretch`, `baselineskip`, brokenpenalties, brokenpenalty, clubpenalties, clubpenalty, direction, displaywidowpenalties, displaywidowpenalty, doublehyphendemerits, emergencyextrastretch, emergencyleftskip, emergencyrightskip, emergencystretch, exhyphenpenalty, finalhyphendemerits, fitnessclasses, hangafter, hangindent, hsize, hyphenationmode, hyphenpenalty, interlinepenalties, interlinepenalty, lastlinefit, leftskip, lefttwindemerits, linebreakchecks, linebreakoptional, linepenalty, lineskip, lineskiplimit, looseness, orphanlinefactors, orphanpenalties, parfillleftskip, parfillrightskip, parinitleftskip, parinitrightskip, parpasses, parshape, pretolerance, protrudechars, rightskip, righttwindemerits, shapingpenaltiesmode, shapingpenalty, singlelinepenalty, toddlerpenalties, tolerance, tracingfitness, tracingparagraphs, tracingpasses, widowpenalties, widowpenalty. There is no need to set them (at all) because the usual TEX parameters apply when they are absent.

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 the following fields: demerits, looseness, prevdepth, prevgraf.

A list can be 'prepared' for a linebreak call with the next function. Normally the linebreak routine will do this. The return values are pointers to some relevant nodes.

```
function tex.preparelinebreak (
    <t:direct>  listhead
)
    return
        <t:node>, -- nodelist
        <t:table> -- info
        <t:direct>, -- par (head)
        <t:direct>, -- tail
        <t:direct>, -- parinitleftskip
        <t:direct>, -- parinitrightskip
        <t:direct>, -- parfillleftskip
        <t:direct>  -- parfillrightskip
end
```

```
function tex.snapshotpar ( <t:integer> bitset )
    return <t:integer> -- state (bitset)
end
```

The bitset is made from:

0x00000001	hsize	0x00000400	lastline	0x00100000	shapingpenalty
0x00000002	skip	0x00000800	linepenalty	0x00200000	orphanpenalty
0x00000004	hang	0x00001000	clubpenalty	0x00400000	toddlerpenalty
0x00000008	indent	0x00002000	widowpenalty	0x00800000	emergency
0x00000010	parfill	0x00004000	displaypenalty	0x01000000	parpasses
0x00000020	adjust	0x00008000	brokenpenalty	0x02000000	singlelinepenalty
0x00000040	protrude	0x00010000	demerits	0x04000000	hyphenpenalty
0x00000080	tolerance	0x00020000	shape	0x08000000	exhyphenpenalty
0x00000100	stretch	0x00040000	line	0x10000000	linebreakchecks
0x00000200	looseness	0x00080000	hyphenation	0x20000000	twindemerits

This one is handy when you mess with lists and want to take some parameters into account that matter when building a paragraph. The returned fields are: hangafter, hangindent, hsize, leftskip, parindent, parshape, rightskip.

```
function tex.getparstate ( )
    return <t:table>
end
```

A par shape normally is discarded when the paragraph ends but we can continue using it if needed. In that case we can shift the current array and either or not rotate.

```
function tex.shiftparshape (
    <t:integer> shift,
    <t:boolean> rotate
)
    -- no return values
end
```

A specification, like **\parshape** or **\widowpenalties** can be fetched with:

```
function tex.getspecification ( <t:string> name )
    return <t:table>
end
```

11.4 The configuration

The global texconfig table 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. Watch out: some keys are different from LuaTEX, which is a side effect of a more granular and dynamic memory management.

key	type	default	comment
buffersize	number/table	1000000	input buffer bytes
filesize	number/table	1000	max number of open files
fontsize	number/table	250	number of permitted fonts
hashsize	number/table	150000	number of hash entries
inputsize	number/table	10000	maximum input stack
languagesize	number/table	250	number of permitted languages
marksize	number/table	50	number of mark classes
nestsize	number/table	1000	max depth of nesting
nodesize	number/table	1000000	max node memory (various size)
parametersize	number/table	20000	max size of parameter stack
poolsize	number/table	10000000	max number of string bytes
savesize	number/table	100000	mas size of save stack
stringsize	number/table	150000	max number of strings
tokensize	number/table	1000000	max token memory
expandsize	number/table	10000	max expansion nesting
propertiessize	number	0	initial size of node properties table
functionsize	number	0	initial size of Lua functions table
errorlinesize	number	79	how much or an error is shown

`halferrorlinesize`	number	50	idem
`formatname`	string		
`jobname`	string		
`starttime`	number		for testing only
`useutctime`	number		for testing only
`permitloadlib`	number		for testing only

If no format name or jobname is given on the command line, the related keys will be tested first instead of simply quitting. The statistics library has methods for tracking down how much memory is available and has been configured. The size parameters take a number (for the maximum allocated size) or a table with three possible keys: `size`, `plus` (for extra size) and `step` for the increment when more memory is needed. They all start out with a hard coded minimum and also have an hard coded maximum, the the configured size sits somewhere between these.

11.5 Input and output

This library takes care of the low-level I/O interface: writing to the log file and/or the console. The log file is registered with the following function:

```
function texio.setlogfile ( <t:file> handle )
    -- no return values
end
```

When TEX serializes something it uses a seslector to determine where it goes. The public selectors are:

`0x01`	logfile
`0x02`	terminal
`0x03`	terminal_and_logfile

Internal we have a string selector, Lua buffer selector, and a so called pseudo selector that is used when we want to show the context of an error and that keeps track of the position. These are not opened up.

We start with `texio.write`. Without the `target` argument, it writes all given strings to the same location(s) that TEX writes messages to at that moment. If **`\batchmode`** is in effect, it writes only to the log, otherwise it writes to the log and the terminal. A target can be a number or string.

```
function texio.write ( <t:string> target, <t:string> s, ...)
    -- no return values
end
```

```
function texio.write ( <t:string> s, ... )
    -- no return values
end
```

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.

The next function behaves like the above, but makes 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.

One reason why log output can slow down a run is that the engine works piecewise instead of printing lines. Deep down many writes go character by character because messages can occur everywhere during the expansion process.

```
function texio.writenl ( <t:string> s, ... )
    -- no return values
end
```

The selector variants below always expect a selector, so there is no misunderstanding if `logfile` is a string or selector.

```
function texio.writeselector ( <t:string> s, ... )
    -- no return values
end
```

```
function texio.writeselectornl ( <t:string> s, ... )
    -- no return values
end
```

```
function texio.writeselectorlf ( <t:string> s, ... )
    -- no return values
end
```

The next function should be used with care. It acts as **\endinput** but at the Lua end. You can use it to (sort of) force a jump back to TEX. Normally a Lua call will just collect prints and at the end bump an input level and flush these prints. This function can help you stay at the current level but you need to know what you're doing (or more precise: what TEX is doing with input).

```
function texio.closeinput ( )
    -- no return values
end
```

math

12 Math

12.1 Introduction

There is a lot to tell about math typesetting in LuaMetaTEX but plenty is covered in articles, progress reports and manuals. Here we limit ourselves to some basics. This chapter mostly contains information that is not presented elsewhere. Because math in regular TEX is basically frozen and other macro packaged depend on that, the extensions we have in LuaMetaTEX are mainly useful for ConTEXt. Even there we don't use all features, because completely opening up and providing ways to control every aspect also served the purpose of testing: it just comes with the package.

This chapter is a variant on the one in the old LuaMetaTEX manual and it might evolve a bit. We will not discuss the many options that the engine provides, at least not now. There is an extensive "Math in ConTEXt" that shows the state of the art and serves as reference. In due time we might write some more about what happens deep down in the engine, although already plenty has been published during the upgrade, about dealing with math fonts as well as experimenting with new features. Because all gets wrapped in high level interfaces there is not that much need (nor audience) for endless explanations anyway. There are also examples given in the chapter that discusses all primitives. Most ConTEXt users will never see these low level math commands!

12.2 Traditional alongside OpenType

Because we started in 2019 from LuaTEX, by the end of 2021 this chapter started with this, even if we already reworked the engine:

> "At this point there is no difference between LuaMetaTEX and LuaTEX with respect to math. Well, this might no longer be true because we have more control options that define default behavior and also have a more extensive scaling model. Anyway, it should not look worse, and maybe even a bit better. 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.
>
> You might be surprised that we don't use all these new control features in ConTEXt LMTX, but who knows what might happen because users drive it. The main reason for adding so much is that I decided it made more sense to be complete now than gradually add more and more. At some point we should be able to say 'This is it'. Also, when looking at these features, you need to keep in mind that when it comes to math, LATEX is the dominant macro package and it never needed these engine features, so most are probably just here for exploration purposes."

Although we still process math as TEX does, there have been some fundamental changes to the machinery. Most of that is discussed in documents that come with ConTEXt and in Mikael Sundqvist math manual. Together we explored some new ways to deal with math spacing, penalties, fencing, operators, fractions, atoms and other features of the TEX engine. We started from the way ConTEXt used the already present functionality combine with sometimes somewhat dirty (but on the average working well) tricks.

Much in LuaMetaTeX math handling is about micro-typography and for us the results are quite visible. But, as far as we know, there have never been complaints or demands in the direction of the features discussed here. Also, TeX math usage outside ConTeXt is rather chiseled in stone (already for nearly three decades) so we don't expect other macro packages to use the new features anyway. Anyway, after spending a real lot of time on this we both decided that we're mostly feature complete.

12.3 Intermezzo

It is important to understand a bit how TeX handles math. The math engine is a large subsystem and basically can be divided in two parts: convert sequential input into a list of nodes where math related ones actually are sort of intermediate and therefore called noads.

In text mode entering `abc` results in three glyph nodes and `a b c` in three glyph nodes separated by (spacing) glue. Successive glyphs can be transformed in the font engine later on, just as hyphenation directive can be added. Eventually one (normally) gets a mix of glyphs, font kerns from a sequence of glyphs

In math mode `abc` results in three simple ordinary noads and `a b c` is equivalent to that: three noads. But `a bc` results in two ordinary noads where the second one has a sublist of two ordinary noads. Because characters have class properties, `( a + b = c )` results in a simple open noad, a simple ordinary, a simple binary, a simple ordinary, a simple relation, a simple ordinary and simple close noad. The next samples show a bit of this; in order to see the effects of spacing between ordinary atoms set it to `9mu`.

```
$a b c$ \quad $a bc$ \quad $abc$
```

$a\,b\,c$ varvar varvar $\quad$ $a\,b\,c$ varvar varvar $\quad$ $a\,b\,c$ varvar varvar

With `\tracingmath 1` we get this logged:

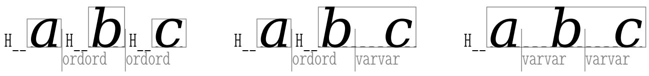
```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "61
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "62
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "63
```

```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "61
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "62
\noad[ord][...]
```

```
.\nucleus
..\mathchar[ord][...], family "0, character "63
```

```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "61
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "62
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "63
```

```
${a} {b} {c}$ \quad ${a} {bc}$ \quad ${abc}$
```



If the previous log surprises you, that might be because in ConTEXt we set up the engine differently: curly braces don't create ordinary atoms. However, when we set **\mathgroupingmode** 0 we return to what the engine normally does.

```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "61
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "62
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "63
```

```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\mathchar[ord][...], family "0, character "61
\noad[ord][...]
.\nucleus
..\submlist[0][...][tracing depth 5 reached]
```

```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\submlist[0][...][tracing depth 5 reached]
```

A warning is in place: tracing in LuaMetaTEX gets extended when we feel the need to gat more feedback from the engine. But it will only be more.

From the first example you can imagine what these sub lists look like: a list of ordinary atoms. The final list that is mix of nodes and yet unprocessed noads get fed into the math-to-hlist function and eventually the noads become glyphs, boxes, kerns, glue and whatever makes sense. A lot goes on there: think scripts, fractions, fences, accents, radicals, spacing, break control.

An example of more tricky scanning is shown here:

```
a +   1 \over 2   + b
a +  {1}\over{2}  + b
a + {{1}\over{2}} + b
```

In this case the `\over` makes TEX reconsider the last noad, remove if from the current list and save it for later, then scan for a following atom a single character turned atom or a braced sequence that then is an ordinary noad. In the end a fraction noad is made. When that gets processed later specific numerator and denominator styles get applied (explicitly entered style nodes of course overload this for the content). The fact that this construct is all about (implicit) ordinary noads, themselves captured in noads, combined with the wish for enforced consistent positioning of numerator and denominator, plus style overload, color support and whatever comes to mind means that in practice one will use a `\frac` macro that provides all that control.[14]

A similar tricky case is this:

```
      ( a +       ( b - c        ) + d        )
\left ( a + \left ( b - c \right ) + d \right )
```

Here the first line creates a list of noads but the second line create a fenced structure that is handled as a whole in order to make the fences match.[15] A fence noad will not break across lines as it is boxed and that is the reason why macro packages have these `\bigg` macros: they explicitly force a size using some trickery. In LuaMetaTEX a fence object can actually be unpacked when the class is configured as such. It is one of the many extensions we have.

There are some peculiar cases that one can run into but that actually are mentioned in the TEX book. Often these reasons for intentional side effects become clear when one thinks of the average usage but unless one is willing to spend time on the 'fine points of math' they can also interfere with intentions. The next bits of code are just for the reader to look at. Try to predict the outcome. Watch out: in LMTX the outcome is not what one gets by default in LuaTEX, pdfTEX or regular TEX.[16]

```
$ 1 {\red +} 2$\par
$ 1 \color[red]{+} 2$\par
$ 1 \mathbin{\red +} 2$\par
$ a + - b + {- b} $
$ a \pm - b - {+ b} $
$ - b $
$ {- b} $
```

The message here is that when a user is coding the mindset with respect to grouping using curly braces has to be switched to math mode too. And how many users really read the relevant chapters of the TEX book a couple of times (as much makes only sense after playing with math in TEX)? Even if one doesn't grasp everything it's a worthwhile read. Also consider this: did you really ask for an ordinary

[14] There are now a `\Uover` primitives that look ahead and then of course still treat curly braces as math lists to be picked up.
[15] Actually instead of such a structure there could have been delimiters with backlinks but one never knows what happens with these links when processing passes are made so that fragility is avoided.
[16] One can set `\mathgroupingmode = 0` to get close.

atom when you uses curly braces where no lists were expected? And what would have happened when ordinary related spacing had been set to non-zero?

All the above (and plenty more) is why in ConTEXt LMTX we make extensive use of some LuaMetaTEX features, like: additional atom classes, configurable inter atom spacing and penalties, pairwise atom rules that can change classes, class based rendering options, more font parameters, configurable style instead of hard coded ones in constructs, more granular spacing, etc. That way we get quite predictable results but also drop some older (un)expected behavior and side effects. It is also why we cannot show many examples in the LuaMetaTEX manual: it uses ConTEXt and we see no reason to complicate out lives (and spend energy on) turning off all the nicely cooperating features (and then for sure forgetting one) just for the sake of demos. It also gave us the opportunity to improve existing mechanisms and/or at least simplify their sometimes complex code.

One last word here about sequences of ordinary atoms: the traditional code path feeds ordinary atoms into a ligature and kerning routine and does that when it encounters one. However, in OpenType we don't have ligatures not (single) kerns so there that doesn't apply. As we're not aware of traditional math fonts with ligatures and no one is likely to use these fonts with LuaMetaTEX the ligature code has been disabled.[17] The kerning has been redone a bit so that it permits us to fine tune spacing (which in ConTEXt we control with goodie files). The mentioned routine can also add italic correction, but that happens selectively because it is driven by specifications and circumstances. It is one of the places where the approach differs from the original, if only for practical reasons.

In addition to what we explained above, we mention the **`\beginmathgroup`** and **`\endmathgroup`** primitives behave like **`\begingroup`** and **`\endgroup`** but restore a style change inside the group. Style changes are actually injecting s special style noad which makes them sort of persistent till the next explicit change which can be confusing. This additional grouping model compensates for that.

12.4 Unicode math characters

For various reasons we need to encode a math character in a 32 bit number and because we often also need to keep track of families and classes the range of characters is limited to 20 bits. There are upto 64 classes (which is a lot more than in LuaTEX) and 64 families (less than in LuaTEX). The upper limit of characters is less that what Unicode offers but for math we're okay. If needed we can provide less families.

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 the larger character codes argument on the left hand side of the equals sign. The number variants of some primitives might be dropped in favor of the primitives that read more than one separate value (class, family and code). All relevant primitives are explained in the primitives chapter.

A delimiter in traditional TEX combines two definitions: the regular character and the way it can become a larger (extensible) one. The small character is just like a math character but the larger one can come from a different font (family). However, in OpenType math fonts the larger sizes (variants) and extensibles (parts) come from the same font. For that reason LuaMetaTEX also accepts a math character when a delimited specifier is expected. It basically means that we could remove delimiters as such from the engine. After all, when we let Lua load a traditional font we can as well use virtual fonts to handle the variants and extensibles, which is indeed the case when we support the jmh fonts.

[17] It might show up in a different way if we feel the need in which case it's more related to runtime patches to fonts and class bases ligature building.

12.5 Math classes

Most characters belong to a so called math class which can be set for each character if needed. There are upto 64 classes of which at this moment about 20 are predefined so, taking some future usage by the engine into account,you can assume 32 upto 60 to be available for any purpose. The number of families has been reduced from 256 to 64 which is plenty for daily use in an OpenType setup. If we ever need to expand the Unicode range there will be less families or we just go for a larger internal record. The values of begin and end classes and the number of classes can be fetched from the Lua status table. There are callbacks that makes it possible to report user classes when there is the need.

12.6 Setting up the engine

Rendering math has long been dominated by TEX but that changed when Microsoft came with OpenType math: an implementation as well as a font. Some of that was modeled after TEX and some was dictated (we think) by the way word processors deal with math. For instance, traditional TEX math has a limited set of glyph properties and therefore has a somewhat complex interplay between width and italic correction. There are no kerns, contrary to OpenType math fonts that provides staircase kerns. Interestingly TEX does have some ligature building going on in the engine.

In traditional TEX italic correction gets added to the width and selectively removed later (or compensated by some shift and/or cheating with the box width). When we started with LuaTEX we had to gamble quite a bit about how to apply parameters and glyph properties which resulted in different code paths, heuristics, etc. That worked on the average but fonts are often not perfect and when served as an example for another one the bad bits can be inherited. That said, over time the descriptions improved and this is what the OpenType specification has to say about italic correction now[18]:

1. When a run of slanted characters is followed by a straight character (such as an operator or a delimiter), the italics correction of the last glyph is added to its advance width.

2. When positioning limits on an N-ary operator (e.g., integral sign), the horizontal position of the upper limit is moved to the right by half the italics correction, while the position of the lower limit is moved to the left by the same distance.

3. When positioning superscripts and subscripts, their default horizontal positions are also different by the amount of the italics correction of the preceding glyph.

The first rule is complicated by the fact that 'followed' is vague: in TEX the sequence `$ a b c def $` results in six separate atoms, separated by inter atom spacing. The characters in these atoms are the nucleus and there can be a super- and/or subscript attached and in LuaMetaTEX also a prime, super-prescript and/or sub-prescript.

The second rule comes from TEX and one can wonder why the available top accent anchor is not used. Maybe because bottom accent anchors are missing? Anyway, we're stuck with this now.

The third rule also seems to come from TEX. Take the '*f*' character: in TEX fonts that one has a narrow width and part sticks out (in some even at the left edge). That means that when the subscript gets attached it will move inwards relative to the real dimensions. Before the superscript an italic correction is added so what that correction is non-zero the scripts are horizontally shifted relative to each other.

18 `https://docs.microsoft.com/en-us/typography/opentype/spec/math`

Now look at this specification of staircase kerns[19]:

> The MathKernInfo table provides mathematical kerning values used for kerning of subscript and superscript glyphs relative to a base glyph. Its purpose is to improve spacing in situations such as omega with superscript f or capital V with subscript capital A.
>
> Mathematical kerning is height dependent; that is, different kerning amounts can be specified for different heights within a glyph's vertical extent. For any given glyph, different values can be specified for four corner positions, top-right, to-left, etc., allowing for different kerning adjustments according to whether the glyph occurs as a subscript, a superscript, a base being kerned with a subscript, or a base being kerned with a superscript.

Again we're talking super- and subscripts and should we now look at the italic correction or assume that the kerns do the job? This is a mixed bag because scripts are not always (single) characters. We have to guess a bit here. After years of experimenting we came to the conclusion that it will never be okay so that's why we settled on controls and runtime fixes to fonts.

This means that processing math is controlled by **\mathfontcontrol**, a numeric bitset parameter. The recommended bits are marked with a star but it really depends on the macro package to set up the machinery well. Of course one can just enable all and see what happens.[20] A list of possible control bits can be found in the primitives chapter where we discuss this parameter.

So, to summarize: the reason for this approach is that traditional and OpenType fonts have different approaches (especially when it comes to dealing with the width and italic corrections) and is even more complicated by the fact that the fonts are often inconsistent (within and between). In ConTEXt we deal with this by runtime fixes to fonts. In any case the Cambria font is taken as reference.

It is important to notice that in ConTEXt we no longer use italic correction at all. After many experiments Mikael Sundvist and I settled on a different approach where we use true widths, proper anchors, a new set of corner kerns, additional parameters and more. We tweak the fonts to match this model which in our opinion gives better results and less interference. We could actually simplify the engine and kick italics out of math but for the moment we keep it around so that we can show improvements in manuals and articles.

12.7 Math styles

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. Beware that contrary to LuaTEX this is now a proper number so you need to use **\number** or **\the** in order to serialize it.

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}
```

[19] Idem.

[20] This model was more granular and could even be font (and character) specific but that was dropped because fonts are too inconsistent and an occasional fit is more robust that a generally applied rule.

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

This gives:

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

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

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

[a : 0] (**x**:**d**:**4**) [b:0] (**y**:**s**:**6**) [c:0] (**z**:**ss**:**6**) [d:0]

[a : 2] (**x**:**t**:**6**) [b:2] (**y**:**ss**:**6**) [c:2] (**z**:**ss**:**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.

So far for the more LuaTEXish approach. One problem with **\mathstyle** is that when you got it, and want to act upon it, you need to remap it onto say **\scriptstyle** which can be done with an eight branched **\ifcase**. This is why we also have a more efficient alternative that you can use in macros:

```
\normalexpand{ ... \givenmathstyle\the\mathstyle      ...}
\normalexpand{ ... \givenmathstyle\the\mathstackstyle ...}
```

This new primitive **\givenmathstyle** accepts a numeric value. The **\mathstackstyle** primitive is just a bonus (it complements **\mathstack**).

The styles that the different math components and their sub components start out with are no longer hard coded but can be set at runtime:

primitive name	default
\Umathoverlinevariant	cramped
\Umathunderlinevariant	normal
\Umathoverdelimitervariant	small
\Umathunderdelimitervariant	small
\Umathdelimiterovervariant	normal
\Umathdelimiterundervariant	normal
\Umathhextensiblevariant	normal
\Umathvextensiblevariant	normal
\Umathfractionvariant	cramped
\Umathradicalvariant	cramped
\Umathdegreevariant	doublesuperscript
\Umathaccentvariant	cramped
\Umathtopaccentvariant	cramped
\Umathbottomaccentvariant	cramped
\Umathoverlayaccentvariant	cramped
\Umathnumeratorvariant	numerator
\Umathdenominatorvariant	denominator
\Umathsuperscriptvariant	superscript
\Umathsubscriptvariant	subscript
\Umathprimevariant	superscript
\Umathstackvariant	numerator

These defaults remap styles are as follows:

default	result	mapping
`cramped`	cramp the style	D' D' T' T' S' S' SS' SS'
`subscript`	smaller and cramped	S' S' S' S' SS' SS' SS' SS'

`small`	smaller	S S S S SS SS SS SS
`superscript`	smaller	S S S S SS SS SS SS
`smaller`	smaller unless already SS	S S' S S' SS SS' SS SS'
`numerator`	smaller unless already SS	S S' S S' SS SS' SS SS'
`denominator`	smaller, all cramped	S' S' S' S' SS' SS' SS' SS'
`doublesuperscript`	smaller, keep cramped	S S' S S' SS SS' SS SS'

The main reason for opening this up was that it permits experiments and removed hard coded internal values. But as these defaults served well for decades there are no real reasons to change them.

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: `\mathstack`:

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

The `\mathstack` command will scan the next brace and start a new math group with the correct (numerator) math style. The `\mathstackstyle` primitive relates to this feature.

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.

Because internally the eight styles are represented as numbers some of the new primitives that relate to them also work with numbers and often you can use them mixed. The `\tomathstyle` prefix converts a symbolic style into a number so `\number\tomathstyle\crampedscriptstyle` gives 5.

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	$b^{x=xx}_{x=xx}$
script	$b^{x=xx}_{x=xx}$
crampedscript	$b^{x=xx}_{x=xx}$

Now we set the following parameters using **\setmathspacing** that accepts two class identifier, a style and a value.

```
\setmathspacing 0 3 \scriptstyle = 30mu
\setmathspacing 0 3 \scriptstyle = 30mu
```

This gives a different result:

style	example
default	$b^{x\quad =xx}_{x=xx}$
script	$b^{x\quad =xx}_{x\quad =xx}$
crampedscript	$b^{x=xx}_{x=xx}$

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

```
\setmathspacing 0 3 \scriptstyle        = 30mu
\setmathspacing 0 3 \scriptstyle        = 30mu
\setmathspacing 0 3 \crampedscriptstyle = 30mu
\setmathspacing 0 3 \crampedscriptstyle = 30mu
```

Now we get:

style	example
default	$b^{x\quad =xx}_{x\quad =xx}$
script	$b^{x\quad =xx}_{x\quad =xx}$
crampedscript	$b^{x\quad =xx}_{x\quad =xx}$

12.8 Math parameters

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 turning some hard codes properties into parameters.

The next needs checking ...

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 an **\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

In addition to the above official OpenType font parameters we have these (the undefined will get presets, quite likely zero):

primitive name	description
\Umathconnectoroverlapmin	
\Umathsubsupshiftdown	
\Umathfractiondelsize	
\Umathnolimitsupfactor	a multiplier for the way limits are shifted up and down
\Umathnolimitsubfactor	a multiplier for the way limits are shifted up and down
\Umathaccentbasedepth	the complement of **\Umathaccentbaseheight**
\Umathflattenedaccentbasedepth	the complement of **\Umathflattenedaccentbaseheight**
\Umathspacebeforescript	
\Umathprimeraise	
\Umathprimeraisecomposed	
\Umathprimeshiftup	the prime variant of **\Umathsupshiftup**
\Umathprimespaceafter	the prescript variant of **\Umathspaceafterscript**
\Umathprimeshiftdrop	the prime variant of **\Umathsupshiftdrop**
\Umathskeweddelimitertolerance	
\Umathaccenttopshiftup	the amount that a top accent is shifted up
\Umathaccentbottomshiftdown	the amount that a bottom accent is shifted down
\Umathaccenttopovershoot	
\Umathaccentbottomovershoot	
\Umathaccentsuperscriptdrop	
\Umathaccentsuperscriptpercent	
\Umathaccentextendmargin	margins added to automatically extended accents
\Umathflattenedaccenttopshiftup	the amount that a wide top accent is shifted up
\Umathflattenedaccentbottomshiftdown	the amount that a wide bottom accent is shifted down
\Umathdelimiterpercent	
\Umathdelimitershortfall	
\Umathradicalextensiblebefore	
\Umathradicalextensibleafter	

These relate to the font parameters and in ConTEXt we assign some different defaults and tweak them in the goodie files:

font parameter	primitive name	default
`MinConnectorOverlap`	**\Umathconnectoroverlapmin**	0
`SubscriptShiftDownWithSuperscript`	**\Umathsubsupshiftdown**	inherited
`FractionDelimiterSize`	**\Umathfractiondelsize**	undefined
`FractionDelimiterDisplayStyleSize`	**\Umathfractiondelsize**	undefined
`NoLimitSubFactor`	**\Umathnolimitsupfactor**	0
`NoLimitSupFactor`	**\Umathnolimitsubfactor**	0
`AccentBaseDepth`	**\Umathaccentbasedepth**	reserved
`FlattenedAccentBaseDepth`	**\Umathflattenedaccentbasedepth**	reserved
`SpaceBeforeScript`	**\Umathspacebeforescript**	0
`PrimeRaisePercent`	**\Umathprimeraise**	0
`PrimeRaiseComposedPercent`	**\Umathprimeraisecomposed**	0
`PrimeShiftUp`	**\Umathprimeshiftup**	0
`PrimeShiftUpCramped`	**\Umathprimeshiftup**	0
`PrimeSpaceAfter`	**\Umathprimespaceafter**	0
`PrimeBaselineDropMax`	**\Umathprimeshiftdrop**	0

SkewedDelimiterTolerance	**\Umathskeweddelimitertolerance**	0
AccentTopShiftUp	**\Umathaccenttopshiftup**	undefined
AccentBottomShiftDown	**\Umathaccentbottomshiftdown**	undefined
AccentTopOvershoot	**\Umathaccenttopovershoot**	0
AccentBottomOvershoot	**\Umathaccentbottomovershoot**	0
AccentSuperscriptDrop	**\Umathaccentsuperscriptdrop**	0
AccentSuperscriptPercent	**\Umathaccentsuperscriptpercent**	0
AccentExtendMargin	**\Umathaccentextendmargin**	0
FlattenedAccentTopShiftUp	**\Umathflattenedaccenttopshiftup**	undefined
FlattenedAccentBottomShiftDown	**\Umathflattenedaccentbottomshiftdown**	undefined
DelimiterPercent	**\Umathdelimiterpercent**	0
DelimiterShortfall	**\Umathdelimitershortfall**	0

These parameters not only provide a bit more control over rendering, they also can be used in compensating issues in font, because no font is perfect. Some are the side effects of experiments and they have CamelCase companions in the `MathConstants` table. For historical reasons the names are a bit inconsistent as some originate in TeX so we prefer to keep those names. Not many users will mess around with these font parameters anyway.[21]

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.

There are quite some parameters that can be set and there are eight styles, which means a lot of keying in. For that reason is is possible to set parameters groupwise:

primitive name	D	D'	T	T'	S	S'	SS	SS'
\alldisplaystyles	+	+						
\alltextstyles			+	+				
\allscriptstyles					+	+		
\allscriptscriptstyles							+	+
\allmathstyles	+	+	+	+	+	+	+	+
\allmainstyles								
\allsplitstyles	+	+	+	+	−	−	−	−
\allunsplitstyles					+	+	+	+
\alluncrampedstyles	+		+		+		+	
\allcrampedstyles		+		+		+		+

These groups are especially handy when you set up inter atom spacing, pre- and post atom penalties and atom rules.

We already introduced the font specific math parameters but we tell abit more about them and how they relate to the original TeX font dimensions.

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

[21] I wonder if some names should change, so that decision is pending.

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`
`\Umathaccentbaseheight`	`xheight` `AccentBaseHeight`
`\Umathflattenedaccentbaseheight`	`xheight` `FlattenedAccentBaseHeight`
[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** D, D'	default_rule_thickness+abs(math_x_height)/4 RadicalDisplayStyleVerticalGap
[3] **\Umathradicalvgap** T, T', S, S', SS, SS'	default_rule_thickness+abs(default_rule_thickness)/4 RadicalVerticalGap
[2] **\Umathradicaldegreebefore**	<not set> RadicalKernBeforeDegree
[2] **\Umathradicaldegreeafter**	<not set> RadicalKernAfterDegree
[2,7] **\Umathradicaldegreeraise**	<not set> RadicalDegreeBottomRaisePercent
[4] **\Umathspaceafterscript**	script_space SpaceAfterScript
\Umathstackdenomdown D, D'	denom1 StackBottomDisplayStyleShiftDown
\Umathstackdenomdown T, T', S, S', SS, SS'	denom2 StackBottomShiftDown
\Umathstacknumup D, D'	num1 StackTopDisplayStyleShiftUp
\Umathstacknumup T, T', S, S', SS, SS'	num3 StackTopShiftUp
\Umathstackvgap D, D'	7*default_rule_thickness StackDisplayStyleGapMin
\Umathstackvgap T, T', S, S', SS, SS'	3*default_rule_thickness 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 D	sup1 SuperscriptShiftUp
\Umathsupshiftup T, S, SS,	sup2 SuperscriptShiftUp
\Umathsupshiftup D', T', S', SS'	sup3 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

A few notes:

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.

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.

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.

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.

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

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.

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.

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

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

As this mostly refers to LuaTEX there is more to tell about how LuaMetaTEX deals with it. However, it is enough to know that much more behavior is configurable.

You can let the engine ignore a parameter with **\setmathignore**, like:

```
\setmathignore \Umathspacebeforescript 1
\setmathignore \Umathspaceafterscript  1
```

Be aware of the fact that a global setting can get unnoticed by users because there is no warning that some parameter is ignored.

There are a couple of parameters that don't relate to the font but are more generally influencing the appearances. Some were added for experimenting.

This is not complete

primitive	meaning
\Umathextrasubpreshift	
\Umathextrasubprespace	
\Umathextrasubshift	
\Umathextrasubspace	
\Umathextrasuppreshift	
\Umathextrasupprespace	
\Umathextrasupshift	
\Umathextrasupspace	
\Umathprimeshiftdrop	

12.9 Math spacing

Besides the parameters mentioned in the previous sections, there are also primitives to control the math spacing table (as explained in Chapter 18 of the TEXbook). This happens per class pair. Because we have many possible classes, we no longer have the many primitives that LuaTEX has but you can define then using the generic **\setmathspacing** primitive:

```
\def\Umathordordspacing     {\setmathspacing 0 0 }
```

```
\def\Umathordordopenspacing {\setmathspacing 0 4 }
```

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

```
\setmathspacing 1 0 \displaystyle=4mu plus 2mu % op ord Umathopordspacing
```

The atom pairs known by the engine are all initialized by `initex` to the values mentioned in the table in Chapter 18 of the TEXbook.

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. As a bonus we also introduced the `\tinymuskip` and `\pettymuskip` primitives, just because we consider these fundamental, but they are not assigned internally to atom spacing combinations.

In LuaMetaTEX we go a bit further. Any named dimension, glue and mu glue register as well as the constants with these properties can be bound to a pair by prefixing `\setmathspacing` by `\inherited`.

Careful readers will realize that there are also primitives for the items marked * in the TEXbook. These will actually be used because we pose no restrictions. However, you can enforce the remapping rules to conform to the rules of TEX (or yourself).

Every class has a set of spacing parameters and the more classes you define the more pairwise spacing you need to define. However, you can default to an existing class. By default all spacing is zero and you can get rid of the defaults inherited from good old TEX with `\resetmathspacing`. You can alias class spacing to an exiting class with `\letmathspacing`:

```
\letmathspacing class displayclass textclass scriptclass scriptscriptclass
```

Instead you can copy spacing with `\copymathspacing`:

```
\copymathspacing class parentclass
```

Specific paring happens with `\setmathspacing`:

```
\setmathspacing leftclass rightclass style value
```

Unless we have a frozen parameter, the prefix `\inherited` makes it possible to have a more dynamic relationship: the used value resolves to the current value of the given register. Possible values are the usual mu skip register, a regular skip or dimension register, or just some mu skip value.

A similar set of primitives deals with rules. These remap pairs onto other pairs, so `\setmathatomrule` looks like:

```
\setmathatomrule oldleftclass oldrightclass newleftclass newrightclass
```

The `\letmathatomrule` and `\copymathatomrule` primitives take two classes where the second is the parent.

The `\setmathprepenalty` and `\setmathpostpenalty` primitives take a class and penalty (integer) value. These are injected before and after atoms with the given class where a penalty of 10000 is a signal to ignore it.

The engine control options for a class can be set with `\setmathoptions`. The possible options are discussed elsewhere. This primitive takes a class number and an integer (bitset). For all these setters the ConTEXt math setup gives examples.

Math is processed in two passes. The first pass is needed to intercept for instance `\over`, one of the few TEX commands that actually has a preceding argument. There are often lots of curly braces used in math and these can result in a nested run of the math sub engine. However, you need to be aware of the fact that some properties are kind of global to a formula and the last setting (for instance a family switch) wins. This also means that a change (or again, the last one) in math parameters affects the whole formula. In LuaMetaTEX we have changed this model a bit. One can argue that this introduces an incompatibility but it's hard to imagine a reason for setting the parameters at the end of a formula run and assume that they also influence what goes in front.

```
$
                                                x \subscript  {-}
     \frozen\Umathsubshiftdown\textstyle  0pt   x \subscript  {0}
    {\frozen\Umathsubshiftdown\textstyle  5pt   x \subscript  {5}}
                                                x \subscript  {0}
    {\frozen\Umathsubshiftdown\textstyle 15pt   x \subscript {15}}
                                                x \subscript  {0}
    {\frozen\Umathsubshiftdown\textstyle 20pt   x \subscript {20}}
                                                x \subscript  {0}
     \frozen\Umathsubshiftdown\textstyle 10pt   x \subscript {10}
                                                x \subscript  {0}
$
```

The `\frozen` prefix does the magic: it injects information in the math list about the set parameter.

In LuaTEX 1.10+ the last setting, the `10pt` drop wins, but in LuaMetaTEX you will see each local setting taking effect. The implementation uses a new node type, parameters nodes, so you might encounter these in an unprocessed math list. The result looks as follows:

$x_{-}\, x_0 x_5 x_0 x_{15}\, x_0\, x_{20}\, x_0\, x_{10}\, x_0$

The `\mathatom` primitive is the generic one and it accepts a couple of keywords:

to be checked

keyword	argument	meaning
`attr`	int int	attributes to be applied to this atom
`leftclass`	class	the left edge class that determines spacing etc
`rightclass`	class	the right edge class that determines spacing etc
`class`	class	the general class
`unpack`		unpack this atom in inline math
`source`	int	a symbolic index of the resulting box
`textfont`		use the current text font
`mathfont`		use the current math font
`limits`		put scripts on top and below
`nolimits`		force scripts to be postscripts
`nooverflow`		keep (extensible) within target dimensions

options	int	bitset with options
void		discard content and ignore dimensions
phantom		discard content but retain dimensions

To what extend the options kick in depends on the class as well where and how the atom is used.

The original TEX engines has three atom modifiers: **\displaylimits**, **\limits**, and **\nolimits**. These look back to the last atom and set a limit related signal. Just to be consistent we have some more of that: **\Umathadapttoleft**, **\Umathadapttoright**, **\Umathuseaxis**, **\Umathnoaxis**, **\Umathphantom**, **\Umathvoid**, **\Umathsource**, **\Umathopenupheight**, **\Umathopenupdepth**, **\Umathlimits**, **\Umathnolimits**. The last two are equivalent to the lowercase ones with the similar names. Al these modifiers are cheap primitives and one can wonder if they are needed but that also now also applies to the original three. We could stick to one modifier that takes an integer but let's not diverge too much from the original concept.

The **\nonscript** primitive injects a glue node that signals that the next glue is to be ignored when we are in script or scriptscript mode. The **\noatomruling** does the same but this time the signal is that no inter-atom rules need to be applied.

13 PDF

13.1 Introduction

There is no backend, not even a dvi one. In ConTEXt the main backend is a pdf backend and it is written in Lua. The pdf format makes it possible to embed jpeg and png encoded images as well as pdf images. All these have to be dealt with in Lua. Although we can parse pdf files with Lua, the engine has a dedicated pdf library on board written by Paweł Jackowski.

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 the document root and those are accessed more directly.

Although everything in pdf is basically an object we have to wrap a few in so called userdata Lua objects.

PDF	Lua
null	<t:nil>
boolean	<t:boolean>
integer	<t:integer>
float	<t:number>
name	<t:string>
string	<t:string>
array	<t:userdata>
dictionary	<t:userdata>
stream	<t:userdata>
reference	<t:userdata>

The interface is rather limited to creating an instance and getting objects and values. Aspects like compression and encryption are mostly dealt with automatically. In ConTEXt users use an interface layer around these, if they use this kind of low level code at all as it assumes familiarity with how pdf is constructed.

13.2 Lua interfaces

13.2.1 Opening and closing

There are two ways to open a pdf file:

```
function pdfe.open ( <t:string> filename )
    return <t:pdf>  -- pdffile
end

function pdfe.openfile( <t:file> filehandle )
    return <t:pdf> -- pdffile
end
```

Instead of from file, we can read from a string:

```
function pdfe.new ( <t:string>  somestring, <t:integer> somelength )
    return <t:pdf> -- pdffile
end
```

Closing the instance is done with:

```
function pdfe.close ( <t:pdf> pdffile )
    -- no return values
end
```

When we used `pdfe.open` the library manages the file and closes it when done. You can check if a document opened as expected by calling:

```
function pdfe.getstatus ( <t:pdf> pdffile )
    return <t:integer> -- status
end
```

A table of possible return codes can be queried with:

```
function pdfe.getstatusvalues ( )
    return <t:table> -- values
end
```

Currently we have these:

`-2`	is protected
`-1`	failed to open
`0`	not encrypted
`1`	is decrypted

An encrypted document can be decrypted by the next command where instead of either password you can give `nil` and hope for the best:

```
function pdfe.unencrypt (
    <t:pdf>    pdffile,
    <t:string> userpassword,
    <t:string> ownerpassword
)
    return <t:integer> -- status
end
```

13.2.2 Getting basic information

A successfully opened document can provide some information:

```
function pdfe.getsize( <t:pdf> pdffile )
    return <t:integer> -- nofbytes
end
```

```
function pdfe.getversion( <t:pdf> pdffile )
    return
        <t:integer>, -- major
```

```
        <t:integer>  -- minor
end

function pdfe.getnofobjects ( <t:pdf> pdffile )
    return <t:integer> -- nofobjects
end

function pdfe.getnofpages ( <t:pdf> pdffile )
    return <t:integer> -- nofpages
end

function pdfe.memoryusage ( <t:pdf> pdffile )
    return
        <t:integer>, -- bytes
        <t:integer>  -- waste
end
```

13.2.3 The main structure

For accessing the document structure you start with the so called catalog, a dictionary:

```
function pdfe.getcatalog( <t:pdf> pdffile )
    return <t:userdata> -- dictionary
end
```

The other two root dictionaries are accessed with:

```
function pdfe.gettrailer ( <t:pdf> pdffile )
    return <t:userdata> -- dictionary
end

function pdfe.getinfo ( <t:pdf> pdffile )
    return <t:userdata> -- dictionary
end
```

13.2.4 Getting content

A specific page can conveniently be reached with the next command, which returns a dictionary.

```
function pdfe.getpage ( <t:pdf> pdffile, <t:integer> pagenumber )
    return <t:userdata> -- dictionary
end
```

Another convenience command gives you the (bounding) box of a (normally page) which can be inherited from the document itself. An example of a valid box name is `MediaBox`.

```
function pdfe.getbox ( <t:pdf> pdffile, <t:string> boxname )
    return <t:table> -- boundingbox
end
```

13.2.5 Getters

Common values in dictionaries and arrays are strings, integers, floats, booleans and names (which are also strings) and these are also normal Lua objects. In some cases a value is a userdata object and you can use this helper to get some more information:

```
function pdfe.type ( <t:whatever> value )
    return type -- string
end
```

Stings are special because internally they are delimited by parenthesis (often `pdfdoc` encoding) or angle brackets (hexadecimal or 16 bit Unicode).

```
function pdfe.getstring (
    <t:userdata> object,
    <t:string>   key | <t:integer> index
)
    return
        <t:string>  -- decoded value
end
```

When you ask for more you get more:

```
function pdfe.getstring (
    <t:userdata> object,
    <t:string>   key | <t:integer> index,
    <t:boolean>  more
)
    return
        <t:string>, -- original
        <t:boolean>  -- hexencoded
end
```

Basic types are fetched with:

```
function pdfe.getinteger ( <t:userdata>, <t:string> key | <t:integer> index )
    return <t:integer> -- value
end

function pdfe.getnumber ( <t:userdata>, <t:string> key | <t:integer> index )
    return <t:number> -- value
end

function pdfe.getboolean ( <t:userdata>, <t:string> key | <t:integer> index )
    return <t:boolean> -- value
end
```

A name is (in the pdf file) a string prefixed by a slash, like `<< /Type /Foo >>`, for instance keys in a dictionary or keywords in an array or constant values.

```
function pdfe.getname ( <t:userdata>, <t:string> key | <t:integer> index )
    return <t:string> -- value
```

```
end
```

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.

```
function pdfe.getdictionary ( <t:userdata>, <t:string> key | <t:integer> index )
    return <t:userdata> -- dictionary
end
```

```
function pdfe.getarray ( <t:userdata>, <t:string> key | <t:integer> index )
    return <t:userdata> -- array
end
```

```
function pdfe.getstream ( <t:userdata>, <t:string> key | <t:integer> index )
    return
        <t:userdata> -- stream
        <t:userdata> -- dictionary
end
```

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])
```

13.2.6 Streams

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:

```
function pdfe.openstream ( <t:userdata> stream, <t:boolean> decode)
    return <t:boolean> okay
end
```

```
function pdfe.closestream ( <t:userdata> stream )
    -- no return values
end
```

```
function pdfe.readfromstream ( <t:userdata> stream )
    return
        <t:string>  str,
        <t:integer> size
end
```

```
function pdfe.readwholestream ( <t:userdata> stream, <t:boolean> decode)
    return
```

```
        <t:string>  str,
        <t:integer> size
end
```

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 `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.

13.2.7 Low level getters

In addition to the getters described before, there is also a bit lower level interface available.

```
function pdfe.getfromdictionary ( <t:userdata>, <t:integer> index )
    return
        <t:string>   key,
        <t:string>   type,
        <t:whatever> value,
        <t:whatever> detail
end
```

```
function pdfe.getfromarray ( <t:userdata>, <t:integer> index )
    return
        <t:integer>  type,
        <t:whatever> value,
        <t:integerr> detail
end
```

The `type` is one of the following:

0	none	3	integer	6	string	9	stream
1	null	4	number	7	array	10	reference
2	boolean	5	name	8	dictionary		

This list was acquired with:

```
function pdfe.getfieldtypes ( )
    return <t:table> -- types
end
```

Here `detail` is a bitset with possible bits:

0	plain	8	base16	32	utf16be
1	encoded	16	base85	64	utf16le
2	decoded				

This time we used:

```
function pdfe.getencodingvalues ( )
```

```
    return <t:table> -- values
end
```

13.2.8 Getting tables

All entries in a dictionary or table can be fetched with the following commands where the return values are a hashed or indexed table.

```
function pdfe.dictionarytotable ( <t:userdata> )
    return <t:table> -- hash
end
```

```
function pdfe.arraytotable ( <t:userdata> )
    return <t:table> -- array
end
```

You can get a list of pages with:

```
function pdfe.pagestotable(<t:pdf> pdffile)
    return {
        {
            <t:userdata>, -- dictionary
            <t:integer>,  -- size
            <t:integer>,  -- objectnumber
        },
        ...
    }
end
```

13.2.9 References

In order to access a pdf file efficiently there is lazy evaluation of references so when you run into a reference as value or array entry you have to resolve it explicitly. An unresolved references object can be resolved with:

```
function pdfe.getfromreference( <t:integer> reference ) -- NEEDS CHECKING
    return
        <t:integer>,  -- type
        <t:whatever>, -- value
        <t:whatever>  -- detail
```

So, as second value you can get back a new `pdfe` userdata object that you can query.

nodes

14 Nodes

14.1 Introduction

The (to be) typeset content is collected in a double linked list of so called nodes. A node is an array of values. When looked at from the Lua end you can either seen them as **<t:userdata>** or as **<t:integer>**. In the case of userdata you access fields like this:

```
local width = foo.width -- foo is userdata
```

while the indexed variant uses:

```
local width = nodes.direct.getwidth(foo) -- foo is an integer
```

In ConTeXt we mostly use the second variant but it's a matter of taste so users can you whatever they like most. When you print a userdata node you see something like this:

```
<node : nil <= 214984 => nil : glyph unset>
<node : nil <= 359410 => nil : hlist unknown>
<node : nil <= 343406 => nil : glue userskip>
```

The number in the middle is the one you would also see if you use the indexed approach and often these numbers are kind of large. A number 13295 doesn't mean that we have that many nodes. The engine has a large array of memory words (pairs of 32 bit integers) and a node is a slice of then with the index pointing to where we start. So, if we have a node that has 5 value pairs, the slice runs from 13295 upto 13299 that consume 40 bytes.

In this chapter we introduce the nodes that are exposed to the user. We will discuss the relevant fields as well as ways to access them. Because there are similar fields in different nodes, we can share accessors.

It is important to notice that not all fields that can be accessed (set and get) are under full user control. For instance, in math we have a `noad` type that is actually shared between several construct (like atoms, accents and fences) and not all parameters make sense for each of them. Some properties are set while the formula is assembled. It fits in the LuaMetaTeX concept to open up everything but abusing this can lead to side effects. It makes no sense to add all kind of safeguards against wrong or unintended usage because in the end only a few users will go that low level anyway.

Not all fields mentioned are accessible in the userdata variant. It is also good to notice that some fields are fabricated, for instance `total` is the sum of `height` and `depth`.

14.2 Lua node representation

As mentioned, nodes are represented in Lua as user data objects with a variable set of fields or by a numeric identifier when requested and we showed that when you print a node user data object you will see these numbers.

0	hlist	3	insert	6	boundary	9	par
1	vlist	4	mark	7	disc	10	dir
2	rule	5	adjust	8	whatsit	11	math

12	glue	18	noad	24	mathtextchar	31	alignrecord
13	kern	19	radical	25	subbox	32	attribute
14	penalty	20	fraction	26	submlist	33	gluespec
15	style	21	accent	27	delimiter	34	temp
16	choice	22	fence	28	glyph	35	split
17	parameter	23	mathchar	29	unset		

You can ask for a list of fields with `node.fields` and for valid subtypes with `node.subtypes`. There are plenty specific field values and you can some idea about them by calling `tex.get*values()` which returns a table if numbers (exclusive numbers or bits). We use these to get the tables that are shown with each node type.

There are a lot of helpers and below we show them per node type. In later sections some will come back organized by type of usage. Trivial getters and setters will not be discussed. It's good to know that some getters take more arguments where the second one can for instance trigger more return values. The number of arguments to a setter can also be more than a few. As with everything LuaMetaTEX the ConTEXt sources can also be seen as a reference.

14.3 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: `next`, `id` and `subtype`. 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`.

14.3.1 hlist and vlist, aka boxes

These lists share fields and subtypes although some subtypes can only occur in horizontal lists while others are unique for vertical lists.

fields

anchor	integer	height	dimension	shift	dimension
attr	attribute	hoffset	dimension	source	integer
axis	integer	id	integer	state	integer
depth	dimension	index	integer	subtype	integer
direction	integer	list	nodelist	target	integer
doffset	dimension	next	node	total	integer
except	nodelist	orientation	integer	width	dimension
exdepth	integer	post	nodelist	woffset	dimension
geometry	integer	postadjust	nodelist	xoffset	dimension
glueorder	integer	pre	nodelist	yoffset	dimension
glueset	integer	preadjust	nodelist		
gluesign	integer	prev	node		

subtypes

0	unknown	14	hdelimiter	28	scripts
1	line	15	vdelimiter	29	over
2	box	16	overdelimiter	30	under
3	indent	17	underdelimiter	31	accent
4	container	18	numerator	32	radical
5	alignment	19	denominator	33	fence
6	cell	20	modifier	34	rule
7	equation	21	fraction	35	ghost
8	equationnumber	22	nucleus	36	mathtext
9	math	23	sup	37	insert
10	mathchar	24	sub	38	local
11	mathpack	25	prime	39	left
12	hextensible	26	prepost	40	right
13	vextensible	27	degree	41	middle

directionvalues

0x00	lefttoright
0x01	righttoleft

listgeometryvalues

0x01	offset	0x04	anchor
0x02	orientation		

listanchorvalues

0x01	leftorigin	0x08	centerheight
0x02	leftheight	0x09	centerdepth
0x03	leftdepth	0x0A	halfwaytotal
0x04	rightorigin	0x0B	halfwayheight
0x05	rightheight	0x0C	halfwaydepth
0x06	rightdepth	0x0D	halfwayleft
0x07	centerorigin	0x0E	halfwayright

listsignvalues

0x0100	negatex
0x0200	negatey

The `shift` is a displacement perpendicular to the character (horizontal) or line (vertical) progression direction.

The `orientation`, `woffset`, `hoffset`, `doffset`, `xoffset` and `yoffset` fields are special. They can be used to make the backend rotate and shift boxes which can be handy in for instance vertical typesetting. Because they relate to (and depend on the) the backend they are not discussed here (yet). The `pre` and `post` fields refer to migrated material in both list types, while the adjusted variants only make sense in horizontal lists.

direct helpers

addxoffset addyoffset appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getanchors getattributelist getattributes getboth getbox getclass getdepth getdirection getexcept getgeometry getglue getheight getid getindex getinputfields getlist getlistdimensions getnext getnodes getnormalizedline getoffsets getorientation getpenalty getpost getpre getprev getshift getspeciallist getstate getsubtype gettotal getwhd getwidth getwordrange hasdimensions hasgeometry hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid iszeroglue kerning lastnode length ligaturing migrate mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setanchors setattributelist setattributes setboth setbox setclass setdepth setdirection setexcept setgeometry setglue setheight setindex setinputfields setlink setlist setnext setoffsets setorientation setpenalty setpost setpre setprev setshift setspeciallist setsplit setstate setsubtype setwhd setwidth showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: preadjust postadjust axis exdepth
no set: preadjust postadjust axis exdepth total

14.3.2 rule

Contrary to traditional TEX, LuaTEX has more subtypes subtypes because we also use rules to store reuseable objects and images. However, in LuaMetaTEX these are gone but we reserve these subtypes. Apart form the basic rules a lot is up to the backend.

fields

attr	attribute	left	dimension	subtype	integer
char	integer	next	node	thickness	integer
data	integer	off	integer	total	dimension
depth	dimension	on	integer	width	dimension
fam	integer	options	integer	xoffset	dimension
font	integer	prev	node	yoffset	dimension
id	integer	right	dimension		

subtypes

0	normal	4	outline	8	fraction
1	empty	5	user	9	radical
2	strut	6	over	10	box
3	virtual	7	under	11	image

ruleoptionvalues

0x01	horizontal	0x04	thickness
0x02	vertical	0x08	running

The width, height and depth of regular rules defaults to the special value of −1073741824 which indicates a running rule that adapts its dimensions to the box that it sits in.

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. The `xoffset` and `yoffset` fields can be used to shift rules. Because they relate to (and depend on the) the backend they are not discussed here (yet). Of course all this assumes that the backend deals with it. Internally fields with different names can use the same variable, depending on the subtype; dedicated names just make more sense.

direct helpers

`addmargins addxoffset addyoffset appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getchar getcharspec getdata getdepth getdiscpart getfam getheight getid getnext getnodes getoffsets getoptions getpenalty getprev getruledimensions getspeciallist getsubtype gettotal getwhd getwidth getwordrange hasdimensions hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setchar setcharspec setdata setdepth setdiscpart setfam setfont setheight setlink setnext setoffsets setoptions setpenalty setprev setruledimensions setruledimensions setspeciallist setsplit setsubtype setwhd setwidth showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `thickness`
no set: `total thickness`

14.3.3 insert

This node relates to the **`\insert`** primitive and support the fields:

fields

`attr`	attribute	`id`	integer	`prev`	node
`cost`	integer	`index`	integer	`subtype`	integer
`depth`	dimension	`list`	nodelist		
`height`	dimension	`next`	node		

Here the subtype indicates the class of the insert and that number is also used to access the box, dimen and skip registers that relate to the insert, if we use inserts in the traditional way.

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getdepth getdiscpart getheight getid getindex getlist getnext getnodes getprev getspeciallist getsubtype gettotal getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing migrate mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setdepth setdiscpart setheight setindex setlink setlist setnext setprev setspeciallist setsplit setsubtype settotal showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `-`
no set: `-`

14.3.4 mark

This one relates to the `\marks` primitive and only has a few fields, one being a token list as field which is kind of rare.

fields

attr	attribute	mark	tokenlist	prev	node
class	integer	next	node	subtype	integer
id	integer				

subtypes

0	set	1	reset

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getdata getid getindex getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setdata setindex setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: -
no set: -

14.3.5 adjust

This node results from `\vadjust` usage:

fields

attr	attribute	id	integer	prev	node
depthafter	dimension	next	node	subtype	integer

subtypes

0	pre	2	local
1	post		

adjustoptionvalues

0x00	none	0x08	depthafter
0x01	before	0x10	depthcheck
0x02	baseline	0x20	depthlast
0x04	depthbefore	0x40	except

direct helpers

`nothing (yet)`

userdata helpers

`nothing (yet)`

userdata helpers

no get:
no set:

14.3.6 disc (discretionary)

The **`\discretionary`**, **`\explicitdiscretionary`** and **`\automaticdiscretionary`** primitives as well as the discretionary that comes from hyphenation all have the pre, post and replace lists. Because these lists have head and tail pointers the getters and setters handle this for you.

fields

attr	attribute	options	integer	prev	node
class	integer	penalty	integer	replace	nodelist
id	integer	post	nodelist	subtype	integer
next	node	pre	nodelist		

subtypes

0	discretionary	3	math
1	explicit	4	regular
2	automatic		

discoptionvalues

0x00000000	normalword	0x00000040	noitaliccorrection
0x00000001	preword	0x00000080	nozeroitaliccorrection
0x00000002	postword	0x00010000	userfirst
0x00000010	preferbreak	0x40000000	userlast
0x00000020	prefernobreak		

direct helpers

appendaftertail beginofmath checkdiscretionaries checkdiscretionary collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getclass getdisc getid getnext getnodes getoptions getpenalty getpost getpre getprev getreplace getspeciallist getsubtype getwordrange hasdiscoption hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyph protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setclass setdisc setlink setnext setoptions setpenalty setpost setpre setprev setreplace setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyph unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: -
no set: -

14.3.7 math

Math nodes represent the boundaries of a math formula, normally wrapped between `$` and `$`. The glue fields are only used when the `surround` field is zero.

fields

attr	attribute	pretolerance	integer	stretchorder	integer
id	integer	prev	node	subtype	integer
next	node	shrink	dimension	surround	integer
options	integer	shrinkorder	integer	tolerance	integer
penalty	integer	stretch	dimension	width	dimension

subtypes

0	beginmath	2	beginbrokenmath
1	endmath	3	endbrokenmath

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions effectiveglue endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders `freeze getattributelist getattributes getboth getbox getglue getid getkern getnext getnodes getoptions getprev getspeciallist getsubtype getwidth getwordrange hasglyph hpack hyphenating ignoremathskip isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid iszeroglue kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setglue setkern setlink setnext setoptions setprev setspeciallist setsplit setsubtype setwidth showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `tolerance pretolerance`
no set: `tolerance pretolerance`

14.3.8 glue

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 `skip`. The structure that represents the glue components of a skip internally is called a `gluespec`. In LuaMetaTEX we don't use the spec itself but just its values.

fields

`attr`	attribute	`leader`	nodelist	`shrinkorder`	integer
`callback`	integer	`next`	node	`stretch`	dimension
`data`	integer	`options`	integer	`stretchorder`	integer
`font`	integer	`prev`	node	`subtype`	integer
`id`	integer	`shrink`	dimension	`width`	dimension

subtypes

0	userskip	12	tabskip	24	intermathskip
1	lineskip	13	spaceskip	25	ignored
2	baselineskip	14	xspaceskip	26	page
3	parskip	15	zerospaceskip	27	mathskip
4	abovedisplayskip	16	parfillleftskip	28	thinmuskip
5	belowdisplayskip	17	parfillskip	29	medmuskip
6	abovedisplayshortskip	18	parinitleftskip	30	thickmuskip
7	belowdisplayshortskip	19	parinitrightskip	31	conditionalmathskip
8	leftskip	20	indentskip	32	rulebasedmathskip
9	rightskip	21	lefthangskip	33	muglue
10	topskip	22	righthangskip	34	leaders
11	splittopskip	23	correctionskip	35	cleaders

`36`	xleaders	`37`	gleaders	`38`	uleaders

glueoptionvalues

`0x00`	normal	`0x04`	islimited
`0x01`	noautobreak	`0x08`	limit
`0x02`	hasfactor	`0x10`	uleadersline

Note that we use the key `width` in both horizontal and vertical glue. This suited the TEX internals well so we decided to stick to that naming.

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 `effectiveglue` function that takes a glue node and a parent (hlist or vlist) returns the effective width of that glue item. When you pass `true` as third argument the value will be rounded.

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions effectiveglue endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getdata getfont getglue getid getleader getnext getnodes getoptions getprev getspeciallist getsubtype getwhd getwidth getwordrange getxscale getyscale hasdimensions hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid iszeroglue kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setdata setfont setglue setleader setlink setnext setoptions setprev setspeciallist setsplit setsubtype setwhd setwidth showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `callback`
no set: `callback`

14.3.9 gluespec

Internally LuaMetaTEX (like its ancestors) also uses nodes to store data that is not seen in node lists. For instance the state of expression scanning (**`\dimexpr`** etc.) and conditionals (**`\ifcase`** etc.) is also kept in lists of nodes. A glue, which has five components, is stored in a node as well, so, where most registers store just a number, a skip register (of internal quantity) uses a pointer to a glue spec node. It has similar fields as glue nodes, which is not surprising because in the past (and other engines than LuaTEX) a glue node also has its values stored in a glue spec. This has some advantages because often the values are the same, so for instance spacing related skips were not resolved immediately but pointed to the current value of a space related internal register (like `\spaceskip`). But, in LuaTEX and therefore LuaMetaTEX we do resolve these quantities immediately and we put the current values in the glue nodes.

fields

`id`	integer	`shrinkorder`	integer	`stretchorder`	integer
`next`	node	`stretch`	dimension	`width`	dimension
`shrink`	dimension				

You will only find these nodes in a few places, for instance when you query an internal quantity. In principle we could do without them as we have interfaces that use the five numbers instead. For compatibility reasons we keep glue spec nodes exposed but this might change in the future. Of course there are no subtypes here because it's just a data store.

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getglue getid getnext getnodes getprev getspeciallist getsubtype getwidth getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid iszeroglue kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setglue setlink setnext setprev setspeciallist setsplit setsubtype setwidth showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `next`
no set: `next`

14.3.10 kern

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

fields

`attr`	attribute	`kern`	dimension	`prev`	node
`expansion`	integer	`next`	node	`subtype`	integer
`id`	integer				

subtypes

0	userkern	5	rightmarginkern	10	mathshapekern
1	accentkern	6	leftcorrectionkern	11	leftmathslackkern
2	fontkern	7	rightcorrectionkern	12	rightmathslackkern
3	italiccorrection	8	spacefontkern	13	horizontalmathkern
4	leftmarginkern	9	mathkern	14	verticalmathkern

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getexpansion getid getkern getkerndimension getnext getnodes getprev getspeciallist getsubtype getwidth getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setexpansion setkern setlink setnext setprev setspeciallist setsplit setsubtype setwidth showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: -
no set: -

14.3.11 penalty

The **`\penalty`** command is one that generates these nodes. There is not much to tell about them, apart from that in LuaMetaTEX they have options and a possible spread related `nepalty` field that is used internally.

fields

`attr`	attribute	`next`	node	`prev`	node
`id`	integer	`options`	integer	`subtype`	integer
`nepalty`	integer	`penalty`	integer		

subtypes

0	userpenalty	5	toddlerpenalty	10	beforedisplaypenalty
1	linebreakpenalty	6	singlelinepenalty	11	afterdisplaypenalty
2	linepenalty	7	finalpenalty	12	equationnumberpenalty
3	wordpenalty	8	mathprepenalty		
4	orphanpenalty	9	mathpostpenalty		

penaltyoptionvalues

0x0000	normal	0x0080	club
0x0001	mathforward	0x0100	broken
0x0002	mathbackward	0x0200	shaping
0x0004	orphaned	0x0400	double
0x0008	widowed	0x0800	doubleused
0x0010	clubbed	0x1000	factorused
0x0020	toddlered	0x2000	endofpar
0x0040	widow		

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getid getnext getnodes getoptions getpenalty getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setlink setnext setoptions setpenalty setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `nepalty`
no set: `nepalty`

14.3.12 glyph

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. Glyph nodes are relatively large and have many fields.

fields

`attr`	attribute	`index`	integer	`scale`	dimension
`char`	integer	`language`	integer	`script`	integer
`control`	integer	`left`	dimension	`slant`	integer
`data`	integer	`lhmin`	integer	`state`	integer
`depth`	dimension	`next`	node	`subtype`	integer
`discpart`	integer	`options`	integer	`total`	dimension
`expansion`	integer	`prev`	node	`weight`	integer
`font`	integer	`properties`	integer	`width`	dimension
`group`	integer	`protected`	integer	`xoffset`	dimension
`height`	dimension	`raise`	dimension	`xscale`	dimension
`hyphenate`	integer	`rhmin`	integer	`yoffset`	dimension
`id`	integer	`right`	dimension	`yscale`	dimension

subtypes

`0`	unset	`8`	relation	`16`	over
`1`	character	`9`	open	`17`	fraction
`2`	ligature	`10`	close	`18`	radical
`3`	delimiter	`11`	punctuation	`19`	middle
`4`	extensible	`12`	variable	`20`	prime
`5`	ordinary	`13`	active	`21`	accent
`6`	operator	`14`	inner		
`7`	binary	`15`	under		

glyphoptionvalues

`0x00000000`	normal	`0x00000400`	mathdiscretionary
`0x00000001`	noleftligature	`0x00000800`	mathsitalicstoo
`0x00000002`	norightligature	`0x00001000`	mathartifact
`0x00000004`	noleftkern	`0x00002000`	weightless
`0x00000008`	norightkern	`0x00004000`	spacefactoroverload
`0x00000010`	noexpansion	`0x00008000`	checktoddler
`0x00000020`	noprotrusion	`0x00010000`	checktwin
`0x00000040`	noitaliccorrection	`0x00020000`	istoddler
`0x00000080`	nozeroitaliccorrection	`0x00100000`	userfirst
`0x00000100`	applyxoffset	`0x40000000`	userlast
`0x00000200`	applyyoffset		

glyphdiscvalues

`0x01`	normal	`0x04`	mathematics
`0x02`	explicit	`0x05`	syllable
`0x03`	automatic		

discpartvalues

`0x00`	unset	`0x03`	replace
`0x01`	pre	`0x04`	always
`0x02`	post		

glyphprotectionvalues

`0x00`	unset	`0x02`	math
`0x01`	text		

The `width`, `height` and `depth` values are read-only. In LuaTEX `expansion` has been introduced as part of the separation between front- 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.

direct helpers

`addmargins addxoffset addxymargins addyoffset appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getchar getchardict getcharspec getclass getcontrol getcornerkerns getdata getdepth getexpansion getfont getglyphdata getglyphdimensions getheight getid getinputfields getlanguage getnext getnodes getoffsets getoptions getprev getscale getscales getscript getslant getspeciallist getstate getsubtype gettotal getweight getwhd getwidth getwordrange getxscale getxyscales getyscale hasdimensions hasglyph hasglyphoption hpack hyphenating isboth ischar isdirect isglyph isitalicglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyph protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setchar setchardict setcharspec setclass setcontrol setdata setexpansion setfont setglyphdata setinputfields setlanguage setlink setnext setoffsets setoptions setprev setscale setscales setscript setslant setspeciallist setsplit setstate setsubtype setweight setwhd setxyscales showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyph unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack xscaled yscaled`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `raise`
no set: `total raise`

14.3.13 boundary

This node relates to the **`\noboundary`**, **`\boundary`**, **`\protrusionboundary`**, **`\wordboundary`** etc. These are relative small nodes that determine what happens before and after them.

fields

`attr`	attribute	`id`	integer	`prev`	node
`data`	integer	`next`	node	`subtype`	integer

subtypes

0	cancel	4	page	8	par
1	user	5	math	9	adjust
2	protrusion	6	optional		
3	word	7	lua		

protrusionboundaryvalues

`0x00`	skipnone	`0x02`	skipprevious
`0x01`	skipnext	`0x03`	skipboth

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getdata getid getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setdata setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

instock inuse todirect

userdata helpers

no get: -
no set: -

14.3.14 par

This node is inserted at the start of a paragraph. You should not mess too much with this one. They are also inserted when `\local...` primitives are used that relate boxes to positions in the line and overload certain parameters that play a role in the line break routine. There are many fields!

fields

adjacentdemerits	node	leftbox	node
adjdemerits	integer	leftboxwidth	dimension
adjustspacing	integer	leftskip	integer
adjustspacingshrink	integer	lefttwindemerits	integer
adjustspacingstep	integer	linebreakchecks	integer
adjustspacingstretch	integer	linepenalty	integer
attr	attribute	lineskip	glue
baselineskip	glue	lineskiplimit	dimension
brokenpenalties	node	looseness	integer
brokenpenalty	integer	middlebox	node
clubpenalties	node	next	node
clubpenalty	integer	orphanpenalties	node
dir	integer	parfillleftskip	glue
displaywidowpenalties	node	parfillrightskip	glue
displaywidowpenalty	integer	parindent	dimension
doublehyphendemerits	integer	parinitleftskip	glue
emergencyextrastretch	dimension	parinitrightskip	glue
emergencyleftskip	glue	parpasses	node
emergencyrightskip	glue	parshape	node
emergencystretch	dimension	pretolerance	integer
endpartokens	tokenlist	prev	node
exhyphenpenalty	integer	prevgraf	integer
finalhyphendemerits	integer	protrudechars	integer
fitnessclasses	node	rightbox	node
hangafter	integer	rightboxwidth	dimension
hangindent	dimension	rightskip	integer
hsize	dimension	righttwindemerits	integer
hyphenationmode	integer	shapingpenaltiesmode	integer
hyphenpenalty	integer	shapingpenalty	integer
id	integer	singlelinepenalty	integer
interlinepenalties	node	state	integer
interlinepenalty	integer	subtype	integer
lastlinefit	integer	toddlerpenalties	node

tolerance	integer
widowpenalties	node
widowpenalty	integer

subtypes

0	vmodepar	2	hmodepar	4	localbreak
1	localbox	3	parameter	5	math

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getdirection getid getnext getnodes getparstate getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes patchparshape prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setdirection setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont validpar verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: dir parpasses linebreakchecks state prevgraf hsize leftskip rightskip hangindent hangafter parindent parfillleftskip parfillrightskip adjustspacing protrudechars emergencystretch looseness lastlinefit linepenalty clubpenalty widowpenalty displaywidowpenalty toddlerpenalties adjdemerits doublehyphendemerits finalhyphendemerits parshape interlinepenalties clubpenalties widowpenalties displaywidowpenalties brokenpenalties orphanpenalties singlelinepenalty baselineskip lineskip lineskiplimit adjustspacingstep adjustspacingshrink adjustspacingstretch endpartokens hyphenationmode shapingpenaltiesmode shapingpenalty parinitleftskip parinitrightskip emergencyleftskip emergencyrightskip emergencyextrastretch fitnessclasses adjacentdemerits hyphenpenalty exhyphenpenalty lefttwindemerits righttwindemerits
no set: dir parpasses linebreakchecks state prevgraf hsize leftskip rightskip hangindent hangafter parindent parfillleftskip parfillrightskip adjustspacing protrudechars emergencystretch looseness lastlinefit linepenalty clubpenalty widowpenalty displaywidowpenalty toddlerpenalties adjdemerits doublehyphendemerits finalhyphendemerits parshape interlinepenalties clubpenalties widowpenalties displaywidowpenalties brokenpenalties orphanpenalties singlelinepenalty baselineskip lineskip lineskiplimit

`adjustspacingstep adjustspacingshrink adjustspacingstretch endpartokens hyphenationmode shapingpenaltiesmode shapingpenalty parinitleftskip parinitrightskip emergencyleftskip emergencyrightskip emergencyextrastretch fitnessclasses adjacentdemerits hyphenpenalty exhyphenpenalty lefttwindemerits righttwindemerits`

14.3.15 dir

Direction nodes mark parts of the running text that need a change of direction and the **`\textdirection`** command generates them. Contrary to LuaTEX we only have two directions.

fields

`attr`	attribute	`level`	integer	`prev`	node
`dir`	integer	`next`	node	`subtype`	integer
`id`	integer				

subtypes

`0`	normal	`1`	cancel

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getdirection getid getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setdirection setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `dir`
no set: `dir`

14.3.16 whatsit

A whatsit node is a real simple one and it only has a subtype. It is even less than a user node (which it actually could be) and uses hardly any memory. What you do with it it entirely up to you: it's is real minimalistic. You can assign a subtype and it has attributes. It is all up to the user (and the backend) how they are handled.

fields

attr	attribute	prev	node
id	integer	subtype	integer
next	node		

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getid getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: -
no set: -

14.3.17 attribute

This is a small node but used a lot. When an attribute is set and travels with a node, we actually have a forward (only) linked list with a head node that keeps a reference count. These lists are (to be) sorted by attribute index. Normally you will *not* mess directly with these list because you can get unwanted side effects.

fields

count	integer	index	integer	subtype	integer
data	integer	next	node	value	integer
id	integer				

subtypes

`0`	list	`1`	value

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getdata getid getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setdata setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `index value`
no set: `data index value`

14.3.18 alignrecord

This node can be encountered in alignments and will eventually become a `hlist` or `vlist` node. It therefore has the same size and fields as those nodes. However, the following fields are overloaded by other parameters: `woffset`, `hoffset`, `doffset`, `xoffset`, `yoffset`, `orientation`, `pre` and `post`. Be careful!

fields

`id`	integer	`prev`	node	`subtype`	integer
`list`	node	`size`	dimension	`width`	dimension
`next`	node				

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getid getnext getnodes getprev getspeciallist getsubtype getwidth getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar` isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setlink setnext setprev setspeciallist setsplit setsubtype setwidth showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: list width size
no set: list width size

14.3.19 unset

This node can be encountered in alignments and will eventually become a hlist or vlist node. It therefore has the same size and fields as those nodes. However, the following fields are (at least temporarily) there and they use the slots of woffset, hoffset, doffset and orientation. Be careful!

fields

attr	attribute	next	node	span	integer
count	integer	prev	node	stretch	dimension
id	integer	shrink	dimension	subtype	integer

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getdepth getexcept getglue getheight getid getinputfields getlist getnext getnodes getprev getspeciallist getsubtype gettotal getwhd getwidth getwordrange hasdimensions hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid iszeroglue kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setdepth setexcept setglue setheight setinputfields setlink setlist setnext setprev setspeciallist setsplit setsubtype setwhd setwidth showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `attr span`
no set: `attr span`

14.4 Math nodes

14.4.1 The concept

Many object fields in math mode are either simple characters in a specific family or math lists or node lists: `mathchar`, `mathtextchar`, subbox and `submlist` and `delimiter`. These are endpoints and therefore the `next` and `prev` fields of these these subnodes are unused.

There is a subset of nodes dedicated to math called noads. These are used for simple atoms, fractions, fences, accents and radicals. When you enter a formula, TEX creates a node list with regular (math) nodes and noads. Then it hands over the list the math processing engine. The result of that is a nodelist without noads. Most of the noads contain subnodes so that the list of possible fields is actually quite small. Math formulas are both a linked list and a tree. For instance in $e = mc^2$ there is a linked list `e = m c` but the `c` has a superscript branch that itself can be a list with branches.

Eventually I might give a more detailed description of the differences between the five noad variants but for now the following has to do. One will quite likely not set that many fields at the Lua end but running over the many sub lists can make sense. One has to know what the engine is doing anyway.

14.4.2 noad

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, vcenter. These all have the same fields, and they are combined into a single node type with separate subtypes for differentiation. However, before reading on you should realize that LuaMetaTEX has an extended math engine. We hale not only more classes, we also have many more keys in the nodes. We won't cover these details here.

fields

`analyzed`	integer	`mainclass`	integer	`scriptstate`	integer
`attr`	attribute	`next`	node	`source`	integer
`depth`	integer	`nucleus`	nodelist	`style`	integer
`extraattr`	attribute	`options`	integer	`sub`	nodelist
`fam`	integer	`prev`	node	`subpre`	nodelist
`height`	integer	`prime`	nodelist	`subshift`	integer
`hlist`	nodelist	`primeshift`	integer	`subtype`	integer
`id`	integer	`rightclass`	integer	`sup`	nodelist
`italic`	integer	`rightslack`	integer	`suppre`	nodelist
`leftclass`	integer	`scriptkern`	integer	`supshift`	integer
`leftslack`	integer	`scriptorder`	integer	`width`	integer

subtypes

0	ordinary	7	variable	14	middle
1	operator	8	active	15	prime
2	binary	9	inner	16	accent
3	relation	10	under	17	fenced
4	open	11	over	18	ghost
5	close	12	fraction	19	vcenter
6	punctuation	13	radical		

noadoptionvalues

0x00000001	axis	0x40000000	followedbyspace
0x00000002	noaxis	0x80000000	proportional
0x00000004	exact	0x100000000	sourceonnucleus
0x00000008	left	0x200000000	fixedsuperorsubscript
0x00000010	middle	0x400000000	fixedsuperandsubscript
0x00000020	right	0x800000000	autobase
0x00000040	adapttoleftsize	0x1000000000	stretch
0x00000080	adapttorightsize	0x2000000000	shrink
0x00000100	nosubscript	0x4000000000	center
0x00000200	nosuperscript	0x8000000000	scale
0x00000400	nosubprescript	0x10000000000	keepbase
0x00000800	nosuperprescript	0x20000000000	single
0x00001000	noscript	0x40000000000	norule
0x00002000	nooverflow	0x80000000000	automiddle
0x00004000	void	0x100000000000	reflected
0x00008000	phantom	0x200000000000	continuation
0x00010000	openupheight	0x400000000000	inheritclass
0x00020000	openupdepth	0x800000000000	discardshapekern
0x00040000	limits	0x1000000000000	realignscripts
0x00080000	nolimits	0x2000000000000	ignoreemptysubscript
0x00100000	preferfontthickness	0x4000000000000	ignoreemptysuperscript
0x00200000	noruling	0x8000000000000	ignoreemptyprimescript
0x00400000	indexedsubscript	0x10000000000000	continuationhead
0x00800000	indexedsuperscript	0x20000000000000	continuationkernel
0x01000000	indexedsubprescript	0x40000000000000	reorderprescripts
0x02000000	indexedsuperprescript	0x80000000000000	ignore
0x04000000	unpacklist	0x100000000000000	nomorescripts
0x08000000	nocheck	0x200000000000000	carryoverclasses
0x10000000	auto	0x400000000000000	usecallback
0x20000000	unrolllist		

In addition to the subtypes (related to classes) that the engines knows of, there can be user defined subtypes. Not all fields make sense for every derives noad: `accent`, `fence`, `fraction` or `radical` but there we (currently) only mention the additional ones. These additional fields are taken from a pool of extra fields. Not all fields are always accessible for these nodes.

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getanchors getattributelist getattributes getboth getbox getcharspec getclass getfam getid getnext getnodes getnucleus getoptions getprev getprime getscripts getspeciallist getsub getsubpre getsubtype getsup getsuppre getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setanchors setattributelist setattributes setboth setbox setcharspec setclass setfam setlink setnext setnucleus setoptions setprev setprime setscripts setspeciallist setsplit setsub setsubpre setsubtype setsup setsuppre showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: hlist italic width height depth style scriptstate analyzed mainclass leftclass rightclass leftslack rightslack subshift supshift primeshift scriptkern extraattr
no set: -

14.4.3 mathchar

The `mathchar` is the simplest subnode field, it contains the character and family for a single glyph object. The family eventually resolves on a reference to a font. Internally this nodes is one of the math kernel nodes.

fields

attr	attribute	id	integer	prev	node
char	integer	index	integer	properties	integer
fam	integer	next	node	subtype	integer
group	integer	options	integer		

kerneloptionvalues

0x01	noitaliccorrection	0x10	fulldiscretionary
0x02	noleftpairkern	0x20	ignoredcharacter
0x04	norightpairkern	0x40	islargeoperator
0x08	autodiscretionary	0x80	hasitalicshape

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getchar getchardict getcharspec getfam getfont getid getnext getnodes getoptions getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setchar setchardict setcharspec setfam setlink setnext setoptions setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: -
no set: -

14.4.4 mathtextchar

The `mathtextchar` 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). Internally this nodes is one of the math kernel nodes.

fields

attr	attribute	id	integer	prev	node
char	integer	index	integer	properties	integer
fam	integer	next	node	subtype	integer
group	integer	options	integer		

kerneloptionvalues

0x01	noitaliccorrection	0x10	fulldiscretionary
0x02	noleftpairkern	0x20	ignoredcharacter
0x04	norightpairkern	0x40	islargeoperator
0x08	autodiscretionary	0x80	hasitalicshape

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getchar getchardict getcharspec getfam getfont getid getnext getnodes getoptions getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setchar setchardict setcharspec setfam setlink setnext setoptions setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: -
no set: -

14.4.5 subbox

These `subbox` subnode is used for subsidiary list items where the `list` points to a 'normal' vbox or hbox.

fields

`attr`	attribute	`list`	nodelist	`prev`	node
`id`	integer	`next`	node	`subtype`	integer

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getid getlist getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setlink setlist setnext setprev setspeciallist setsplit setsubtype showlist `size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: -
no set: -

14.4.6 submlist

In `submlist` subnode the `list` points to a math list that is yet to be converted. Their fields

fields

`attr`	attribute	`list`	nodelist	`prev`	node
`id`	integer	`next`	node	`subtype`	integer

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getid getlist getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setlink setlist setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: -
no set: -

14.4.7 delimiter

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

fields

attr	attribute	largefamily	integer	smallchar	integer
id	integer	next	node	smallfamily	integer
index	integer	prev	node	subtype	integer

The fields largechar and largefamily can be zero, in that case the font that is set for the smallfamily is expected to provide the large version as an extension to the smallchar.

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getchar getchardict getcharspec getclass getfont getid getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setchar setchardict setcharspec setclass setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: index
no set: index

14.4.8 accent

Accent nodes deal with stuff on top or below a math constructs.

fields

attr	attribute	id	integer	subtype	integer
bottomaccent	nodelist	next	node	topaccent	nodelist
bottomovershoot	nodelist	overlayaccent	nodelist	topovershoot	nodelist
fraction	nodelist	prev	node		

subtypes

`0`	bothflexible	`2`	fixedbottom
`1`	fixedtop	`3`	fixedboth

For more fields see noad. At some point we might move fields from that list to here but only when the engine also gets that split.

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getanchors getattributelist getattributes getboth getbottom getbox getclass getdelimiter getfam getid getnext getnodes getnucleus getoptions getprev getprime getscripts getspeciallist getsub getsubpre getsubtype getsup getsuppre gettop getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setanchors setattributelist setattributes setboth setbottom setbox setclass setdelimiter setfam setlink setnext setnucleus setoptions setprev setprime setscripts setspeciallist setsplit setsub setsubpre setsubtype setsup setsuppre settop showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `topovershoot bottomovershoot`
no set: `topovershoot bottomovershoot`

14.4.9 style

These nodes are signals to switch to another math style. Currently the subtype is actually used to store the style but don't rely on that for the future.

fields

`attr`	attribute	`prev`	node	`style`	integer
`id`	integer	`scale`	integer	`subtype`	integer
`next`	node				

subtypes

mathstylenamevalues

`0x00`	display	`0x04`	script
`0x01`	crampeddisplay	`0x05`	crampedscript
`0x02`	text	`0x06`	scriptscript
`0x03`	crampedtext	`0x07`	crampedscriptscript

mathstylevalues

`0x00`	display	`0x04`	script
`0x01`	crampeddisplay	`0x05`	crampedscript
`0x02`	text	`0x06`	scriptscript
`0x03`	crampedtext	`0x07`	crampedscriptscript

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getid getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `scale`
no set: `-`

14.4.10 parameter

These nodes are used to (locally) set math parameters. The subtype reflects a math style.

fields

id	integer	prev	node	subtype	integer
name	integer	style	integer	value	integer
next	node				

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getid getnext getnodes getprev getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: -
no set: -

14.4.11 choice

Most of the fields of this node are lists. Depending on the subtype different field names are used.

fields

attr	attribute	post	nodelist	scriptscript	nodelist
class	integer	pre	nodelist	subtype	integer
display	nodelist	prev	node	text	nodelist
id	integer	replace	nodelist		
next	node	script	nodelist		

subtypes

0	normal	1	discretionary

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getattributelist getattributes getboth getbox getchoice getdisc getid getnext getnodes getpost getpre getprev getreplace getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setattributelist setattributes setboth setbox setchoice setdisc setlink setnext setprev setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: class
no set: class pre post replace

14.4.12 radical

Radical nodes are the most complex as they deal with scripts as well as constructed large symbols. 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.

fields

attr	attribute	id	integer	size	integer
bottom	nodelist	left	nodelist	subtype	integer
degree	nodelist	next	node	top	nodelist
depth	dimension	prev	node		
height	dimension	right	nodelist		

subtypes

0	normal	4	underdelimiter	8	delimited
1	radical	5	overdelimiter	9	hextensible
2	root	6	delimiterunder		
3	rooted	7	delimiterover		

For more fields see noad. At some point we might move fields from that list to here but only when the engine also gets that split.

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getanchors getattributelist getattributes getboth getbottomdelimiter getbox getclass getdegree getdelimiter getfam getid getleftdelimiter getnext getnodes getnucleus getoptions getprev getprime getrightdelimiter getscripts getspeciallist getsub getsubpre getsubtype getsup getsuppre gettopdelimiter getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setanchors setattributelist setattributes setboth setbottomdelimiter setbox setclass setdegree setdelimiter setfam setleftdelimiter setlink setnext setnucleus setoptions setprev setprime setrightdelimiter setscripts setspeciallist setsplit setsub setsubpre setsubtype setsup setsuppre settopdelimiter showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: size height depth
no set: size height depth

14.4.13 fraction

Fraction nodes are also used for delimited cases, hence the `left` and `right` fields among.

fields

attr	attribute	middle	nodelist	rule	dimension
denominator	nodelist	next	node	subtype	integer
hfactor	integer	numerator	nodelist	vfactor	integer
id	integer	prev	node		
left	nodelist	right	nodelist		

subtypes

0	over	3	skewed
1	atop	4	stretched
2	above		

For more fields see noad. At some point we might move fields from that list to here but only when the *engine also gets that split.*

direct helpers

`appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getanchors getattributelist getattributes getboth getbox getclass getdelimiter getdenominator getfam getid getleftdelimiter getnext getnodes getnumerator getoptions getprev getrightdelimiter getspeciallist getsubtype getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setanchors setattributelist setattributes setboth setbox setclass setdelimiter setdenominator setfam setleftdelimiter setlink setnext setnumerator setoptions setprev setrightdelimiter setspeciallist setsplit setsubtype showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack`

userdata helpers

`instock inuse todirect`

userdata helpers

no get: `rule hfactor vfactor`
no set: `rule hfactor vfactor`

14.4.14 fence

Fence nodes come in pairs but either one can be a dummy (this period driven empty fence). Some of these fields are used by the renderer and might get adapted in the process.

fields

`attr`	attribute	`id`	integer	`subtype`	integer
`bottom`	integer	`nestingfactor`	integer	`top`	integer
`bottomovershoot`	dimension	`next`	node	`topovershoot`	dimension
`delimiter`	nodelist	`prev`	node	`variant`	integer

subtypes

`0`	unset	`2`	middle	`4`	operator
`1`	left	`3`	right	`5`	no

For more fields see noad*. At some point we might move fields from that list to here but only when the engine also gets that split.*

direct helpers

appendaftertail beginofmath checkdiscretionaries collapsing copyonly count dimensions endofmath exchange findattribute findattributerange findnode firstchar firstglyph firstglyphnode firstitalicglyph flattendiscretionaries flattenleaders freeze getanchors getattributelist getattributes getboth getbottom getbottomdelimiter getbox getclass getdelimiter getdepth getfam getheight getid getnext getnodes getprev getspeciallist getsubtype gettop gettopdelimiter gettotal getwordrange hasglyph hpack hyphenating isboth ischar isdirect isglyph isloop isnext isnextchar isnextglyph isprev isprevchar isprevglyph issimilarglyph isspeciallist isvalid kerning lastnode length ligaturing mlisttohlist naturalhsize naturalwidth newcontinuationatom newmathglyph newtextglyph patchattributes prependbeforehead protectglyphs protectglyphsnone protrusionskippable rangedimensions removefromlist repack reverse setanchors setattributelist setattributes setboth setbottom setbottomdelimiter setbox setclass setdelimiter setdepth setfam setheight setlink setnext setprev setspeciallist setsplit setsubtype settop settopdelimiter showlist size slide softenhyphens startofpar tonode tovaliddirect traversechar traversecontent traverseglyph traverseitalic traverseleader traverselist unprotectglyphs unprotectglyphsnone unsetattributes usesfont verticalbreak vpack

userdata helpers

instock inuse todirect

userdata helpers

no get: `nestingfactor topovershoot bottomovershoot`
no set: `nestingfactor topovershoot bottomovershoot`

14.5 Helpers

14.5.1 Introduction

The userdata node variant has accessors on that object but when we use the indexed variant we use functions. As a consequence there are more helpers for direct nodes that for userdata nodes and many of them accept more arguments or have multiple return values. When you use ConTEXt you will notice that instead of the `node.direct` namespace we use `nuts`. Among the reasons is that we had an intermediate variant in ConTEXt MkIV before we had these direct nodes. That variant was more efficient than the userdata accessors and triggered the introduction of direct nodes after which we dropped the intermediate variant. So, for ConTEXt users direct nodes are nuts.

14.5.2 Housekeeping

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

```
function node.types ( )
    return <t:table> -- identifiers
end
```

This shows the names of the nodes and their internal numbers. Not all nodes are visible unless one goes really deep down into lists. The next two convert a name to its internal numeric representation and vise versa. The numbers don't relate to importance or some ordering; they just appear in the order that is handy for the engine. Commands like this are rather optimized so performance should be ok but you can of course always store the id in a Lua number.

```
function node.id ( <t:string> name )
    return <t:integer> -- identifier
end
```

```
function node.type ( <t:integer> identifier )
    return <t:string> -- name
end
```

This function returns an indexed table with valid field names for a particular type of node. Some fields (like `total`) can be constructed from other fields.

```
function node.fields ( <t:integer> identifier | <t:string> name )
    return <t:table> -- fields
end
```

The `hasfield` function returns a boolean that is only true if `n` is actually a node, and it has the field.This function probably is not that useful but some nodes don't have a `subtype`, `attr` or `prev` field and this is a way to test for that.

```
function node.direct.hasfield ( <t:direct> n | <t:string> name )
    return <t:boolean> -- okay
end
```

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 pass a second argument the subtype will be set too.

```
function node.direct.new (
    <t:number> id | <t:string> name
)
    return <t:direct.> -- node
end
```

```
function node.direct.new (
    <t:number> id | <t:string> name,
    <t:number> | <t:string> subtype
)
    return <t:direct.> -- node
end
```

As already has been mentioned, you are responsible for making sure that nodes created this way are used only once, and are freed when you don't pass them back somehow.

The next one frees 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. Fields that point to nodes or lists are flushed too. So, when you used their content for something else you need to set them to nil first.

```
function node.direct.free ( <t:direct> n )
    return <t:direct> -- next
end
```

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

```
function node.direct.flush ( <t:direct> n )
    -- no return values
end
```

A list starting with node `n` can be flushed from TEX's memory too. 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.

```
function node.direct.flushlist ( <t:direct> n )
    -- no return values
end
```

When you free for instance a discretionary node, `flushlist` is applied to the `pre`, `post`, `replace` so you don't need to do that yourself. Assigning them `nil` won't free those lists!

This 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.

```
function node.direct.copy ( <t:direct> n )
    return <t:direct> -- copy
end
```

A deep copy of the node list that starts at `n` can be created too. If `m` is also given, the copy stops just before node `m`.

```
function node.direct.copylist ( <t:direct> n )
    return <t:direct> -- copy
end
```

```
function node.direct.copylist ( <t:direct> n, <t:direct> m )
    return <t:direct> -- copy
end
```

Note that you cannot copy attribute lists this way. However, there is normally no need to copy attribute lists because when you do assignments to the `attr` field or make changes to specific attributes, the needed copying and freeing takes place automatically. When you change a value of an attribute *in* a list, it will affect all the nodes that share that list.

```
function node.direct.write ( <t:direct> n )
    -- no return values
end
```

This function will append a node list to TEX's 'current list'. The node list is not deep-copied! There is no error checking either! You might need to enforce horizontal mode in order for this to work as expected.

14.5.3 Manipulating lists

Unless there is a bug or a callback messes up a node list is dual linked. In original TEX nodes had to be small so nodes only had a next pointer. If you run into an issue you can use the next helper to sure that the node list is double linked.

```
function node.direct.slide ( <t:direct> n)
    return <t:direct> -- tail
end
```

In most cases TEX itself only uses `next` pointers but your other callbacks might expect proper `prev` pointers too. So, when you run into issues or are in doubt, apply the slide function before you return the list. You can also get the tail without sliding:

```
function node.direct.tail ( <t:direct> n )
    return <t:direct> -- tail
end
```

For tracing purposes we have a few counters. The first one 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.

```
function node.direct.length (
    <t:direct> n
)
    return <t:integer>
end

function node.direct.length (
    <t:direct> n,
    <t:direct> m
)
    return <t:integer>
end
```

The second one 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.

```
function node.direct.count (
    <t:integer> id,
    <t:direct>  n
)
    return <t:integer>
end

function node.direct.count (
    <t:integer> id,
    <t:direct>  n,
    <t:direct>  m
)
    return <t:integer>
```

```
end
```

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. When the third argument is passed, the node is freed.

```
function node.direct.remove ( <t:direct> head, <t:direct> current )
    return
        <t:direct> head,
        <t:direct> current,
        <t:direct> removed
end

function node.direct.remove ( <t:direct> head, <t:direct> current, <t:boolean> free)
    return
        <t:direct> -- head,
        <t:direct> -- current
end
```

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.

```
function node.direct.insertbefore (
    <t:direct> head,
    <t:direct> current,
    <t:direct> new
)
    return
        <t:direct>, -- head
        <t:direct>  -- new
end
```

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.

```
function node.direct.insertafter (
    <t:direct> head,
    <t:direct> current,
    <t:direct> new
)
    return
        <t:direct>, -- head
        <t:direct>  -- new
end
```

You can also mess with the list by changing the next or prev fields, using:

```
function node.direct.setprev ( <t:direct> n, <t:direct> prv                ) end
function node.direct.setnext ( <t:direct> n, <t:direct> nxt                ) end
function node.direct.setboth ( <t:direct> n, <t:direct> prv, <t:direct> nxt ) end
```

The next function pops the last node from TEX's 'current list'. It returns that node, or `nil` if the current list is empty.

```
function node.direct.lastnode ( )
    return <t:direct> n
end
```

This helper returns the location of the first match at or after node `n`:

```
function node.direct.findnode ( <t:direct> n, <t:integer> subtype )
    return <t:direct> -- n
end

function node.direct.findnode ( <t:direct> n )
    return
        <t:direct>, -- n
        <t:integer> -- subtype
end
```

14.5.4 Traversing

The easiest do-it-yourself approach to run over a list of nodes is to use one of the following functions:

```
function node.direct.getnext ( <t:direct> n )
    return <t:direct> | <t:nil>
end

function node.direct.getprev ( <t:direct> n )
    return <t:direct> | <t:nil>
end

function node.direct.getboth ( <t:direct> n )
    return
        <t:direct> | <t:nil>, -- prev
        <t:direct> | <t:nil>  -- next
end
```

Instead of using these you can use one of the iterators that loops over the node list that starts at `n`.

```
function node.direct.traverse ( <t:direct> n )
    return
        <t:direct>  t,
        <t:integer> id,
        <t:integer> subtype
end
```

Typically code looks like this:

```
for n in node.traverse(head) do
    -- whatever
end
```

which 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
        -- whatever
    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.

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`, kind of like:

```
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
```

This and the previously discussed `traverse` are the only traverses provided for userdata nodes.

```
function node.direct.traverseid ( <t:integer> id, <t:direct> n )
```

```
    return
        <t:direct>  t,
        <t:integer> subtype
end
```

The `traversechar` iterator loops over the `glyph` nodes in a list. Only nodes with a subtype less than 256 are seen.

NEEDS CHECKING: protected check

```
function node.direct.traversechar ( <t:direct> n )
    return
        <t:direct>,  -- n
        <t:integer>, -- char
        <t:integer>  -- font
end
```

The `traverseglyph` iterator loops over a list and returns the list and filters all glyphs:

```
function node.direct.traverseglyph ( <t:direct> n )
    return
        <t:direct>,  -- n
        <t:integer>, -- char
        <t:integer>  -- font
end
```

This iterator loops over the `hlist` and `vlist` nodes in a list. The four return values can save some time compared to fetching these fields but in practice you seldom need them all.

```
function node.direct.traverselist ( <t:direct> n )
    return
        <t:direct>,  -- n
        <t:integer>, -- identifier
        <t:integer>, -- subtype
        <t:direct>   -- list
end
```

This iterator loops over nodes that have content: `hlist`, `vlist`, `glue` with leaders, `glyph`, `disc` and `rule` nodes.

```
function node.direct.traversecontent ( <t:direct> n )
    return
        <t:direct>,  -- n
        <t:integer>, -- identifier
        <t:integer>, -- subtype
        <t:direct>   -- listorleader
end
```

The traversers also support backward traversal. An optional extra boolean triggers this. Yet another optional boolean will automatically start at the end of the given list. So, if we want both we use:

```
function node.direct.traverse (
```

```
    <t:direct>  n,
    <t:boolean> reverse,
    <t:boolean> startatend
)
    return
        <t:direct>  t,
        <t:integer> id,
        <t:integer> subtype
end
```

14.5.5 Glyphs

Glyphs have a lot of parameters and there are many setters and getters that can access them. Some generic ones, like `getwidth` are discussed in other subsections, some are more specific to glyphs:

```
function node.direct.getslant  ( <t:direct> g ) return <t:integer> end
function node.direct.getweight ( <t:direct> g ) return <t:integer> end
```

and

```
function node.direct.setslant  ( <t:direct> g, <t:integer> slant  ) end
function node.direct.setweight ( <t:direct> g, <t:integer> weight ) end
```

14.5.6 Glue

You can set the five properties of a glue in one go. If a non-numeric value is passed the property becomes zero.

```
function node.direct.setglue ( <t:direct> n )
    -- no return values
end

function node.direct.setglue (
    <t:direct>  n,
    <t:integer> width,
    <t:integer> stretch,
    <t:integer> shrink,
    <t:integer> stretchorder,
    <t:integer> shrinkorder
)
    -- no return values
end
```

When you pass values, only arguments that are numbers are assigned so the next call will only adapt the width and shrink.

```
node.direct.setglue(n,655360,false,65536)
```

When a list node is passed, you set the glue, order and sign instead. The next call will return five values or nothing when no glue is passed.

```
function node.direct.getglue ( <t:direct> n )
    return
        <t:integer>, -- width
        <t:integer>, -- stretch
        <t:integer>, -- shrink
        <t:integer>, -- stretchorder
        <t:integer>  -- shrinkorder
```

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.

This function returns `true` when the width, stretch and shrink properties are all zero.

```
function node.direct.iszeroglue ( <t:direct> n )
    return <t:boolean> -- allzero
end
```

Glue is not only, well, glue. The to be filled space can also be occupied by a rule, boxes, glyphs and what more. You can get the list that makes this with:

```
function node.direct.getleader ( <t:direct> n )
    return <t:direct> -- list
end
```

and set the list with

```
function node.direct.setleader ( <t:direct> n, <t:direct> l | <t:nil> )
    -- no return values
end
```

14.5.7 Attributes

Assignments to attributes registers result in assigning lists with set attributes to nodes and the implementation is non-trivial because the value that is attached to a node is essentially a (sorted) 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.

An `attribute` comes in two variants, indicated by subtype. Because attributes are stored in a sorted linked list, and because they are shared, the first node is a list reference node and the following ones are value nodes. So, most attribute nodes are value nodes. These are forward linked lists. Because there are assumptions to how these list are build you should rely on the helpers, also because details might change.

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

```
function node.direct.currentattr()
    return <t:direct> -- list
end
```

The intended usage of `currentattr` is as follows (we use the userdata interface here):

```
local x1 = node.new("glyph")
```

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

or:

```
local x1 = node.new("glyph")
local x2 = node.new("glyph")
local ca = node.currentattr()
x1.attr = ca
x2.attr = ca
```

The attribute lists are reference counted and the assignment takes care of incrementing the count. 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.

```
<number> v = node.hasattribute ( <node> n, <number> id )
<number> v = node.hasattribute ( <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`.

```
function node.direct.getattribute ( <t:direct> n, <t:integer> id )
    return <t:integer> -- value
end
```

The previous function 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.

```
function node.direct.findattribute ( <t:direct> n, <t:integer> id )
    return
        <t:integer>, -- value
        <t:direct>   -- node
end
```

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.

```
function node.direct.setattribute ( <t:direct> n, <t:integer> id, <t:integer> value )
    -- no return values
end
```

Sets the attribute with number `id` to the value `value`. Duplicate assignments are ignored.

```
function node.direct.unsetattribute ( <t:direct> n, <t:integer> id )
    return <t:integer> -- value
end
```

```
function node.direct.unsetattribute ( <t:direct> n, <t:integer> id, <t:integer> value
  )
    return <t:integer> -- value
end
```

Unsets the attribute with number `id`. If `value` is also supplied, it will only perform this operation if the value matches `value`. Missing attributes or attribute-value pairs are ignored. If the attribute was actually deleted, the function returns its old value, otherwise it returns `nil`.

14.5.8 Glyph handling

Processing a character stream into a visual representation using glyphs is one of the important processes in the engine. In TEX82 this happens in two places. When the text is read ligaturing and kerning takes place and the list can, if needed, be packed into a box because the dimensions are now known. When that list is to become a paragraph it might be that lines get split and when a word can be hyphenated the ligaturing and kerning is reverted, the word gets hyphenated, ligatures and kerns get reapplied and the process goes on.

In OpenType processing characters is way more complex. Even if we delegate this to a library, the fact that we have a mix of text and whatever, potential hyphenation as well as spaces turned glue, means that we need to do some juggling with nodes. For that reason hyphenation (of the whole list), ligaturing and kerning has been split into clearly separates stages. One can still apply the original TEX variants but in practice it is Lua that does the juggling of nodes in more complex situations. And we're not only talking of font processing. For instance, additional inter-character kerning can be done in Lua too.

This all means that we have quite a repertoire of helpers that deal with glyph processing efficiently.

We can locate 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. The `char` and `glyph` variants check for the protected field being (yet) unset or (already) set.

```
function node.direct.firstglyphnode ( <t:direct> n )
    return <t:direct> -- n
end

function node.direct.firstglyphnode ( <t:direct> n, <t:direct> m )
    return <t:direct> -- n
end
```

The next functions can be used to determine if processing is needed. We distinguish between a character (unprocessed) and a glyph (processed or unprocessed). When we check for a glyph there are three possible outcomes:

```
function node.direct.isglyph ( <t:direct> n )
    return
        <t:nil>,
        <t:nil>
end

function node.direct.isglyph ( <t:direct> n )
    return
        <t:false>,
        <t:integer> -- identifier
end
```

```
function node.direct.isglyph ( <t:direct> n )
    return
        <t:integer>, -- character
        <t:integer>  -- font
end
```

Checking for a processed character is more complicated. If the glyph has been processed and the protected property has been set, we get this:

```
function node.direct.ischar ( <t:direct> n )
    return <t:false>
end
```

If that's not the case additional arguments are checked. If we don't pass a valid integer, the character value is returned:

```
function node.direct.ischar ( <t:direct> n, <t:integer> font )
    return <t:integer> -- character
end
```

btu when we passed a font identifier indeed we check if that one matches the one in the glyph and if not again we get:

```
function node.direct.ischar ( <t:direct> n, <t:integer> font )
    return <t:false> --
end
```

From there on we check for more arguments to match the glyph fields:

```
function node.direct.ischar (
    <t:direct>  n,
    <t:integer> font,
    <t:integer> data
)
    return <t:false> | <t:integer> -- character

end

function node.direct.ischar (
    <t:direct>  n,
    <t:integer> font,
    <t:integer> data,
    <t:integer> state
)
    return <t:false> | <t:integer> -- character

end

function node.direct.ischar (
    <t:direct>  n,
    <t:integer> font,
    <t:integer> scale,
    <t:integer> xscale,
```

```
    <t:integer> yscale,
)
    return <t:false> | <t:integer> -- character

end

function node.direct.ischar (
    <t:direct>  n,
    <t:integer> font,
    <t:integer> data,
    <t:integer> scale,
    <t:integer> xscale,
    <t:integer> yscale,
)
    return <t:false> | <t:integer> -- character

end
```

There are reasons for these combined tests and they can be found in the ConTEXt font handler. A related helper is one that compares the `font`, `data`, `scale`, `xscale`, `yscale`, `slant` and `weight`.

```
function node.direct.issimilarglyph ( <t:direct>  one, <t:direct> two )
    return <t:boolean> -- similar
end
```

This function returns the first glyph or disc node in the given list:

```
function node.direct.hasglyph ( <t:direct> n )
    return <t:direct> -- n
end
```

Traditional TEX ligature processing can be achieved with the next helper. This assumes that the ligature information is present in the font. In ConTEXt we call this base mode processing.

```
function node.direct.ligaturing ( <t:direct> first )
    return
        <t:direct>, -- head
        <t:direct>, -- tail
        <t:boolean> -- success
end

function node.direct.ligaturing ( <t:direct> first, <t:direct> last )
    return
        <t:direct>, -- head
        <t:direct>, -- tail
        <t:boolean> -- success
end
```

Traditional TEX font kern processing can be achieved with the next helper. This assumes that the kern information is present in the font. In ConTEXt we call this base mode processing.

```
function node.direct.kerning ( <t:direct> first )
    return
```

```
        <t:direct>, -- head
        <t:direct>, -- tail
        <t:boolean> -- success
end
```

```
function node.direct.kerning ( <t:direct> first, <t:direct> last )
    return
        <t:direct>, -- head
        <t:direct>, -- tail
        <t:boolean> -- success
end
```

When processing is done, you can mark the glyph nodes as protected in order to prevent redundant processing, for instance because boxed material gets unboxed. Where in LuaTEX the subtype gets changed by adding or subtracting 256, in LuaMetaTEX we have a dedicated (small) protection field.

```
function node.direct.protectglyph ( <t:direct> n )
    -- no return values
end
```

```
function node.direct.protectglyphs ( <t:direct> first, <t:direct> last )
    -- no return values
end
```

The opposite action can also be done.

```
function node.direct.unprotectglyph ( <t:direct> n )
    -- no return values
end
```

```
function node.direct.unprotectglyphs ( <t:direct> first, <t:direct> last )
    -- no return values
end
```

The next function checks if protrusion is active at a line boundary, in which case the glyph node can be skipped. It's not that useful in the end.

```
function node.direct.protrusionskipable ( <t:direct> n )
    return <t:boolean> -- skippable
end
```

Once we're done we can freeze leaders: apply the glue to the leader and freeze the boxes or whatever is at hand.

```
function node.direct.flattenleaders ( <t:direct> n )
    return
        <t:direct>, -- head
        <t:integer> -- count
end
```

14.5.9 Discretionaries

Discretionaries and glyphs are the carriers of text. Where the core of `glyph` nodes are the `font` and `char` fields, in `disc` nodes we have to focus on the `pre`, `post` and `replace` fields. These point to linked lists that are a mix of glyph, kerns and (in LuaMetaTEX fixed width) glue. here are the accessors:[22]

```
function node.direct.getpost ( <t:direct> d, <t:boolean> tailtoo )
    return
        <t:direct>, -- head
        <t:direct>  -- tail
end

function node.direct.getpre ( <t:direct> d, <t:boolean> tailtoo )
    return
        <t:direct>, -- head
        <t:direct>  -- tail
end

function node.direct.getreplace ( <t:direct> d, <t:boolean> tailtoo )
    return
        <t:direct>, -- head
        <t:direct>  -- tail
end

function node.direct.getdisc ( <t:direct> d, <t:boolean> tailtoo )
    return
        <t:direct>, -- prehead
        <t:direct>, -- posthead
        <t:direct>, -- replacehead
        <t:direct>, -- pretail
        <t:direct>, -- posttail
        <t:direct>  -- replacetail
end
```

We also have setters:

```
function node.direct.setpost    ( <t:direct> d, <t:direct> | <t:nil> ) end
function node.direct.setpre     ( <t:direct> d, <t:direct> | <t:nil> ) end
function node.direct.setreplace ( <t:direct> d, <t:direct> | <t:nil> ) end
```

A major update can be done with this one:

```
function node.direct.setdisc (
    <t:direct>,            -- discretionary
    <t:direct>  | <t:nil>, -- pre
    <t:direct>  | <t:nil>, -- post
    <t:direct>  | <t:nil>, -- replace
    <t:subtype> | <t:nil>, -- subtype
    <t:subtype> | <t:nil>  -- penalty
)
```

[22] These are a bit more generic because they also return fields from `choice` nodes and possibly `hlist` and `vlist` nodes.

```
    -- no return values
end
```

From this you can deduce that we can also say:

```
function node.direct.getpenalty ( <t:direct> d )
    return <t:integer> -- penalty
end

function node.direct.setpenalty ( <t:direct> d, <t:integer> penalty )
    -- no return value
end
```

The next pair targets glyphs and normally you will not use the setter, because the engine takes care of setting that state.

```
function node.direct.getdiscpart ( <t:direct> g )
    return <t:integer> -- bitset
end

function node.direct.setdiscpart ( <t:direct> g, <t:integer> bitset )
    -- no return value
end
```

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 routine kicks in. You can call this function to check the list for issues with disc nodes.

```
function node.direct.checkdiscretionary ( <t:direct> n )
    -- no return values
end
```

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).

```
function node.direct.checkdiscretionaries ( <t:direct> head )
    -- no return values
end
```

This function will remove the discretionaries in the list and inject the replace field when set.

```
function node.direct.flattendiscretionaries ( <t:direct> n )
    return
        <t:direct>, -- head
        <t:integer> -- count
end
```

14.5.10 Packaging and dimensions

At some point a node list has to be packed in either a horizontal or vertical box. There are restrictions to what can get packed, for instance you cannot have glyphs in a vertical list.

The hpack 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.

```
function node.direct.hpack (
    <t:direct> list
)
    return
        <t:direct>, -- box
        <t:integer> -- badness
end

function node.direct.hpack (
    <t:direct>  list,
    <t:integer> width,
    <t:string>  info -- "additional" | "exactly"
)
    return
        <t:direct>, -- box
        <t:integer> -- badness
end

function node.direct.hpack (
    <t:direct>  list,
    <t:integer> width,
    <t:string>  info, -- "additional" | "exactly"
    <t:integer> direction
)
    return
        <t:direct>, -- box
        <t:integer> -- badness
end
```

The vpack 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.

```
function node.direct.vpack (
    <t:direct> list
)
    return
        <t:direct>, -- box
        <t:integer> -- badness
end

function node.direct.vpack (
    <t:direct>  list,
    <t:integer> height,
```

```
    <t:string>  info -- "additional" | "exactly"
)
    return
        <t:direct>, -- box
        <t:integer> -- badness
end
```

```
function node.direct.vpack (
    <t:direct>  list,
    <t:integer> height,
    <t:string>  info, -- "additional" | "exactly"
    <t:integer> direction
)
    return
        <t:direct>, -- box
        <t:integer> -- badness
end
```

This function calculates the natural in-line dimensions of the node list starting at node `first` and terminating just before node `last` (or the end of the list, if there is no second argument). The return values are scaled points.

```
function node.direct.dimensions (
    <t:direct> first,
    <t:direct> last
)
    return
        <t:integer>, -- width
        <t:integer>, -- height
        <t:integer>  -- depth
end
```

This alternative 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].glueset,
    tex.box[0].gluesign,
    tex.box[0].glueorder,
    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`.

```
function node.direct.dimensions (
    <t:number>  glueset,
```

```
    <t:integer> gluesign
    <t:integer> glueorder,
    <t:direct>  first,
    <t:direct>  last
)
    return
        <t:integer>, -- width
        <t:integer>, -- height
        <t:integer>  -- depth
end
```

This alternative saves a few lookups and can be more convenient in some cases:

```
function node.direct.rangedimensions (
    <t:direct> parent,
    <t:direct> first,
    <t:direct> last
)
    return
        <t:integer>, -- width
        <t:integer>, -- height
        <t:integer>  -- depth
end
```

If you only need the width, a simple and somewhat more efficient variant is this, where again `last` is optional:

```
function node.direct.naturalwidth (
    <t:direct> first,
    <t:direct> last
)
    return <t:integer> -- width
end
```

More low level are the following helpers. They accept various kind of nodes `hlist`, `vlist`, `unset`, `rule`, `glyph` or `glue` (because these can have a leader).

```
function node.direct.getwhd ( <t:direct> n )
    return
        <t:dimension>, -- width
        <t:dimension>, -- height
        <t:dimension>  -- depth
end
```

In case of as glyph you can also get the expansion:

```
function node.direct.getwhd ( <t:direct> n, <t:true> expansion )
    return
        <t:dimension>, -- width
        <t:dimension>, -- height
        <t:dimension>, -- depth
        <t:integer>    -- expansion
```

```
end
```

The getwidth accepts even more node types: hlist, vlist, unset, align, rule, glue, gluespec, glyph, kern and math (surround).

```
function node.direct.getwidth ( <t:direct> n )
    return <t:dimension> -- width
end
```

And for glyphs:

```
function node.direct.getwidth ( <t:direct> n, <t:true> expansion )
    return
        <t:dimension>,-- width
        <t:dimension> -- expansion
end
```

The getter for height operates on hlist, vlist, unset, rule, insert and fence.

```
function node.direct.getheight ( <t:direct> n )
    return <t:dimension> -- height
end
```

For the depth we have a different repertoire: hlist, vlist, unset, rule, insert, glyph and fence.

```
function node.direct.getdepth ( <t:direct> n )
    return <t:dimension> -- depth
end
```

For hlist, vlist, unset, rule, insert_node:, glyph and fence we can get the total of height and depth:

```
function node.direct.gettotal ( <t:direct> n )
    return <t:dimension> -- height + depth
end
```

Only hlist and vlist have a (vertical or horizontal) shift:

```
function node.direct.getshift ( <t:direct> n )
    return <t:dimension> -- shift
end
```

This one is only valid for glyph and kern nodes:

```
function node.direct.getexpansion ( <t:direct> n )
    return <t:dimension> -- expansion
end
```

Before we move on we mention the setters:

```
function node.direct.setwidth     ( <t:direct> n, <t:dimension> width     ) end
function node.direct.setheight    ( <t:direct> n, <t:dimension> height    ) end
function node.direct.setdepth     ( <t:direct> n, <t:dimension> depth     ) end
function node.direct.setshift     ( <t:direct> n, <t:dimension> shift     ) end
```

```
function node.direct.setexpansion ( <t:direct> n, <t:integer>   expansion ) end
```

The combined one ignores values that are no number, so passing (e.g.) `nil` or (nicer) `false` will retain the value.

```
function nodedirect.setwhd (
    <t:direct>    node,
    <t:dimension> width,
    <t:dimension> height,
    <t:dimension> depth,
    -- no return values
end
```

These `hlist` and `vlist` nodes (but others as well have) a field called `list`:

```
function node.direct.getlist ( <t:direct> b )
    return <t:direct> -- list
end
```

```
function node.direct.setlist ( <t:direct> b, <t:direct> list )
    -- nothing to return
end
```

When a list is packages, glue is resolved and the list node gets its glue properties set so that the backend can apply the stretch and shrink to the glue amount. There might be situations where you want to do this explicitly, which is why we provide:

```
function node.direct.freeze ( <t:direct> b )
    -- nothing to return
end
```

In LuaMetaTEX we can handle nested marks, inserts and adjusts, and pre and post material can get bound to a box. We can use these to access them:

```
function node.direct.getpost ( <t:direct> b, <t:boolean> tailtoo )
    return
        <t:direct>, -- head
        <t:direct>  -- tail
end
```

```
function node.direct.getpre ( <t:direct> b, <t:boolean> tailtoo )
    return
        <t:direct>, -- head
        <t:direct>  -- tail
end
```

and these to set them, although they are unlikely candidates for that.

```
function node.direct.setpost ( <t:direct> b, <t:direct> | <t:nil> ) end
function node.direct.setpre  ( <t:direct> b, <t:direct> | <t:nil> ) end
```

14.5.11 Math

We start with the function that runs the internal ‘mlist to hlist’ conversion that turns a the yet unprocessed math list into a horizontal list. The interface is the same as for the callback callback mlisttohlist.

```
function node.direct.mlisttohlist (
    <t:direct>  list,
    <t:string>  displaytype,
    <t:boolean> penalties
)
   <t:direct> -- result
end
```

When you have a horizontal list with math you can locate the relevant portion with:

```
function node.direct.beginofmath ( <t:direct> n ) return <t:direct> end
function node.direct.endofmath   ( <t:direct> n ) return <t:direct> end
```

You can for instance use these helpers to skip over math in case you're processing text.

The math noads have a nucleus and scripts. In LuaMetaTEX we have the usual super- and subscript but also prescripts and a primescript, so five scripts in total so naturally we have getters for these:

```
function node.direct.getnucleus ( <t:direct> n ) return <t:direct> | <t:nil> end
function node.direct.getprime   ( <t:direct> n ) return <t:direct> | <t:nil> end
function node.direct.getsup     ( <t:direct> n ) return <t:direct> | <t:nil> end
function node.direct.getsub     ( <t:direct> n ) return <t:direct> | <t:nil> end
function node.direct.getsuppre  ( <t:direct> n ) return <t:direct> | <t:nil> end
function node.direct.getsubpre  ( <t:direct> n ) return <t:direct> | <t:nil> end
```

plus:

```
function node.direct.getscripts ( <t:direct> n )
    return
        <t:direct>, -- primescript
        <t:direct>, -- superscript
        <t:direct>, -- subscript
        <t:direct>, -- superprescript
        <t:direct>  -- subprescript
end
```

These are complemented by setters. When the second argument is not passes (or nil) the field is reset.

```
function node.direct.setnucleus ( <t:direct> n, <t:direct> nucleus        ) end
function node.direct.setprime   ( <t:direct> n, <t:direct> primescript    ) end
function node.direct.setsup     ( <t:direct> n, <t:direct> superscript    ) end
function node.direct.setsub     ( <t:direct> n, <t:direct> subscript      ) end
function node.direct.setsuppre  ( <t:direct> n, <t:direct> superprescript ) end
function node.direct.setsubpre  ( <t:direct> n, <t:direct> subprescript   ) end
```

And of course:

```
function node.direct.getscripts (
    <t:direct> primescript,
    <t:direct> superscript,
    <t:direct> subscript,
    <t:direct> superprescript,
    <t:direct> subprescript
)
    -- no return values
end
```

In the discretionaries subsection we mention accessing pre, post and replace fields. These functions can also be used for choice nodes. Discussing this is currently beyond this manual.

14.5.12 SyncTEX

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).

```
function node.direct.setsynctexfields ( <t:integer> fileid, <t:integer> line )
    -- no return values
end
```

```
function node.direct.getsynctexfields ( <t:direct> n )
    return
        <t:integer>, -- fileid
        <t:integer>  -- line
end
```

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) and there are different incompatible versions floating around. Even more important to notice is that the engine doesn't do anything with this so support is upto Lua.

14.5.13 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. You go from user data nodes and there numeric references and back with:

```
function node.todirect ( <t:node>   n) return <t:direct> end
function node.tonode   ( <t:direct> d) return <t:node>   end
```

The user data model is rather robust as it is a virtual interface with some additional checking while the more direct access which uses the node numbers directly. However, even with user data you can get into troubles when you free nodes that are no longer allocated or mess up lists. If you apply tostring to a node you see its internal (direct) number and id.

The userdata model provides key based access while the direct model always accesses fields via functions:

```
local c = nodeobject.char
local c = 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.

In the direct namespace there are more 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.

In previous sections we only show the functions in the `node.direct` namespace. The following functions are available in both `node` and `node.direct`:

`checkdiscretionaries`	`effectiveglue`	`free`
`checkdiscretionary`	`endofmath`	`getattribute`
`copy`	`fields`	`getcachestate`
`copylist`	`findattribute`	`getfield`
`count`	`flattendiscretionaries`	`getfielderror`
`currentattributes`	`flushlist`	`getglue`
`dimensions`	`flushnode`	`getpropertiestable`

getproperty
gluetostring
hasattribute
hasfield
hasglyph
hpack
hyphenating
id
insertafter
insertbefore
isnode
iszeroglue
kerning
lastnode
length
ligaturing
makeextensible
mlisttohlist
new
protectglyph
protectglyphs
protrusionskippable
rangedimensions
remove
serialized
setattribute
setfield
setfielderror
setglue
setproperty
show
size
slide
subtypes
tail
todirect
tonode
tostring
traverse
traverseid
type
types
unprotectglyph
unprotectglyphs
unsetattribute
usedlist
usesfont
vpack
write

In ConTEXt these are duplicated in `nodes.nuts` so that is the reference. Quite some functions gets mapped onto the `nodes` namespace. In addition we emulate some userdata functions and add some of our own. We show them here because this manual takes ConTEXt as reference.

node.direct	node	nodes
addmargins		
addxoffset		
addxymargins		
addyoffset		
appendaftertail		
beginofmath		
checkdiscretionaries	⋆	
checkdiscretionary	⋆	
collapsing		
copy	⋆	⋆
copylist	⋆	⋆
copyonly		
count	⋆	
currentattributes	⋆	⋆
dimensions	⋆	
effectiveglue	⋆	
endofmath	⋆	
exchange		
fields	⋆	⋆
findattribute	⋆	
findattributerange		
findnode		
firstchar		
firstglyph		
firstglyphnode		
firstitalicglyph		
flattendiscretionaries	⋆	
flattenleaders		
flushlist	⋆	⋆
flushnode	⋆	⋆
free	⋆	
freeze		
getanchors		
getattribute	⋆	⋆
getattributelist		
getattributes		
getboth		⋆
getbottom		
getbottomdelimiter		
getbox		⋆
getcachestate	⋆	
getchar		⋆
getchardict		
getcharspec		
getchoice		
getclass		
getcontrol		
getcornerkerns		
getdata		
getdegree		
getdelimiter		
getdenominator		
getdepth		
getdirection		

getdisc		
getdiscpart		
getexcept		
getexpansion		
getfam		
getfield	⋆	⋆
getfielderror	⋆	
getfont		⋆
getgeometry		
getglue	⋆	
getglyphdata		
getglyphdimensions		
getheight		
getid		⋆
getindex		
getinputfields		
getkern		
getkerndimension		
getlanguage		
getleader		⋆
getleftdelimiter		
getlist		⋆
getlistdimensions		
getnext		⋆
getnodes		
getnormalizedline		
getnucleus		
getnumerator		
getoffsets		
getoptions		
getorientation		
getparstate		
getpenalty		
getpost		
getpre		
getprev		⋆
getprime		
getpropertiestable	⋆	
getproperty	⋆	⋆
getreplace		
getrightdelimiter		
getruledimensions		
getscale		
getscales		
getscript		
getscripts		
getshift		
getslant		
getspeciallist		
getstate		
getsub		
getsubpre		
getsubtype		⋆
getsup		
getsuppre		
gettop		
gettopdelimiter		
gettotal		
getusage		
getusedattributes		
getweight		
getwhd		
getwidth		
getwordrange		
getxscale		
getxyscales		
getyscale		
gluetostring	⋆	
hasattribute	⋆	⋆
hasdimensions		
hasdiscoption		
hasfield	⋆	⋆
hasgeometry		
hasglyph	⋆	
hasglyphoption		
hasusage		
hpack	⋆	⋆
hyphenating	⋆	
id	⋆	
ignoremathskip		
insertafter	⋆	⋆
insertbefore	⋆	⋆
isboth		
ischar		
isdirect		⋆
isglyph		
isitalicglyph		
isloop		
isnext		
isnextchar		
isnextglyph		
isnode	⋆	⋆
isprev		
isprevchar		
isprevglyph		
issimilarglyph		
isspeciallist		
isvalid		

iszeroglue	⋆	
kerning	⋆	
lastnode	⋆	
length	⋆	
ligaturing	⋆	
makeextensible	⋆	
migrate		
mlisttohlist	⋆	
naturalhsize		
naturalwidth		
new	⋆	⋆
newcontinuationatom		
newmathglyph		
newtextglyph		
patchattributes		
patchparshape		
prependbeforehead		
protectglyph	⋆	
protectglyphs	⋆	
protectglyphsnone		
protrusionskippable	⋆	
rangedimensions	⋆	
remove	⋆	⋆
removefromlist		
repack		
reverse		
serialized	⋆	⋆
setanchors		
setattribute	⋆	⋆
setattributelist		
setattributes		
setboth		⋆
setbottom		
setbottomdelimiter		
setbox		⋆
setchar		⋆
setchardict		
setchoice		
setclass		
setcontrol		
setdata		
setdegree		
setdelimiter		
setdenominator		
setdepth		
setdirection		
setdisc		
setdiscpart		
setexcept		
setexpansion		
setfam		
setfield	⋆	⋆
setfielderror	⋆	
setfont		⋆
setgeometry		
setglue	⋆	
setglyphdata		
setheight		
setindex		
setinputfields		
setkern		
setlanguage		
setleader		⋆
setleftdelimiter		
setlink		⋆
setlist		⋆
setnext		⋆
setnucleus		
setnumerator		
setoffsets		
setoptions		
setorientation		
setpenalty		
setpost		
setpre		
setprev		⋆
setprime		
setproperty	⋆	⋆
setreplace		
setrightdelimiter		
setruledimensions		
setscale		
setscales		
setscript		
setscripts		
setshift		
setslant		
setspeciallist		
setsplit		
setstate		
setsub		
setsubpre		
setsubtype		
setsup		
setsuppre		
settop		
settopdelimiter		
settotal		

setweight		
setwhd		
setwidth		
show	⋆	
size	⋆	
slide	⋆	
softenhyphens		
startofpar		⋆
subtypes	⋆	⋆
tail	⋆	⋆
todirect	⋆	
tonode	⋆	⋆
tostring	⋆	⋆
tovaliddirect		
traverse	⋆	⋆
traversechar		
traversecontent		
traverseglyph		
traverseid	⋆	⋆
traverseitalic		
traverseleader		
traverselist		
type	⋆	
types	⋆	
unprotectglyph	⋆	
unprotectglyphs	⋆	
unsetattribute	⋆	⋆
unsetattributes		
usedlist	⋆	⋆
usesfont	⋆	
verticalbreak		
vpack	⋆	⋆
write	⋆	⋆
xscaled		
yscaled		

The following functions are in the ConTEXt `nodes` namespace but don't come from the library. Again, we show them here because ConTEXt is the reference.

nodes	nodes.nuts	node
aligned		
append	⋆	
apply	⋆	
applyvisuals	⋆	
astable		
basepoints		
concat	⋆	
copy_node		
countall	⋆	
delete	⋆	
firstdirinbox		
flush	⋆	
fullhpack	⋆	
getattr	⋆	
idsandsubtypes		
idstostring		
insertlistafter	⋆	
installattributehandler		
is_display_math	⋆	
isnut	⋆	
leftmarginwidth		
link	⋆	
linked	⋆	
list		
listtoutf		
locate	⋆	
maxboxwidth		
nopts		
packlist		
points		
prepend	⋆	
print		
pts		
rehpack	⋆	
repackhlist	⋆	
replace	⋆	
report		
rightmarginwidth		
serialize		
serializebox		
setattr	⋆	
setattrlist	⋆	
setboxtonaturalwd		
showboxes		
showlist		
showsimplelist		
somepenalty	⋆	
somespace	⋆	
splitbox	⋆	
stripdiscretionaries		
takeattr	⋆	
takebox	⋆	

tobasepoints		topoints	
tocentimeters		toscaledpoints	
tociceros		tosequence	⋆
todidots		totable	
todimen		totree	
toinches		toutf	
tomillimeters		upcomingproperties	
tonodes	⋆	vianodes	⋆
tonut	⋆	vianuts	⋆
topicas		visualizebox	

We have quite some helpers and some accept different node types. Here is the repertoire:

14.5.14 Properties

Attributes are a convenient way to relate extra information to a node. You can assign them at the TEX end as well as at the Lua end and consult them at the Lua end. One big advantage is that they obey grouping. They are linked lists and normally checking for them is pretty efficient, even if you use a lot of them. A macro package has to provide some way to manage these attributes at the TEX end because otherwise clashes in their usage can occur.

Each node also can have a properties table and you can assign values to this table using the `setproperty` function and get properties using the `getproperty` function. Managing properties is way more demanding than managing attributes.

Take the following example:

```
\directlua {
    local n = node.new("glyph")

    node.setproperty(n,"foo")
    print(node.getproperty(n))

    node.setproperty(n,"bar")
    print(node.getproperty(n))

    node.free(n)
}
```

This will print `foo` and `bar` which in itself is not that useful when multiple mechanisms want to use this feature. A variant is:

```
\directlua {
    local n = node.new("glyph")

    node.setproperty(n,{ one = "foo", two = "bar" })
    print(node.getproperty(n).one)
    print(node.getproperty(n).two)

    node.free(n)
}
```

This time we store two properties with the node. It really makes sense to have a table as property because that way we can store more. But in order for that to work well you need to do it this way:

```
\directlua {
    local n = node.new("glyph")

    local t = node.getproperty(n)

    if not t then
        t = { }
        node.setproperty(n,t)
    end
    t.one = "foo"
    t.two = "bar"

    print(node.getproperty(n).one)
    print(node.getproperty(n).two)

    node.free(n)
}
```

Here our own properties will not overwrite other users properties unless of course they use the same keys. So, eventually you will end up with something:

```
\directlua {
    local n = node.new("glyph")

    local t = node.getproperty(n)

    if not t then
        t = { }
        node.setproperty(n,t)
    end
    t.myself = { one = "foo", two = "bar" }

    print(node.getproperty(n).myself.one)
    print(node.getproperty(n).myself.two)

    node.free(n)
}
```

This assumes that only you use `myself` as subtable. The possibilities are endless but care is needed. For instance, the generic font handler that ships with ConTEXt uses the `injections` subtable and you should not mess with that one!

There are a few helper functions that you normally should not touch as user: `getpropertiestable` and will give the table that stores properties (using direct entries) and you can best not mess too much with that one either because LuaMetaTEX itself will make sure that entries related to nodes will get wiped when nodes get freed, so that the Lua garbage collector can do its job. In fact, the main reason why we have this mechanism is that it saves the user (or macro package) some work. One can easily write a property mechanism in Lua where after a shipout properties gets cleaned up but it's not entirely trivial to make sure that with each freed node also its properties get freed, due to the fact that there can be nodes left over for a next page. And having a callback bound to the node deallocator would add way to much overhead.

When we copy a node list that has a table as property, there are several possibilities: we do the same as a new node, we copy the entry to the table in properties (a reference), we do a deep copy of a table in the properties, we create a new table and give it the original one as a metatable. After some experiments (that also included timing) with these scenarios we decided that a deep copy made no sense, nor did nilling. In the end both the shallow copy and the metatable variant were both ok, although the second one is slower. The most important aspect to keep in mind is that references to other nodes in properties no longer can be valid for that copy. We could use two tables (one unique and one shared) or metatables but that only complicates matters.

When defining a new node, we could already allocate a table but it is rather easy to do that at the lua end e.g. using a metatable `__index` method. That way it is under macro package control. When deleting a node, we could keep the slot (e.g. setting it to false) but it could make memory consumption raise unneeded when we have temporary large node lists and after that only small lists. Both are not done because in the end this is what happens now: when a node is copied, and it has a table as property, the new node will share that table. The copy gets its own table with the original table as metatable.

A few more experiments were done. For instance: copy attributes to the properties so that we have fast access at the Lua end. In the end the overhead is not compensated by speed and convenience, in fact, attributes are not that slow when it comes to accessing them. So this was rejected.

Another experiment concerned a bitset in the node but again the gain compared to attributes could be neglected and given the small amount of available bits it also demands a pretty strong agreement over what bit represents what, and this is unlikely to succeed in the TEX community. It doesn't pay off.

Just in case one wonders why properties make sense: it is not so much speed that we gain, but more convenience: storing all kinds of (temporary) data in attributes is no fun and this mechanism makes sure that properties are cleaned up when a node is freed. Also, the advantage of a more or less global properties table is that we stay at the Lua end. An alternative is to store a reference in the node itself but that is complicated by the fact that the register has some limitations (no numeric keys) and we also don't want to mess with it too much.

tokens

15 Tokens

15.1 Introduction

If a TEX programmer talks tokens (and nodes) the average user can safely ignore it. Often it is enough to now that your input is tokenized which means that one or more characters in the input got converted into some efficient internal representation that then travels through the system and triggers actions. When you see an error message with TEX code, the reverse happened: tokens were converted back into commands that resemble the (often expanded) input.

There are not that many examples here because the functions discusses here are often not used directly but instead integrated in a bit more convenient interfaces. However, in due time more examples might show up here.

15.2 Lua token representation

A token is an 32 bit integer that encodes a command and a value, index, reference or whatever goes with a command. The input is converted into a token and the body of macros are stored as linked list of tokens. In the later case we combine a token and a next pointer in what is called a memory word. If we see tokens in Lua we don't get the integer but a userdata object that comes with accessors.

Unless you're into very low level programming the likelihood of encountering tokens is low. But related to tokens is scanning so that is what we cover here in more detail.

15.3 Helpers

15.3.1 Basics

References to macros are stored in a table along with some extra properties but in the end they travel around as tokens. The same is true for characters, they are also encoded in a token. We have three ways to create a token:

```
function token.create ( <t:integer> value )
    return <t:token> -- userdata
end

function token.create ( <t:integer> value, <t:integer> command)
    return <t:token> -- userdata
end

function token.create ( <t:string> csname )
    return <t:token> -- userdata
end
```

An example of the first variant is `token.create(65)`. When we print (inspect) this in ConTEXt we get:

```
<lua token : 476151 == letter 65>={
 ["category"]="letter",
```

```
 ["character"]="A",
 ["id"]=476151,
}
```

If we say `token.create(65,12)` instead we get:

```
<lua token : 476151 == other_char 65>={
 ["category"]="other",
 ["character"]="A",
 ["id"]=476151,
}
```

An example of the third call is `token.create("relax")`. This time get:

```
<lua token : 580111 == relax : relax 0>={
 ["active"]=false,
 ["cmdname"]="relax",
 ["command"]=16,
 ["csname"]="relax",
 ["expandable"]=false,
 ["frozen"]=false,
 ["id"]=580111,
 ["immutable"]=false,
 ["index"]=0,
 ["instance"]=false,
 ["mutable"]=false,
 ["noaligned"]=false,
 ["permanent"]=false,
 ["primitive"]=true,
 ["protected"]=false,
 ["tolerant"]=false,
}
```

Another example is `token.create("dimen")`:

```
<lua token : 467905 == dimen : register 3>={
 ["active"]=false,
 ["cmdname"]="register",
 ["command"]=121,
 ["csname"]="dimen",
 ["expandable"]=false,
 ["frozen"]=false,
 ["id"]=467905,
 ["immutable"]=false,
 ["index"]=3,
 ["instance"]=false,
 ["mutable"]=false,
 ["noaligned"]=false,
 ["permanent"]=false,
 ["primitive"]=true,
 ["protected"]=false,
```

```
 ["tolerant"]=false,
}
```

The most important properties are `command` and `index` because the combination determines what it does. The macros (here primitives) have a lot of extra properties. These are discusses in the low level manuals.

You can check if something is a token with the next function; when a token is passed the return value is the string literal `token`.

```
function token.type ( <t:whatever> )
    return <t:string> "token" | <t:nil>
end
```

A maybe more natural test is:

```
function token.istoken ( <t:whatever> )
    return <t:boolean> -- success
end
```

Internally we can see variables like `cmd`, `chr`, `tok` and such, where the later is a combination of the first two. The `create` variant that take two integers relate to this. Of course you need to know what the magic numbers are. Passing weird numbers can give side effects so don't expect too much help with that. 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:

```
\protected\def\MyMacro#1{\dimen 0 = \numexpr #1 + 10 \relax}
```

A macro like this is actually a little program:

```
467922   19   49  match                argument 1
580083   20    0  end match
--------------
467931  121    3  register             dimen
580013   12   48  other char           0 (U+00030)
582314   10   32  spacer
582312   12   61  other char           = (U+0003D)
580193   10   32  spacer
582783   81   75  some item            numexpr
582310   21    1  parameter reference
190952   10   32  spacer
582785   12   43  other char           + (U+0002B)
476151   10   32  spacer
580190   12   49  other char           1 (U+00031)
582265   12   48  other char           0 (U+00030)
467939   10   32  spacer
580045   16    0  relax                relax
```

The first column shows indices in token memory where we have a token combined with a next pointer. So, in slot `467931` we have both a token and a pointer to slot `580013`.

There is another way to create a token.

```
function token.new ( <t:string> command, <t:integer> value )
    return <t:token>
end

function token.new ( <t:integer> value, <t:integer> command )
    return <t:token>
end
```

Watch the order of arguments. We not have four ways to create a token

```
<lua token : 580087 == letter 65>={
 ["category"]="letter",
 ["character"]="A",
 ["id"]=580087,
}
```

namely:

```
token.new("letter",65)
token.new(65,11)
token.create(65,11)
token.create(65)
```

You can test if a control sequence is defined with:

```
function token.isdefined ( <t:string> t )
    return <t:boolean> -- success
end
```

The engine was never meant to be this open which means that in various places the assumption is that tokens are valid. However, it is possible to create tokens that make little sense in some context and can even make the system crash. When possible we catch this but checking everywhere would bloat the code and harm performance. Compare this to changing a few bytes in a binary that at some point create can havoc.

15.3.2 Getters

The userdata objects have a virtual interface that permits access by fieldname. Instead you can use one of the getters.

```
function token.getcommand ( <t:token> t ) return <t:integer> end
function token.getindex   ( <t:token> t ) return <t:integer> end
function token.getcmdname ( <t:token> t ) return <t:string>  end
function token.getcsname  ( <t:token> t ) return <t:string>  end
function token.getid      ( <t:token> t ) return <t:integer> end
function token.getactive  ( <t:token> t ) return <t:boolean> end
```

If you want to know what the possible values are, you can use:

```
function token.getrange (
    <t:token> | <t:integer>
)
```

```
return
    <t:integer>, -- first
    <t:integer>  -- last
end
```

We can also ask for the macro properties but instead you can just fetch the bit set that describes them.

```
function token.getexpandable ( <t:token> t ) return <t:boolean> end
function token.getprotected  ( <t:token> t ) return <t:boolean> end
function token.getfrozen     ( <t:token> t ) return <t:boolean> end
function token.gettolerant   ( <t:token> t ) return <t:boolean> end
function token.getnoaligned  ( <t:token> t ) return <t:boolean> end
function token.getprimitive  ( <t:token> t ) return <t:boolean> end
function token.getpermanent  ( <t:token> t ) return <t:boolean> end
function token.getimmutable  ( <t:token> t ) return <t:boolean> end
function token.getinstance   ( <t:token> t ) return <t:boolean> end
function token.getconstant   ( <t:token> t ) return <t:boolean> end
```

The bit set can be fetched with:

```
function token.getflags ( <t:token> t )
    return <t:integer> -- bit set
end
```

The possible flags are:

0x000001	frozen	0x000080	untraced	0x004000	conditional
0x000002	permanent	0x000100	global	0x008000	value
0x000004	immutable	0x000200	tolerant	0x010000	semiprotected
0x000008	primitive	0x000400	protected	0x020000	inherited
0x000010	mutable	0x000800	overloaded	0x040000	constant
0x000020	noaligned	0x001000	aliased	0x080000	deferred
0x000040	instance	0x002000	immediate		

The number of parameters of a macro can be queried with:

```
function token.getparameters ( <t:token> t )
    return <t:integer>
end
```

The three properties that are used to identify a token can be fetched with:

```
function token.getcmdchrcs ( <t:token> t )
    return
        <t:integer>, -- command (cmd)
        <t:integer>, -- value   (chr)
        <t:integer>  -- index   (cs)
end
```

A simpler call is:

```
function token.getcstoken ( <t:string> csname )
```

```
    return <t:integer> -- token number
end
```

A table with relevant properties of a token (or control sequence) can be fetched with:

```
function token.getfields ( <t:token> token )
    return <t:table> -- fields
end

function token.getfields ( <t:string> csname )
    return <t:table> -- fields
end
```

15.3.3 Setters

The `setmacro` function can be called with a different amount of arguments, where the prefix list comes last. Examples of prefixes are `global` and `protected`.

```
function token.setmacro (
    <t:string> csname
)

function token.setmacro (
    <t:integer> catcodetable,
    <t:string>  csname
)
    -- no return values
end

function token.setmacro (
    <t:string> csname,
    <t:string> content
)
    -- no return values
end

function token.setmacro (
    <t:integer> catcodetable,
    <t:string>  csname,
    <t:string>  content
)
    -- no return values
end

function token.setmacro (
    <t:string> csname,
    <t:string> content,
    <t:string> prefix
 -- there can be more prefixes
)
    -- no return values
```

```
end

function token.setmacro (
    <t:integer> catcodetable,
    <t:string>  csname,
    <t:string>  content,
    <t:string>  prefix
 -- there can be more prefixes
)
    -- no return values
end
```

A macro can also be queried:

```
function token.getmacro (
    <t:string>  csname,
    <t:boolean> preamble,
    <t:boolean> onlypreamble
)
    return <t:string>
end
```

The various arguments determine what you get:

```
\def\foo#1{foo: #1}
```

```
\ctxlua{context.type(token.getmacro("foo"))}
\ctxlua{context.type(token.getmacro("foo",true))}
\ctxlua{context.type(token.getmacro("foo",false,true))}
```

We get:

```
foo: #1
#1->foo:
#1
```

The meaning can be fetched as string or table:

```
function token.getmeaning (
    <t:string>  csname,
)
    return <t:string>
end

function token.getmeaning (
    <t:string>  csname,
    <t:true>    astable,
    <t:boolean> subtables,
    <t:boolean> originalindices -- special usage
)
    return <t:table>
end
```

The name says it:

```
function token.undefinemacro ( <t:string> csname)
    -- no return values
end
```

Expanding a macro happens in a ‘local control’ context which makes it immediate, that is, while running Lua code.

```
function token.expandmacro ( <t:string> csname)
    -- no return values
end
```

This means that:

```
\def\foo{\scratchdimen100pt \edef\oof{\the\scratchdimen}}
% used in:
\startluacode
token.expandmacro("foo")
context(token.getmacro("oof"))
\stopluacode
```

gives: 100.0pt, because when `getmacro` is called the expansion has been performed. You can consider this a sort of subrun (local to the main control loop).

The next helper creates a token that refers to a Lua function with an entry in the table that you can access with `lua.getfunctionstable`. It is the companion to **`\luadef`**. When the first (and only) argument is true the size will preset to the value of `texconfig.functionsize`.

```
function token.setlua (
    <t:string>  csname,
    <t:integer> id,
    <t:string>  prefix
 -- there can be more prefixes
)
    return <t:token>
end
```

15.3.4 Writers

In the `tex` library we have various ways to print something back to the input and the these print helpers in most cases also accept tokens. The `token.putnext` function is rather tolerant with respect to its arguments and there can be multiple. As with most prints, a new input level is created.

```
function token.putnext ( <t:string> | <t:number> | <t:token> | <t:table> )
    -- no return values
end
```

Here are some examples. We save some scanned tokens and flush them

```
local t1 = token.scannext()
local t2 = token.scannext()
```

```
local t3 = token.scannext()
local t4 = token.scannext()
-- watch out, we flush in sequence
token.putnext { t1, t2 }
-- but this one gets pushed in front
token.putnext ( 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.

This putter is actually a bit more flexible because the following input also works out okay:

```
\def\foo#1{[#1]}

\directlua {
    local list = { 101, 102, 103, token.create("foo"), "{abracadabra}" }
    token.putnext("(the)")
    token.putnext(list)
    token.putnext("(order)")
    token.putnext(unpack(list))
    token.putnext("(is reversed)")
}
```

We get this:

```
(is reversed)efg[abracadabra](order)efg[abracadabra](the)
```

So, strings get converted to individual tokens according to the current catcode regime and numbers become characters also according to this regime. A more low level, single token push back is the next one, it does the same as when TEX itself puts a token back into the input, something that for instance happens when an integer is scanned and the last scanned token is not a digit.

```
function token.putback ( <t:token> )
    -- no return values
end
```

You can force an ‘expand step’ with the following function. What happens depends on the input and scanner states TEX is.

```
function token.expand ( )
    -- no return values
end
```

15.3.5 Scanning

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 okay so 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 LuaMetaTEX is to provide a minimal set of tools and no solutions. The scanner functions are probably the most intriguing.

We start with token scanners. The first one just reads the next token from the current input (file, token list, Lua output) while the second variant expands the next token, which can push back results and make us enter a new input level, and then reads a token from what is then the input.

```
function token.scannext ( )
    return <t:token>
end

function token.scannextexpanded ( )
    return <t:token>
end
```

This is a simple scanner that picks up a character:

```
function token.scannextchar ( )
    return <t:string>
end
```

We can look ahead, that is: pick up a token and push a copy back into the input. The second helper first expands the upcoming token and the third one is the peek variant of `scannextchar`.

```
function token.peeknext ( )
    return <t:token>
end

function token.peeknextexpanded ( )
    return <t:token>
end

function token.peeknextchar ( )
    return <t:token>
end
```

We can skip tokens with the following two helpers where the second one first expands the upcoming token

```
function token.skipnext ( )
    -- no return values
end

function token.skipnextexpanded ( )
    -- no return values
end
```

The next token can be converted into a combination of command and value. The second variant shown below first expands the upcoming token.

```
function token.scancmdchr ( )
    return
        <t:integer>, -- command a.k.a cmd
        <t:integer>, -- value   a.k.a chr
end
```

```
function token.scancmdchrexpanded ( )
    return
        <t:integer>, -- command a.k.a cmd
        <t:integer>, -- value   a.k.a chr
end
```

We have two keywords scanners. The first scans how TEX does it: a mixture of lower- and uppercase. The second is case sensitive.

```
function token.scankeyword ( <t:string> keyword )
    return <t:boolean> -- success
end
```

```
function token.scankeywordcs ( <t:string> keyword )
    return <t:boolean> -- success
end
```

The integer, dimension and glue scanners take an extra optional argument that signals that en optional equal is permitted. The next function errors when the integer exceeds the maximum that TEX likes: 2147483647.

```
function token.scaninteger ( <t:boolean> optionalequal )
    return <t:integer>
end
```

Cardinals are unsigned integers:

```
function token.scancardinal ( <t:boolean> optionalequal )
    return <t:cardinal>
end
```

When an integer or dimension is wrapped in curly braces, like `{123}` and `{4.5pt}`, you can use one of the next two. Of course unwrapped integers and dimensions are also read.

```
function token.scanintegerargument ( <t:boolean> optionalequal )
    return <t:integer>
end
```

```
function token.scandimensionargument (
    <t:boolean> infinity,
    <t:boolean> mu,
    <t:boolean> optionalequal
)
    return <t:integer>
end
```

When we scan for a float, we also accept an exponent, so `123.45` and `-1.23e45` are valid:

```
function token.scanfloat ( )
    return <t:number>
end
```

Contrary to the previous scanner here we don't handle the exponent:

```
function token.scanreal ( )
    return <t:number>
end
```

In Lua a very precise representation of a float is the hexadecimal notation. In addition to regular floating point, optionally with an exponent, you can also have `0x1.23p45`.

```
function token.scanluanumber ( )
    return <t:number>
end
```

Integers can be signed:

```
function token.scanluainteger ( )
    return <t:integer>
end
```

while cardinals (Modula2 speak) are unsigned: unsigned

```
function token.scanluacardinal ( )
    return <t:cardinal>
end
```

122345

```
function token.scanscale ( )
    return <t:integer>
end
```

A posit is (in LuaMetaTEX) a float packed into an integer, but contrary to a scaled value it can have exponents. Here `12.34` gives `1549208125` and Here `12.34e5` gives `2114670912`. Because we have integers we can store them in LuaMetaTEX float registers. Optionally you can return a float instead of the integer that encodes the posit.

```
function token.scanposit (
    <t:boolean> optionalqual,
    <t:boolean> float
)
    return <t:integer> | <t:float>
end
```

In (traditional) TEX we don't really have floats. If we enter for instance a dimension in point units, we actually scan for two 16 bit integers that will be packed into a 32 bit integer. The next scanner expects a number plus a unit, like `pt`, `cm` and `em`, but also handles user defined units, like in ConTEXt `tw`.

```
function token.scandimension (
    <t:boolean> infinity,
    <t:boolean> mu,
    <t:boolean> optionalequal
)
    return <t:integer>
end
```

A glue (spec) is a dimension with optional stretch and/or shrink, like `12pt plus 4pt minus 2pt` or `10pt plus 1 fill`. The glue scanner returns five values:

```
function token.scanglue (
    <t:boolean> mu,
    <t:boolean> optionalequal
)
    return
        <t:integer>, -- amount
        <t:integer>, -- stretch
        <t:integer>, -- shrink
        <t:integer>, -- stretchorder
        <t:integer>  -- shrinkorder
end
```

```
function token.scanglue (
    <t:boolean> mu,
    <t:boolean> optionalequal,
    <t:true>
)
    return {
        <t:integer>, -- amount
        <t:integer>, -- stretch
        <t:integer>, -- shrink
        <t:integer>, -- stretchorder
        <t:integer>  -- shrinkorder
    }
end
```

The skip scanner does the same but returns a `gluespec` node:

```
function token.scanskip (
    <t:boolean> mu,
    <t:boolean> optionalequal
)
    return <t:node> -- gluespec
end
```

There are several token scanners, for instance one that returns a table:

```
function token.scantoks (
    <t:boolean> macro,
    <t:boolean> expand
)
    -- return <t:table> -- tokens
end
```

Here `token.scantoks()` will return `{123}` as

```
{
 "<lua token : 589866 == other_char 49>",
 "<lua token : 589867 == other_char 50>",
```

```
 "<lua token : 589870 == other_char 51>",
}
```

The next variant returns a token list:

```
function token.scantokenlist (
    <t:boolean> macro,
    <t:boolean> expand
)
    return <t:token> -- tokenlist
end
```

Here we get the head of a token list:

```
<lua token : 590083 => 169324 : refcount>={
 ["active"]=false,
 ["cmdname"]="escape",
 ["command"]=0,
 ["expandable"]=false,
 ["frozen"]=false,
 ["id"]=590083,
 ["immutable"]=false,
 ["index"]=0,
}
```

This scans a single character token with specified catcode (bit) sets:

```
function token.scancode ( <t:integer> catcodes )
    return <t:string> -- character
end
```

This scans a single character token with catcode letter or other:

```
function token.scantokencode ( )
    -- return <t:token>
end
```

The difference between `scanstring` and `scanargument` is that the first returns a string given between `{}`, as `\macro` or as sequence of characters with catcode 11 or 12 while the second also accepts a `\cs` which then get expanded one level unless we force further expansion.

```
function token.scanstring ( <t:boolean> expand )
    return <t:string>
end

function token.scanargument ( <t:boolean> expand )
    return <t:string>
end
```

So the `scanargument` function expands the given argument. When a braced argument is scanned, expansion can be prohibited by passing `false` (default is `true`). In case of a control sequence passing `false` will result in a one-level expansion (the meaning of the macro).

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\oof{oof}
\def\foo{foo-\oof}
```

we get:

name	result	
`\directlua{token.scanstring()}{foo}`	`foo`	full expansion
`\directlua{token.scanstring()}foo`	`foo`	letters and others
`\directlua{token.scanstring()}\foo`	`foo-oof`	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.

A variant is the following which give a bit more control over what doesn't get expanded:

```
function token.scantokenstring (
    <t:boolean> noexpand,
    <t:boolean> noexpandconstant,
    <t:boolean> noexpandparameters
)
    return <t:string>
end
```

Here's one that can scan a delimited argument:

```
function token.scandelimited (
    <t:integer> leftdelimiter,
    <t:integer> rightdelimiter,
    <t:boolean> expand
)
    return <t:string>
end
```

A word is a sequence of what TeX calls letters and other characters. The optional `keep` argument endures that trailing space and **`\relax`** tokens are pushed back into the input.

```
function token.scanword ( <t:boolean> keep )
    return <t:string>
end
```

Here we do the same but only accept letters:

```
function token.scanletters ( <t:boolean> keep )
    return <t:string>
end
```

```
function token.scankey ( )
```

```
    return <t:string>
end
```

We can pick up a string that stops at a specific character with the next function, which accepts two such sentinels (think of a comma and closing bracket).

```
function token.scanvalue ( <t:integer> one, <t:integer> two )
    return <t:string>
end
```

This returns a single (utf) character. Special input like back slashes, hashes, etc. are interpreted as characters.

```
function token.scanchar ( )
    return <t:string>
end
```

This scanner looks for a control sequence and if found returns the name. Optionally leading spaces can be skipped.

```
function token.scancsname ( <t:boolean> skipspaces )
    return <t:string> | <t:nil>
end
```

The next one returns an integer instead:

```
function token.scancstoken ( <t:boolean> skipspaces )
    return <t:integer> | <t:nil>
end
```

This is a straightforward simple scanner that expands next token if needed:

```
function token.scantoken ( )
    return <t:token>
end
```

Then next scanner picks up a box specification and returns a `[h|v]list` node. There are two possible calls. The first variant expects a **\hbox**, **\vbox** etc. The second variant scans for an explicitly passed box type: `hbox`, `vbox`, `vbox` or `dbox`.

```
function token.scanbox ( )
    return <t:node> -- box
end
```

```
function token.scanbox ( <t:string> boxtype )
    return <t:node> -- box
end
```

This scans and returns a so called ‘detokenized’ string:

```
function token.scandetokened ( <t:boolean> expand )
    return <t:string>
end
```

In the next function we check if a specific character with catcode letter or other is picked up.

```
function token.isnextchar ( <t:integer> charactercode  )
    return <t:boolean>
end
```

15.3.6 Gobbling

You can gobble up an integer or dimension with the following helpers. An error is silently ignored.

```
function token.gobbleinteger ( <t:boolean> optionalequal )
    -- no return values
end
```

```
function token.gobbledimension ( <t:boolean> optionalequal )
    -- no return values
end
```

This is a nested gobbler:

```
function token.gobble ( <t:token> left, <t:token> right )
    -- no return values
end
```

and this a nested grabber that returns a string:

```
function token.grab ( <t:token> left, <t:token> right )
    return <t:string>
end
```

15.3.7 Macros

This is a nasty one. It pick up two tokens. Then it checks if the next character matches the argument and if so, it pushes the first token back into the input, otherwise the second.

```
function token.futureexpand ( <t:integer> charactercode )
    -- no return values
end
```

The `pushmacro` and `popmacro` function are still experimental and can be used to get and set an existing macro. The push call returns a user data object and the pop takes such a userdata object. These object have no accessors and are to be seen as abstractions.

```
function token.pushmacro ( <t:string> csname )
    return <t:userdata>
end
```

```
function token.pushmacro ( <t:integer> token )
    return <t:userdata> -- entry
end
```

```
function token.popmacro ( <t:userdata> entry )
```

```
    -- return todo
end
```

This saves a Lua function index on the save stack. When a group is closes the function will be called.

```
function token.savelua ( <t:integer> functionindex, <t:boolean> backtrack )
    -- no return values
end
```

The next function serializes a token list:

```
function token.serialize ( )
    return <t:string>
end
```

The function is somewhat picky so give van example in ConTeXt speak:

```
\startluacode
    local t = token.scantokenlist()
    local s = token.serialize(t)
    context.type(tostring(t)) context.par()
    context.type(s)           context.par()
    context(s)                context.par()
\stopluacode {before\hskip10pt after}
```

The serialize expects a token list as scanned by `scantokenlist` which starts with token that points to the list and maintains a reference count, which in this context is irrelevant but is used in the engine to prevent duplicates; for instance the `\let` primitive just points to the original and bumps the count.

```
<lua token : 67914 => 584772 : refcount>
before\hskip 10pt after
```

before after

You can interpret a string as TeX input with embedded macros expanded, unless they are unexpandable.

```
function token.getexpansion ( <t:string> code )
    return <t:string> -- result
end
```

Here is an example:

```
          \def\foo{foo}
\protected\def\oof{oof}

\startluacode
context.type(token.getexpansion("test \relax"))
context.par()
context.type(token.getexpansion("test \\relax{!} \\foo\\oof"))
\stopluacode
```

Watch how the single backslash actually is a Lua escape that results in a newline:

```
test
```

```
elax
test \relax{!} foo\oof
```

You can also specify a catcode table identifier:

```
function token.getexpansion (
    <t:integer> catcodetable,
    <t:string>  code
)
    return <t:string> -- result
end
```

15.3.8 Information

In some cases you signal to Lua what data type is involved. The list of known types are available with:

```
function token.getfunctionvalues ( )
    return <t:table>
end
```

0x00	none	0x04	skip	0x08	node
0x01	integer	0x05	boolean	0x09	direct
0x02	cardinal	0x06	float	0x0A	conditional
0x03	dimension	0x07	string		

The names of command is made available with:

```
function token.getcommandvalues ( )
    return <t:table>
end
```

0x00	escape	0x13	match
0x01	left_brace	0x14	end_match
0x02	right_brace	0x15	parameter_reference
0x03	math_shift	0x16	end_paragraph
0x04	alignment_tab	0x17	end_job
0x05	end_line	0x18	delimiter_number
0x06	parameter	0x19	char_number
0x07	superscript	0x1A	math_char_number
0x08	subscript	0x1B	mark
0x09	ignore	0x1C	node
0x0A	spacer	0x1D	xray
0x0B	letter	0x1E	make_box
0x0C	other_char	0x1F	hmove
0x0D	active_char	0x20	vmove
0x0E	comment	0x21	un_hbox
0x0F	invalid_char	0x22	un_vbox
0x10	relax	0x23	remove_item
0x11	alignment	0x24	hskip
0x12	end_template	0x25	vskip

0x26 mskip
0x27 kern
0x28 mkern
0x29 leader
0x2A legacy
0x2B local_box
0x2C halign
0x2D valign
0x2E vrule
0x2F hrule
0x30 insert
0x31 vadjust
0x32 ignore_something
0x33 after_something
0x34 penalty
0x35 begin_paragraph
0x36 italic_correction
0x37 accent
0x38 math_accent
0x39 discretionary
0x3A equation_number
0x3B math_fence
0x3C math_component
0x3D math_modifier
0x3E math_fraction
0x3F math_choice
0x40 vcenter
0x41 case_shift
0x42 message
0x43 catcode_table
0x44 end_local
0x45 lua_function_call
0x46 lua_protected_call
0x47 lua_semiprotected_call
0x48 begin_group
0x49 end_group
0x4A explicit_space
0x4B boundary
0x4C math_radical
0x4D math_script
0x4E math_shift_cs
0x4F end_cs_name
0x50 char_given
0x51 some_item
0x52 internal_toks
0x53 register_toks
0x54 internal_integer
0x55 register_integer
0x56 internal_attribute
0x57 register_attribute
0x58 internal_posit
0x59 register_posit
0x5A internal_dimension
0x5B register_dimension
0x5C internal_glue
0x5D register_glue
0x5E internal_muglue
0x5F register_muglue
0x60 lua_value
0x61 iterator_value
0x62 font_property
0x63 auxiliary
0x64 hyphenation
0x65 page_property
0x66 box_property
0x67 specification
0x68 define_char_code
0x69 define_family
0x6A math_parameter
0x6B math_style
0x6C set_font
0x6D define_font
0x6E integer
0x6F posit
0x70 dimension
0x71 gluespec
0x72 mugluespec
0x73 index
0x74 mathspec
0x75 fontspec
0x76 specificationspec
0x77 association
0x78 interaction
0x79 register
0x7A combine_toks
0x7B arithmic
0x7C prefix
0x7D let
0x7E shorthand_def
0x7F def
0x80 set_box
0x81 undefined_cs
0x82 expand_after
0x83 no_expand
0x84 input
0x85 lua_call
0x86 lua_local_call
0x87 begin_local

0x88 if_test
0x89 cs_name
0x8A convert
0x8B the
0x8C get_mark
0x8D call
0x8E protected_call
0x8F semi_protected_call
0x90 constant_call
0x91 tolerant_call
0x92 tolerant_protected_call
0x93 tolerant_semi_protected_call
0x94 deep_frozen_end_template
0x95 deep_frozen_dont_expand
0x96 deep_frozen_keep_constant
0x97 internal_glue_reference
0x98 register_glue_reference
0x99 internal_muglue_reference
0x9A register_muglue_reference
0x9B specification_reference
0x9C internal_box_reference
0x9D internal_toks_reference
0x9E register_toks_reference
0x9F specification_reference
0xA0 unit_reference
0xA1 internal_integer_reference
0xA2 register_integer_reference
0xA3 internal_attribute_reference
0xA4 register_attribute_reference
0xA5 internal_posit_reference
0xA6 register_posit_reference
0xA7 internal_dimension_reference
0xA8 register_dimension_reference

The complete list of primitives can be fetched with the next one:

```
function token.getprimitives ( )
    return {
        { <t:integer>, <t:integer>, <t:string> }, -- command, value, name
        ...
    }
end
```

The numbers shown below can change if we add or reorganize primitives, although this seldom happens. The list gives an impression how primitives are grouped.

4	0	\aligntab
6	0	\parametermark
6	0	\alignmark
16	0	\relax
16	1	\norelax
18	1	\span
18	2	\omit
18	3	\aligncontent
18	4	\noalign
18	5	\realign
18	6	\cr
18	7	\crcr
22	0	\par
22	3	\localbreakpar
23	0	\end
23	1	\dump
24	0	\delimiter
24	1	\Udelimiter
25	0	\char
25	1	\glyph
26	0	\mathchar
26	1	\Umathchar
26	2	\mathdictionary
26	3	\mathclass
26	4	\nomathchar
27	0	\mark
27	1	\marks
27	2	\clearmarks
27	3	\flushmarks
29	0	\show
29	1	\showbox
29	2	\showthe
29	3	\showlists
29	4	\showgroups
29	5	\showstack
29	6	\showcodestack
29	7	\showtokens
29	8	\showifs
30	0	\box
30	1	\copy

30	3	\lastbox
30	4	\tsplit
30	5	\vsplit
30	6	\dsplit
30	7	\tpack
30	8	\vpack
30	9	\dpack
30	10	\hpack
30	11	\vtop
30	12	\vbox
30	13	\dbox
30	14	\hbox
30	15	\insertbox
30	16	\insertcopy
30	17	\localleftboxbox
30	18	\localrightboxbox
30	19	\localmiddleboxbox
31	0	\moveright
31	1	\moveleft
32	0	\lower
32	1	\raise
33	0	\unhbox
33	1	\unhcopy
33	2	\unhpack
34	0	\unvbox
34	1	\unvcopy
34	2	\unvpack
34	15	\insertunbox
34	16	\insertuncopy
34	20	\pagediscards
34	21	\splitdiscards
35	0	\unkern
35	1	\unpenalty
35	2	\unskip
35	3	\unboundary
36	0	\hfil
36	1	\hfill
36	2	\hss
36	3	\hfilneg
36	4	\hskip
37	0	\vfil
37	1	\vfill
37	2	\vss
37	3	\vfilneg
37	4	\vskip
38	0	\mskip
38	1	\mathatomskip
39	0	\kern
39	1	\hkern
39	2	\vkern
40	0	\mkern
41	0	\leaders
41	1	\cleaders
41	2	\xleaders
41	3	\gleaders
41	4	\uleaders
42	0	\shipout
43	0	\localleftbox
43	1	\localrightbox
43	2	\localmiddlebox
43	4	\resetlocalboxes
44	0	\halign
45	0	\valign
46	0	\vrule
46	1	\novrule
46	2	\srule
46	3	\virtualvrule
47	0	\hrule
47	1	\nohrule
47	3	\virtualhrule
48	0	\insert
49	0	\vadjust
50	0	\ignorespaces
50	1	\ignorepars
50	2	\ignorearguments
50	3	\ignoreupto
50	4	\ignorenestedupto
50	5	\ignorerest
51	0	\aftergroup
51	1	\aftergrouped
51	2	\afterassignment
51	3	\afterassigned
51	4	\atendofgroup
51	5	\atendofgrouped
51	6	\atendoffile
51	7	\atendoffiled
52	0	\penalty
52	1	\hpenalty
52	2	\vpenalty
53	0	\noindent
53	1	\indent
53	2	\quitvmode
53	3	\undent
53	4	\snapshotpar
53	5	\parattribute
53	6	\wrapuppar
54	0	\/
54	0	\explicititaliccorrection

54	1	\forcedleftcorrection
54	2	\forcedrightcorrection
55	0	\accent
56	0	\mathaccent
56	1	\Umathaccent
57	0	\discretionary
57	1	\explicitdiscretionary
57	1	\-
57	2	\automaticdiscretionary
58	0	\leqno
58	1	\eqno
59	1	\left
59	2	\middle
59	3	\right
59	4	\Uoperator
59	5	\Uvextensible
59	6	\Uleft
59	7	\Umiddle
59	8	\Uright
60	0	\mathord
60	1	\mathop
60	2	\mathbin
60	3	\mathrel
60	4	\mathopen
60	5	\mathclose
60	6	\mathpunct
60	8	\mathinner
60	9	\underline
60	10	\overline
60	18	\mathatom
61	0	\displaylimits
61	1	\Umathlimits
61	1	\limits
61	2	\Umathnolimits
61	2	\nolimits
61	3	\Umathadapttoleft
61	4	\Umathadapttoright
61	5	\Umathuseaxis
61	6	\Umathnoaxis
61	7	\Umathphantom
61	8	\Umathvoid
61	9	\Umathsource
61	10	\Umathopenupheight
61	11	\Umathopenupdepth
62	0	\above
62	1	\abovewithdelims
62	2	\over
62	3	\overwithdelims
62	4	\atop
62	5	\atopwithdelims
62	6	\Uabove
62	7	\Uabovewithdelims
62	8	\Uover
62	9	\Uoverwithdelims
62	10	\Uatop
62	11	\Uatopwithdelims
62	12	\Uskewed
62	13	\Uskewedwithdelims
62	14	\Ustretched
62	15	\Ustretchedwithdelims
63	0	\mathchoice
63	1	\mathdiscretionary
63	2	\mathstack
64	0	\vcenter
65	0	\lowercase
65	1	\uppercase
66	0	\message
66	1	\errmessage
67	0	\savecatcodetable
67	1	\restorecatcodetable
67	2	\initcatcodetable
68	0	\endlocalcontrol
69	0	\luafunctioncall
69	1	\luabytecodecall
72	0	\begingroup
72	1	\beginsimplegroup
72	2	\beginmathgroup
73	0	\endgroup
73	1	\endsimplegroup
73	2	\endmathgroup
74	0	\explicitspace
74	0	\
75	0	\noboundary
75	1	\boundary
75	2	\protrusionboundary
75	3	\wordboundary
75	4	\pageboundary
75	5	\mathboundary
75	6	\optionalboundary
75	7	\luaboundary
76	0	\radical
76	1	\Uradical
76	2	\Uroot
76	3	\Urooted
76	4	\Uunderdelimiter
76	5	\Uoverdelimiter
76	6	\Udelimiterunder
76	7	\Udelimiterover

76	8	\Udelimited
76	9	\Uhextensible
77	0	\nonscript
77	1	\noatomruling
77	2	\subscript
77	3	\superscript
77	4	\superprescript
77	5	\subprescript
77	6	\nosubscript
77	7	\nosuperscript
77	8	\nosubprescript
77	9	\nosuperprescript
77	10	\indexedsubscript
77	11	\indexedsuperscript
77	12	\indexedsubprescript
77	13	\indexedsuperprescript
77	14	\primescript
77	15	\noscript
78	0	\Ustartmath
78	1	\Ustopmath
78	2	\Ustartdisplaymath
78	3	\Ustopdisplaymath
78	4	\Ustartmathmode
78	5	\Ustopmathmode
79	0	\endcsname
81	0	\lastpenalty
81	1	\lastkern
81	2	\lastskip
81	3	\lastboundary
81	4	\lastnodetype
81	5	\lastnodesubtype
81	6	\inputlineno
81	7	\badness
81	8	\overshoot
81	9	\luatexversion
81	10	\luatexrevision
81	11	\currentgrouplevel
81	12	\currentgrouptype
81	13	\currentstacksize
81	14	\currentiflevel
81	15	\currentiftype
81	16	\currentifbranch
81	17	\gluestretchorder
81	18	\glueshrinkorder
81	19	\fontid
81	20	\glyphxscaled
81	21	\glyphyscaled
81	22	\fontcharwd
81	23	\fontcharht
81	24	\fontchardp
81	25	\fontcharic
81	26	\fontcharta
81	27	\fontcharba
81	28	\scaledfontcharwd
81	29	\scaledfontcharht
81	30	\scaledfontchardp
81	31	\scaledfontcharic
81	32	\scaledfontcharta
81	33	\scaledfontcharba
81	34	\fontspecid
81	35	\fontspecscale
81	36	\fontspecxscale
81	37	\fontspecyscale
81	38	\fontspecslant
81	39	\fontspecweight
81	40	\fontspecifiedsize
81	41	\fontmathcontrol
81	42	\fonttextcontrol
81	43	\mathscale
81	44	\mathstyle
81	45	\mathmainstyle
81	46	\mathparentstyle
81	47	\mathstylefontid
81	48	\mathstackstyle
81	49	\mathcharclass
81	50	\mathcharfam
81	51	\mathcharslot
81	52	\scaledslantperpoint
81	53	\scaledinterwordspace
81	54	\scaledinterwordstretch
81	55	\scaledinterwordshrink
81	56	\scaledexheight
81	57	\scaledemwidth
81	58	\scaledextraspace
81	59	\scaledmathaxis
81	60	\scaledmathexheight
81	61	\scaledmathemwidth
81	62	\lastarguments
81	63	\parametercount
81	64	\parameterindex
81	65	\insertprogress
81	66	\leftmarginkern
81	67	\rightmarginkern
81	68	\parshapelength
81	69	\parshapeindent
81	70	\parshapewidth
81	70	\parshapedimen
81	71	\gluestretch

81	72	\glueshrink
81	73	\mutoglue
81	74	\gluetomu
81	75	\numexpr
81	76	\floatexpr
81	77	\dimexpr
81	78	\glueexpr
81	79	\muexpr
81	80	\numexpression
81	81	\dimexpression
81	82	\lastchknumber
81	83	\lastchkdimension
81	84	\numericscale
81	85	\numericscaled
81	86	\indexofregister
81	87	\indexofcharacter
81	88	\mathatomglue
81	89	\lastleftclass
81	90	\lastrightclass
81	91	\lastatomclass
81	92	\nestedloopiterator
81	93	\previousloopiterator
81	94	\currentloopiterator
81	95	\currentloopnesting
81	96	\lastloopiterator
81	97	\lastpartrigger
81	98	\lastparcontext
81	99	\lastpageextra
82	0	\output
82	1	\everypar
82	2	\everymath
82	3	\everydisplay
82	4	\everyhbox
82	5	\everyvbox
82	6	\everymathatom
82	7	\everyjob
82	8	\everycr
82	9	\everytab
82	10	\errhelp
82	11	\everybeforepar
82	12	\everyeof
84	38	\discretionaryoptions
84	38	\setfontid
84	38	\glyphslant
84	38	\catcodetable
84	38	\glyphscriptscriptscale
84	38	\textdirection
84	38	\outputbox
84	38	\glyphscale
84	38	\endlinechar
84	38	\glyphxscale
84	38	\hyphenationmode
84	38	\language
84	38	\mathleftclass
84	38	\localbrokenpenalty
84	38	\eufactor
84	38	\mathdirection
84	38	\pardirection
84	38	\glyphtextscale
84	38	\prebinoppenalty
84	38	\mathendclass
84	38	\protrudechars
84	38	\uchyph
84	38	\binoppenalty
84	38	\mathrightclass
84	38	\pretolerance
84	38	\newlinechar
84	38	\relpenalty
84	38	\localtolerance
84	38	\glyphscriptscale
84	38	\mathbeginclass
84	38	\localinterlinepenalty
84	38	\adjustspacing
84	38	\overloadmode
84	38	\linedirection
84	38	\prerelpenalty
84	38	\setlanguage
84	38	\localpretolerance
84	38	\glyphweight
84	38	\glyphyscale
84	38	\glyphoptions
84	39	\tolerance
84	40	\linepenalty
84	41	\hyphenpenalty
84	42	\exhyphenpenalty
84	43	\clubpenalty
84	44	\widowpenalty
84	45	\displaywidowpenalty
84	46	\brokenpenalty
84	47	\predisplaypenalty
84	48	\postdisplaypenalty
84	49	\preinlinepenalty
84	50	\postinlinepenalty
84	51	\preshortinlinepenalty
84	52	\postshortinlinepenalty
84	53	\shortinlineorphanpenalty
84	54	\interlinepenalty
84	55	\doublehyphendemerits

84	56	\finalhyphendemerits
84	57	\adjdemerits
84	58	\doublepenaltymode
84	59	\delimiterfactor
84	60	\looseness
84	61	\time
84	62	\day
84	63	\month
84	64	\year
84	65	\showboxbreadth
84	66	\showboxdepth
84	67	\shownodedetails
84	68	\hbadness
84	69	\vbadness
84	70	\pausing
84	71	\tracingonline
84	72	\tracingmacros
84	73	\tracingstats
84	74	\tracingparagraphs
84	75	\tracingpages
84	76	\tracingoutput
84	77	\tracinglostchars
84	78	\tracingcommands
84	79	\tracingrestores
84	80	\tracingassigns
84	81	\tracinggroups
84	82	\tracingifs
84	83	\tracingmath
84	84	\tracinglevels
84	85	\tracingnesting
84	86	\tracingalignments
84	87	\tracinginserts
84	88	\tracingmarks
84	89	\tracingadjusts
84	90	\tracinghyphenation
84	91	\tracingexpressions
84	92	\tracingnodes
84	93	\tracingfullboxes
84	94	\tracingpenalties
84	95	\tracinglooseness
84	96	\tracinglists
84	97	\tracingpasses
84	98	\tracingfitness
84	99	\tracingtoddlers
84	100	\tracingorphans
84	101	\tracingloners
84	102	\outputpenalty
84	103	\maxdeadcycles
84	104	\hangafter
84	105	\floatingpenalty
84	106	\globaldefs
84	107	\fam
84	108	\escapechar
84	109	\spacechar
84	110	\defaulthyphenchar
84	111	\defaultskewchar
84	112	\lefthyphenmin
84	113	\righthyphenmin
84	114	\holdinginserts
84	115	\holdingmigrations
84	116	\errorcontextlines
84	117	\nospaces
84	118	\parametermode
84	119	\glyphdatafield
84	120	\glyphstatefield
84	121	\glyphscriptfield
84	122	\exhyphenchar
84	123	\adjustspacingstep
84	124	\adjustspacingstretch
84	125	\adjustspacingshrink
84	126	\predisplaydirection
84	127	\lastlinefit
84	128	\savingvdiscards
84	129	\savinghyphcodes
84	130	\matheqnogapstep
84	131	\mathdisplayskipmode
84	132	\mathscriptsmode
84	133	\mathlimitsmode
84	134	\mathnolimitsmode
84	135	\mathrulesmode
84	136	\mathrulesfam
84	137	\mathpenaltiesmode
84	138	\mathcheckfencesmode
84	139	\mathslackmode
84	140	\mathsurroundmode
84	141	\mathdoublescriptmode
84	142	\mathfontcontrol
84	143	\mathdisplaymode
84	144	\mathdictgroup
84	145	\mathdictproperties
84	146	\predisplaygapfactor
84	147	\firstvalidlanguage
84	148	\automatichyphenpenalty
84	149	\explicithyphenpenalty
84	150	\exceptionpenalty
84	151	\luacopyinputnodes
84	152	\automigrationmode
84	153	\normalizelinemode

84	154	\normalizeparmode
84	155	\mathspacingmode
84	156	\mathgroupingmode
84	157	\mathgluemode
84	158	\mathinlinepenaltyfactor
84	159	\mathdisplaypenaltyfactor
84	160	\supmarkmode
84	161	\autoparagraphmode
84	162	\shapingpenaltiesmode
84	163	\shapingpenalty
84	164	\singlelinepenalty
84	165	\lefttwindemerits
84	166	\righttwindemerits
84	167	\alignmentcellsource
84	168	\alignmentwrapsource
84	169	\linebreakpasses
84	170	\linebreakoptional
84	171	\linebreakchecks
84	172	\variablefam
84	173	\mathpretolerance
84	174	\mathtolerance
84	175	\spacefactormode
84	176	\spacefactorshrinklimit
84	177	\spacefactorstretchlimit
84	178	\spacefactoroverload
84	179	\boxlimitmode
84	180	\scriptspacebeforefactor
84	181	\scriptspacebetweenfactor
84	182	\scriptspaceafterfactor
90	0	\parindent
90	1	\mathsurround
90	2	\lineskiplimit
90	3	\hsize
90	4	\vsize
90	5	\maxdepth
90	6	\splitmaxdepth
90	7	\boxmaxdepth
90	8	\hfuzz
90	9	\vfuzz
90	10	\delimitershortfall
90	11	\nulldelimiterspace
90	12	\scriptspace
90	13	\predisplaysize
90	14	\displaywidth
90	15	\displayindent
90	16	\overfullrule
90	17	\hangindent
90	18	\emergencystretch
90	19	\emergencyextrastretch
90	20	\glyphxoffset
90	21	\glyphyoffset
90	22	\pxdimen
90	23	\tabsize
90	24	\pageextragoal
90	25	\ignoredepthcriterion
90	26	\shortinlinemaththreshold
92	3	\initialpageskip
92	3	\lineskip
92	3	\initialtopskip
92	3	\additionalpageskip
92	4	\baselineskip
92	5	\parskip
92	6	\abovedisplayskip
92	7	\belowdisplayskip
92	8	\abovedisplayshortskip
92	9	\belowdisplayshortskip
92	10	\leftskip
92	11	\rightskip
92	12	\topskip
92	13	\splittopskip
92	14	\tabskip
92	15	\spaceskip
92	16	\xspaceskip
92	17	\parfillleftskip
92	18	\parfillrightskip
92	18	\parfillskip
92	19	\parinitleftskip
92	20	\parinitrightskip
92	21	\emergencyleftskip
92	22	\emergencyrightskip
92	23	\mathsurroundskip
92	24	\maththreshold
94	1	\pettymuskip
94	2	\tinymuskip
94	3	\thinmuskip
94	4	\medmuskip
94	5	\thickmuskip
98	0	\hyphenchar
98	1	\skewchar
98	2	\lpcode
98	3	\rpcode
98	4	\efcode
98	5	\cfcode
98	6	\fontdimen
98	7	\scaledfontdimen
99	0	\spacefactor
99	1	\prevdepth
99	2	\prevgraf

99	3	\interactionmode
99	4	\insertmode
100	0	\hyphenation
100	1	\patterns
100	2	\prehyphenchar
100	3	\posthyphenchar
100	4	\preexhyphenchar
100	5	\postexhyphenchar
100	6	\hyphenationmin
100	7	\hjcode
101	0	\pagegoal
101	1	\pagevsize
101	2	\pagetotal
101	3	\pagedepth
101	4	\pageexcess
101	5	\pagelastheight
101	6	\pagelastdepth
101	7	\deadcycles
101	8	\insertpenalties
101	9	\insertheights
101	10	\insertstoring
101	11	\insertdistance
101	12	\insertmultiplier
101	13	\insertlimit
101	14	\insertstorage
101	15	\insertpenalty
101	16	\insertmaxdepth
101	17	\insertheight
101	18	\insertdepth
101	19	\insertwidth
101	20	\pagestretch
101	21	\pagefistretch
101	22	\pagefilstretch
101	23	\pagefillstretch
101	24	\pagefilllstretch
101	25	\pageshrink
101	26	\pagelaststretch
101	27	\pagelastfistretch
101	28	\pagelastfilstretch
101	29	\pagelastfillstretch
101	30	\pagelastfilllstretch
101	31	\pagelastshrink
102	0	\wd
102	1	\ht
102	2	\dp
102	3	\boxdirection
102	4	\boxgeometry
102	5	\boxorientation
102	6	\boxanchor
102	7	\boxanchors
102	8	\boxsource
102	9	\boxtarget
102	10	\boxxoffset
102	11	\boxyoffset
102	12	\boxxmove
102	13	\boxymove
102	14	\boxtotal
102	15	\boxshift
102	16	\boxadapt
102	17	\boxrepack
102	18	\boxfreeze
102	19	\boxlimitate
102	20	\boxfinalize
102	21	\boxlimit
102	22	\boxstretch
102	23	\boxshrink
102	24	\boxattribute
102	25	\boxvadjust
103	0	\parshape
103	0	\interlinepenalties
103	0	\toddlerpenalties
103	0	\parpasses
103	0	\clubpenalties
103	0	\orphanlinefactors
103	0	\parpassesexception
103	0	\orphanpenalties
103	0	\adjacentdemerits
103	0	\brokenpenalties
103	0	\displaywidowpenalties
103	0	\mathbackwardpenalties
103	0	\widowpenalties
103	0	\fitnessclasses
103	0	\mathforwardpenalties
104	0	\catcode
104	1	\lccode
104	2	\uccode
104	3	\sfcode
104	4	\hccode
104	5	\hmcode
104	6	\amcode
104	7	\cccode
104	8	\mathcode
104	9	\Umathcode
104	10	\delcode
104	11	\Udelcode
105	0	\textfont
105	1	\scriptfont
105	2	\scriptscriptfont

106		\Umath...
107	0	\displaystyle
107	1	\crampeddisplaystyle
107	2	\textstyle
107	3	\crampedtextstyle
107	4	\scriptstyle
107	5	\crampedscriptstyle
107	6	\scriptscriptstyle
107	7	\crampedscriptscriptstyle
107	8	\alldisplaystyles
107	9	\alltextstyles
107	10	\allscriptstyles
107	11	\allscriptscriptstyles
107	12	\allmathstyles
107	13	\allmainstyles
107	14	\allsplitstyles
107	15	\allunsplitstyles
107	16	\alluncrampedstyles
107	17	\allcrampedstyles
107	18	\currentlysetmathstyle
107	19	\givenmathstyle
107	20	\scaledmathstyle
108	0	\nullfont
109	0	\font
119	0	\associateunit
120	0	\batchmode
120	1	\nonstopmode
120	2	\scrollmode
120	3	\errorstopmode
121	0	\float
121	1	\count
121	2	\attribute
121	3	\dimen
121	4	\skip
121	5	\muskip
121	6	\toks
122	0	\etoks
122	1	\toksapp
122	2	\etoksapp
122	3	\tokspre
122	4	\etokspre
122	5	\xtoks
122	6	\gtoksapp
122	7	\xtoksapp
122	8	\gtokspre
122	9	\xtokspre
123	0	\advance
123	1	\multiply
123	2	\divide
123	3	\edivide
123	4	\rdivide
123	5	\advanceby
123	6	\multiplyby
123	7	\divideby
123	8	\edivideby
123	9	\rdivideby
124	0	\frozen
124	1	\permanent
124	2	\immutable
124	3	\mutable
124	4	\noaligned
124	5	\instance
124	6	\untraced
124	7	\global
124	8	\tolerant
124	9	\protected
124	10	\overloaded
124	11	\aliased
124	12	\immediate
124	13	\deferred
124	14	\semiprotected
124	15	\enforced
124	17	\inherited
124	18	\constant
124	19	\retained
124	20	\constrained
124	21	\long
124	22	\outer
125	0	\glet
125	1	\let
125	2	\futurelet
125	3	\futuredef
125	4	\letcharcode
125	5	\swapcsvalues
125	6	\letprotected
125	7	\unletprotected
125	8	\letfrozen
125	9	\unletfrozen
125	10	\letcsname
125	11	\gletcsname
125	12	\lettonothing
125	13	\glettonothing
125	14	\lettolastnamedcs
126	0	\chardef
126	1	\mathchardef
126	2	\Umathchardef
126	3	\Umathdictdef
126	4	\floatdef

126	5	\countdef
126	6	\attributedef
126	7	\dimendef
126	8	\skipdef
126	9	\muskipdef
126	10	\toksdef
126	11	\luadef
126	12	\integerdef
126	13	\parameterdef
126	14	\positdef
126	15	\dimensiondef
126	16	\gluespecdef
126	17	\mugluespecdef
126	18	\fontspecdef
126	19	\specificationdef
127	0	\edef
127	1	\def
127	2	\xdef
127	3	\gdef
127	4	\edefcsname
127	5	\defcsname
127	6	\xdefcsname
127	7	\gdefcsname
127	8	\cdef
127	9	\cdefcsname
128	0	\setbox
130	0	\expandafter
130	1	\unless
130	2	\futureexpand
130	3	\futureexpandis
130	4	\futureexpandisap
130	5	\expandafterspaces
130	6	\expandafterpars
130	7	\expandtoken
130	8	\expandcstoken
130	9	\expand
130	10	\expandtoks
130	11	\expandactive
130	12	\semiexpand
130	13	\expandedafter
130	14	\expandparameter
131	0	\noexpand
132	0	\input
132	1	\eofinput
132	2	\endinput
132	3	\scantokens
132	4	\scantextokens
132	5	\tokenized
132	6	\retokenized
132	7	\quitloop
132	8	\quitloopnow
135	0	\beginlocalcontrol
135	1	\localcontrol
135	2	\localcontrolled
135	3	\localcontrolledloop
135	4	\expandedloop
135	5	\unexpandedloop
135	6	\localcontrolledrepeat
135	7	\expandedrepeat
135	8	\unexpandedrepeat
135	9	\localcontrolledendless
135	10	\expandedendless
135	11	\unexpandedendless
136	2	\fi
136	3	\else
136	4	\or
136	5	\orelse
136	6	\orunless
136	7	\if
136	8	\ifcat
136	9	\ifnum
136	10	\ifabsnum
136	11	\ifzeronum
136	12	\ifintervalnum
136	13	\iffloat
136	14	\ifabsfloat
136	15	\ifzerofloat
136	16	\ifintervalfloat
136	17	\ifdim
136	18	\ifabsdim
136	19	\ifzerodim
136	20	\ifintervaldim
136	21	\ifodd
136	22	\ifvmode
136	23	\ifhmode
136	24	\ifmmode
136	25	\ifinner
136	26	\ifvoid
136	27	\ifhbox
136	28	\ifvbox
136	29	\iftok
136	30	\ifcstok
136	31	\ifx
136	32	\iftrue
136	33	\iffalse
136	34	\ifchknum
136	35	\ifchknumber
136	36	\ifnumval

136	37	\ifcmpnum
136	38	\ifchkdim
136	39	\ifchkdimension
136	40	\ifdimval
136	41	\ifcmpdim
136	42	\ifcase
136	43	\ifdefined
136	44	\ifcsname
136	45	\ifincsname
136	46	\iffontchar
136	47	\ifcondition
136	48	\ifflags
136	49	\ifempty
136	50	\ifrelax
136	51	\ifboolean
136	52	\ifnumexpression
136	53	\ifdimexpression
136	54	\iflastnamedcs
136	55	\ifmathparameter
136	56	\ifmathstyle
136	57	\ifarguments
136	58	\ifparameters
136	59	\ifparameter
136	60	\ifhastok
136	61	\ifhastoks
136	62	\ifhasxtoks
136	63	\ifhaschar
136	64	\ifinsert
136	65	\ifinalignment
136	66	\ifcramped
137	0	\csname
137	1	\lastnamedcs
137	2	\begincsname
137	3	\futurecsname
138	0	\number
138	1	\tointeger
138	2	\tohexadecimal
138	3	\toscaled
138	4	\tosparsescaled
138	5	\todimension
138	6	\tosparsedimension
138	7	\tomathstyle
138	8	\directlua
138	9	\luafunction
138	10	\luabytecode
138	11	\expanded
138	12	\semiexpanded
138	13	\string
138	14	\csstring
138	15	\csactive
138	16	\csnamestring
138	17	\detokenized
138	18	\detokened
138	19	\romannumeral
138	20	\meaning
138	21	\meaningfull
138	22	\meaningless
138	23	\meaningasis
138	24	\meaningful
138	25	\meaningles
138	26	\tocharacter
138	27	\luaescapestring
138	28	\fontname
138	29	\fontspecifiedname
138	30	\jobname
138	31	\formatname
138	32	\luatexbanner
138	33	\fontidentifier
139	0	\the
139	1	\thewithoutunit
139	2	\detokenize
139	3	\expandeddetokenize
139	4	\protecteddetokenize
139	5	\protectedexpandeddetokenize
139	6	\unexpanded
140	0	\currentmarks
140	1	\topmarks
140	2	\firstmarks
140	3	\botmarks
140	4	\splitfirstmarks
140	5	\splitbotmarks
140	6	\topmark
140	7	\firstmark
140	8	\botmark
140	9	\splitfirstmark
140	10	\splitbotmark

This is a curious one: it returns the number of steps that a hash lookup took:

```
function token.locatemacro ( <t:string> name )
    return <t:integer> - steps
end
```

We used this helper when deciding on a reasonable hash size. Of the many primitives there are a few that need more than one lookup step:

steps	total	macros
1	1117	...
2	14	boxshrink dsplit dump everytab fontcharic fontmathcontrol glet glueshrink if lower number pagestretch tabskip vfil
3	3	cr etoksapp gluestretch

libraries

16 Libraries

16.1 Introduction

The engines has quite some libraries built in of which some are discussed in dedicated chapters. Not all libraries will be detailed here, for instance, so called optional libraries depend on system libraries and usage is wrapped in modules because we delegate as much as possible to Lua.

16.2 Third party

There is not much to tell here other than it depends on the Lua symbols being visible and the Lua version matching. We don't use this in ConTEXt and have a different mechanism instead: optional libraries.

16.3 Core

The core libraries are those that interface with TEX and MetaPost, these are discussed in dedicated chapters:

chapter	library
Lua	`lua luac`
TEX	`status tex texio`
MetaPost	`mp`
Nodes	`node`
Tokens	`token`
Callbacks	`callback`
Fonts	`font`
Languages	`language`
Libraries	`library`

Some, like `node`, `token` and `tex` provide a lot of functions but most are used in more higher level ConTEXt specific functions and interfaces. This means that in the code you will more often font `nodes` and `tokens` being used as well as functions that the macro package adds to the various built-in libraries.

16.4 Auxiliary

16.4.1 Extensions

These are the libraries that are needed to implement various subsystems, like for instance the backend and image inclusion. Although much can be done in pure Lua for performance reasons helpers make sense. However, we try to minimize this, which means that for instance the `zip` library provides what we need for (de)compressing for instance pdf streams but that unzipping files is done with Lua code wrapped around the core zip routines. The same is true for png inclusion: all that was done in pure Lua but a few critical helpers were translated to C.

Some libraries extend existing ones, like for instance `file`, `io` and `os` and `string`.

16.4.2 Extra file helpers

The original `lfs` module has been adapted a bit to our needs but for practical reasons we kept the namespace. In LuaMetaTEX we operate in utf8 so for MS Windows system interfaces we convert from and to Unicode16.

The `attributes` checker returns a table with details.

```
function lfs.attributes ( <t:string> name )
    return <t:table> -- details
end
```

The table has the following fields:

field	type	meaning
`mode`	string	`file directory link other`
`size`	integer	bytes
`modification`	integer	time
`access`	integer	time
`change`	integer	time
`permissions`	string	`rwxrwxrwx`
`nlink`	integer	number of links

If you're not interested in details, then the next calls are more efficient:

```
function lfs.isdir           ( <t:string> name ) return <t:boolean> end
function lfs.isfile          ( <t:string> name ) return <t:boolean> end
function lfs.iswriteabledir  ( <t:string> name ) return <t:boolean> end
function lfs.iswriteablefile ( <t:string> name ) return <t:boolean> end
function lfs.isreadabledir   ( <t:string> name ) return <t:boolean> end
function lfs.isreadablefile  ( <t:string> name ) return <t:boolean> end
```

The current (working) directory is fetch with:

```
function lfs.currentdir ( )
    return <t:string> -- directory
end
```

These three return `true` is the action was a success:

```
function lfs.chdir ( <t:string> name ) return <t:boolean> end
function lfs.mkdir ( <t:string> name ) return <t:boolean> end
function lfs.rmdir ( <t:string> name ) return <t:boolean> end
```

Here the second and third argument are optional:

```
function lfs.touch (
    <t:string>  name,
    <t:integer> accesstime,
    <t:integer> modificationtime
)
    return <t:boolean> -- success
```

```
end
```

The `dir` function is a traverser which in addition to the name returns some more properties. Keep in mind that the traverser loops over a directory and that it doesn't run well when used nested. This is a side effect of the operating system. It is also the reason why we return some properties because querying them via `attributes` would interfere badly. The directory iterator has two variants:

```
for
    <t:string> name,
    <t:string> mode
in lfs.dir (
    <t:string> name
)
    -- actions
end
```

This one provides more details:

```
for
    <t:string>  name,
    <t:string>  mode,
    <t:integer> size,
    <t:integer> mtime
in lfs.dir (
    <t:string>  name,
    <t:true>
)
    -- actions
end
```

Here the boolean indicates if we want a symlink (`true`) or hard link (`false`).

```
function lfs.link (
    <t:string>  source,
    <t:string>  target,
    <t:boolean> symlink
)
    return <t:boolean> -- success
end
```

The next one is sort of redundant but explicit:

```
function lfs.symlink (
    <t:string>  source,
    <t:string>  target,
)
    return <t:boolean> -- success
end
```

Helpers like these are a bit operating system and user permission dependent:

```
function lfs.setexecutable ( <t:string> name )
```

```
    return <t:boolean> -- success
end

function lfs.symlinktarget ( <t:string> name )
    return <t:string> -- target
end
```

16.4.3 Reading from a file

Because we load fonts in Lua and because these are binary files we have some helpers that can read integers of various kind and some more. Originally we did this in pure Lua, which actually didn't perform that bad but this is of course more efficient.

We have readers for signed and unsigned, little and big endian. All return a (64 bit) Lua integer.

```
function fio.readcardinal1 ( <t:file> handle ) return <t:integer> end
function fio.readcardinal2 ( <t:file> handle ) return <t:integer> end
function fio.readcardinal3 ( <t:file> handle ) return <t:integer> end
function fio.readcardinal4 ( <t:file> handle ) return <t:integer> end

function fio.readcardinal1le ( <t:file> handle ) return <t:integer> end
function fio.readcardinal2le ( <t:file> handle ) return <t:integer> end
function fio.readcardinal3le ( <t:file> handle ) return <t:integer> end
function fio.readcardinal4le ( <t:file> handle ) return <t:integer> end

function fio.readinteger1 ( <t:file> handle ) return <t:integer> end
function fio.readinteger2 ( <t:file> handle ) return <t:integer> end
function fio.readinteger3 ( <t:file> handle ) return <t:integer> end
function fio.readinteger4 ( <t:file> handle ) return <t:integer> end

function fio.readinteger1le ( <t:file> handle ) return <t:integer> end
function fio.readinteger2le ( <t:file> handle ) return <t:integer> end
function fio.readinteger3le ( <t:file> handle ) return <t:integer> end
function fio.readinteger4le ( <t:file> handle ) return <t:integer> end
```

These float readers are rather specific for fonts:

```
function fio.readfixed2 ( <t:file> handle ) return <t:number>  end
function fio.readfixed4 ( <t:file> handle ) return <t:number>  end
function fio.read2dot14 ( <t:file> handle ) return <t:number>  end
```

Of these two the first reads a line and the second a string the C way, so ending with a newline and null character:

```
function fio.readcline   ( <t:file> handle ) return <t:string> end
function fio.readcstring ( <t:file> handle ) return <t:string> end
```

The next set of readers reads multiple integers in one call:

```
function fio.readbytes (
    <t:file> handle
)
```

```
    return <t:integer> -- one or more
end

function fio.readintegertable (
    <t:file>    handle,
    <t:integer> size,
    <t:integer> bytes
)
    return <t:table>
end

function fio.readcardinaltable (
    <t:file>    handle,
    <t:integer> size,
    <t:integer> bytes
)
    return <t:table>
end

function fio.readbytetable (
    <t:file> handle
)
    return <t:table>
end
```

In case we need a random access the following have to be used:

```
function fio.setposition  ( <t:file> handle, <t:integer> ) return <t:integer> end
function fio.getposition  ( <t:file> handle              ) return <t:integer> end
function fio.skipposition ( <t:file> handle, <t:integer> ) return <t:integer> end
```

The library also provide a few writers:

```
function fio.writecardinal1 ( <t:file> handle, <t:integer> value ) end
function fio.writecardinal2 ( <t:file> handle, <t:integer> value ) end
function fio.writecardinal3 ( <t:file> handle, <t:integer> value ) end
function fio.writecardinal4 ( <t:file> handle, <t:integer> value ) end

function fio.writecardinal1le ( <t:file> handle, <t:integer> value ) end
function fio.writecardinal2le ( <t:file> handle, <t:integer> value ) end
function fio.writecardinal3le ( <t:file> handle, <t:integer> value ) end
function fio.writecardinal4le ( <t:file> handle, <t:integer> value ) end
```

16.4.4 Reading from a string

These readers take a string and position. We could have used a userdata approach but it saves little. (Nowadays we can more easily store the position with the userdata so maybe some day . . .).

```
function sio.readcardinal1 ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readcardinal2 ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readcardinal3 ( <t:string> s, <t:integer> p ) return <t:integer> end
```

```
function sio.readcardinal4 ( <t:string> s, <t:integer> p ) return <t:integer> end

function sio.readcardinal1le ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readcardinal2le ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readcardinal3le ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readcardinal4le ( <t:string> s, <t:integer> p ) return <t:integer> end

function sio.readinteger1 ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readinteger2 ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readinteger3 ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readinteger4 ( <t:string> s, <t:integer> p ) return <t:integer> end

function sio.readinteger1le ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readinteger2le ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readinteger3le ( <t:string> s, <t:integer> p ) return <t:integer> end
function sio.readinteger4le ( <t:string> s, <t:integer> p ) return <t:integer> end
```

Here are the (handy for fonts) float readers:

```
function sio.readfixed2 ( <t:string> s, <t:integer> p ) return <t:number> end
function sio.readfixed4 ( <t:string> s, <t:integer> p ) return <t:number> end
function sio.read2dot14 ( <t:string> s, <t:integer> p ) return <t:number> end
```

A C line (terminated by a newline) and string (terminated by null) are read by:

```
function sio.readcline   ( <t:string> s, <t:integer> p ) return <t:string> end
function sio.readcstring ( <t:string> s, <t:integer> p ) return <t:string> end

function sio.readbytes (
    <t:string>  str,
    <t:integer> pos
)
    return <t:integer> -- one or more
end

function sio.readintegertable (
    <t:string>  str,
    <t:integer> pos,
    <t:integer> size,
    <t:integer> bytes
)
    return <t:table>
end

function sio.readcardinaltable (
    <t:string>  str,
    <t:integer> pos,
    <t:integer> size,
    <t:integer> bytes
)
    return <t:table>
end
```

```
function sio.readbytetable (
    <t:string>  str,
    <t:integer> pos
)
    return <t:table>
end
```

Here are a few straightforward converters:

```
function sio.tocardinal1 ( <t:string> ) return <t:integer> end
function sio.tocardinal2 ( <t:string> ) return <t:integer> end
function sio.tocardinal3 ( <t:string> ) return <t:integer> end
function sio.tocardinal4 ( <t:string> ) return <t:integer> end
```

```
function sio.tocardinal1le ( <t:string> ) return <t:integer> end
function sio.tocardinal2le ( <t:string> ) return <t:integer> end
function sio.tocardinal3le ( <t:string> ) return <t:integer> end
function sio.tocardinal4le ( <t:string> ) return <t:integer> end
```

16.4.5 Extra file helpers

This function gobble characters upto a newline. When characters are gobbled. `true` is returned when we end up at a newline or when something is gobbled before the file ends, other wise we get `false`. A `nil` return value indicates a bad handle.

```
function io.gobble( <t:file> )
    return <t:boolean> | <t:nil>
end
```

Function like type io.open `io.popen` are patched to support files on MS Windows that use wide Unicode.

16.4.6 Extra operating system helpers

The `os` library has a few extra functions and variables so for complete overview you need to look in the Lua manual.

We can sleep for the given number of seconds. When the optional `units` arguments is (for instance) 1000 we assume milliseconds.

```
function os.sleep (
    <t:integer>  seconds,
    <t:integers> units
)
    -- no return values
end
```

The `os.uname` function returns a table with specific operating system information acquired at runtime. The fields in the returned table are: `machine`, `nodename`, `release`, `sysname`, `version`.

```
function os.uname ( )
```

```
    return <t:table>
```

The os.gettimeofday function returns the current 'Unix time', but as a float. Keep in mind that there might be platforms where this function is not available.

```
function os.gettimeofday ( )
    return <t:number>
end
```

When we execute a command the return code is returned. Interpretation is up to the caller.

```
function os.execute ( <t:string> )
    return <t:integer> -- return code
end
```

This one enable interpreting ansi escape sequences in the console. It is only implemented for MS Windows. In ConTEXt you can run with --ansi.

```
function os.enableansi ( )
    return <t:boolean>
end
```

This one only returns something useful for MS Windows. One can of course just set your the system for utf8. It's just a reporter meant for debugging issues.

```
function os.getcodepage ( )
    return
        <t:integer> oemcodepage,
        <t:integer> applicationcodepage
end
```

The os.setenv function sets a variable in the environment. Passing nil instead of a value string will remove the variable.

```
function os.setenv (
    <t:string> key,
    <t:string> value
)
    -- no return values
end
```

The possible values of os.type are: unix, windows.

```
local currenttype = os.type
```

The os.name string gives a more precise indication of the operating system. The possible values are: bsd, freebsd, generic, gnu, linux, macosx, windows.

```
local currentname = os.name
```

On MS Windows the original os.rename, os.remove and os.getenv functions are replaced by variants that interface to and convert from Unicode16 to utf8.

16.4.7 Extra string helpers

The `string` library has gotten a couple of extra functions too, some of which are iterators. There are some Unicode related helpers too. When we started Lua had no utf8 function, now it has a few, but we keep using our own, if only because they were there before. We also add plenty extra functions in the `string` name space at the Lua end.

This first function runs over a string and pick sup single characters:

```
for <t:string> c in string.characters ( <t:string> s ) do
    -- some action
end
```

```
\startluacode
for c in string.characters("τεχ") do
    context("[%02X]",string.byte(c))
end
\stopluacode
```

gives: [CF][84][CE][B5][CF][87].

```
for <t:string> l, <t:string> r in string.characterpairs ( <t:string> s ) do
    -- some action
end
```

```
\startluacode
for l, r in string.characterpairs("τεχ") do
    context("[%02X %02X]",string.byte(l),string.byte(r))
end
\stopluacode
```

gives: [CF 84][CE B5][CF 87].

```
for <t:string> c in string.utfcharacters( <t:string> s ) do
    -- some action
end
```

```
\startluacode
for c in string.utfcharacters("τεχ") do
    context("[%s]",c)
end
\stopluacode
```

gives: [τ][ε][χ].

Instead of getting strings back we can also get integers.

```
for <t:integer> c in string.bytes ( <t:string> s ) do
    -- some action
end
```

```
\startluacode
for b in string.bytes("τεχ") do
```

```
    context("[%02X]",b)
end
\stopluacode
```

gives: [CF][84][CE][B5][CF][87].

```
for <t:integer> l, <t:integer> r in string.bytepairs ( <t:string> s ) do
    -- some action
end
```

```
\startluacode
for l, r in string.bytepairs("τεχ") do
    context("[%02X %02X]",l,r)
end
\stopluacode
```

gives: [CF 84][CE B5][CF 87].

```
for <t:integer> u in string.utfvalues( <t:string> s ) do
    -- some action
end
```

```
\startluacode
for c in string.utfvalues("τεχ") do
    context("[%U]",c)
end
\stopluacode
```

gives: [U+003C4][U+003B5][U+003C7].

The `bytetable` function splits a string in bytes.

```
function string.bytetable ( <s:string> s) do
    return <t:table> -- with bytes
end
```

Here is a line splitter:

```
function string.linetable  ( <s:string> s) do
    return <t:table> -- with lines
end
```

This one converts an integer (code point) into an utf string:

```
function string.utfcharacter ( <t:string> s )
    return <t:string>
end
```

We also have a variant that takes a table. The table can have integers, strings, and subtables.

```
function string.utfcharacter ( <t:table> s )
    return <t:string>
end
```

This an utf8 variant of `string.byte` and it returns the code points of the split on the stack.

```
function string.utfvalue ( <t:string> s )
    return <t:integer> -- zero or more
end
```

Instead of a list on the stack you can get a table:

```
function string.utfvaluetable ( <t:string> s )
    return <t:table> -- indexed
end
```

The name says it all:

```
function string.utflength ( <tr:string> s )
    return <t:integer>
end
```

Here we split a string in characters that are collected in an indexed table:

```
function string.utfcharactertable ( <t:string> s )
    return <t:table> -- indexed
end
```

In ConTEXt we mostly use `string.formatters` which is often more efficient then `string.format` and also has additional formatting options, one being for instance `N` which is like `f` but strips trailing zero and returns efficient zeros and ones. Here is a similar low level formatter:

```
function string.f6 ( <t:number> n )
    return <t:string>
end

function string.f6 ( <t:number> n,  <t:string> f )
    return <t:string>
end
```

In the first case it returns a string with at most 6 digits while the second one uses given format but tail strips the result.

```
function string.tounicode16 ( <t:integer> code  ) return <t:string> end

function string.toutf8  ( <t:table>   codes ) return <t:string> end
-------- string.toutf16 ( <t:table>   codes ) return <t:string> end
function string.toutf32 ( <t:table>   codes ) return <t:string> end
```

The next one has quite some variation in calling:

```
function string.utf16toutf8 ( <t:string> str, <t:true> )
    return <t:string>  -- big endian
end

function string.utf16toutf8 ( <t:string> str, <t:false> )
    return <t:string> -- little endian
end
```

```
function string.utf16toutf8 ( <t:string> str, <t:nil>, <t:true> )
    return <t:string> -- check bom, default to big endian
end

function string.utf16toutf8 ( <t:string> str, <t:nil>, <t:false> )
    return <t:string> end -- check bom, default to little endian
end

function string.utf16toutf8 ( <t:string> str, <t:nil>, <t:nil> )
    return <t:string> end -- check bom, default to little endian
end
```

The next packer is used for creating bitmaps:

```
function string.packrowscolumns ( <t:table> data )
    return <t:string>
end
```

For example:

```
\startluacode
local t = {
    { 65, 66, 67 },
    { 68, 69, 70 },
}
context(string.packrowscolumns(t))
\stopluacode
```

gives: ABCDEF

While:

```
\startluacode
local t = {
    { { 114, 103, 98 }, { 114, 103, 98 } },
    { { 114, 103, 98 }, { 114, 103, 98 } },
}

context(string.packrowscolumns(t))
\stopluacode
```

gives: rgbrgbrgbrgb

A string with hexadecimals can be converted with the following. Spaces are ignored. We use this for instance in the MetaPost potrace interface to permits nice input.

```
function string.hextocharacters ( <t:string> data )
    return <t:string>
end
```

So:

```
\startluacode
```

```
local t = [[
  414243 44 4546 47
  414243 44 4546 47
]]

context(string.hextocharacters(t))
\stopluacode
```

gives: ABCDEFGABCDEFG

These take strings and return integers:

```
function string.octtointeger ( <t:string> octstr ) return <t:integer> end
function string.dectointeger ( <t:string> decstr ) return <t:integer> end
function string.hextointeger ( <t:string> hexstr ) return <t:integer> end
function string.chrtointeger ( <t:string> chrstr ) return <t:integer> end
```

16.4.8 Extra table helpers

This returns the keys of the given table:

```
function table.getkeys ( < t:table> )
    return <t:table>
end
```

16.4.9 Byte encoding and decoding

We use some helpers from `pplib`.

```
function basexx.encode16  ( <t:string> str, <t:boolean> newline )
    return <t:string>
end
function basexx.encode64  ( <t:string> str, <t:boolean> newline )
    return <t:string>
end
function basexx.encode85  ( <t:string> str, <t:boolean> newline )
    return <t:string>
end

function basexx.decode16  ( <t:string> str ) return <t:string> end
function basexx.decode64  ( <t:string> str ) return <t:string> end
function basexx.decode85  ( <t:string> str ) return <t:string> end

function basexx.encodeRL  ( <t:string> str ) return <t:string> end
function basexx.decodeRL  ( <t:string> str ) return <t:string> end

function basexx.encodeLZW ( <t:string> str ) return <t:string> end
function basexx.decodeLZW ( <t:string> str ) return <t:string> end
```

The last two functions accept an optional bitset with coder flags that we leave for the user to ponder about. The `newline` directive in the first three is optional.

16.4.10 png decoding

These function started out as pure Lua functions (extrapolated from the descriptions in the standard) but eventually became library helpers. It is worth noticing that pdf supports jpeg directly so there we can just use Lua to interpret the file and pass relevant data. Support for png is actually just support for png compression, so there we need to do more work and filter the content:

```
function decode.applyfilter (
    <t:string>  data,
    <t:integer> nx,
    <t:integer> ny,
    <t:integer> slice
)
    return <t:string>
end
```

We also need to split off the mask as ie becomes a separate object:

```
function decode.splitmask (
    <t:string>  data,
    <t:integer> nx,
    <t:integer> ny,
    <t:integer> bpp,
    <t:integer> bytes
)
    return
        <t:string>, -- bitmap
        <t:string>  -- mask
end
```

If present we have to deinterlace:

```
function decode.interlace (
    <t:string>  data,
    <t:integer> nx,
    <t:integer> ny,
    <t:integer> slice,
    <t:integer> pass
)
    return <t:string>
end
```

And maybe expand compressed:

```
function decode.expand (
    <t:string>  data,
    <t:integer> nx,
    <t:integer> ny,
    <t:integer> parts,
    <t:integer> xline,
    <t:integer> factor
```

```
)
    return <t:string>
end
```

These are just helpers that permit integration in the ConTEXt graphic ecosystem (including MetaPost):

```
function decode.tocmyk ( <t:string data )
    return <t:string>
end
```

For usage see the ConTEXt sources.

```
function decode.tomask (
    <t:string>  content,
    <t:string>  mapping,
    <t:integer> xsize,
    <t:integer> ysize,
    <t:integer> colordepth
)
    return <t:string>
end
```

There are to variants:

```
function decode.makemask (
    <t:string>  content,
    <t:integer> mapping
)
    return <t:string>
end
```

```
function decode.makemask (
    <t:string> content,
    <t:table>  mapping
)
    return <t:string>
end
```

16.4.11 MD5 hashing

In the meantime we use some helpers from `pplib` because we have that anyway. These are useful when we need a reasonable unique hash of limited length:

```
function md5.sum ( <t:string> ) return <t:string> end
function md5.hex ( <t:string> ) return <t:string> end
function md5.HEX ( <t:string> ) return <t:string> end
```

Using a hexadecimal representation of the 16 byte calculated checksum is less sensitive for escaping. This:

```
\startluacode
context.type(md5.HEX("normally this is unique enough"))
```

```
\stopluacode
```

gives: 3C1F10E596B1D1972CF5D1078796C97D.

16.4.12 SHA2 hashing

Because pplib comes with some SHA2 support we can borrow its helpers instead of the Lua code we used before (which was anyway fun to write).

```
function sha2.digest256 ( <t:string> data ) return <t:string> end
function sha2.digest384 ( <t:string> data ) return <t:string> end
function sha2.digest512 ( <t:string> data ) return <t:string> end
function sha2.sum256    ( <t:string> data ) return <t:string> end
function sha2.sum384    ( <t:string> data ) return <t:string> end
function sha2.sum512    ( <t:string> data ) return <t:string> end
function sha2.hex256    ( <t:string> data ) return <t:string> end
function sha2.hex384    ( <t:string> data ) return <t:string> end
function sha2.hex512    ( <t:string> data ) return <t:string> end
function sha2.HEX256    ( <t:string> data ) return <t:string> end
function sha2.HEX384    ( <t:string> data ) return <t:string> end
function sha2.HEX512    ( <t:string> data ) return <t:string> end
```

The number refers to bytes, so with 256 we get a 32 byte hash that we show in hexadecimal because that is less sensitive for escaping:

```
\startluacode
context.type(sha2.HEX256("normally this is unique enough"))
\stopluacode
```

gives: D1F1E826197E80BB3860BA279C2D46652C37D4D56B5B7CFD7881450FCF0161F4.

16.4.13 AES encryption

In the next encryption functions the key should be 16, 24 or 32 bytes long.

```
function aes.encode (
    <t:string> data,
    <t:string> key
)
    return <t:string>
end

function aes.decode (
    <t:string> data,
    <t:string> key
)
    return <t:string>
end
```

This returns a string. The default length is 16; the optional length is limited to 32.

```
function aes.random ( <t:integer> length )
    return <t:string>
end
```

Here is an example:

```
\startluacode
context.type ( basexx.encode16 ( aes.encode (
    "normally this is unique enough",
    "The key of live!"
) ) )
\stopluacode
```

This gives: `6A19333F6D2D25B4FF47C5B5631D825696454361601673C1ADFFE7161C7F00C3`, where we hexed the result because it is unlikely to be valid utf8.

16.4.14 ZIP (de)compression

We provide the minimum needed to support compression in the backend but even this limited set makes it possible to implement a zip file compression utility which is indeed what we do in ConTEXt. We use `minizip` as codebase, without the zip utility code. The meaning and application of the various arguments can be found (and are better explained) on the internet.

```
function xzip.compress (
    <t:string>  data,
    <t:integer> compresslevel,
    <t:integer> method,
    <t:integer> window,
    <t:integer> memory,
    <t:integer> strategy
)
    return <t:string>
end

function xzip.compresssize (
    <t:string>  data,
    <t:integer> buffersize,
    <t:integer> compresslevel,
    <t:integer> window
)
    return <t:string>
end

function xzip.decompress (
    <t:string>  data,
    <t:integer> window
)
    return <t:string>
end

function xzip.decompresssize (
```

```
    <t:string>  data,
    <t:integer> targetsize,
    <t:integer> window
)
    return <t:string>
end

function xzip.adler32 (
    <t:string>  buffer,
    <t:integer> checksum
)
    return <t:integer>
end

function xzip.crc32 (
    <t:string>  buffer,
    <t:integer> checksum
)
    return <t:integer>
end
```

16.4.15 Potrace

The excellent potrace manual explains everything about this library therefore here we just show the interface. Possible fields in specification are: `bytes`, `height`, `negate`, `nx`, `ny`, `swap`, `value`, `width`

```
function potrace.new ( <t:table> specification )
    return <t:userdata> -- instance
end

function potrace.free ( <t:userdata> instance)
    -- no return values
end
```

The process is controlled by the specification: `negate`, `optimize`, `policy`, `size`, `threshold`, `tolerance`, `value`, where permitted policy values are `black`, `left`, `majority`, `minority`, `random`, `right`, `white`.

```
function potrace.process ( <t:userdata> instance, <t:table> specification )
    return <t:boolean> -- success
end
```

Results are collected in a table that we can feed into MetaPost, The table has subtables per traced shape and these contain indexed tables with two (pair) or six (curve) entries. There is a boolean `sign` field and an integer `index` field. In the next function only the first argument is mandate.

```
function potrace.totable  (
    <t:userdata> instance,
    <t:boolean>  debug,
    <t:integer>  first,
    <t:integer>  last
```

```
)
    return <t:table>
end
```

16.4.16 Sparse hashes

The sparse library is just there because we use similar code to store all these character related codes that way (`\lccode`) and such). The entries can be 1, 2 or 4 bytes wide.

```
function sparse.new (
    <t:integer> bytes,
    <t:integer> default
)
    return <t:userdata>
end
```

You set a value by index. Optionally there can be the `"global"` keyword before the second argument.

```
function sparse.set (
    <t:userdata> instance,
    <t:integer>  index,
    <t:integer>  value
)
    return <t:integer>
end
```

We get back integers as that is what we store:

```
function sparse.get ( <t:userdata> instance ) return <t:integer> end
function sparse.min ( <t:userdata> instance ) return <t:integer> end
function sparse.max ( <t:userdata> instance ) return <t:integer> end
```

The range is fetched with:

```
function sparse.range ( <t:userdata> instance )
    return
        <t:integer>, -- min
        <t:integer>  -- max
end
```

We can iterate over the hash:

```
for
    <t:integer> index,
    <t:integer> value
in sparse.traverse (
    <t:userdata> instance
) do
    -- actions
end
```

This is a somewhat strange one but it permits packing all values in a string. It's another way to create bitmaps.

```
function sparse.concat (
    <t:userdata> instance
    <t:integer>  min,
    <t:integer>  max,
    <t:integer>  how  -- 0=byte, 1=lsb 2=msb
)
    return <t:string>
end
```

Setting values obeys grouping in TEX, but we can restore any time:

```
function sparse.restore ( <t:userdata> instance )
    -- nothing to return
end
```

We can also wipe all values:

```
function sparse.wipe (<t:userdata> instance )
    -- nothing to return
end
```

16.4.17 Posits

We implement posits as userdata . We use the library from the posit team, although it is not complete so we might roll out our own variant (as we need less anyway). The advance of userdata is that we can use the binary and relation operators.

Here are the housekeeping functions. Some are more tolerant with respect to arguments, take the allocator:

```
function posit.new (              ) return <t:posit> end
function posit.new ( <t:string> s ) return <t:posit> end
function posit.new ( <t:number> n ) return <t:posit> end
```

When a posit is expected a number or string is also accepted which is then converted to a posit.

```
function posit.copy      ( <t:posit>  p ) return <t:posit>   end
function posit.tostring  ( <t:posit>  p ) return <t:string>  end
function posit.tonumber  ( <t:posit>  p ) return <t:number>  end
function posit.integer   ( <t:posit>  p ) return <t:integer  end
function posit.rounded   ( <t:posit>  p ) return <t:integer> end
function posit.toposit   ( <t:number> n ) return <t:posit>   end
function posit.fromposit ( <t:posit>  p ) return <t:number>  end

function posit.NaN ( <t:posit> p ) return <t:boolean> end
function posit.NaR ( <t:posit> p ) return <t:boolean> end
```

Here are the logical operators:

```
function posit.bor  ( <t:posit> p1, <t:posit> p2 ) return <t:posit> end
function posit.bxor ( <t:posit> p1, <t:posit> p2 ) return <t:posit> end
function posit.band ( <t:posit> p1, <t:posit> p2 ) return <t:posit> end
```

Ans shifters:

```
function posit.shift  ( <t:posit> p1, <t:integer> n ) return <t:posit> end
function posit.rotate ( <t:posit> p,  <t:integer> n ) return <t:posit> end
```

There is a limited repertoire of math functions (basically what we needed for MetaPost):

```
function posit.abs   ( <t:posit> p ) return <t:posit> end
function posit.conj  ( <t:posit> p ) return <t:posit> end
function posit.acos  ( <t:posit> p ) return <t:posit> end
function posit.asin  ( <t:posit> p ) return <t:posit> end
function posit.atan  ( <t:posit> p ) return <t:posit> end
function posit.ceil  ( <t:posit> p ) return <t:posit> end
function posit.cos   ( <t:posit> p ) return <t:posit> end
function posit.exp   ( <t:posit> p ) return <t:posit> end
function posit.exp2  ( <t:posit> p ) return <t:posit> end
function posit.floor ( <t:posit> p ) return <t:posit> end
function posit.log   ( <t:posit> p ) return <t:posit> end
function posit.log10 ( <t:posit> p ) return <t:posit> end
function posit.log1p ( <t:posit> p ) return <t:posit> end
function posit.log2  ( <t:posit> p ) return <t:posit> end
function posit.logb  ( <t:posit> p ) return <t:posit> end
function posit.round ( <t:posit> p ) return <t:posit> end
function posit.sin   ( <t:posit> p ) return <t:posit> end
function posit.sqrt  ( <t:posit> p ) return <t:posit> end
function posit.tan   ( <t:posit> p ) return <t:posit> end
```

```
function posit.modf ( <t:posit> p )
    return
        <t:posit>,
        <t:posit>
end
```

```
function posit.min ( <t:posit> p1, <t:posit> p2 ) return <t:posit> end
function posit.max ( <t:posit> p1, <t:posit> p2 ) return <t:posit> end
function posit.pow ( <t:posit> p1, <t:posit> p2 ) return <t:posit> end
```

16.4.18 Complex numbers

```
function xcomplex.new ( )
    return <t:complex>
end
```

```
function xcomplex.new (
    <t:number> re,
    <t:number> im
)
```

```
    return <t:complex>
end

function xcomplex.tostring ( <t:complex> z )
    return <t:string>
end

function xcomplex.topair ( <t:complex> z )
    return
        <t:number>, -- re
        <t:number>  -- im
end
```

There is a bunch of functions that take a complex number:

```
function xcomplex.abs   ( <t:complex> z ) return <t:complex> end
function xcomplex.arg   ( <t:complex> z ) return <t:complex> end
function xcomplex.imag  ( <t:complex> z ) return <t:complex> end
function xcomplex.real  ( <t:complex> z ) return <t:complex> end
function xcomplex.onj   ( <t:complex> z ) return <t:complex> end
function xcomplex.proj  ( <t:complex> z ) return <t:complex> end
function xcomplex.exp   ( <t:complex> z ) return <t:complex> end
function xcomplex.log   ( <t:complex> z ) return <t:complex> end
function xcomplex.sqrt  ( <t:complex> z ) return <t:complex> end
function xcomplex.sin   ( <t:complex> z ) return <t:complex> end
function xcomplex.cos   ( <t:complex> z ) return <t:complex> end
function xcomplex.tan   ( <t:complex> z ) return <t:complex> end
function xcomplex.asin  ( <t:complex> z ) return <t:complex> end
function xcomplex.acos  ( <t:complex> z ) return <t:complex> end
function xcomplex.atan  ( <t:complex> z ) return <t:complex> end
function xcomplex.sinh  ( <t:complex> z ) return <t:complex> end
function xcomplex.cosh  ( <t:complex> z ) return <t:complex> end
function xcomplex.tanh  ( <t:complex> z ) return <t:complex> end
function xcomplex.asinh ( <t:complex> z ) return <t:complex> end
function xcomplex.acosh ( <t:complex> z ) return <t:complex> end
function xcomplex.atanh ( <t:complex> z ) return <t:complex> end

function xcomplex.pow   ( <t:complex> z1, <t:complex> z2 ) return <t:complex> end
```

We added the `cerf` functions but none can wonder if we should carry that burden around (instead of just assuming a library to be used).

The complex error function `erf(z)`:

```
function cerf.erf ( <t:complex> z )
    return <t:complex>
end
```

The complex complementary error function `erfc(z) = 1 - erf(z)`:

```
function cerf.erfc ( <t:complex> z )
    return <t:complex>
```

```
end
```

The underflow-compensating function `erfcx(z) = exp(z^2) erfc(z)`:

```
function cerf.erfcx ( <t:complex> z )
    return <t:complex>
end
```

The imaginary error function `erfi(z) = -i erf(iz)`:

```
function cerf.erfi ( <t:complex> z )
    return <t:complex>
end
```

Dawson's integral `D(z) = sqrt(pi)/2 * exp(-z^2) * erfi(z)`:

```
function cerf.dawson ( <t:complex> z )
    return <t:complex>
end
```

The convolution of a Gaussian and a Lorentzian:

```
function cerf.voigt (
    <t:number> n1,
    <t:number> n2,
    <t:number> n3
)
    return <t:number>
end
```

The half width at half maximum of the Voigt profile:

```
function cerf.voigt_hwhm (
    <t:number> n1,
    <t:number> n2
)
    return <t:number>
end
```

16.4.19 Decimal numbers

Because in MetaPost we support the decimal number system, we also provide this at the TeX end Apart from the usual support for operators there are some functions available.

```
function xdecimal.new ( )              return <t:decimal> end
function xdecimal.new ( <t:number> n ) return <t:decimal> end
function xdecimal.new ( <t:string> s ) return <t:decimal> end

function xdecimal.copy     ( <t:decimal> a ) return <t:decimal> end
function xdecimal.tostring ( <t:decimal> a ) return <t:string>  end
function xdecimal.tonumber ( <t:decimal> a ) return <t:number>  end

function xdecimal.setprecision ( <t:integer> digits )
```

```
    --nothing to return
end

function xdecimal.getprecision ( )
    return <t:integer>
end

function xdecimal.bor  ( <t:decimal> a, <t:decimal> b ) return <t:decimal> end
function xdecimal.bxor ( <t:decimal> a, <t:decimal> b ) return <t:decimal> end
function xdecimal.band ( <t:decimal> a, <t:decimal> b ) return <t:decimal> end

function xdecimal.shift  ( <t:decimal> a, <t:integer> n ) return <t:decimal> end
function xdecimal.rotate ( <t:decimal> a, <t:integer> n ) return <t:decimal> end

function xdecimal.abs   ( <t:decimal> a ) return <t:decimal> end
function xdecimal.trim  ( <t:decimal> a ) return <t:decimal> end
function xdecimal.conj  ( <t:decimal> a ) return <t:decimal> end
function xdecimal.abs   ( <t:decimal> a ) return <t:decimal> end
function xdecimal.sqrt  ( <t:decimal> a ) return <t:decimal> end
function xdecimal.ln    ( <t:decimal> a ) return <t:decimal> end
function xdecimal.log   ( <t:decimal> a ) return <t:decimal> end
function xdecimal.exp   ( <t:decimal> a ) return <t:decimal> end
function xdecimal.minus ( <t:decimal> a ) return <t:decimal> end
function xdecimal.plus  ( <t:decimal> a ) return <t:decimal> end

function xdecimal.min ( <t:decimal> a, <t:decimal> b ) return <t:decimal> end
function xdecimal.max ( <t:decimal> a, <t:decimal> b ) return <t:decimal> end
function xdecimal.pow ( <t:decimal> a, <t:decimal> b ) return <t:decimal> end
```

16.4.20 Math helpers

The xmath library provides function and a few constants:

```
local infinty    = xmath.inf
local notanumber = xmath.nan
local pi         = xmath.pi
```

There are more helpers than the average used needs. We also use these to extend the MetaPost repertoire.

```
function xmath.acos       ( <t:number> a )              return <t:number> end
function xmath.acosh      ( <t:number> a )              return <t:number> end
function xmath.asin       ( <t:number> a )              return <t:number> end
function xmath.asinh      ( <t:number> a )              return <t:number> end
function xmath.atan       ( <t:number> a )              return <t:number> end
function xmath.atan       ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.atan2      ( <t:number> a )              return <t:number> end
function xmath.atan2      ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.atanh      ( <t:number> a )              return <t:number> end
function xmath.cbrt       ( <t:number> a )              return <t:number> end
function xmath.ceil       ( <t:number> a )              return <t:number> end
```

```
function xmath.copysign   ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.cos        ( <t:number> a )               return <t:number> end
function xmath.cosh       ( <t:number> a )               return <t:number> end
function xmath.deg        ( <t:number> a )               return <t:number> end
function xmath.erf        ( <t:number> a )               return <t:number> end
function xmath.erfc       ( <t:number> a )               return <t:number> end
function xmath.exp        ( <t:number> a )               return <t:number> end
function xmath.exp2       ( <t:number> a )               return <t:number> end
function xmath.expm1      ( <t:number> a )               return <t:number> end
function xmath.fabs       ( <t:number> a )               return <t:number> end
function xmath.fdim       ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.floor      ( <t:number> a )               return <t:number> end
function xmath.fmax       ( ... )                        return <t:number> end
function xmath.fmin       ( ... )                        return <t:number> end
function xmath.fmod       ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.frexp      ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.gamma      ( <t:number> a )               return <t:number> end
function xmath.hypot      ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.isfinite   ( <t:number> a )               return <t:number> end
function xmath.isinf      ( <t:number> a )               return <t:number> end
function xmath.isnan      ( <t:number> a )               return <t:number> end
function xmath.isnormal   ( <t:number> a )               return <t:number> end
function xmath.j0         ( <t:number> a )               return <t:number> end
function xmath.j1         ( <t:number> a )               return <t:number> end
function xmath.jn         ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.ldexp      ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.lgamma     ( <t:number> a )               return <t:number> end
function xmath.l0         ( <t:number> a )               return <t:number> end
function xmath.l1         ( <t:number> a )               return <t:number> end
function xmath.ln         ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.log        ( <t:number> a [,b])           return <t:number> end
function xmath.log10      ( <t:number> a )               return <t:number> end
function xmath.log1p      ( <t:number> a )               return <t:number> end
function xmath.log2       ( <t:number> a )               return <t:number> end
function xmath.logb       ( <t:number> a )               return <t:number> end
function xmath.modf       ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.nearbyint  ( <t:number> a )               return <t:number> end
function xmath.nextafter  ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.pow        ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.rad        ( <t:number> a )               return <t:number> end
function xmath.remainder  ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.remquo     ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.round      ( <t:number> a )               return <t:number> end
function xmath.scalbn     ( <t:number> a, <t:number> b ) return <t:number> end
function xmath.sin        ( <t:number> a )               return <t:number> end
function xmath.sinh       ( <t:number> a )               return <t:number> end
function xmath.sqrt       ( <t:number> a )               return <t:number> end
function xmath.tan        ( <t:number> a )               return <t:number> end
function xmath.tanh       ( <t:number> a )               return <t:number> end
```

```
function xmath.tgamma     ( <t:number> a )              return <t:number> end
function xmath.trunc      ( <t:number> a )              return <t:number> end
function xmath.y0         ( <t:number> a )              return <t:number> end
function xmath.y1         ( <t:number> a )              return <t:number> end
function xmath.yn         ( <t:number> a )              return <t:number> end
```

```
function xmath.fma (
    <t:number> a,
    <t:number> b,
    <t:number> c
)
    return <t:number>
end
```

16.5 Optional

16.5.1 Loading

The optional libraries are (indeed) optional. Compilation of LuaMetaTEX doesn't depend on them being present. Loading (and binding) is delayed. In practice we only see a few being of interest and used, like `zint` for barcodes, `mysql` for database processing and `graphicmagick` for an occasional runtime conversion. Some are just there to show the principles and were used to test the interfaces and loading.

A library can be loaded, and thereby registered in the 'optional' namespace, assuming that `--permitloadlib` is given with:

```
function library.load (
    <t:string> filename,
    <t:string> openname,
)
    return
        <t:function>, -- target
        <t:string>    -- foundname
end
```

but there are no guarantees that it will work.

16.5.2 Management

```
foreign
optional
```

16.5.3 TDS (kpse)

The optional `kpse` library deals with lookups in the TEX Directory Structure and before it can be used it has to be initialized:

```
function optional.kpse.initialize ( <t:string> filename )
```

```
    return <t:boolean>
end
```

By setting the program name the library knows in what namespace to resolve filenames and variables.

```
function optional.kpse.set_program_name (
    <t:string> binaryname,
    <t:string> programname
)
    -- no return values
end
```

The main finder has one or more arguments. When the second and later arguments can be a boolean, string or number. The boolean indicates if the file must exist. A string sets the file type and a number does the same.

```
function optional.kpse.find_file(
    <t:string>  filename,
    <t:string>  filetype.
    <t:boolean> mustexist
)
    return <t:string>
end
```

You can also ask for a list of found files:

```
function optional.kpse.find_files (
    <t:string> userpath,
    <t:string> filename
)
    return <t:table>
end
```

These return variables, values and paths:

```
function optional.kpse.expand_path   ( <t:string> name ) return <t:string> end
function optional.kpse.expand_var    ( <t:string> name ) return <t:string> end
function optional.kpse.expand_braces ( <t:string> name ) return <t:string> end
function optional.kpse.var_value     ( <t:string> name ) return <t:string> end
```

If possible this returns the (first found) filename that is readable:

```
function optional.kpse.readable_file ( <t:string> filename )
    return <t:string>
end
```

The list of supported file types can be fetched with:

```
function optional.kpse.get_file_types ( )
    return <t:table>
end
```

16.5.4 Graphics

ghostscript

The ghostscript library has to be initialized:

```
function optional.ghostscript.initialize ( <t:string> filename )
    return <t:boolean>
end
```

A conversion is executed with the following command. Here the table is a mixed list of strings and numbers that represent the otherwise command like arguments.

```
function optional.ghostscript.execute ( <t:table> )
    return
        <t:boolean>, -- success
        <t:string>,  -- result
        <t:string>   -- message
end
```

graphicsmagick

The graphicsmagick library has to be initialized:

```
function optional.graphicsmagick.initialize ( <t:string> filename )
    return <t:boolean>
end
```

A conversion is executed with the following command.

```
function optional.graphicsmagick.execute (
    {
        inputfilename  = <t:string>,
        outputfilename = <t:string>,
        blur           = {
            radius = <t:number>,
            sigma  = <t:number>,
        },
        noise          - {
            type   = <t:integer>,
        },
    }
)
    return <t:boolean>
end
```

The noise types can be fetched with:

```
function optional.graphicsmagick.noisetypes ( )
    return <t:table>
end
```

imagemagick

The imagemagick library is initialized with:

```
function optional.imagemagick.initialize ( <t:string> filename )
    return <t:boolean>
end
```

After that you can execute convert commands. The options table is a sequence of strings, numbers and booleans that gets passes, in the same order, but where a boolean becomes one of the strings true or false.

```
function optional.imagemagick.execute (
    {
        inputfilename  = <t:string>,
        outputfilename = <t:string>,
        options        = <t:table>,
    }
)
    return <t:boolean>
end
```

zint

The zint library is initialized with:

```
function optional.zint.initialize ( <t:string> filename )
    return <t:boolean>
end
```

As with the other graphic libraries we execute a command but here we implement a converter a bit more specific because we want back a result that we can handle in a combination of TEX and MetaPost.

```
function optional.zint.execute (
    {
        code   = <t:integer>,
        text   = <t:string>,
        option = <t:string>, -- "square"
    }
)
    return <t:table>
end
```

We get back a table that has graphic components, where each components table can zero or more subtables.

```
result = {
    rectangles = {
        { <t:integer> x, <t:integer> y, <t:integer> w, <t:integer> h }, ...
    },
    hexagons = {
        { <t:integer> x, <t:integer> y, <t:integer> d }, ...
```

```
    },
    circles = {
        { <t:integer> x, <t:integer> y, <t:nteger> d }, ...
    },
    strings = {
        { <t:integer> x, <t:integer> y, <t:integer> s, <t:string> t }, ...
    }
}
```

16.5.5 Compression

lz4

The library is initialized with:

```
function optional.lz4.initialize ( )
    return <t:boolean> -- success
end
```

There are compressors and decompressors. If you want the more efficient decompressor, make sure to save the size with the compressed stream and pass that when decompressing.

```
function optional.lz4.compress (
    <t:string>  data,
    <t:integer> acceleration -- default 1
)
    return <t:string>
end

function optional.lz4.decompresssize (
    <t:string>  data,
    <t:integer> size
)
    return <t:string>
end
```

These are the frame based variants:

```
function optional.lz4.framecompress ( <t:string> data )
    return <t:string>
end

function optional.lz4.framedecompress ( return <t:string>  )
    return <t:string>
end
```

lzma

The library is initialized with:

```
function optional.lzma.initialize ( )
```

```
    return <t:boolean> -- success
end
```

The compressor can take an estimated size which makes it possible to preallocate a buffer.

```
function optional.lzma.compress (
    <t:string>  data,
    <t:integer> level,
    <t:integer> size   -- estimated
)
    return <t:string>
end
```

The decompressor can be told what the final size is which is more efficient.

```
function optional.lzma.decompress (
    <t:string>  data,
    <t:integer> size   -- estimated
)
    return <t:string>
end
```

lzo

The library is initialized with:

```
function optional.lzo.initialize ( )
    return <t:boolean> -- success
end
```

There is not much to tell about:

```
function optional.lzo.compress ( <t:string> data )
    return <t:string>
end
```

and

```
function optional.lzo.decompresssize (
    <t:string>  data,
    <t:integer> size
)
    return <t:string>
end
```

zstd

The library is initialized with:

```
function optional.zstd.initialize ( )
    return <t:boolean> -- success
end
```

The compressor:

```
function optional.zstd.compress (
    <t:string>  data,
    <t:integer> level
)
    return <t:string>
end
```

The decompressor:

```
function optional.zstd.decompress ( <t:string> data )
    return <t:string>
end
```

16.5.6 Databases

mysql

We start with the usual initializer:

```
function optional.mysql.initialize ( )
    return <t:boolean> -- success
end
```

Opening the database is done with:

```
function optional.mysql.open (
    <t:string>  database,
    <t:string>  username,
    <t:string>  password,
    <t:string>  host,
    <t:integer> port -- optional
)
    return <t:userdata> -- instance
end
```

The database is kept ‘open’ but can be closed with:

```
function optional.mysql.close ( <t:userdata> instance )
    -- no return values
end
```

A query is executed with:

```
function optional.mysql.execute (
    <t:userdata> instance,
    <t:string>   query,
    <t:function> callback
)
    return <t:boolean> -- success
```

```
end
```

The callback is a Lua function that looks like this:

```
function callback(nofcolumns,values,fields)
    ...
end
```

It gets called for every row of the result. The fields table is only filled the first time, if at all.

This interface is rather minimalistic but in ConTEXt we wrap all in a more advanced setup. It's among the oldest Lua code in the distribution and evolved with the possibilities (client as well as external libraries) and is quite performing also due to the use of templates, caching, built-in conversions etc.

If there is an error we can fetch the message with:

```
function optional.mysql.getmessage ( <t:userdata> instance )
    return <t:string> | <t:nil> -- last error message
end
```

postgress

This library has the same interface as the `mysql` interface, so it can be used instead.

sqlite

This library has the same interface as the `mysql` interface, so it can be used instead. The only function that differs is the opener:

```
function optional.sqlite.open ( <t:string> filename )
    return <t:userdata> -- instance
end
```

16.5.7 Whatever

cerf

This library is plugged in the `xcomplex` so there is no need to discuss it here unless we decide to move it to an optional loaded library, which might happen eventually (depends on need).

curl

The library is initialized with:

```
function optional.curl.initialize ( )
    return <t:boolean> -- success
end
```

The fetcher stays kind of close to how the library wants it so we have no fancy interface. We have pairs where the first member is an integer indicating the option. The library only has string and integer options so booleans are effective zeros or ones. A Lua boolean therefore becomes an integer.

```
function optional.curl.fetch (
    {
        <t:integer>, <t:string> | <t:integer> | <t:boolean>,
        ...
    }
)
end
```

A url can be (un)escaped:

```
function optional.curl.escape ( <t:string> data )
    return <t:string>
end
```

```
function optional.curl.unescape ( <t:string> data )
    return <t:string>
end
```

The current version of the library:

```
function optional.curl.getversion ( )
    return <t:string>
end
```

hb

This module is mostly there to help Idris Hamid (The Oriental TEX Project develop his fonts in such away that they work with other libraries (also uniscribe). We need to initialize this library with the following function. Best have the library in the TEX tree because either more are present or the operating system updates them. As we don't use this in ConTEXt we're also not sure of things work ok but we can assume stable interfaces anyway. See the plugin module for more info.

```
function optional.hb.initialize ( )
    return <t:boolean> -- success
end
```

It probably makes sense to check for the version because (in the TEXLive code base) it is one of the most frequently updated code bases and for TEX stability and predictability (when working on a specific project) is important. When you initialize

```
function optional.hb.getversion ( )
    return <t:string>
end
```

```
function optional.hb.getshapers ( )
    return <t:table> -- strings
end
```

```
function optional.hb.loadfont (
    <t:integer> id,
    <t:string>  name
)
```

```
    return <t:userdata> -- instance
end
```

A run over characters happens with the next one. You get back a list of tables that specify to be handled glyphs. The interface is pretty much the same as what Kai Eigner came up with at the time he wanted to compare the results with the regular font loader, for which the LuaTEX and LuajitTEX) ffi interfaces were used.

```
function optional.hb.shapestring (
    <t:userdata> font,
    <t:string>   script,
    <t:string>   language,
    <t:string>   direction,
    <t:table>    shapers,
    <t:table>    features,
    <t:string>   text
    <t:boolean>  reverse
    <t:integer>  utfbits, -- default 8
)
    return {
        {
            <t:integer>, -- codepoint
            <t:integer>, -- cluster
            <t:integer>, -- xoffset
            <t:integer>, -- yoffset
            <t:integer>, -- xadvance
            <t:integer>, -- uadvance
        },
        ...
    }
end
```

mujs

This is just a fun experiment that permits JavaScript code to be used instead of Lua. It was actually one of the first optional libraries I played with and as with the other optionals there is a module that wraps it. The library is initialized with:

```
function optional.mujs.initialize ( )
    return <t:boolean> -- success
end
```

There are a few ‘mandate’ callbacks than need to be implemented:

```
function optional.mujs.setfindfile (
    function ( <t:string> name )
        return <t:string>
    end
)
    -- no return values
end
```

```
function optional.mujs.setopenfile  (
    function ( <t:string> name )
        return <t:integer> id
    end
)
    -- no return values
end

function optional.mujs.setclosefile (
    function ( <t:integer> id )
        -- no return values
    end
)
    -- no return values
end

function optional.mujs.setreadfile (
    function ( <t:integer> id )
        return <t:string> | <t:nil>
    end
)
    -- no return values
end

function optional.mujs.setseekfile (
    function ( <t:integer> id, <t:integer> position )
        return <t:integer>
    end
)
    -- no return values
end

function optional.mujs.setconsole ( )
    function ( <t:string> category, <t:string> message )
        -- no return values
    end
)
    -- no return values
end
```

The library implements a few JavaScript functions, like the ones printing to TEX, they take an optional catcodes reference:

```
texprint (catcodes, ...)
texsprint(catcodes, ...)
```

and a reporter:

```
console  (category, message)
```

The next function resets the interpreter:

```
function optional.mujs.reset ( )
    -- no return value
end
```

A snippet of JavaScript can be executed with:

```
function optional.mujs.execute ( <t:string> filename )
    -- no return value
end
```

This loads a JavaScript file:

```
function optional.mujs.dofile ( <t:string> filename )
    -- no return value
end
```

openssl

We use this module for some pdf features. Given the frequent updates to the (external) code base, it's for sure not something one wants in the engine. We use only a small subset of functionality. The library is initialized with:

```
function optional.openssl.initialize ( )
    return <t:boolean> -- success
end
```

When signing succeeds the first return value is `true` and possibly there is a string as second return value. When `false` is returned the second argument is an error code.

```
function optional.openssl.sign (
    {
        certfile   = <t:string>,
        datafile   = <t:string>,
        data       = <t:string>,
        password   = <t:string>,
        resultfile = <t:string>,
    }
)
    return
        <t:boolean>, -- success
        <t:string> | <t:integer> | <t:nil>
end
```

Verifying needs similar data:

```
function optional.openssl.verify (
    {
        certfile - <t:string>,
        datafile - <t:string>,
        data     - <t:string>,
        signature- <t:string>,
        password - <t:string>,
```

```
    }
)
    return
        <t:boolean>, -- success
        <t:integer> | <t:nil>
end
```

This needs no explanation:

```
function optional.openssl.getversion ( )
    return <t:integer>
end
```