hans hagen

about

luatex and context

Contents

Introduction

During the development of LuaTEX we wrapped up the state of affairs in articles and reports. Upto version 0.50 we tagged them as 'MkIV' (the transition from MkII), while for the next 0.25 versions we bundled them in 'hybrid' (the rewrite of ConTEXt). The next series goes under the name `about` as one might wonder what all this LuaTEX and ConTEXt is about. After all we've now reached a state where we can think about future applications instead of improving older features as that process is ongoing.

As we're a bit beyond experimenting now, the focus will be on practical usage and of course we target on applications that the Lua and TEX combination makes possible, either new or in a renewed form. Some of the chapters will eventually become part of manuals.

As with the two preceding collections of LuaTEX development stories, 'mk' and 'hybrid', this one, called 'about', covers a stretch of development, mostly between versions 0.50 and 0.75. The forth stretch, upto 1.00 is covered in 'still'.

Hans Hagen
Hasselt NL
2013–2015

`http://www.luatex.org`
`http://www.pragma-ade.com`

1 Math stackers

1.1 Introduction

In the next sections I will discuss the way we deal with stacked content in ConTeXt MkIV and in particular extensible characters. The mechanism describe here is actually more generic and can also deal with regular text. The stacker code is an evolution of the mechanisms that combine math arrows with text. From the users perspective there is not that much difference with the old methods because in practice 'defined' commands are used and their name stayed. However, we use different definition and setup commands and provide much more control. The new implementation is leaner but not meaner and fits the way MkIV is set up.

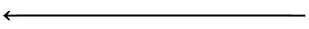
How does Lua fits in? We use a helper in order to determine some characteristics of extensibles, but we could have done without. We also use some new LuaTeX math primitives and of course we depend on OpenType font technoloygy.

1.2 Extensibles

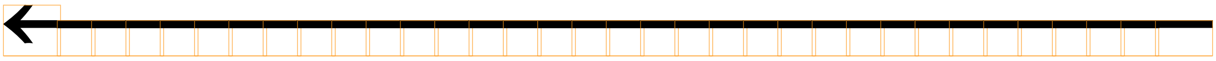
The command `\leftarrowfill` was introduced in plain TeX and gives, as the name indicates, a ⟵ that stretches itself so that it takes the available space. Take the following example:

```
\hbox to 4cm{\leftarrowfill}
```

This will make an arrow of length 4cm:



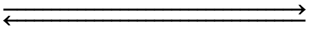
This arrow is made out of small snippets:



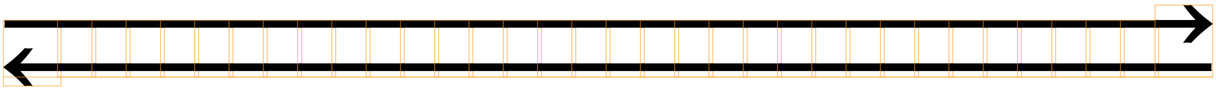
Here is another one:

```
\hbox to 4cm{\rightoverleftarrowfill}
```

or:



This time we have three different snippets:



The TeX engine has a concept of extensible characters. In fact there are two mechanisms: there is a list of larger glyphs and when that list is exhausted larger characters can be constructed out of snippets. Examples are left and right fences in math like braces and brackets, and, also in math, some top and bottom accents.

For reasons unknown to me, some of these extensibles are handled by the engine directly, using properties of a font, while others are composed using macros. Given that TeX is quite popular for typesetting scientific articles it is beyond my understanding why no one decided to provide some more fonts and/or extend the TeX engine. After all, the whole idea of Donald Knuth with TeX was that it could be adapted to future needs by its users. And so, more that 30 years after TeX and macro packages showed up we're stuck with not only incomplete fonts, but also an engine that was never adapted to demands.

1.3 The traditional way

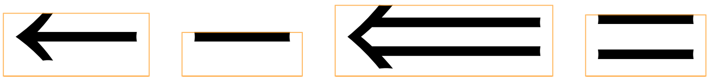
In ConTeXt we have support for extensibles built into the core but it uses the traditional approach: take some snippets and paste them together, making sure to achieve some overlap and get rid of side bearings. In terms of TeX code this can best be illustrated with the plain TeX definition of such a command:

```
\def\leftarrowfill
  {$%
   \mathsurround0pt%
   \mathord\leftarrow\mkern-7mu%
   \cleaders\hbox{$\mkern-2mu\smash-\mkern-2mu$}\hfill
   \mkern-7mu\smash-%
   $}
```

Here we create a tight formula starting with a `leftarrow`, ending with a minus sign and glued together with the number of minus signs that are needed to fill the available space. This macro eventually expands to something like this (a bit spaced out):

```
\def\leftarrowfill { $
  % \leftarrow = \mathchardef\leftarrow="3220 in plain but in
  % unicode it's character 0x2190 so we use that one here
  \mathsurround=0pt
  \mathord{\mathchar"2190}
  \mkern-7mu
  \cleaders
   \hbox { $
     \mkern-2mu
     \mathchoice
      {\setbox0\hbox{$\displaystyle     -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\textstyle        -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\scriptstyle      -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\scriptscriptstyle-$}\ht0=0pt\dp0=0pt\box0}
     \mkern-2mu
    $ }
   \hfill
  \mkern-7mu
  \mathchoice
   {\setbox0\hbox{$\displaystyle     -$}\ht0=0pt\dp0=0pt\box0}
   {\setbox0\hbox{$\textstyle        -$}\ht0=0pt\dp0=0pt\box0}
   {\setbox0\hbox{$\scriptstyle      -$}\ht0=0pt\dp0=0pt\box0}
   {\setbox0\hbox{$\scriptscriptstyle-$}\ht0=0pt\dp0=0pt\box0}
$ }
```

If you look at the code you see a few hacks. First of all we see that we need to add kerns in order to make the symbols overlap. For the middle shapes this is understandable as there we don't want rounding errors to lead to gaps. Also, because the minus in Computer Modern (and therefore Latin Modern) has rounded tips, we need to make sure that we end up beyond the tips. Next we see two blobs of `mathchoice`. This primitive chooses one of the four variants and switches to the right math style. It packages the minus and smashes it. In our case smashing makes not much sense as the arrowhead has height and depth anyway, but it's a side effect of using general purpose macros that there can be some unneeded overhead.



Above you see the two characters that traditionally are combined into a leftward pointing arrows. Watch the whitespace on the left and right of the actual glyph.

1.4 The new way

These zero height and depth don't show up in our rendered examples. Why is this? The reason is that I cheated a bit. I used this to get the arrow:[1]

```
\mathstylehbox{\Umathaccent\fam\zerocount"21C4{\hskip4cm}}
```

The ConTeXt support macro `\mathstylehbox` is an efficient variant of `\mathchoice`. More significant is that we don't assemble the arrow, but just put it as an accent on top of a skip. The `\Umathaccent` primitive will assemble the long arrow for us, using information in the font. If we look into the definition of the (Latin Modern) font in MkIV we see this:

```
[8592]={
 ["boundingbox"]={ 57, -10, 942, 510 },
 ["class"]="base",
 ["index"]=1852,
 ["math"]={
  ["horiz_parts"]={
   {
    ["advance"]=507,
    ["end"]=169,
    ["extender"]=0,
```

---

[1] In this example I misuse the accent placement mechanism. Upto LuaTeX 0.75 that was the way to go.

```
      ["glyph"]=984274,
      ["start"]=0,
     },
     {
      ["advance"]=337,
      ["end"]=337,
      ["extender"]=1,
      ["glyph"]=984275,
      ["start"]=337,
     },
     {
      ["advance"]=507,
      ["end"]=0,
      ["extender"]=0,
      ["glyph"]=984276,
      ["start"]=169,
     },
    },
    ["horiz_variants"]={ 10229 },
   },
   ["name"]="arrowleft",
   ["width"]=1000,
  }
```

This arrow symbol comes in two sizes. The extra size is mentioned in `horiz_variants`. When no more variants are seen, it switches to the extensible definition, that uses `horiz_parts`. The dimensions are in basepoints, the references to glyphs are decimal. The `end` and `start` fields specify the overlap. When `extender` equals 1 it signals a repeatable snippet.

In the TEX engine the slot allocated for the left arrow symbol has a `next` pointer to a larger shape. Here there is only one such shape but when there are more they form a linked list. The the last one in such a list gets the specification of the extenders.

We hard-coded the width to 4cm so how does it work when the arrow has to adapt itself? There are two cases there. When we are putting text on top of or below an arrow, we know what the width is because we can measure the text. But when we use the arrow as a filler, we have to leave it to the engine to arrange it. In recent LUATEX the definition can be as simple as:

```
\def\leftarrowfill{\leaders "2190 \hfill}
```

or:

```
\def\leftarrowfill{\mathstylehbox{\leaders"2190\hfill}}
```

In fact, we can use this new LUATEX extension to `\leaders` to replace the accent hacks as well.

1.5 Wrapping it in macros

If this was all, we would be done in a few lines of definitions but as usual there is more involved: especially text. The prerequisites can be summarized as follows:

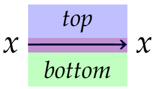
- The width of the extensible need to adapt itself automatically.
- We need to be able to control horizontal and vertical offsets.
- We best have a math as well as a text variant (which is handy for chemistry).
- For historic reasons we need to deal with optional arguments in a special (reverse) way.
- We need alternatives for extensibles on top, in the middle and at the bottom.

Using a low level command we can do this:

```
$x \directmathextensible{"2192}{top}{bottom} x$
```

$$x \xrightarrow[bottom]{top} x$$

This is not that exiting too look at, but the next might be:



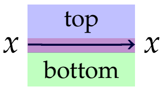
Here we have turned on a tracker:

```
\enabletrackers[math.stackers.texts]
```

The toppart is transparent blue, the middlepart transparent red and the bottom part becomes transparent green. When the areas overlap you see the mixed color.

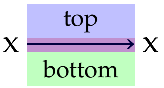
Before we explore some options, we show some variants. Often extensibles are used in math mode, if only because they originate in math and come from math fonts.

```
$x \textstacker{"2192}{top}{bottom} x$
```



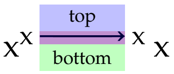
These commands also work outside math mode:

```
x \textstacker{"2192}{top}{bottom} x
```



and to some extend can adapt themselves:

```
x\high{x \textstacker{"2192}{top}{bottom} x} x
```

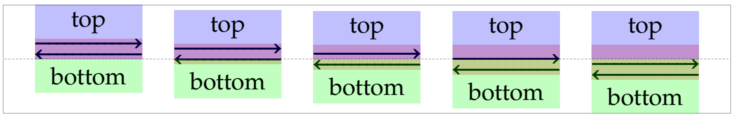


1.6 Influencing the spacing

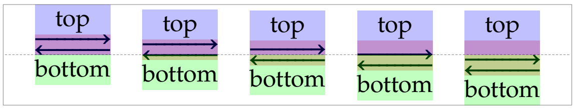
We will use the text example to illustrate some options.

```
\ruledhbox \bgroup \quad
    \setupmathstackers[location=top]%
    \textstacker{"21C4}{top}{bottom}\quad
    \setupmathstackers[location=high]%
    \textstacker{"21C4}{top}{bottom}\quad
    \setupmathstackers[location=middle]%
    \textstacker{"21C4}{top}{bottom}\quad
    \setupmathstackers[location=low]%
    \textstacker{"21C4}{top}{bottom}\quad
    \setupmathstackers[location=bottom]%
    \textstacker{"21C4}{top}{bottom}\quad
\egroup
```

You can set up extensibles to be shifted up and down.



The above rendering uses the default spacing. When we set all values to zero we get this:



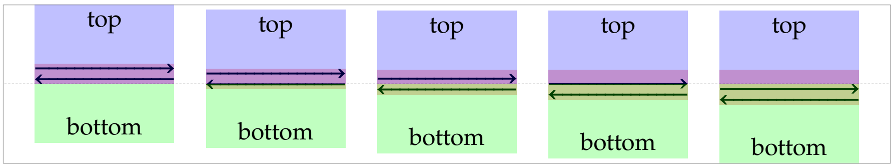
The setup looks like this:

```
\setupmathstackers
  [voffset=\zeropoint,
   hoffset=\zeropoint,
   minheight=\exheight,
   mindepth=\zeropoint,
   minwidth=\zeropoint]
```

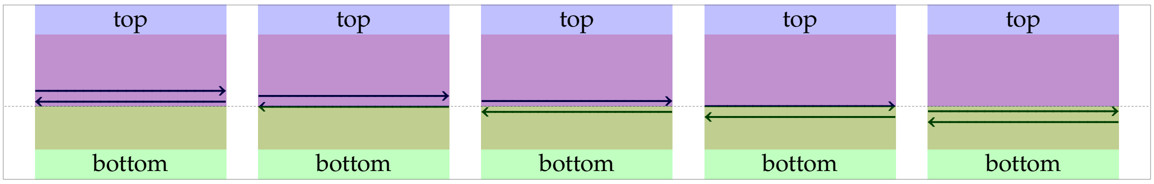
and gives a pretty tight rendering. The default values are:

```
\setupmathstackers
  [voffset=.25\exheight,
   hoffset=.5\emwidth,
   minheight=\exheight,
   mindepth=\zeropoint,
   minwidth=\emwidth]
```

When we set `voffset` to twice the ex-height and `hoffset` to the em-width we get:



We can enforce a (consistent) height and depth of the extensible by setting the minimum values:

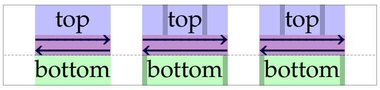


1.7 A neat feature

A more obscure feature relates to the visual appearance. When we put something on top of for instance an arrow, it sometimes looks better when we only consider the middle part. Watch the following:

```
\ruledhbox \bgroup \quad
    \setupmathstackers[offset=normal]%
    \textstacker{"21C4}{top}{bottom}\quad
    \setupmathstackers[offset=min]%
    \textstacker{"21C4}{top}{bottom}\quad
    \setupmathstackers[offset=max]%
    \textstacker{"21C4}{top}{bottom}\quad
\egroup
```

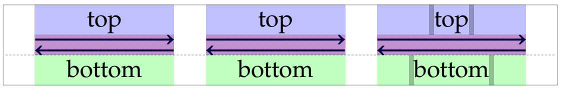
The `min` and `max` values will add extra offsets that relate to the width of the edge snippets.



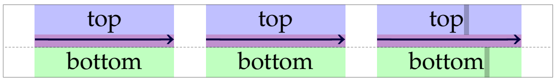
In this case both have the same result but the difference becomes clear when we set the `hoffset` to the em-width. In the case of `min` we don't add some extra space if the `hoffset` is applied.

```
\ruledhbox \bgroup \quad
    \setupmathstackers[offset=normal]%
    \textstacker{"21C4}{top}{bottom}\quad
    \setupmathstackers[offset=min]%
    \textstacker{"21C4}{top}{bottom}\quad
    \setupmathstackers[offset=max]%
    \textstacker{"21C4}{top}{bottom}\quad
\egroup
```

Of course in this example we have a symmetrical correction.



A one sided arrow behaves different:



1.8 The user interface

It all starts out with categories. We have a couple of predefined categories in the core. The `mathematics` category typesets the top and bottom texts in mathmode, while the `text` category doesn't. The `reverse` category swaps its arguments. There are `upper` and `under` categories too.

As with most ConTEXt mechanisms inheritance is part of the picture:

```
\definemathextensibles [mine] [big] [offset=min]
```

You can change settings with:

```
\setupmathstackers [mine] [big] [voffset=\exheight]
```

For downward compatibility we also provide these:

```
\definemathextensibles [normal] [hoffset=0.5\emwidth]
\definemathextensibles [none]   [hoffset=\zeropoint]
\definemathextensibles [small]  [hoffset=1\emwidth]
\definemathextensibles [medium] [hoffset=1.5\emwidth]
\definemathextensibles [big]    [hoffset=2\emwidth]
```

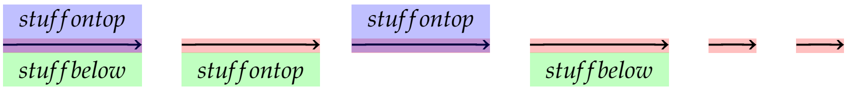
They inherit from `mathematics` so choosing this also forces the top and bottomtexts to be typeset in math mode.

These commands don't define extensibles, they only provide a way to categorize them. There are couple of definers and one reason for that is that we want to define downward compatible commands.

```
\definemathextensible [reverse] [xleftarrow]  ["2190]
\definemathextensible [reverse] [xrightarrow] ["2192]
```

The `x` in the name is sort of standard for an extensible symbol with optionally some text on top or below. The reverse forced compatible behaviour.

```
\xrightarrow{stuff below} {stuff on top} \quad
\xrightarrow{stuff on top}               \quad
\xrightarrow{}            {stuff on top} \quad
\xrightarrow{stuff below} {}             \quad
\xrightarrow{}            {}             \quad
\xrightarrow                             \quad
```



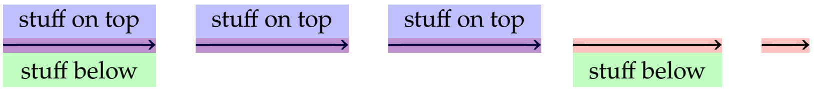
New in MkIV is the `t` variant that typesets the text as (indeed) text. In addition we have a normal-order `m` variant:

```
\definemathextensible [text] [tleftarrow]  ["2190]
\definemathextensible [text] [trightarrow] ["2192]

\definemathextensible [mathematics] [mleftarrow]  ["2190]
\definemathextensible [mathematics] [mrightarrow] ["2192]
```

This time the order is always top first and bottom next:

```
\trightarrow{stuff on top} {stuff below} \quad
\trightarrow{stuff on top} {}            \quad
\trightarrow{stuff on top}               \quad
\trightarrow{}             {stuff below} \quad
\trightarrow                             \quad
```

So we get:

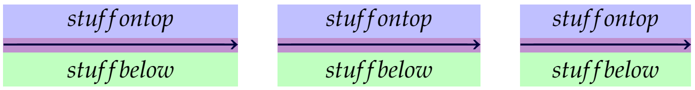


As you can see, there is an optional first argument that specifies the category that applies. This permits you to define extra commands that have their own (spacing) properties.

Earlier on we saw that defined commands can be forced into a category:

```
\trightarrow[big]   {stuff on top} {stuff below} \quad
\trightarrow[medium]{stuff on top} {stuff below} \quad
\trightarrow[small] {stuff on top} {stuff below}
```

Here we get:



A variation on this kind of extensibles are over- and underarrows. This time the text is the nucleus.

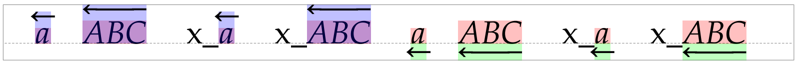
```
\definemathoverextensible  [top]    [overleftarrow]   ["2190]
\definemathoverextensible  [top]    [overrightarrow]  ["2192]

\definemathunderextensible [bottom] [underleftarrow]  ["2190]
\definemathunderextensible [bottom] [underrightarrow] ["2192]
```

In action this looks like:

```
\ruledhbox \bgroup $ \quad
```

```
     \overleftarrow {a}  \quad    \overleftarrow {ABC}  $ \quad
  x_{\overleftarrow {a}} \quad x_{\overleftarrow {ABC}} $ \quad
     \underleftarrow{a}  \quad    \underleftarrow{ABC}  $ \quad
  x_{\underleftarrow{a}} \quad x_{\underleftarrow{ABC}} $ \quad
$ \egroup
```

Here we also have tracing enabled, and we also show the bounding box:



This leaves us one command: the one that defines the basic filler:

```
\definemathextensiblefiller [leftarrowfill]  ["2190]
\definemathextensiblefiller [rightarrowfill] ["2192]
```

Commands defined like this will stretch themselves to fit the circumstances, and normally they will fill op the available space.

```
\hbox to 4cm {from here \leftarrowfill\ to there}
\hbox to 8cm {from there \rightarrowfill\ to here}
```

These commands (like the others) work in text mode as well as in math mode.

from here ⟵ to there
from there ⟶ to here

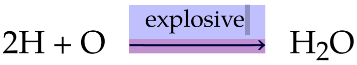
1.9 Special cases

One of the reasons why the arrows mechanism has always been somewhat configureable is that we need arrows in the chemistry code.

```
\definemathextensibles
  [chemistry]
  [offset=max,
   left=\enspace,
   right=\enspace,
   hoffset=.5\emwidth]

\definemathextensible [chemistry] [cleftarrow]          ["2190]
\definemathextensible [chemistry] [crightarrow]         ["2192]
\definemathextensible [chemistry] [crightoverleftarrow] ["21C4]
```

```
2H + O \crightarrow{explosive}\ H\low{2}O
```

Of course normally such code is wrapped into the chemistry environments and support macros.



If you want something else than an extensible you can use definitions like the following:

```
\definemathtriplet [tripleta]
\definemathtriplet [text] [tripletb]
\definemathtriplet [text] [tripletc] [\otimes]
```

```
\tripleta{\ominus}{top}{botom} and
\tripletb{\oplus} {top}{botom} and
\tripletc         {top}{botom}
```

$\overset{\text{top}}{\underset{\text{botom}}{\ominus}}$ and $\overset{\text{top}}{\underset{\text{botom}}{\oplus}}$ and $\overset{\text{top}}{\underset{\text{botom}}{\otimes}}$

As optional first argument you can pass a category.

```
\tripleta[mathematics]{\ominus}{top}{botom} and
\tripletb[mathematics]{\oplus}{top}{botom} and
\tripletc[mathematics]{top}{botom}
```

Which gives:

$\overset{top}{\underset{botom}{\ominus}}$ and $\overset{top}{\underset{botom}{\oplus}}$ and $\overset{top}{\underset{botom}{\otimes}}$

Instead of `mathematics` you could have given its synonym `math`. Keep in mind that categories are shared among stackers. There is also a direct command:

```
before \mathtriplet{\otimes}{top}{botom} after
```

1.10 An overview

We end with showing a list of extensibles that come with the font used here, the TEXGyre Pagella. First we load a module:

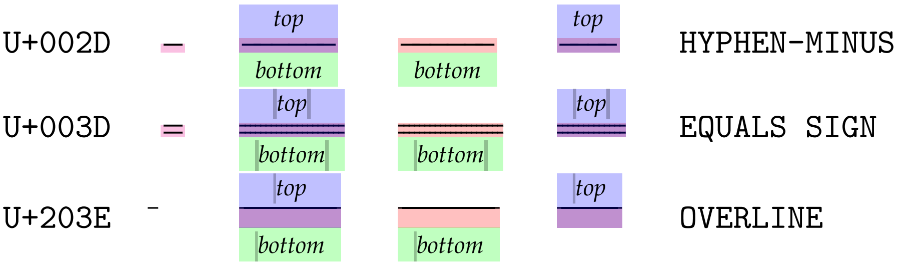
```
\usemodule[s][math-extensibles]
```

This module provides a couple of commands that typesets a table with the extensibles as known in ConTEXt. Beware: not all fonts have all those characters.

A second command is:

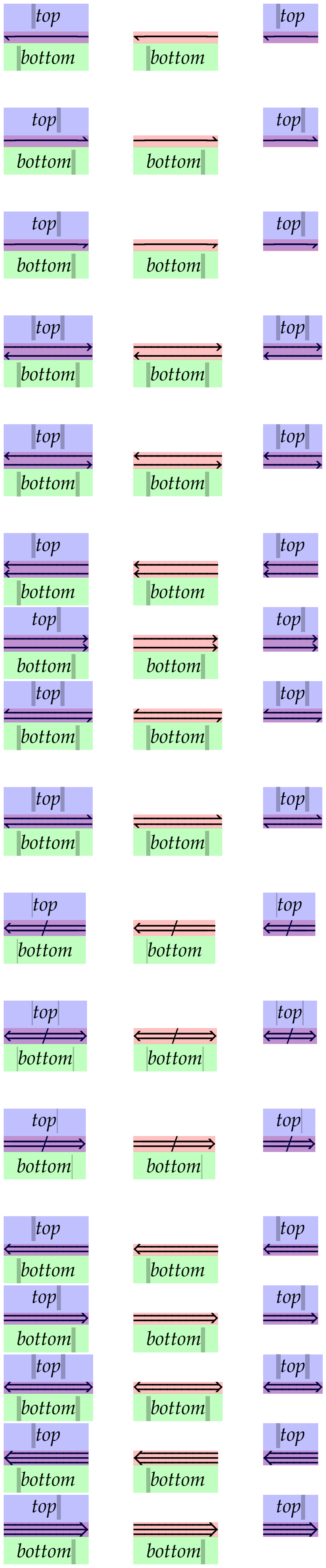
```
\showmathextensibles[alternative=a]
```

This commands shows the base glyph, and the stretched variant with text on top and below. When no symbol is found in the font a rule is rendered.



U+002D	−	*top* *bottom*	*bottom*	*top*	HYPHEN-MINUS
U+003D	=	*top* *bottom*	*bottom*	*top*	EQUALS SIGN
U+203E	‾	*top* *bottom*	*bottom*	*top*	OVERLINE

U+2190	←	$\overset{top}{\underset{bottom}{\longleftarrow}}$	$\underset{bottom}{\longleftarrow}$	$\overset{top}{\leftarrow}$	LEFTWARDS ARROW
U+2192	→	$\overset{top}{\underset{bottom}{\longrightarrow}}$	$\underset{bottom}{\longrightarrow}$	$\overset{top}{\rightarrow}$	RIGHTWARDS ARROW
U+2194	↔	$\overset{top}{\underset{bottom}{\longleftrightarrow}}$	$\underset{bottom}{\longleftrightarrow}$	$\overset{top}{\leftrightarrow}$	LEFT RIGHT ARROW
U+219A	↚	$\overset{top}{\underset{bottom}{\nleftarrow}}$	$\underset{bottom}{\nleftarrow}$	$\overset{top}{\nleftarrow}$	LEFTWARDS ARROW WITH STROKE
U+219B	↛	$\overset{top}{\underset{bottom}{\nrightarrow}}$	$\underset{bottom}{\nrightarrow}$	$\overset{top}{\nrightarrow}$	RIGHTWARDS ARROW WITH STROKE
U+219C	?	$\overset{top}{\underset{bottom}{}}$	$\underset{bottom}{}$	$\overset{top}{}$	LEFTWARDS WAVE ARROW
U+219D	?	$\overset{top}{\underset{bottom}{}}$	$\underset{bottom}{}$	$\overset{top}{}$	RIGHTWARDS WAVE ARROW
U+219E	↞	$\overset{top}{\underset{bottom}{\twoheadleftarrow}}$	$\underset{bottom}{\twoheadleftarrow}$	$\overset{top}{\twoheadleftarrow}$	LEFTWARDS TWO HEADED ARROW
U+21A0	↠	$\overset{top}{\underset{bottom}{\twoheadrightarrow}}$	$\underset{bottom}{\twoheadrightarrow}$	$\overset{top}{\twoheadrightarrow}$	RIGHTWARDS TWO HEADED ARROW
U+21A2	↢	$\overset{top}{\underset{bottom}{\leftarrowtail}}$	$\underset{bottom}{\leftarrowtail}$	$\overset{top}{\leftarrowtail}$	LEFTWARDS ARROW WITH TAIL
U+21A3	↣	$\overset{top}{\underset{bottom}{\rightarrowtail}}$	$\underset{bottom}{\rightarrowtail}$	$\overset{top}{\rightarrowtail}$	RIGHTWARDS ARROW WITH TAIL
U+21A4	↤	$\overset{top}{\underset{bottom}{\mapsfrom}}$	$\underset{bottom}{\mapsfrom}$	$\overset{top}{\mapsfrom}$	LEFTWARDS ARROW FROM BAR
U+21A6	↦	$\overset{top}{\underset{bottom}{\mapsto}}$	$\underset{bottom}{\mapsto}$	$\overset{top}{\mapsto}$	RIGHTWARDS ARROW FROM BAR
U+21A9	↩	$\overset{top}{\underset{bottom}{\hookleftarrow}}$	$\underset{bottom}{\hookleftarrow}$	$\overset{top}{\hookleftarrow}$	LEFTWARDS ARROW WITH HOOK
U+21AA	↪	$\overset{top}{\underset{bottom}{\hookrightarrow}}$	$\underset{bottom}{\hookrightarrow}$	$\overset{top}{\hookrightarrow}$	RIGHTWARDS ARROW WITH HOOK
U+21AB	↫	$\overset{top}{\underset{bottom}{\looparrowleft}}$	$\underset{bottom}{\looparrowleft}$	$\overset{top}{\looparrowleft}$	LEFTWARDS ARROW WITH LOOP
U+21AC	↬	$\overset{top}{\underset{bottom}{\looparrowright}}$	$\underset{bottom}{\looparrowright}$	$\overset{top}{\looparrowright}$	RIGHTWARDS ARROW WITH LOOP
U+21AD	↭	$\overset{top}{\underset{bottom}{\leftrightsquigarrow}}$	$\underset{bottom}{\leftrightsquigarrow}$	$\overset{top}{\leftrightsquigarrow}$	LEFT RIGHT WAVE ARROW
U+21AE	↮	$\overset{top}{\underset{bottom}{\nleftrightarrow}}$	$\underset{bottom}{\nleftrightarrow}$	$\overset{top}{\nleftrightarrow}$	LEFT RIGHT ARROW WITH STROKE
U+21B9	?	$\overset{top}{\underset{bottom}{}}$	$\underset{bottom}{}$	$\overset{top}{}$	LEFTWARDS ARROW TO BAR OVER RIGHTWARDS ARROW TO BAR
U+21BC	↼	$\overset{top}{\underset{bottom}{\leftharpoonup}}$	$\underset{bottom}{\leftharpoonup}$	$\overset{top}{\leftharpoonup}$	LEFTWARDS HARPOON WITH BARB UPWARDS



U+21BD	↽	LEFTWARDS HARPOON WITH BARB DOWNWARDS
U+21C0	⇀	RIGHTWARDS HARPOON WITH BARB UPWARDS
U+21C1	⇁	RIGHTWARDS HARPOON WITH BARB DOWNWARDS
U+21C4	⇄	RIGHTWARDS ARROW OVER LEFTWARDS ARROW
U+21C6	⇆	LEFTWARDS ARROW OVER RIGHTWARDS ARROW
U+21C7	⇇	LEFTWARDS PAIRED ARROWS
U+21C9	⇉	RIGHTWARDS PAIRED ARROWS
U+21CB	⇋	LEFTWARDS HARPOON OVER RIGHTWARDS HARPOON
U+21CC	⇌	RIGHTWARDS HARPOON OVER LEFTWARDS HARPOON
U+21CD	⇍	LEFTWARDS DOUBLE ARROW WITH STROKE
U+21CE	⇎	LEFT RIGHT DOUBLE ARROW WITH STROKE
U+21CF	⇏	RIGHTWARDS DOUBLE ARROW WITH STROKE
U+21D0	⇐	LEFTWARDS DOUBLE ARROW
U+21D2	⇒	RIGHTWARDS DOUBLE ARROW
U+21D4	⇔	LEFT RIGHT DOUBLE ARROW
U+21DA	⇚	LEFTWARDS TRIPLE ARROW
U+21DB	⇛	RIGHTWARDS TRIPLE ARROW

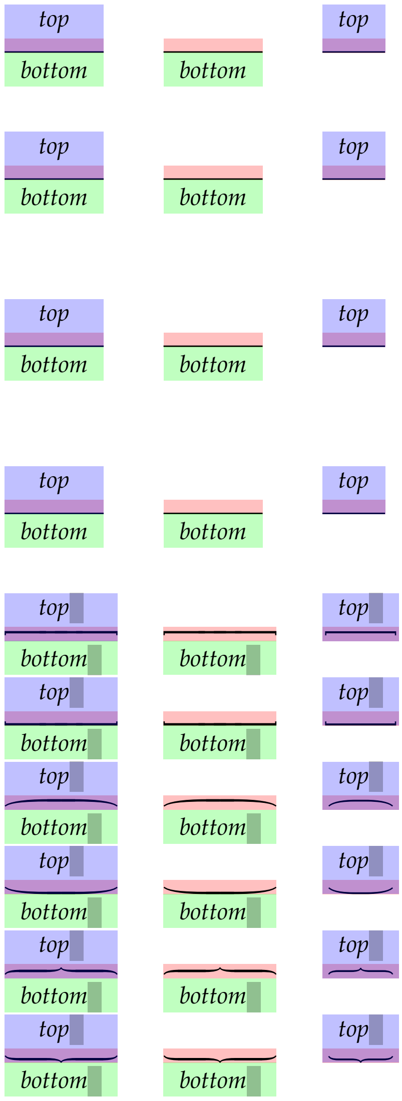
U+21DC	⇜	top ⇜ bottom	⇜ bottom	top ⇜	LEFTWARDS SQUIGGLE ARROW
U+21E0	?	top bottom	bottom	top	LEFTWARDS DASHED ARROW
U+21E4	?	top bottom	bottom	top	LEFTWARDS ARROW TO BAR
U+21E5	?	top bottom	bottom	top	RIGHTWARDS ARROW TO BAR
U+21E6	⇦	top ⇦ bottom	⇦ bottom	top ⇦	LEFTWARDS WHITE ARROW
U+21E8	⇨	top ⇨ bottom	⇨ bottom	top ⇨	RIGHTWARDS WHITE ARROW
U+21F4	?	top bottom	bottom	top	RIGHT ARROW WITH SMALL CIRCLE
U+21F6	⇶	top ⇶ bottom	⇶ bottom	top ⇶	THREE RIGHTWARDS ARROWS
U+21F7	?	top bottom	bottom	top	LEFTWARDS ARROW WITH VERTICAL STROKE
U+21F8	?	top bottom	bottom	top	RIGHTWARDS ARROW WITH VERTICAL STROKE
U+21F9	?	top bottom	bottom	top	LEFT RIGHT ARROW WITH VERTICAL STROKE
U+21FA	?	top bottom	bottom	top	LEFTWARDS ARROW WITH DOUBLE VERTICAL STROKE
U+21FB	?	top bottom	bottom	top	RIGHTWARDS ARROW WITH DOUBLE VERTICAL STROKE
U+21FC	?	top bottom	bottom	top	LEFT RIGHT ARROW WITH DOUBLE VERTICAL STROKE
U+21FD	?	top bottom	bottom	top	LEFTWARDS OPEN-HEADED ARROW
U+21FE	?	top bottom	bottom	top	RIGHTWARDS OPEN-HEADED ARROW
U+21FF	?	top bottom	bottom	top	LEFT RIGHT OPEN-HEADED ARROW
U+2261	≡	top ≡ bottom	≡ bottom	top ≡	IDENTICAL TO
U+2262	≢	top bottom	bottom	top	NOT IDENTICAL TO

U+2263	≣	top ≣ bottom	≣ bottom	top ≣	STRICTLY EQUIVALENT TO
U+23B4	⎴	top ⎴ bottom	⎴ bottom	top ⎴	TOP SQUARE BRACKET
U+23B5	⎵	top ⎵ bottom	⎵ bottom	top ⎵	BOTTOM SQUARE BRACKET
U+23DC	⏜	top ⏜ bottom	⏜ bottom	top ⏜	TOP PARENTHESIS
U+23DD	⏝	top ⏝ bottom	⏝ bottom	top ⏝	BOTTOM PARENTHESIS
U+23DE	⏞	top ⏞ bottom	⏞ bottom	top ⏞	TOP CURLY BRACKET
U+23DF	⏟	top ⏟ bottom	⏟ bottom	top ⏟	BOTTOM CURLY BRACKET
U+27F4	⟴	top bottom	bottom	top	RIGHT ARROW WITH CIRCLED PLUS
U+27F5	⟵	top ⟵ bottom	⟵ bottom	top ⟵	LONG LEFTWARDS ARROW
U+27F6	⟶	top ⟶ bottom	⟶ bottom	top ⟶	LONG RIGHTWARDS ARROW
U+27F7	⟷	top ⟷ bottom	⟷ bottom	top ⟷	LONG LEFT RIGHT ARROW
U+27F8	⟸	top ⟸ bottom	⟸ bottom	top ⟸	LONG LEFTWARDS DOUBLE ARROW
U+27F9	⟹	top ⟹ bottom	⟹ bottom	top ⟹	LONG RIGHTWARDS DOUBLE ARROW
U+27FA	⟺	top ⟺ bottom	⟺ bottom	top ⟺	LONG LEFT RIGHT DOUBLE ARROW
U+27FB	⟻	top ⟻ bottom	⟻ bottom	top ⟻	LONG LEFTWARDS ARROW FROM BAR
U+27FC	⟼	top ⟼ bottom	⟼ bottom	top ⟼	LONG RIGHTWARDS ARROW FROM BAR
U+27FD	⟽	top ⟽ bottom	⟽ bottom	top ⟽	LONG LEFTWARDS DOUBLE ARROW FROM BAR
U+27FE	⟾	top ⟾ bottom	⟾ bottom	top ⟾	LONG RIGHTWARDS DOUBLE ARROW FROM BAR
U+27FF	⟿	top bottom	bottom	top	LONG RIGHTWARDS SQUIGGLE ARROW

U+2900	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
U+2901	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS TWO-HEADED ARROW WITH DOUBLE VERTICAL STROKE
U+2902	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS DOUBLE ARROW WITH VERTICAL STROKE
U+2903	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS DOUBLE ARROW WITH VERTICAL STROKE
U+2904	?	*top* *bottom*	*bottom*	*top*	LEFT RIGHT DOUBLE ARROW WITH VERTICAL STROKE
U+2905	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS TWO-HEADED ARROW FROM BAR
U+2906	⤆	*top* ⤆ *bottom*	⤆ *bottom*	*top* ⤆	LEFTWARDS DOUBLE ARROW FROM BAR
U+2907	⤇	*top* ⤇ *bottom*	⤇ *bottom*	*top* ⤇	RIGHTWARDS DOUBLE ARROW FROM BAR
U+290C	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS DOUBLE DASH ARROW
U+290D	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS DOUBLE DASH ARROW
U+290E	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TRIPLE DASH ARROW
U+290F	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS TRIPLE DASH ARROW
U+2910	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS TWO-HEADED TRIPLE DASH ARROW
U+2911	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS ARROW WITH DOTTED STEM
U+2914	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
U+2915	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE

U+2916	?	top bottom	bottom	top	RIGHTWARDS TWO-HEADED ARROW WITH TAIL
U+2917	?	top bottom	bottom	top	RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE
U+2918	?	top bottom	bottom	top	RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2919	?	top bottom	bottom	top	LEFTWARDS ARROW-TAIL
U+291A	?	top bottom	bottom	top	RIGHTWARDS ARROW-TAIL
U+291B	?	top bottom	bottom	top	LEFTWARDS DOUBLE ARROW-TAIL
U+291C	?	top bottom	bottom	top	RIGHTWARDS DOUBLE ARROW-TAIL
U+291D	?	top bottom	bottom	top	LEFTWARDS ARROW TO BLACK DIAMOND
U+291E	?	top bottom	bottom	top	RIGHTWARDS ARROW TO BLACK DIAMOND
U+291F	?	top bottom	bottom	top	LEFTWARDS ARROW FROM BAR TO BLACK DIAMOND
U+2920	?	top bottom	bottom	top	RIGHTWARDS ARROW FROM BAR TO BLACK DIAMOND
U+2933	?	top bottom	bottom	top	WAVE ARROW POINTING DIRECTLY RIGHT
U+2938	?	top bottom	bottom	top	RIGHT-SIDE ARC CLOCKWISE ARROW
U+2939	?	top bottom	bottom	top	LEFT-SIDE ARC ANTICLOCKWISE ARROW
U+293E	?	top bottom	bottom	top	LOWER RIGHT SEMICIRCULAR CLOCKWISE ARROW

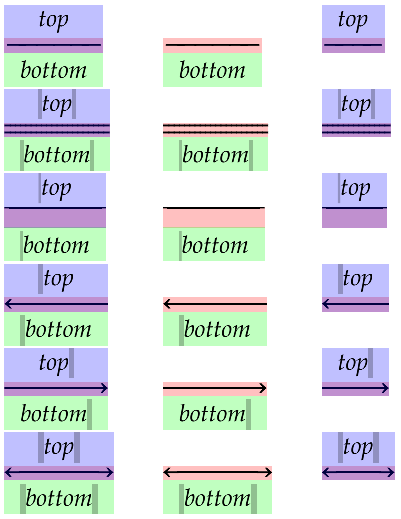
U+293F	?	*top* *bottom*	*bottom*	*top*	LOWER LEFT SEMICIRCULAR ANTICLOCKWISE ARROW
U+2945	?	*top* *bottom*	*bottom*	*top*	RIGHTWARDS ARROW WITH PLUS BELOW
U+2946	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS ARROW WITH PLUS BELOW
U+2970	?	*top* *bottom*	*bottom*	*top*	RIGHT DOUBLE ARROW WITH ROUNDED HEAD
U+2A17	?	*top* *bottom*	*bottom*	*top*	INTEGRAL WITH LEFTWARDS ARROW WITH HOOK
U+2B30	?	*top* *bottom*	*bottom*	*top*	LEFT ARROW WITH SMALL CIRCLE
U+2B31	⬱	*top* ⬱ *bottom*	⬱ *bottom*	*top* ⬱	THREE LEFTWARDS ARROWS
U+2B32	?	*top* *bottom*	*bottom*	*top*	LEFT ARROW WITH CIRCLED PLUS
U+2B33	⬳	*top* *bottom*	*bottom*	*top*	LONG LEFTWARDS SQUIGGLE ARROW
U+2B34	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
U+2B35	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED ARROW WITH DOUBLE VERTICAL STROKE
U+2B36	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED ARROW FROM BAR
U+2B37	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED TRIPLE DASH ARROW
U+2B38	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS ARROW WITH DOTTED STEM
U+2B39	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
U+2B3A	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE



U+2B3B	?	LEFTWARDS TWO-HEADED ARROW WITH TAIL
U+2B3C	?	LEFTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE
U+2B3D	?	LEFTWARDS TWO-HEADED ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2B3F	?	WAVE ARROW POINTING DIRECTLY LEFT
U+FE3B4	⎴	EXTENSIBLE OF 0x03B4
U+FE3B5	⎵	EXTENSIBLE OF 0x03B5
U+FE3DC	⏜	EXTENSIBLE OF 0x03DC
U+FE3DD	⏝	EXTENSIBLE OF 0x03DD
U+FE3DE	⏞	EXTENSIBLE OF 0x03DE
U+FE3DF	⏟	EXTENSIBLE OF 0x03DF

`\showmathextensibles[alternative=b]`

This command typesets a list with Unicode entries and defined commands. There are empty entries due to lack of glyphs in the used font. Not all characters have an associated command. Some have multiple commands with different math classes.

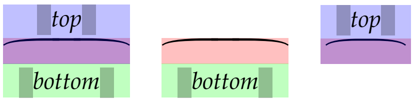
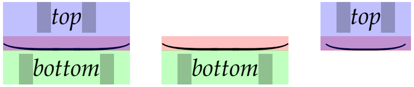
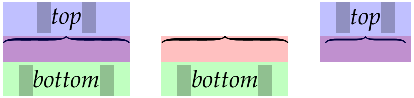
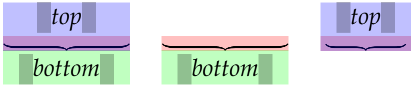
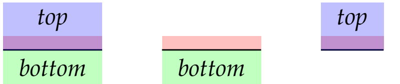
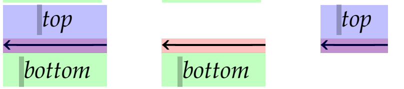
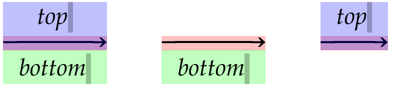
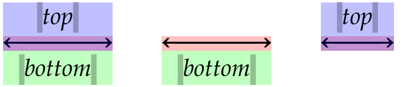
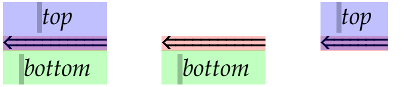
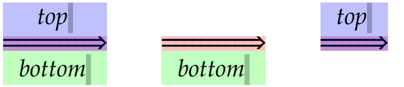
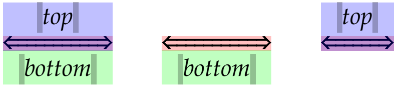
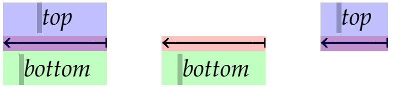
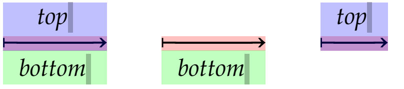
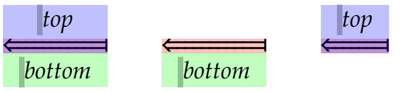
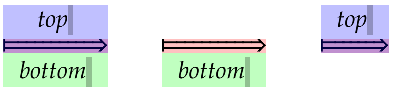
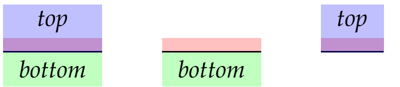
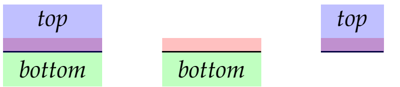
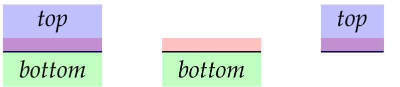


U+002D	−	HYPHEN-MINUS
U+003D	=	EQUALS SIGN
U+203E	‾	OVERLINE
U+2190	←	LEFTWARDS ARROW
U+2192	→	RIGHTWARDS ARROW
U+2194	↔	LEFT RIGHT ARROW

U+219A	↚	top ↚ bottom	↚ bottom	top ↚	LEFTWARDS ARROW WITH STROKE
U+219B	↛	top ↛ bottom	↛ bottom	top ↛	RIGHTWARDS ARROW WITH STROKE
U+219C	?	top bottom	bottom	top	LEFTWARDS WAVE ARROW
U+219D	?	top bottom	bottom	top	RIGHTWARDS WAVE ARROW
U+219E	↞	top ↞ bottom	↞ bottom	top ↞	LEFTWARDS TWO HEADED ARROW
U+21A0	↠	top ↠ bottom	↠ bottom	top ↠	RIGHTWARDS TWO HEADED ARROW
U+21A2	↢	top ↢ bottom	↢ bottom	top ↢	LEFTWARDS ARROW WITH TAIL
U+21A3	↣	top ↣ bottom	↣ bottom	top ↣	RIGHTWARDS ARROW WITH TAIL
U+21A4	↤	top ↤ bottom	↤ bottom	top ↤	LEFTWARDS ARROW FROM BAR
U+21A6	↦	top ↦ bottom	↦ bottom	top ↦	RIGHTWARDS ARROW FROM BAR
U+21A9	↩	top ↩ bottom	↩ bottom	top ↩	LEFTWARDS ARROW WITH HOOK
U+21AA	↪	top ↪ bottom	↪ bottom	top ↪	RIGHTWARDS ARROW WITH HOOK
U+21AB	↫	top ↫ bottom	↫ bottom	top ↫	LEFTWARDS ARROW WITH LOOP
U+21AC	↬	top ↬ bottom	↬ bottom	top ↬	RIGHTWARDS ARROW WITH LOOP
U+21AD	↭	top ↭ bottom	↭ bottom	top ↭	LEFT RIGHT WAVE ARROW
U+21AE	↮	top ↮ bottom	↮ bottom	top ↮	LEFT RIGHT ARROW WITH STROKE
U+21B9	?	top bottom	bottom	top	LEFTWARDS ARROW TO BAR OVER RIGHTWARDS ARROW TO BAR
U+21BC	↼	top ↼ bottom	↼ bottom	top ↼	LEFTWARDS HARPOON WITH BARB UPWARDS
U+21BD	↽	top ↽ bottom	↽ bottom	top ↽	LEFTWARDS HARPOON WITH BARB DOWNWARDS
U+21C0	⇀	top ⇀ bottom	⇀ bottom	top ⇀	RIGHTWARDS HARPOON WITH BARB UPWARDS

U+21C1	$\rightharpoondown$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\rightharpoondown}}$	$\underset{\textit{bottom}}{\rightharpoondown}$	$\overset{\textit{top}}{\rightharpoondown}$	RIGHTWARDS HARPOON WITH BARB DOWNWARDS
U+21C4	$\rightleftarrows$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\rightleftarrows}}$	$\underset{\textit{bottom}}{\rightleftarrows}$	$\overset{\textit{top}}{\rightleftarrows}$	RIGHTWARDS ARROW OVER LEFTWARDS ARROW
U+21C6	$\leftrightarrows$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\leftrightarrows}}$	$\underset{\textit{bottom}}{\leftrightarrows}$	$\overset{\textit{top}}{\leftrightarrows}$	LEFTWARDS ARROW OVER RIGHTWARDS ARROW
U+21C7	$\leftleftarrows$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\leftleftarrows}}$	$\underset{\textit{bottom}}{\leftleftarrows}$	$\overset{\textit{top}}{\leftleftarrows}$	LEFTWARDS PAIRED ARROWS
U+21C9	$\rightrightarrows$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\rightrightarrows}}$	$\underset{\textit{bottom}}{\rightrightarrows}$	$\overset{\textit{top}}{\rightrightarrows}$	RIGHTWARDS PAIRED ARROWS
U+21CB	$\leftrightharpoons$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\leftrightharpoons}}$	$\underset{\textit{bottom}}{\leftrightharpoons}$	$\overset{\textit{top}}{\leftrightharpoons}$	LEFTWARDS HARPOON OVER RIGHTWARDS HARPOON
U+21CC	$\rightleftharpoons$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\rightleftharpoons}}$	$\underset{\textit{bottom}}{\rightleftharpoons}$	$\overset{\textit{top}}{\rightleftharpoons}$	RIGHTWARDS HARPOON OVER LEFTWARDS HARPOON
U+21CD	$\nLeftarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\nLeftarrow}}$	$\underset{\textit{bottom}}{\nLeftarrow}$	$\overset{\textit{top}}{\nLeftarrow}$	LEFTWARDS DOUBLE ARROW WITH STROKE
U+21CE	$\nLeftrightarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\nLeftrightarrow}}$	$\underset{\textit{bottom}}{\nLeftrightarrow}$	$\overset{\textit{top}}{\nLeftrightarrow}$	LEFT RIGHT DOUBLE ARROW WITH STROKE
U+21CF	$\nRightarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\nRightarrow}}$	$\underset{\textit{bottom}}{\nRightarrow}$	$\overset{\textit{top}}{\nRightarrow}$	RIGHTWARDS DOUBLE ARROW WITH STROKE
U+21D0	$\Leftarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\Leftarrow}}$	$\underset{\textit{bottom}}{\Leftarrow}$	$\overset{\textit{top}}{\Leftarrow}$	LEFTWARDS DOUBLE ARROW
U+21D2	$\Rightarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\Rightarrow}}$	$\underset{\textit{bottom}}{\Rightarrow}$	$\overset{\textit{top}}{\Rightarrow}$	RIGHTWARDS DOUBLE ARROW
U+21D4	$\Leftrightarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\Leftrightarrow}}$	$\underset{\textit{bottom}}{\Leftrightarrow}$	$\overset{\textit{top}}{\Leftrightarrow}$	LEFT RIGHT DOUBLE ARROW
U+21DA	$\Lleftarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\Lleftarrow}}$	$\underset{\textit{bottom}}{\Lleftarrow}$	$\overset{\textit{top}}{\Lleftarrow}$	LEFTWARDS TRIPLE ARROW
U+21DB	$\Rrightarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\Rrightarrow}}$	$\underset{\textit{bottom}}{\Rrightarrow}$	$\overset{\textit{top}}{\Rrightarrow}$	RIGHTWARDS TRIPLE ARROW
U+21DC	$\leftsquigarrow$	$\overset{\textit{top}}{\underset{\textit{bottom}}{\leftsquigarrow}}$	$\underset{\textit{bottom}}{\leftsquigarrow}$	$\overset{\textit{top}}{\leftsquigarrow}$	LEFTWARDS SQUIGGLE ARROW
U+21E0	?	$\overset{\textit{top}}{\underset{\textit{bottom}}{}}$	$\underset{\textit{bottom}}{}$	$\overset{\textit{top}}{}$	LEFTWARDS DASHED ARROW
U+21E4	?	$\overset{\textit{top}}{\underset{\textit{bottom}}{}}$	$\underset{\textit{bottom}}{}$	$\overset{\textit{top}}{}$	LEFTWARDS ARROW TO BAR

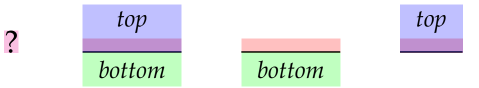
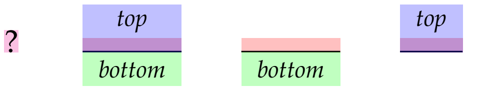
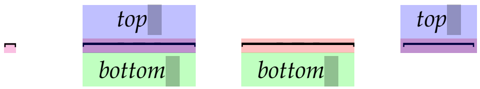
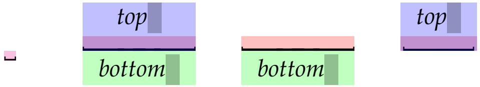
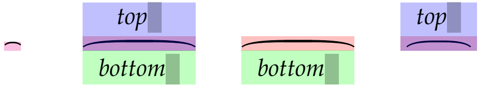
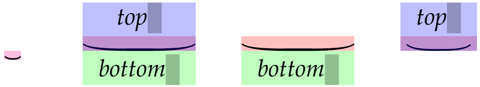
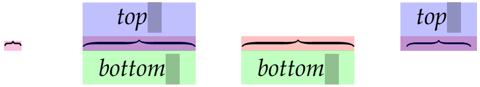
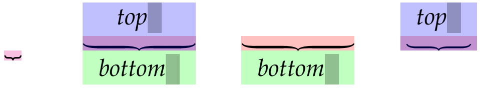
U+21E5	?	top bottom	bottom	top	RIGHTWARDS ARROW TO BAR
U+21E6	⇦	top bottom	bottom	top	LEFTWARDS WHITE ARROW
U+21E8	⇨	top bottom	bottom	top	RIGHTWARDS WHITE ARROW
U+21F4	?	top bottom	bottom	top	RIGHT ARROW WITH SMALL CIRCLE
U+21F6	⇶	top bottom	bottom	top	THREE RIGHTWARDS ARROWS
U+21F7	?	top bottom	bottom	top	LEFTWARDS ARROW WITH VERTICAL STROKE
U+21F8	?	top bottom	bottom	top	RIGHTWARDS ARROW WITH VERTICAL STROKE
U+21F9	?	top bottom	bottom	top	LEFT RIGHT ARROW WITH VERTICAL STROKE
U+21FA	?	top bottom	bottom	top	LEFTWARDS ARROW WITH DOUBLE VERTICAL STROKE
U+21FB	?	top bottom	bottom	top	RIGHTWARDS ARROW WITH DOUBLE VERTICAL STROKE
U+21FC	?	top bottom	bottom	top	LEFT RIGHT ARROW WITH DOUBLE VERTICAL STROKE
U+21FD	?	top bottom	bottom	top	LEFTWARDS OPEN-HEADED ARROW
U+21FE	?	top bottom	bottom	top	RIGHTWARDS OPEN-HEADED ARROW
U+21FF	?	top bottom	bottom	top	LEFT RIGHT OPEN-HEADED ARROW
U+2261	≡	top bottom	bottom	top	IDENTICAL TO
U+2262	≢	top bottom	bottom	top	NOT IDENTICAL TO
U+2263	≣	top bottom	bottom	top	STRICTLY EQUIVALENT TO
U+23B4	⎴	top bottom	bottom	top	TOP SQUARE BRACKET
U+23B5	⎵	top bottom	bottom	top	BOTTOM SQUARE BRACKET

U+23DC	⏜		TOP PARENTHESIS
U+23DD	⏝		BOTTOM PARENTHESIS
U+23DE	⏞		TOP CURLY BRACKET
U+23DF	⏟		BOTTOM CURLY BRACKET
U+27F4	⟴		RIGHT ARROW WITH CIRCLED PLUS
U+27F5	⟵		LONG LEFTWARDS ARROW
U+27F6	⟶		LONG RIGHTWARDS ARROW
U+27F7	⟷		LONG LEFT RIGHT ARROW
U+27F8	⟸		LONG LEFTWARDS DOUBLE ARROW
U+27F9	⟹		LONG RIGHTWARDS DOUBLE ARROW
U+27FA	⟺		LONG LEFT RIGHT DOUBLE ARROW
U+27FB	⟻		LONG LEFTWARDS ARROW FROM BAR
U+27FC	⟼		LONG RIGHTWARDS ARROW FROM BAR
U+27FD	⟽		LONG LEFTWARDS DOUBLE ARROW FROM BAR
U+27FE	⟾		LONG RIGHTWARDS DOUBLE ARROW FROM BAR
U+27FF	⟿		LONG RIGHTWARDS SQUIGGLE ARROW
U+2900	?		RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
U+2901	?		RIGHTWARDS TWO-HEADED ARROW WITH DOUBLE VERTICAL STROKE

U+2902	?	top bottom	bottom	top	LEFTWARDS DOUBLE ARROW WITH VERTICAL STROKE
U+2903	?	top bottom	bottom	top	RIGHTWARDS DOUBLE ARROW WITH VERTICAL STROKE
U+2904	?	top bottom	bottom	top	LEFT RIGHT DOUBLE ARROW WITH VERTICAL STROKE
U+2905	?	top bottom	bottom	top	RIGHTWARDS TWO-HEADED ARROW FROM BAR
U+2906	⤆	top ⤆ bottom	⤆ bottom	top ⤆	LEFTWARDS DOUBLE ARROW FROM BAR
U+2907	⤇	top ⤇ bottom	⤇ bottom	top ⤇	RIGHTWARDS DOUBLE ARROW FROM BAR
U+290C	?	top bottom	bottom	top	LEFTWARDS DOUBLE DASH ARROW
U+290D	?	top bottom	bottom	top	RIGHTWARDS DOUBLE DASH ARROW
U+290E	?	top bottom	bottom	top	LEFTWARDS TRIPLE DASH ARROW
U+290F	?	top bottom	bottom	top	RIGHTWARDS TRIPLE DASH ARROW
U+2910	?	top bottom	bottom	top	RIGHTWARDS TWO-HEADED TRIPLE DASH ARROW
U+2911	?	top bottom	bottom	top	RIGHTWARDS ARROW WITH DOTTED STEM
U+2914	?	top bottom	bottom	top	RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
U+2915	?	top bottom	bottom	top	RIGHTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2916	?	top bottom	bottom	top	RIGHTWARDS TWO-HEADED ARROW WITH TAIL
U+2917	?	top bottom	bottom	top	RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE

U+2918	?	top bottom	bottom	top	RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2919	?	top bottom	bottom	top	LEFTWARDS ARROW-TAIL
U+291A	?	top bottom	bottom	top	RIGHTWARDS ARROW-TAIL
U+291B	?	top bottom	bottom	top	LEFTWARDS DOUBLE ARROW-TAIL
U+291C	?	top bottom	bottom	top	RIGHTWARDS DOUBLE ARROW-TAIL
U+291D	?	top bottom	bottom	top	LEFTWARDS ARROW TO BLACK DIAMOND
U+291E	?	top bottom	bottom	top	RIGHTWARDS ARROW TO BLACK DIAMOND
U+291F	?	top bottom	bottom	top	LEFTWARDS ARROW FROM BAR TO BLACK DIAMOND
U+2920	?	top bottom	bottom	top	RIGHTWARDS ARROW FROM BAR TO BLACK DIAMOND
U+2933	?	top bottom	bottom	top	WAVE ARROW POINTING DIRECTLY RIGHT
U+2938	?	top bottom	bottom	top	RIGHT-SIDE ARC CLOCKWISE ARROW
U+2939	?	top bottom	bottom	top	LEFT-SIDE ARC ANTICLOCKWISE ARROW
U+293E	?	top bottom	bottom	top	LOWER RIGHT SEMICIRCULAR CLOCKWISE ARROW
U+293F	?	top bottom	bottom	top	LOWER LEFT SEMICIRCULAR ANTICLOCKWISE ARROW
U+2945	?	top bottom	bottom	top	RIGHTWARDS ARROW WITH PLUS BELOW
U+2946	?	top bottom	bottom	top	LEFTWARDS ARROW WITH PLUS BELOW

U+2970	?	*top* *bottom*	*bottom*	*top*	RIGHT DOUBLE ARROW WITH ROUNDED HEAD
U+2A17	?	*top* *bottom*	*bottom*	*top*	INTEGRAL WITH LEFTWARDS ARROW WITH HOOK
U+2B30	?	*top* *bottom*	*bottom*	*top*	LEFT ARROW WITH SMALL CIRCLE
U+2B31	⬱	*top* ⬱ *bottom*	⬱ *bottom*	*top* ⬱	THREE LEFTWARDS ARROWS
U+2B32	?	*top* *bottom*	*bottom*	*top*	LEFT ARROW WITH CIRCLED PLUS
U+2B33	⬳	*top* *bottom*	*bottom*	*top*	LONG LEFTWARDS SQUIGGLE ARROW
U+2B34	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
U+2B35	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED ARROW WITH DOUBLE VERTICAL STROKE
U+2B36	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED ARROW FROM BAR
U+2B37	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED TRIPLE DASH ARROW
U+2B38	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS ARROW WITH DOTTED STEM
U+2B39	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
U+2B3A	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2B3B	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED ARROW WITH TAIL
U+2B3C	?	*top* *bottom*	*bottom*	*top*	LEFTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE

U+2B3D		LEFTWARDS TWO-HEADED ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2B3F		WAVE ARROW POINTING DIRECTLY LEFT
U+FE3B4		EXTENSIBLE OF 0x03B4
U+FE3B5		EXTENSIBLE OF 0x03B5
U+FE3DC		EXTENSIBLE OF 0x03DC
U+FE3DD		EXTENSIBLE OF 0x03DD
U+FE3DE		EXTENSIBLE OF 0x03DE
U+FE3DF		EXTENSIBLE OF 0x03DF

1.11 Remark

The number of extensions to the LuaTeX core math engine is not that large and mostly involves more control over spacing and support for Unicode math as OpenType math extensibles. However, a few years after writing this chapter the machinery was cleaned up a bit and in the process some more control was added to constructors for radicals, fractions and delimiters. The spacing and composition can be controlled in a bit more detail using keywords (and dimensions). Because in ConTeXt we already have mechanisms in place not much of that new functionality is used (yet). Also, in the meantime ConTeXt evolved further. This chapter is just a snapshot and it might even render a bit different in more recent versions of ConTeXt and/or LuaTeX. After all, it was written as part of the development story.

2 Speed

2.1 Introduction

In the 'mk' and `hybrid` progress reports I have spend some words on speed. Why is speed this important?

In the early days of ConTEXt I often had to process documents with thousands of pages and hundreds of thousands of hyperlinks. You can imagine that this took a while, especially when all kind of ornaments had to be added to the page: backgrounds, buttons with their own backgrounds and offsets, hyperlink colors dependent on their state, etc. Given that multiple runs were needed, this could mean that you'd leave the machine running all night in order to get the final document.

It was the time when computers got twice the speed with each iteration of hardware, so I suppose that it would run substantially faster on my current laptop, an old Dell M90 workhorse. Of course a recently added SSD drive adds a boost as well. But still, processing such documents on a machine with a 8Mhz 286 processor and 640 megabytes of memory was close to impossible. But, when I compare the speed of core duo M90 with for instance an M4600 with a i5 CPU running the same clock speed as the M90, I see a factor 2 improvement at most. Of course going for a extremely clocked desktop will be much faster, but we're no longer seeing a tenfold speedup every few years. On the contrary: we see a shift multiple cores, often running at a lower clock speed, with the assumption that threaded applications are used. This scales perfectly for web services and graphic manipulations but not so much for TEX. If we want go faster, we need to see where we can be more efficient within more or less frozen clock speeds.

Of course there are some developments that help us. First of all, for programs like TEX clever caching of files by the operating system helps a lot. Memory still becomes faster and CPU cached become larger too. For large documents with lots of resources an SSD works out great. As Lua uses floating point, speedup in that area also help with LuaTEX. We use virtual machines for TEX related services and for some reason that works out quite well, as the underlying operating system does lots of housekeeping in parallel. But, with all maxing out, we finally end up at the software itself, and in TEX this boils down to a core of compiled code along with lots of macro expansions and interpret Lua code.

In the end, the question remains what causes excessive runtimes. Is it the nature of the TEX expansion engine? Is it bad macro writing? Is there too much overhead? If you notice how fast processing the TEX book goes on modern hardware it is clear that the core engine is not the problem. It's no big deal to get 100 pages per second on documents that use relative a simple page builder and have macros that lack a flexible user interface.

Take the following example:

```
\starttext
\dorecurse{1000}{test\page}
```

```
\stoptext
```

We do nothing special here. We use the default Latin Modern fonts and process single words. No burden is put on the pagebuilder either. This way we get on a 2.33 Ghz T7600 CPU a performance of 185 pages per second.[2] The estimated Lua overhead in this 1000 page document is some 1.5 to 2 seconds. The following table shows the performance on such a test document with different page numbers in pps (reported pages per second).

# pages	pps
1	2
10	15
100	90
1000	185
10000	215

The startup time, measured on a zero page document, is 0.5 seconds. This includes loading the format, loading the embedded Lua scripts and initializing them, initializing and loading the file database, locating and loading some runtime files and loading the absolute minumum number of fonts: a regular and math Latin Modern. A few years before this writing that was more than a second, and the gain is due to a slightly faster Lua interpreter as well as improvements in ConTeXt.

So why does this matter at all, if on a larger document the startup time can be neglected? It does because when I have to implement a style for a project or are developing some functionality a fast edit–run–preview cycle is a must, if only because even a few second wait feels uncomfortable. On the other hand, when I process a manual of say 150 pages, which uses some tricks to explain matters, I don't care if the processing rate is between 5 and 15 pages per second, simply because you get (done) what you asked for. It mostly has to do with feeling comfortable.

There is one thing to keep in mind: such measurements can vary over time, as they depend on several factors. Even in the trivial case we need to:

- load macros and Lua code
- load additional files
- initialize the system, think of fonts and languages
- package the pages, which includes reverting to global document states
- create the final output stream (PDF)

The simple one word per page test is not that slow, and normally for 1000 pages we measure around 200 pps. However, due to some small speedups (that somehow add up) in three months time I could gain a lot:

# pages	Januari	April	May	(2013)
1	2	2	2	

---

[2] In this case the mingw version was used. A version using the native Windows compiler runs somewhat faster, although this depends on the compiler options.[3] With LuajitTeX the 185 pages per second become becomes 195 on a 1000 page document.

10	15	17	17
100	90	109	110
1000	185	234	259
10000	215	258	289

Among the improvements in April were a faster output to the console (first prototyped in Lua, later done in the LuaTeX engine itself), and a couple of low level Lua optimizations. In May a dirty (maybe too tricky) global document state restore trick has beeing introduced. Although these changes give nice speed bump, they will mostly go unnoticed in more realistic documents. There we are happy if we end up in the 20 pps range. So, in practice a more than 10 percent speedup between Januari and April is just a dream.[4]

There are many cases where it does matter to squeeze out every second possible. We run workflows where some six documents are generated from one source. If we forget about the initial overhead of fetching the source from a remote server[5] gaining half a second per document (if we start frech each needs two runs at least) means that the user will see the first result one second faster and have them all in six less than before. In that case it makes sense to identify bottlenecks in the more high level mechanisms.

And this is why during the development of ConTeXt and the transition from MkII to MkIV quite some time has been spent on avoiding bottlenecks. And, at this point we can safely conclude that, in spite of more advanced functionality, the current version of MkIV runs faster than the MkII versions in most cases, especially if you take the additional functionality into account (like Unicode input and fonts).

2.2 The TeX engine

Writing inefficient macros is not that hard. If they are used only a few times, for instance in setting up properties it plays no role. But if they're expanded many times it may make a difference. Because use and development of ConTeXt went hand in hand we always made sure that the overhead was kept at a minimum.

The parbuilder

There are a couple of places where document processing in a traditional TeX engine gets a performance hit. Let's start with the parbuilder. Although the paragraph builder is quite fast it can responsible for a decent amount of runtime. It is also a fact that the parbuilder of the engines derived from original TeX are more complex. For instance, Omega adds bidirectionality to the picture which involves some extra checking as well as more nodes in the list. The pdfTeX engine provides protrusion and expansions, and as that feature was primarily a topic of research it was never optimized.

---

[4] If you wonder why I still bother with such things: sometimes speedups are just a side effect of trying to accomplish something else, like less verbose output in full tracing mode.

[5] In the user interface we report the time it takes to fetch the source so that the typesetter can't be blamed for delays.

In LuaTeX the parbuilder is a mixture of the pdfTeX and Omega builders and adapted to the fact that we have split the hyphenation, ligature building, kerning and breaking a paragraph into lines. The protrusion and expansion code is still there but already for a few years I have alternative code for LuaTeX that simplifies the implementation and could in principle give a speed boost as well but till now we never found time to adapt the engine. Take the following test code:

```
\testfeatureonce{100}{\setbox0\hbox{\tufte \par}} \tufte \par
```

In MkIV we use Lua for doing fonts so when we measure this bit we get the used time for typesetting our `\tufte` quote without breaking it into lines. A normal LuaTeX run needs 0.80 seconds and a LuajitTeX run takes 0.47 seconds.[6]

```
\testfeatureonce{100}{\setbox0\vbox{\tufte \par}} \tufte \par
```

In this case LuaTeX needs 0.80 seconds and LuajitTeX needs 0.50 seconds and as we break the list into lines, we can deduct that close to zero seconds are needed to break 100 samples. This (often used) sample text has the interesting property that it has many hyphenation points and always gives multiple hyphenated lines. So, the parbuilder, if no protrusion and expansion are used, is real fast!

```
\startparbuilder[basic]
  \testfeatureonce{100}{\setbox0\vbox{\tufte \par}} \tufte \par
\stopparbuilder
```

Here we kick in our Lua version of the par builder. This takes 1.50 seconds for LuaTeX and 0.90 seconds for LuajitTeX. So, LuaTeX needs 0.70 seconds to break the quote into lines while LuajitTeX needs 0.43. If we stick to stock LuaTeX, this means that a medium complex paragraph needs 0.007 seconds of Lua time and this is not that is not a time to be worried about. Of course these numbers are not that accurate but the measurements are consistent over multiple runs for a specific combination of LuaTeX and MkIV. On a more modern machine it's probably also close to zero.

These measurements demonstrate that we should add some nuance to the assumption that parbuilding takes time. For this we need to distinguish between traditional TeX and LuaTeX. In traditional TeX you build an horizontal box or vertical box. In TeX speak these are called horizontal and vertical lists. The main text flow is a special case and called the main vertical list, but in this perspective you can consider it to be like a vertical box.

Each vertical box is split into lines. These lines are packed into horizontal boxes. In traditional TeX constructing a list starts with turning references to characters into glyphs and ligatures. Kerns get inserted between characters if the font requests that. When a vertical box is split into lines, discretionary nodes get inserted (hyphenation) and when font expansion or protrusion is enabled extra fonts with expanded dimensions get added.

---

[6] All measurements are on a Dell M90 laptop running Windows 8. I keep using this machine because it has a decent high res 4:3 screen. It's the same machine Luigi Scarso and I used when experimenting with LuajitTeX.

So, in the case of vertical box, building the paragraph is not really distinguished from ligaturing, kerning and hyphenation which means that the timing of this process is somewhat fuzzy. Also, because after the lines are identified some final packing of lines happens and the result gets added to a vertical list.

In LuaTeX all these stages are split into hyphenation, ligature building, kerning, line breaking and finalizing. When the callbacks are not enabled the normal machinery kicks in but still the stages are clearly separated. In the case of ConTeXt the font ligaturing and kerning get preceded by so called node mode font handling. This means that we have extra steps and there can be even more steps before and afterwards. And, hyphenation always happens on the whole list, contrary to traditional TeX that interweaves this. Keep in mind that because we can box and unbox and in that process add extra text the whole process can get repeated several times for the same list. Of course already treated glyphs and kerns are normally kept as they are.

So, because in LuaTeX the process of splitting into lines is separated we can safely conclude that it is real fast. Definitely compared to al the font related steps. So, let's go back to the tests and let's do the following:

```
\testfeatureonce{1000}{\setbox0\hbox{\tufte}}

\testfeatureonce{1000}{\setbox0\vbox{\tufte}}

\startparbuilder[basic]
    \testfeatureonce{1000}{\setbox0\vbox{\tufte}}
\stopparbuilder
```

We've put the text into a macro so that we don't have interference from reading files. The test wrapper does the timing. The following measurements are somewhat rough but repetition gives similar results.[7]

	engine	**method**	**normal**	**hz**
1	luatex	tex hbox	9.64	9.64
2		tex vbox	9.84	10.16
3		lua vbox	17.28	18.43
4	luajittex	tex hbox	6.33	6.33
5		tex vbox	6.53	6.81
6		lua vbox	11.06	11.81

In line 1 we see the basline: hyphenation, processing fonts and hpacking takes 9.74 seconds. In the second line we see that breaking the 1000 paragraphs costs some 0.20 seconds and when expansion is enabled an extra 12 seconds is needed. This means that expansion takes 150% more runtime. If we delegate the task to Lua we need 7.64 seconds for breaking into lines which can not be neglected but is still ok given the fact that we break 1000 paragraphs. But, interesting is to see that our alternative expansion routine only adds 1.33 seconds which is less than 20%. It must be said that the built-in

---

[7] Before and between runs we do a garbage collection.

method is not that efficient by design if only because it started out differently as part of research.

When measured three months later, the numbers for regular LuaTeX (at that time version 0.77) with the latest ConTeXt were: 8.52, 8.72 and 15.40 seconds for the normal run, which demonstrates that we should not draw too many conclusions from such measurements. It's the overal picture that matters.

As with earlier timings, if we use LuajitTeX we see that the runtime of Lua is much lower (due to the virtual machine). Of course we're still 20 times slower than the built-in method but only 10 times slower when we use expansion. To put these numbers in perspective: 5 seconds for 1000 paragraphs.

```
\setupbodyfont[dejavu]

\starttext
  \dontcomplain \dorecurse{1000}{\tufte\par}
\stoptext
```

This results in 295 pages in the default layout and takes 17.8 seconds or 16.6 pages per second. Expansion is not enabled.

That one takes 24.7 seconds and runs at 11.9 pages per second. This is indeed slower but on a bit more modern machine I expect better results. We should also realize that with Dejavu being a relative large font a difficult paragraph like the tufte example gives overfull boxes which in turn is an indication that quite some alternative breaks are tried.

When typeset with Latin Modern we don't get overfull boxes and interesting is that the native method needs less time (15.9 seconds or 14.1 pages per second) while the Lua variant also runs a bit faster: 23.4 or 9.5 pages per second. The number of pages is 223 because this font is smaller by design.

When we disable hyphenation the the Dejavu variant takes 16.5 (instead of 17.8) seconds and 23.1 (instead of 24.7) seconds for Lua, so this process is not that demanding.

For typesetting so many paragraphs without anything special it makes no sense to bother with using a Lua based parbuilder. I must admit that I never had to typeset novels so all my 300 page runs are much longer anyway. Anyway, when at some point we introduce alternative parbuilding to ConTeXt, the speed penalty is probably acceptable.

Just to indicate that predictions are fuzzy: when we put a `\blank` between the paragraphs we end up with 313 pages and the traditional method takes 18.3 while Lua needs 23.6 seconds. One reason for this is that the whitespace is also handled by Lua and in the pagebuilder we do some finalizing, so we suddenly get interference of other processes (as well as the garbage collector). Again an indication that we should not bother too much about speed. I try to make sure that the Lua (as well as TeX) code is reasonably efficient, so in practice it's the document style that is a more important factor than the parbuilder, it being the traditional one or the Lua variant.

Copying boxes

As soon as in ConTeXt you start enhancing the page with headers and footers and backgrounds you will see that the pps rate drops. This is partly due to the fact that suddenly quite some macro expansion takes place in order to check what needs to happen (like font and color switches, offsets, overlays etc). But what has more impact is that we might end up with copying boxes and that takes time. Also, by wrapping and repackaging boxes, we add additional levels of recursion in postprocessing code.

Macro expansion

Taco and I once calculated that MkII spends some 4% of the time in accessing the hash table. This is a clear indication that quite some macro expansions goes on. Due to the fact that when I rewrote MkII into MkIV I no longer had to take memory and other limitations into account, the codebase looks quite different. There we do have more expansion in the mechanism that deals with settings but the body of macros is much smaller and less parameters are passed. So, the overall performance is better.

Fonts

Using a font has several aspects. First you have to define an instance. Then, when you use it for the first time, the font gets loaded from storage, initialized and is passed to TeX. All these steps are quite optimized. If we process the following file:

```
\setupbodyfont[dejavu]

\starttext
    regular, {\it italic}, {\bf bold ({\bi italic})} and $m^a_th$
\stoptext
```

we get reported:

```
loaded fonts      xits-math.otf xits-mathbold.otf
                  dejavuserif-bold.ttf dejavuserif-bolditalic.ttf
                  dejavuserif-italic.ttf dejavuserif.ttf
fonts load time   0.374 seconds
runtime           1.014 seconds, 0.986 pages/second
```

So, six fonts are loaded and because XITS is used we also preload the math bold variant. Loading of text fonts is delayed but in order initialize math we need to preload the math fonts.

If we don't define a bodyfont, a default set gets loaded: Latin Modern. In that case we get:

```
loaded fonts      latinmodern-math.otf
                  lmroman10-bolditalic.otf lmroman12-bold.otf
                  lmroman12-italic.otf lmroman12-regular.otf
```

```
fonts load time  0.265 seconds
runtime          0.874 seconds, 1.144 pages/second
```

Before we had native OpenType Latin Modern math fonts, it took slightly longer because we had to load many small Type1 fonts and assemble a virtual math font.

As soon as you start mixing more fonts and/or load additional weights and styles you will see these times increase. But if you use an already loaded font with a different featureset or scaled differently, the burden is rather low. It is safe to say that at this moment loading fonts is not a bottleneck.

Applying fonts can be more demanding. For instance if you typeset Arabic or Devanagari the amount of node and font juggling definitely influences the total runtime. As the code is rather optimized there is not much we can do about it. It's the price that comes with flexibility. As far as I can tell getting the same results with pdfTeX (if possible at all) or XeTeX is not taking less time. If you've split up your document in separate files you will seldom run more than a dozen pages which is then still bearable.

If you are for instance typesetting a dictionary like document, it does not make sense to do all font switches by switching body fonts. Just defining a couple of font instances makes more sense and comes at no cost. Being already quite efficient given the complexity you should not expect impressive speedups in this area.

Manipulations

The main manipulation that I have to do is to process xml into something readable. Using the built–in parser and mapper already has some advantages and if applied in the right way it's also rather efficient. The more you restrict your queries, the better.

Text manipulations using Lua are often quite fast and seldom the reason for seeing slow processing. You can do lots of things at the Lua end and still have all the ConTeXt magic by using the `context` namespace and function.

Multipass

You can try to save 1 second on a 20 second run but that is not that impressive if you need to process the document three times in order to get your cross references right. Okay you'd save 3 seconds but still to get result you needs some 60 seconds (unless you already have run the document before). If you have a predictable workflow you might know in advance that you only need two runs in case you can enforce that with `--runs=2`. Furthermore you can try to optimize the style by getting rid of redundant settings and inefficient font switches. But no matter what we optimize, unless we have a document with no cross references, sectioning and positioning, you often end up with the extra run, although ConTeXt tries to minimize the number of needed runs needed.

Trial runs

Some mechanisms, like extreme tables, need multiple passes and all but the last one are tagged as trial runs. Because in many cases only dimensions matter, we can disable some time consuming code in such case. For instance, at some point Alan Braslau and I found out that the new chemical manual ran real slow, mainly due to the tens of thousands of MetaPost graphics. Adding support for trial runs to the chemical structure macros gave a fourfold improvement. The manual is still a slow-runner, but that is simply because it has so many runtime generated graphics.

2.3 The METAPOST library

When the MetaPost library got included we saw a drastic speedup in processing document with lots of graphics. However, when MetaPost got a different number system (native, double and decimal) the changed memory model immediately lead to a slow down. On one 150 page manual which a graphic on each page I saw the MetaPost runtime go up from about half a second upto more than 5 seconds. In this case I was able to rewrite some core MetaFun macro to better suit the new model, but you might not be so lucky. So more careful coding is needed. Of course if you only have a few graphics, you can just ignore the change.

2.4 The LUA interpreter

Where the TEX part of LuaTEX is compiled, the Lua code gets interpreted, converted into bytecode, and ran by the virtual machine. Lua is by design quite portable, which means that the virtual machine is not optimized for a specific target. The LuaJIT interpreter on the other hand is written in assembler and available for only some platforms, but the virtual machine is about twice as fast. The just–in–time part of LuaJIT is not if much help and even can slow down processing.

When we moved from Lua 5.1 to 5.2 we found out that there was some speedup but it's hard to say why. There has been changes in the way strings are dealt with (Lua hashes strings) and we use lots of strings, really lots. There has been changes in the garbage collection and during a run lots of garbage needs to be collected. There are some fundamental changes in so called environments and who knows what impact that has.

If you ever tried to measure the performance of Lua, you probably have noticed that it is quite fast. This means that it makes no sense to optimize code that gets visited only occasionally. But some of the ConTEXt code gets exercised a lot, for instance all code that deals with fonts. We use attributes a lot and checking them is for good reason not the fastest code. But given the often advanced functionality that it makes possible we're willing to pay the price. It's also functionality that you seldom need all at the same time and for straightforward text only documents all that code is never executed.

When writing TEX or Lua code I spent a lot of time making it as efficient as possible in terms of performance and memory usage. The sole reason for that is that we happen to process documents where a lot of functionality is combined, so if many small speed–ups accumulate to a noticeable performance gain it's worth the effort.

So, where does Lua influence runtime? First of all we use Lua do deal with all in- and output as well as locating files in the TeX directory structure. Because that code is partly shared with the script manager (`mtxrun`) it is optimized but some more is possible if needed. It is already not the most easy to read code, so I don't want to introduce even more obscurity.

Quite some code deals with loading, preparing and caching fonts. That code is mostly optimized for memory usage although speed is also okay. This code is only called when a font is loaded for the first time (after an update). After that loading is at matter of milliseconds. When a text gets typeset and when fonts are processed in so called node mode, depending on the script and/or enabled features, a substantial amount of time is spent in Lua. There is still a bit complex dealing with inserting kerns but future LuaTeX will carry kerning in the glyph node so there we can gain some runtime.

If a page has 4000 characters and if font features as well as other manipulations demand 10 runs over the text, we have 40.000 checks of nodes and potential actions. Each involves an id check, maybe a subtype check, maybe some attribute checking and possibly some action. So, if we have 200.000 (or more) function calls to the per page TeX end it might add up to a lot. Around the time that we went to Lua 5.2 and played with LuajitTeX, the node accessors have been sped up. This gave indeed a measurable speedup but not on an average document, only on the more extreme documents or features. Because the MkIV Lua code goes from experimental to production to final, some improvements are made in the process but there is not much to gain there. We just have to wait till computers get faster, cpu cache get bigger, branch prediction improves, floating point calculations take less time, memory is speedy, and flash storage is the standard.

The Lua code is plugged into the TeX machinery via callbacks. For instance each time a box is build several callbacks are triggered, even if it's an empty box or just an extra wrapper. Take for instance this:

```
\hbox \bgroup
    \hskip \zeropoint
    \hbox \bgroup
        test
    \egroup
    \hskip \zeropoint
\egroup
```

Of course you won't come up with this code as it doesn't do much good but macros that you use can definitely produce this. For instance, the zero skip can be a left and right margin that happen to be. For 10.000 iterations I measured 0.78 seconds while the text one takes 0.62 seconds:

```
\hbox \bgroup
    \hbox \bgroup
```

```
        test
    \egroup
\egroup
```

Why is this? One reason is that a zero skip results in a node and the more nodes we have the more memory (de)allocation takes place and the more nodes in the list need to be checked. Of course the relative difference is less when we have more text. So how can we improve this? The following variant, at the cost of some testing takes just as much time.

```
\hbox \bgroup
    \hbox \bgroup
        \scratchdimen\zeropoint
        \ifdim\scratchdimen=\zeropoint\else\hskip\scratchdimen\fi
        test
        \ifdim\scratchdimen=\zeropoint\else\hskip\scratchdimen\fi
    \egroup
\egroup
```

As does this one, but the longer the text, the slower it gets as one of the two copies needs to be skipped.

```
\hbox \bgroup
    \hbox \bgroup
        \scratchdimen\zeropoint
        \ifdim\scratchdimen=\zeropoint
            test%
        \else
            \hskip\scratchdimen
            test%
            \hskip\scratchdimen
        \fi
    \egroup
\egroup
```

Of course most speedup is gained when we don't package at all, so if we test before we package but such an optimization is seldom realistic because much more goes on and we cannot check for everything. Also, 10.000 is a lot while 0.10 seconds is something we can live with. By the way, compare the following

```
\hbox \bgroup
    \hskip\zeropoint
    test%
    \hskip\zeropoint
\egroup

\hbox \bgroup
    \kern\zeropoint
    test%
    \kern\zeropoint
\egroup
```

The first variant is less efficient that the second one, because a skip effectively is a glue node pointing to a specification node while a kern is just a simple node with the width stored in it.[8] I must admit that I seldom keep in mind to use kerns instead of skips when possible if only because one needs to be sure to be in the right mode, horizontal or vertical, so additional commands might be needed.

2.5 Macros

Are macros a bottleneck? In practice not really. Of course we have optimized the core ConTEXt macros pretty well, but one reason for that is that we have a rather extensive set of configuration and definition mechanisms that rely heavily on inheritance. Where possible all that code is written in a way that macro expansion won't hurt too much. because of this users themselves can be more liberal in coding. There is a lot going on deep down and if you turn on tracing macros you can get horrified. But, not all shown code paths are entered. During the move (and rewrite) from MkII to MkIV quite some bottlenecks that result from limitations of machines and memory have been removed and as a result the macro expansion part is somewhat faster, which nicely compensates the fact that we have a more advanced but slower inheritance subsystem. Readability of code and speed are probably nicely balanced by now.

Once a macro is read in, its internal representation is pretty efficient. For instance references to macro names are just pointers into a hash table. Of course, when a macro is seen in your source, that name has to be looked up, but that's a fast action. Using short names in the running text for instance really doesn't speed up processing much. Switching font sets on the other hand does, as then quite some checking happens and the related macros are pretty extensive. However, once a font is loaded references to it a pretty fast. Just keep in mind that if you define something inside a group, in most cases it got forgotten. So, if you need something more often, just define it at the outer level.

2.6 Optimizing code

Optimizing only makes sense if used very often and called frequently or when the problem to solve is demanding. An example of code that gets done often is page building, where we pack together many layout elements. Font switches can also be time consuming, if defined wrong. These can happen for instance for formulas, marked words, cross references, margin notes, footnotes (often a complete bodyfont switch), table cells, etc. Yet another is clever vertical spacing that happens between structural elements. All these mechanisms are reasonably optimized.

I can safely say that deep down ConTEXt is no that inefficient, given what it has to do. But when a style for instance does redundant or unnecessary massive font switches you are wasting runtime. I dare to say that instead of trying to speed up code (for instance

---

[8] On the LuaTEX agenda is moving the glue spec into the glue node.

by redefining macros) you can better spend the time in making styles efficient. For instance having 10 `\blank`'s in a row will work out rather well but takes time. If you know that a section head has no raised or lowered text and no math, you can consider using `\definefont` to define the right size (especially if it is a special size) instead of defining an extra bodyfont size and switch to that as it includes setting up related sizes and math.

It might sound like using Lua for some tasks makes ConTEXt slower, but this is not true. Of course it's hard to prove because by now we also have more advanced font support, cleaner math mechanisms, additional features especially in especially structure related mechanisms, etc. There are also mechanisms that are faster, for instance extreme tables (a follow up on natural tables) and mixed column modes. Of course on the previously mentioned 300 page simple paragraphs with simple Latin text the pdfTEX engine is much faster than LuaTEX, also because simple fonts are used. But for many of todays document this engine is no longer an options. For instance in our xml processing in multiple languages, LuaTEX beats pdfTEX. There is not that much to optimize left, so most speed up has to come from faster machines. And this is not much different from the past: processing 300 page document on a 4.7Mhz 8086 architecture was not much fun and we're not even talking of advanced macros here. Faster machines made more clever and user friendly systems possible but at the cost of runtime, to even if machines have become many times faster, processing still takes time. On the other hand, ConTEXt will not become more complex than it is now, so from now on we can benefit from faster cpu's, memory and storage.

3 Math Styles

3.1 Introduction

Because ConTeXt is often considered somewhat less math savvy than for instance LaTeX we have more freedom to experiment with new insights and move forward. Of course ConTeXt always could deal with math, and even provides rather advanced support when it comes to combining fonts, which at some point was needed for a magazine that used two completely different sets of fonts in one issue. Also, many of the mechanisms had ways to influence the rendering, but often by means of constants and flags.

Already in an early stage of LuaTeX we went Unicode and after that the low level code has been cleaned up stepwise. In fact, we probably have less code now than before because we need less hacks. Well, this might not be that true, if we consider that we also introduced code at the Lua end which wasn't there before, but which makes makes support easier.

Because we don't need to support all kind of third party math extensions that themselves might depend on overloading low level implementations, we can rigourously replace mechanisms. In the process we also make things easier to configure, easier to define and we promote some previously low level tuning options at the user level.

Or course, by introducing new features and more options, there is a price to pay in terms of speed, but in practice users will seldom use the more complex constructs thousands of times in one document. Elsewhere arrows and alike were discussed, here I will spend some words on math styles and will use fences and fractions as an example as these mechanisms were used to experiment.

3.2 Math styles

In TeX a formula can used three different sizes of a font: text, script and scriptscript. In addition a formula can be typeset using rules for display math or rules for inline math. This means that we have the following so called math styles:

keyword	**meaning**	**command**
`display`	used for display math	`\displaystyle`
`text`	used for inline math	`\textstyle`
`script`	smaller than text style	`\scriptstyle`
`scriptscript`	smaller than script style	`\scriptscriptstyle`

Each of these commands will force a style but in practice you seldom need to do that because TeX does it automatically. In addition there is are cramped styles with corresponding commands.

$x^2 + \sqrt{x^2 + 2x} + \sqrt{x^2 + 2x}$	`\displaystyle`
$x^2 + \sqrt{x^2 + 2x} + \sqrt{x^2 + 2x}$	`\crampeddisplaystyle`

$\textstyle x^2 + \sqrt{x^2+2x} + \sqrt{x^2+2x}$	`\textstyle`
$\textstyle x^2 + \sqrt{x^2+2x} + \sqrt{x^2+2x}$	`\crampedtextstyle`
$\scriptstyle x^2+\sqrt{x^2+2x}+\sqrt{x^2+2x}$	`\scriptstyle`
$\scriptstyle x^2+\sqrt{x^2+2x}+\sqrt{x^2+2x}$	`\crampedscriptstyle`
$\scriptscriptstyle x^2+\sqrt{x^2+2x}+\sqrt{x^2+2x}$	`\scriptscriptstyle`
$\scriptscriptstyle x^2+\sqrt{x^2+2x}+\sqrt{x^2+2x}$	`\crampedscriptscriptstyle`

Here we applied the styles as follows:

```
$\textstyle x^2 + \sqrt{x^2+2x} + \sqrt{\textstyle x^2+2x}$
```

The differences are subtle: the superscripts in the square root are positioned a bit lower than normal: the radical forces them to be cramped.

$$x^2 + \sqrt{x^2 + 2x} + \sqrt{x^2 + 2x}$$

Although the average user will not bother about styles, a math power user might get excited about the possibility to control the size of fonts being used, of course wit the danger of creating a visually inconsistent document. And, as in ConTEXt we try to avoid such low level commands[9] it will be no surprise that we have ways to set them beforehand.

```
\definemathstyle[mystyle][scriptscript]

$ 2x + \startmathstyle [mystyle] 4y^2 \stopmathstyle = 10 $
```

So, if you want it this ugly, you can get it:

$2x + {\scriptscriptstyle 4y^2} = 10$

A style can be a combination of keywords. Of course we have `display`, `text`, `script` and `scriptscript`. Then there are `uncramped` and `cramped` along with their synonyms `normal` and `packed`. In some cases you can also use `small` and `big` which will promote the size up or down, relative to what we have currently.

A style definition can be combination of such keywords:

```
\definemathstyle[mystyle][scriptscript,cramped]
```

Gradually we will introduce the `mathstyle` keyword in math related setups commands.

In most cases a user will limit the scope of some setting by using braces, like this:

This gives $x{\scriptstyle x}x$: a smaller symbol between two with text size. Equally valid is this:

```
$x\startmathstyle[script]x\stopmathstyle x$
```

---

[9] Although . . . it's pretty hard to convince users to stay away from `\vskip` and friends.

Again we get xxx, but at the cost of more verbose coding.

The use of `{}` (either or not hidden in commands) has a few side effects. In text mode, when we use this at the start of a paragraph, the paragraph will start inside the group and when we end the group, specific settings that were done at that time get lost. So, in practice you will force a paragraph outside the group using `\dontleavehmode`, `\strut`, or one of the indentation commands.

In math mode a new math group is created which limits local style settings to this group. But as such groups also can trigger special kinds of spacing you sometimes don't want that. One pitfall is then to do this:

```
$x\begingroup\setupmathstyle[script]x\endgroup x$
```

Alas, now we get xxx. A `\begingroup` limits the scope of many things but it will not create a math group! This kind of subtle issues is the reason why we have pre-built solutions that take care of style switching, grouping, spacing and positioning.

3.3 Fences

Fences are symbols at the left and right of an expression: braces, brackets, curly braces, and bars are the most well known. Often they are supposed to adapt their size to the content that they wrap. Here you see some in action:

<table>
<tr><td><code>$|x|$</code></td><td>$|x|$</td><td>okay</td></tr>
<tr><td><code>$||x||$</code></td><td>$\|x\|$</td><td>okay</td></tr>
<tr><td><code>$a\left | \frac{1}{b}\right | c$</code></td><td>$a\left|\frac{1}{b}\right|c$</td><td>okay</td></tr>
<tr><td><code>$a\left ||\frac{1}{b}\right ||c$</code></td><td>$a\left|{|}\frac{1}{b}\right|{|}c$</td><td>wrong</td></tr>
<tr><td><code>$a\left ‖ \frac{1}{b}\right ‖ c$</code></td><td>$a\left\|\frac{1}{b}\right\|c$</td><td>okay</td></tr>
</table>

Often authors like to code their math with minimal structure and if you use Unicode characters that is actually quite doable. Just look at the double bar in the example above: if we input `||` we don't get what we want, but with `‖` the result is okay. This is because the `\left` and `\right` commands expect one character. But, even then, coding a bit more verbose sometimes makes sense.

In stock ConTeXt we have a couple of predefined fences:

```
\definemathfence [parenthesis] [left=0x0028,right=0x0029]
\definemathfence [bracket]     [left=0x005B,right=0x005D]
\definemathfence [braces]      [left=0x007B,right=0x007D]
\definemathfence [bar]         [left=0x007C,right=0x007C]
\definemathfence [doublebar]   [left=0x2016,right=0x2016]
\definemathfence [angle]       [left=0x003C,right=0x003E]
```

You use these by name:

```
test $a \fenced[bar]      {\frac{1}{b}} c$ test
test $a \fenced[doublebar]{\frac{1}{b}} c$ test
test $a \fenced[bracket]  {\frac{1}{b}} c$ test
```

and get

test $a\left|\frac{1}{b}\right|c$ test
test $a\left\|\frac{1}{b}\right\|c$ test
test $a\left[\frac{1}{b}\right]c$ test

You can stick to only one fence:

```
\definemathfence [nooffence] [left=0x005B]
```

Here ConTeXt will take care of the dummy fence that TeX expects instead.

on $a\ x005B\frac{1}{b}\ c$ off

You can define new fences and clone existing ones. You can also assign some properties:

```
\definemathfence
  [fancybracket]
  [bracket]
  [command=yes,
   color=blue]
```

```
test $a\fancybracket{\frac{1}{b}}c$ test
test \color[red]{$a\fancybracket{\frac{1}{b}}c$} test
```

The color is only applied to the fence. This makes sense as the formula can follow the main color but influencing the fences is technically somewhat more complex.

test $a\left[\frac{1}{b}\right]c$ test test $a\left[\frac{1}{b}\right]c$ test

Here are some more examples:

```
\definemathfence
  [normalbracket]
  [bracket]
  [command=yes,
   color=blue]

\definemathfence
  [scriptbracket]
  [normalbracket]
  [mathstyle=script]

\definemathfence
  [smallbracket]
  [normalbracket]
  [mathstyle=small]
```

`$a \frac{1}{b} c$`	$a\tfrac{1}{b}c$
`$a \normalbracket{\frac{1}{b} c$}`	$a\left[\frac{1}{b}\right]c$
`$a \scriptbracket{\frac{1}{b} c$}`	$a\left[\scriptstyle\frac{1}{b}\right]c$
`$a \smallbracket{\frac{1}{b} c$}`	$a\left[\scriptstyle\frac{1}{b}\right]c$

As with most commands, the fences inherit from the parents so we can say:

```
\setupmathfences [color=red]
```

and get all our fences colored red. The `command` option results in a command being defined, which saves you some keying.

3.4 Fractions

In TeX the mechanism to put something on top of something else, separated by a horizontal rule, is driven by the `\over` primitive. That one has a (compared to other commands) somewhat different specification, in the sense that one of its arguments sits in front:

```
$ {{2x}\over{x^1}} $
```

Although to some extend this is considered to be more readable, macro packages often provide a `\frac` commands that goes like this:

```
$ \frac{2x}{x^1} $
```

There we have less braces and the arguments come after the command. As with the fences in the previous section, you can define your own fractions:

```
\definemathfraction
  [innerfrac]
  [frac]
  [alternative=inner,
   mathstyle=script,
   color=red]

\definemathfraction
  [outerfrac]
  [frac]
  [alternative=outer,
   mathstyle=script,
   color=blue]
```

The mathstyle and color are already discussed but the `alternative` is specific for these fractions. It determines if the style is applied to the whole fraction or to its components.

```
\startformula
```

```
\outerfrac{2a}{3b} = \innerfrac{2a}{3b} = \frac{2a}{3b}
\stopformula
```

As with fences, the color is only applied to the horizontal bar as there is no other easy way to color that otherwise.

$${\scriptstyle\frac{2a}{3b}} = \frac{2a}{3b} = \dfrac{2a}{3b}$$

As TEX has a couple of low level stackers, we provide an interface to that as well, but we hide the dirty details. For instance you can define left and right fences and influence the rule

```
\definemathfraction[fraca][rule=no,left=0x005B,right=0x007C]
\definemathfraction[fracb][rule=yes,left=0x007B,right=0x007D]
\definemathfraction[fracc][rule=auto,left=0x007C]
\definemathfraction[fracd][rule=yes,rulethickness=2pt,left=0x007C]
```

When `rule` is set to `auto`, we use TEX's values (derived from font metrics) for the thickness of rules, while `yes` triggers usage of the specified `rulethickness`.

```
\startformula
\fraca{a}{b} + \fracb{a}{b} + \fracc{a}{b} + \fracd{a}{b}
\stopformula
```

Gives:

$$\left[\genfrac{}{}{0pt}{}{a}{b}\right| + \left\{\genfrac{}{}{1pt}{}{a}{b}\right\} + \left|\frac{a}{b}\right. + \left|\genfrac{}{}{2pt}{}{a}{b}\right.$$

```
\definemathfraction
  [frace]
  [rule=yes,
   color=blue,
   rulethickness=1pt,
   left=0x005B,
   right=0x007C]
```

This fraction looks as follows (scaled up):

$$\left[\genfrac{}{}{1pt}{}{a}{b}\right|$$

So, the color is applied to the (optional) fences as well as to the (optional) rule. And when you color the whole formula as part of the context, you get

$$\left[\genfrac{}{}{1pt}{}{a}{b}\right|$$

There is a (maybe not so) subtle difference between fences that come with fractions and regular fences, Take these definitions:

```
\definemathfence    [parenta] [left=0x28,right=0x29,command=yes]
\definemathfraction [parentb] [left=0x28,right=0x29,rule=auto]
```

Of course the `b` variant takes less code:

```
\startformula
\parenta{\frac{a}{b}} + \parentb{a}{b}
\stopformula
```

But watch how the parentheses are also larger. At some point ConTeXt will provide a bit more control over this,

$$x28\frac{a}{b}\,x29+\left(\frac{a}{b}\right)$$

You can also influence the width of the rule, but that is not related to the style.

```
\definemathfraction
  [wfrac]
  [margin=.25em]

\definemathfraction
  [wwfrac]
  [margin=.50em]

\startformula
  \frac   {  a } { \frac {  b } {  c } } +
  \wfrac  {  a } { \frac {  b } {  c } } =
  \wwfrac { 2a } { \frac { 2b } { 2c } }
\stopformula
```

Both the nominator and denominator are widened by the margin:

$$\frac{a}{\frac{b}{c}}+\frac{a}{\frac{b}{c}}=\frac{2a}{\frac{2b}{2c}}$$

4 Calling Lua

4.1 Introduction

One evening, on Skype, Luigi and I were pondering about the somewhat disappointing impact of jit in LuajitTeX and one of the reasons we could come up with is that when you invoke Lua from inside TeX each `\directlua` gets an extensive treatment. Take the following:

```
\def\SomeValue#1%
  {\directlua{tex.print(math.sin(#1)/math.cos(2*#1))}}
```

Each time `\SomeValue` is expanded, the TeX parser will do the following:

- It sees `\directlua` and will jump to the related scanner.
- There it will see a `{` and enter a special mode in which it starts collecting tokens.
- In the process, it will expand control sequences that are expandable.
- The scanning ends when a matching `}` is seen.
- The collected tokens are converted into a regular (C) string.
- This string is passed to the `lua_load` function that compiles it into bytecode.
- The bytecode is executed and characters that are printed to TeX are injected into the input buffer.

In the process, some state information is set and reset and errors are dealt with. Although it looks like a lot of actions, this all happens very fast, so fast actually that for regular usage you don't need to bother about it.

There are however applications where you might want to see a performance boost, for instance when you're crunching numbers that end up in tables or graphics while processing the document. Again, this is not that typical for jobs, but with the availability of Lua more of that kind of usage will show up. And, as we now also have LuajitTeX its jitting capabilities could be an advantage.

Back to the example: there are two calls to functions there and apart from the fact that they need to be resolved in the `math` table, they also are executed C functions. As LuaJIT optimizes known functions like this, there can be a potential speed gain but as `\directlua` is parsed and loaded each time, the jit machinery will not do that, unless the same code gets exercised lots of time. In fact, the jit related overhead would be a waste in this one time usage.

In the next sections we will show two variants that follow a different approach and as a consequence can speed up a bit. But, be warned: the impact is not as large as you might expect, and as the code might look less intuitive, the good old `\directlua` command is still the advised method.

Before we move on it's important to realize that a `\directlua` call is in fact a function call. Say that we have this:

```
\def\SomeValue{1.23}
```

This becomes:

```
\directlua{tex.print(math.sin(1.23)/math.cos(2*1.23))}
```

Which in LUA is wrapped up as:

```
function()
    tex.print(math.sin(1.23)/math.cos(2*1.23))
end
```

that gets executed. So, the code is always wrapped in a function. Being a function it is also a closure and therefore local variables are local to this function and are invisible at the outer level.

4.2 Indirect LUA

The first variant is tagged as indirect LUA. With indirect we mean that instead of directly parsing, compiling and executing the code, it is done in steps. This method is not as generic a the one discussed in the next section, but for cases where relatively constant calls are used it is fine. Consider the next call:

```
\def\NextValue
  {\indirectlua{myfunctions.nextvalue()}}
```

This macro does not pass values and always looks the same. Of course there can be much more code, for instance the following is equally valid:

```
\def\MoreValues {\indirectlua{
    for i=1,100 do
        myfunctions.nextvalue(i)
    end
}}
```

Again, there is no variable information passed from TEX. Even the next variant is relative constant:

```
\def\SomeValues#1{\indirectlua{
    for i=1,#1 do
        myfunctions.nextvalue(i)
    end
}}
```

especially when this macro is called many times with the same value. So how does `\indirectlua` work? Well, it's behaviour is in fact undefined! It does, like `\directlua`, parse the argument and makes the string, but instead of calling LUA directly, it will pass the string to a LUA function `lua_call`.

```
lua.call = function(s) load(s)() end
```

The previous definition is quite okay and in fact makes `\indirectlua` behave like `\directlua`. This definition makes

```
\directlua  {tex.print(math.sin(1.23))}
\indirectlua{tex.print(math.sin(1.23))}
```

equivalent calls but the second one is slightly slower, which is to be expected due to the wrapping and indirect loading. But look at this:

```
local indirectcalls = { }

function lua.call(code)
    local fun = indirectcalls[code]
    if not fun then
        fun = load(code)
        if type(fun) ~= "function" then
            fun = function() end
        end
        indirectcalls[code] = fun
    end
    fun()
end
```

This time the code needs about one third of the runtime. How much we gain depends on the size of the code and its complexity, but on the average its's much faster. Of course, during a TEX job only a small part of the time is spent on this, so the overall impact is much smaller, but it makes runtime number crunching more feasible.

If we bring jit into the picture, the situation becomes somewhat more diffuse. When we use LuajitTEX the whole job processed faster, also this part, but because loading and interpreting is more optimized the impact might be less. If you enable jit, in most cases a run is slower than normal. But as soon as you have millions of calls to e.g. type math.sin it might make a difference.

This variant of calling Lua is quite intuitive and also permits us to implement specific solutions because the `lua.call` function can be defined as you with. Of course macro package writers can decide to use this feature too, so you need to beware of unpleasant side effects if you redefine this function.

4.3 Calling LUA

In the process we did some tests with indirect calls in ConTEXt core code and indeed some gain in speed could be noticed. However, many calls get variable input and therefore don't qualify. Also, as a mixture of `\directlua` and `\indirectlua` calls in the source can be confusing it only makes sense to use this feature in real time-critical cases, because even in moderately complex documents there are not that many calls anyway.

The next method uses a slightly different approach. Here we stay at the TEX end, parse some basic type arguments, push them on the Lua stack, and call a predefined function. The amount of parsing TEX code is not less, but especially when we pass numbers stored in registers, no tokenization (serialization of a number value into the input stream) and stringification (converting the tokens back to a Lua number) takes place.

```
\indirectluacall 123
    {some string}
    \scratchcounter
    {another string}
    true
    \dimexpr 10pt\relax
\relax
```

Actually, an extension like this had been on the agenda for a while, but never really got much priority. The first number is a reference to a function to be called.

```
lua.calls = lua.calls or { }
lua.calls[123] = function(s1,n1,s2,b,n2)
    -- do something with
    --
    -- string  s1
    -- number  n1
    -- string  s2
    -- boolean b
    -- number  n2
end
```

The first number to `indirectluacall` is mandate. It can best also be a number that has a function associated in the `lua.calls` table. Following that number and before the also mandate `\relax`, there can be any number of arguments: strings, numbers and booleans.

Anything surrounded by `{}` becomes a string. The keywords `true` and `false` become boolean values. Spaces are skipped and everything else is assumed to be a number. This means that if you omit the final `\relax`, you get a error message mentioning a 'missing number'. The normal number parser applies, so when a dimension register is passed, it is turned into a number. The example shows that wrapping a more verbose dimension into a `\dimexpr` also works.

Performance wise, each string goes from list of tokens to temporary C string to LUA string, so that adds some overhead. A number is more efficient, especially when you pass it using a register. The booleans are simple sequences of character tokens so they are relatively efficient too. Because LUA functions accept an arbitrary number of arguments, you can provide as many as you like, or even less than the function expects: it is all driven by the final `\relax`.

An important characteristic of this kind of call is that there is no `load` involved, which means that the functions in `lua.calls` can be subjected to jitting.

4.4 Name spaces

As with `\indirectlua` there is a potential clash when users mess with the `lua.calls` table without taking the macro package usage into account. It not that complex to define a variant that provides namespaces:

```
\newcount\indirectmain \indirectmain=1
\newcount\indirectuser \indirectuser=2

\indirectluacall \indirectmain
    {function 1}
    {some string}
\relax

\indirectluacall \indirectuser
    {function 1}
    {some string}
\relax
```

A matching implementation is this:

```
lua.calls = lua.calls or { }

local main = { }

lua.calls[1] = function(name,...)
    main[name](...)
end

main["function 1"] = function(a,b,c)
    -- do something with a,b,c
end

local user = { }

lua.calls[2] = function(name,...)
    user[name](...)
end

user["function 1"] = function(a,b,c)
    -- do something with a,b,c
end
```

Of course this is also ok:

```
\indirectluacall \indirectmain 1
    {some string}
\relax

\indirectluacall \indirectuser 1
    {some string}
\relax
```

with:

```
main[1] = function(a,b,c)
    -- do something with a,b,c
end

user[1] = function(a,b,c)
    -- do something with a,b,c
end
```

Normally a macro package, if it wants to expose this mechanism, will provide a more abstract interface that hides the implementation details. In that case the user is not supposed to touch `lua.calls` but this is not much different from the limitations in redefining primitives, so users can learn to live with this.

4.5 Practice

There are some limitations. For instance in ConTEXt we often pass tables and this is not implemented. Providing a special interface for that is possible but does not really help. Often the data passed that way is far from constant, so it can as well be parsed by Lua itself, which is quite efficient. We did some experiments with the more simple calls and the outcome is somewhat disputable. If we replace some of the 'critital' calls we can gain some 3% on a run of for instance the `fonts-mkiv.pdf` manual and a bit more on the command reference `cont-en.pdf`. The first manual uses lots of position tracking (an unfortunate side effect of using a specific feature that triggers continuous tracking) and low level font switches and many of these can benefit from the indirect call variant. The command reference manual uses xml processing and that involves many calls to the xml mapper and also does quite some string manipulations so again there is something to gain there.

The following numbers are just an indication, as only a subset of `\directlua` calls has been replaced. The 166 page font manual processes in about 9 seconds which is not bad given its complexity. The timings are on a Dell Precision M6700 with Core i7 3840QM, 16 GB memory, a fast SSD and 64 bit Windows 8. The binaries were cross compiled mingw 32 bit by Luigi.[10]

	LuaTEX	LuajitTEX	LuajitTEX + jit
direct	8.90	6.95	7.50
indirect	8.65	6.80	7.30

So, we can gain some 3% on such a document and given that we spend probably half the time in Lua, this means that these new features can make Lua run more than 5% faster which is not that bad for a couple of lines of extra code. For regular documents we can forget about jit which confirms earlier experiments. The commands reference has these timings:

---

[10] While testing with several function definitions we noticed that `math.random` in our binaries made jit twice as slow as normal, while for instance `math.sin` was 100 times faster. As the font manual uses the random function for rendering random punk examples it might have some negative impact. Our experience is that binaries compiled with the ms compiler are somewhat faster but as long as the engines that we test are compiled similarly the numbers can be compared.

	LuaTEX	LuajitTEX
direct	2.55	1.90
indirect	2.40	1.80

Here the differences are larger which is due to the fact that we can indirect most of the calls used in this processing. The document is rather simple but as mentioned is encoded in XML and the TEX–XML interface qualifies for this kind of speedups.

As Luigi is still trying to figure out why jitting doesn't work out so well, we also did some tests with (in itself useless) calculations. After all we need proof. The first test was a loop with 100.000 step doing a regular `\directlua`:

```
\directlua {
    local t = { }
    for i=1,10000
        do t[i] = math.sin(i/10000)
    end
}
```

The second test is a bit optimized. When we use jit this kind of optimizations happens automatically for known (!) functions so there is not much won.

```
\directlua {
    local sin = math.sin
    local t = { }
    for i=1,10000
        do t[i] = sin(i/10000)
    end
}
```

We also tested this with `\indirectlua` and therefore defined some functions to test the call variant:

```
lua.calls[1] = function()
    -- overhead
end

lua.calls[2] = function()
    local t = { }
    for i=1,10000 do
        t[i] = math.sin(i/10000) -- naive
    end
end

lua.calls[3] = function()
    local sin = math.sin
    local t = { }
    for i=1,10000 do
```

```
        t[i] = sin(i/10000) -- normal
    end
end
```

These are called with:

```
\indirectluacall0\relax
\indirectluacall1\relax
\indirectluacall2\relax
```

The overhead variant demonstrated that there was hardly any: less than 0.1 second.

		LuaTEX	LuajitTEX	LuajitTEX + jit
directlua	normal	167	64	46
	local	122	57	46
indirectlua	normal	166	63	45
	local	121	56	45
indirectluacall	normal	165	66	48
	local	120	60	47

The results are somewhat disappoint but not that unexpected. We do see a speedup with LuajitTEX and in this case even jitting makes sense. However in a regular typesetting run jitting will never catch up with the costs it carries for the overall process. The indirect call is somewhat faster than the direct call. Possible reasons are that hashing at the Lua end also costs time and the 100.000 calls from TEX to Lua is not that big a burden. The indirect call is therefore also not much faster because it has some additional parsing overhead at the TEX end. That one only speeds up when we pass arguments and even then not always the same amount. It is therefore mostly a convenience feature.

We left one aspect out and that is garbage collection. It might be that in large runs less loading has a positive impact on collecting garbage. We also need to keep in mind that careful application can have some real impact. Take the following example of ConTEXt code:

```
\dorecurse {1000} {

    \startsection[title=section #1]

        \startitemize[n,columns]
            \startitem test \stopitem
            \startitem test \stopitem
            \startitem test \stopitem
            \startitem test \stopitem
        \stopitemize

        \starttabulate[|l|p|]
            \NC test \NC test \NC \NR
            \NC test \NC test \NC \NR
            \NC test \NC test \NC \NR
        \stoptabulate
        test {\setfontfeature{smallcaps} abc} test
        test {\setfontfeature{smallcaps} abc} test
        test {\setfontfeature{smallcaps} abc} test
        test {\setfontfeature{smallcaps} abc} test
        test {\setfontfeature{smallcaps} abc} test
        test {\setfontfeature{smallcaps} abc} test

        \framed[align={lohi,middle}]{test}

        \startembeddedxtable
            \startxrow \startxcell x \stopxcell \startxcell x \stopxcell \stopxrow
            \startxrow \startxcell x \stopxcell \startxcell x \stopxcell \stopxrow
            \startxrow \startxcell x \stopxcell \startxcell x \stopxcell \stopxrow
            \startxrow \startxcell x \stopxcell \startxcell x \stopxcell \stopxrow
            \startxrow \startxcell x \stopxcell \startxcell x \stopxcell \stopxrow
        \stopembeddedxtable

    \stopsection

    \page

}
```

These macros happen to use mechanism that are candidates for indirectness. However, it doesn't happen often you you process thousands of pages with mostly tables and smallcaps (although tabular digits are a rather valid font feature in tables). For instance, in web services squeezing out a few tens of seconds might make sense if there is a large queue of documents.

	LuaTEX	LuajitTEX	LuajitTEX + jit
direct	19.1	15.9	15.8
indirect	18.0	15.2	15.0

Surprisingly, even jitting helps a bit here. Maybe it relates the the number of pages and the amount of calls but we didn't investigate this. By default jitting is off anyway. The impact of indirectness is more than in previous examples.

For this test a file was loaded that redefines some core ConTEXt code. This also has some overhead which means that numbers for the indirect case will be somewhat better if we decide to use these mechanisms in the core code. It is tempting to do that but it involves some work and it's always the question if a week of experimenting and coding will ever be compensated by less. After all, in this last test, a speed of 50 pages per second is not that bad a performance.

When looking at these numbers, keep in mind that it is still not clear if we end up using this functionality, and when ConTEXt will use it, it might be in a way that gives better or worse timings than mentioned above. For instance, storing Lua code in the format is possible, but these implementations force us to serialize the `lua.calls` mechanism and initialize them after format loading. For that reason alone, a more native solution is better.

4.6 Exploration

In the early days of LuaTEX Taco and I discussed an approach similar do registers which means that there is some `\...def` command available. The biggest challenge there is to come up with a decent way to define the arguments. On the one hand, using a hash syntax is natural to TEX, but using names is more natural to Lua. So, when we picked up that thread, solutions like this came up in a Skype session with Taco:

```
\luadef\myfunction#1#2{ tex.print(arg[1]+arg[2]) }
```

The Lua snippet becomes a function with this body:

```
local arg = { #1, #2 } -- can be preallocated and reused
-- the body as defined at the tex end
tex.print(arg[1]+arg[2])
```

Where `arg` is set each time. As we wrapped it in a function we can also put the arguments on the stack and use:

```
\luadef\myfunction#1#2{ tex.print((select(1,...))+(select(2,...)) }
```

Given that we can make select work this way (either or not by additional wrapping). Anyway, both these solutions are ugly and so we need to look further. Also, the `arg` variant mandates building a table. So, a natural next iteration is:

```
\luadef\myfunction a b { tex.print(a+b) }
```

Here it becomes already more natural:

```
local a = #1
local b = #2
-- the body as defined at the tex end
tex.print(a+b)
```

But, as we don't want to reload the body we need to push `#1` into the closure. This is a more static definition equivalent:

```
local a = select(1,...)
local b = select(2,...)
tex.print(a+b)
```

Keep in mind that we are not talking of some template that gets filled in and loaded, but about precompiled functions! So, a `#1` is not really put there but somehow pushed into the closure (we know the stack offsets).

Yet another issue is more direct alias. Say that we define a function at the Lua end and want to access it using this kind of interface.

```
function foo(a,b)
    tex.print(a+b)
end
```

Given that we have something:

```
\luadef \myfunctiona a b { tex.print(a+b) }
```

We can consider:

```
\luaref \myfunctionb 2 {foo}
```

The explicit number is debatable as it can be interesting to permit an arbitrary number of arguments here.

```
\myfunctiona{1}{2}
\myfunctionb{1}{2}
```

So, if we go for:

```
\luaref \myfunctionb {foo}
```

we can use `\relax` as terminator:

```
\myfunctiona{1}{2}
\myfunctionb{1}{2}\relax
```

In fact, the call method discussed in a previous section can be used here as well as it permits less arguments as well as mixed types. Think of this:

```
\luadef \myfunctiona a b c { tex.print(a or 0 + b or 0 + c or 0) }
\luaref \myfunctionb {foo}
```

with

```
function foo(a,b,c)
    tex.print(a or 0 + b or 0 + c or 0)
end
```

This could be all be valid:

```
\myfunctiona{1}{2}{3]\relax
\myfunctiona{1}\relax
\myfunctionb{1}{2}\relax
```

or (as in practice we want numbers):

```
\myfunctiona 1 \scratchcounter 3\relax
\myfunctiona 1 \relax
\myfunctionb 1 2 \relax
```

We basicaly get optional arguments for free, as long as we deal with it properly at the Lua end. The only condition with the `\luadef` case is that there can be no more than the given number of arguments, because that's how the function body gets initialized set up. In practice this is quite okay.

4.7 The follow up

We don't know what eventually will happen with LuaTEX. We might even (at least in ConTEXt) stick to the current approach because there not much to gain in terms of speed, convenience and (most of all) beauty.

Note: In LuaTEX 0.79 onward `\indirectlua` has been implemented as `\luafunction` and the `lua.calls` table is available as `lua.get_functions_table()`. A decent token parser has been discussed at the ConTEXt 2013 conference and will show up in due time. In addition, so called `latelua` nodes support function assignments and `user` nodes support a field for Lua values. Additional information can be associated with any nodes using the properties subsystem.

5 Luigi's nightmare

5.1 Introduction

If you have a bit of a background in programming and watch kids playing video games, either or not on a dedicates desktop machine, a console or even a mobile device, there is a good change that you realize how much processing power is involved. All those pixels get calculated many times per second, based on a dynamic model that not only involves characters, environment, physics and a story line but also immediately reacts on user input.

If on the other hand in your text editor hit the magic key combination that renders a document source into for instance a PDF file, you might wonder why that takes so many seconds. Of course it does matter that some resources are loaded, that maybe images are included, and lots of fuzzy logic makes things happen, but the most important factor is without doubt that TEX macros are not compiled into machine code but into an intermediate representation. Those macros then get expanded, often over and over again, and that a relative slow process. As (local) macros can be redefined any time, the engine needs to take that into account and there is not much caching going on, unless you explicitly define macros that do so. Take this:

```
\def\bar{test}
\def\foo{test \bar\space test}
```

Even if the definition of `\test` stays the same, that if `\bar` can change:

```
\foo \def\bar{foo} \foo
```

There is no mechanism to freeze the meaning of `\bar` in `\foo`, something that is possible in the other language used in ConTEXt:

```
local function bar() context("test") end
function foo() context("test ") bar() context(" test") end
```

Here we can use local functions to limit their scope.

```
foo() local function bar() context("foo") end foo()
```

In a way you can say that TEX is a bit more dynamic that Lua, and optimizing (as well as hardening) it is much more difficult. In ConTEXt we already stretched that to the limits, although occasionally I find ways to speed up a bit. Given that we spend a considerable amount of runtime in Lua it makes sense to see what we can gain there. We have less possible interference and often a more predictable outcome as `bar`s won't suddenly become `foo`s.

Nevertheless, the dynamic nature of both TEX and Lua has some impact on performance, especially when they do most of the work. While in games there are dedicated chips to do tasks, for TEX there aren't. So, we're sort of stuck when it comes to speeding up the process to the level that is similar to advanced games. In the next sections I will discuss a few aspects of possible speedups and the reason why it doesn't work out as expected.

5.2 Jitting

Let's go back once more to Luigi's nightmare of disappointing jit[11] We already know that the virtual machine of LuaJIT is about twice as fast as the standard machine. We also experienced that enabling jit can degrade performance. Although we did observe some real drastic drop in performance when testing functions like `math.random` using the `mingw` compiler, we also saw a performance boost with simple pure Lua functions. In that respect LuaJIT is an impressive effort. So, it makes sense to use LuajitTEX even if in theory it could be faster.

Next some tests will be shown. The timings are snapshots so different versions of LuajitTEX can have different outcomes. The tests are mostly used for discussions between Luigi and me and further experiments and believe me: we've really done all kind of tests to see if we can get some speed out of jitting. After all it's hard to believe that we can't gain something from it, so we might as do something wrong.

Each test is run 5000 times. These are of course non-typical examples but they illustrate the principle. Each time we show two measurements: one with jit turned on, and one with jit off, but in both cases the faster virtual machine is enabled. The times shown are of course dependent on the architecture and operating system, but as we are only interested in relative times it's enough to know that we run 32 bit mingw binaries under 64 bit Windows 8 on a modern quad core Ivy bridge CPU. We did most tests with LuaJIT 2.0.1 but as far as we can see 2.0.2 has a similar performance.

1 simple loops, no function calls

```
return function()
    local a = 0
    for i=1,10000 do
        a = a + i
    end
end
```

```
off  0
on   0
```

2 simple loops, with simple function

```
local function whatever(i)
    return i
end

return function()
```

---

[11] Luigi Scarso is the author of LuajitTEX and we have reported on experiments with this variant of LuaTEX on several occasions.

```
    local a = 0
    for i=1,10000 do
        a = a + whatever(i)
    end
end
```

```
off  0
on   0
```

3 simple loops, with built-in basic functions

```
return function()
    local a = 0
    for i=1,10000 do
        a = a + math.sin(1/i)
    end
end
```

```
off  0
on   0
```

4 simple loops, with built-in simple functions

```
return function()
    local a = 0
    for i=1,1000 do
        local a = a + tonumber(tostring(i))
    end
end
```

```
off  0
on   0
```

5 simple loops, with built-in simple functions

```
local tostring, tonumber = tostring, tonumber
return function()
    local a = 0
    for i=1,1000 do
        local a = a + tonumber(tostring(i))
    end
end
```

```
off  0
on   0
```

6 simple loops, with built-in complex functions

```
return function()
    local a = 0
    local p = (1-lpeg.P("5"))^0 * lpeg.P("5") + lpeg.Cc(0)
    for i=1,100 do
        local a = a + lpeg.match(p,tostring(i))
    end
end
```

```
off  0
on   0
```

7 simple loops, with foreign function

```
return function()
    local a = 0
    for i=1,10000 do
        a = a + font.current()
    end
end
```

```
off  0
on   0
```

8 simple loops, with wrapped foreign functions

```
local fc = font.current

function font.xcurrent()
    return fc()
end

return function()
    local a = 0
    for i=1,10000 do
        a = a + font.xcurrent()
    end
end
```

```
off  0
on   0
```

What we do observe here is that turning on jit doesn't always help. By design the current just-in-time compiler aborts optimization when it sees a function that is not known. This means that in LuajitTeX most code will not get jit, because we use built-in library calls a lot. Also, in version 2.0 we notice that a bit of extra wrapping will make performance worse too. This might be why for us jitting doesn't work out the way it is advertised. Often performance tests are done with simple functions that use built in functions that do get jit. And the more of those are supported, the better it gets. Although, when you profile a ConTeXt run, you will notice that we don't call that many standard library functions, at least not so often that jitting would get noticed.

A safe conclusion is that you can benefit a lot from the fast virtual machine but should check carefully if jit is not having a negative impact. As it is turned on by default in LuaJIT (but off in LuajitTeX) it might as well get unnoticed, especially because there is always a performance gain due to the faster virtual machine and that might show more overall gain than the drawback of jitting unjittable code. It might just be a bit less drastic then possible because of artifacts mentioned here, but who knows what future versions of LuaJIT will bring.

Maybe sometime we can benefit from `ffi` but it makes no sense to mess up the ConTeXt code with related calls: it looks ugly and also makes the code unusable in stock Lua, so it is a a sort of no-go. There are some suggestions in LuaJIT related posts about adapting the code to suit the jitter, but again, that makes no sense. If we need to keep a specific interpreter in mind, we could as well start writing everything in C. So, our hopes are on future versions of stock Lua and LuaJIT. Luigi uncovered the following comment in the source code:

```
/* C functions can have arbitrary side-effects and are not
recorded (yet). */
```

Although the `(yet)` indicates that at some point this restriction can be lifted, we don't expect this to happen soon. And patching the jit machinery ourselves to suite LuaTeX is no option.

There is an important difference between a LuaTeX run and other programs: they are runs and these live short. A lot of code gets executed only once of a few times (like loading fonts), or gets executed in such different ways that (branch) prediction is hard. If you run a web server using Lua it runs for weeks in a row so optimizing a function pays off, given that it gets optimized. When you have a Lua enhanced interactive program, again, the session is long enough to benefit from jitting (if applied). And, when you crunch numbers, it might pay off too. In practice, a TeX run has no such characteristics.

5.3 Implementation

In Lua 5.2 there are some changes in the implementation compared to 5.1 and before. It is hard to measure the impact of that but it's probably a win some here and loose some there situation. A good example is the way Lua deals with strings. Before 5.2 all strings were hashed, but now only short strings are (at most 32 bytes are looked at). Now, consider this:

- In ConTeXt we do all font handling in Lua and that involves lots of tables with lots of (nicely hashed) short keys. So, comparing them is pretty fast.
- We also read a lot from files, and each line passes filters and such before it gets passed to TeX. There hashing is not really needed, although when it gets processed by filters it might as well save some time.

- When we go from TEX to LUA and reverse, lots of strings are involved and many of them are unique and used once. There hashing might bring a penalty.
- When we loop over a string with `gmatch` or some `lpeg` subprogram lots of (small) strings can get created and each gets hashed, even if they have a short livespan.

The above items indicate that we can benefit from hashing but that sometimes it might have a performance hit. My impression is that on the average we're better off by hashing and it's one of the reasons why LUA is so fast (and useable).

In TEX all numbers are integers and in LUA all numbers are floats. On modern computers dealing with floating point is fast and we're not crunching numbers anyway. We definitely would have an issue when numbers were just integers and an upcoming mixed integer/float model might not be in our advantage. We'll see.

I had expected to benefit from bitwise operations but so far never could find a real application in CONTEXT, at least not one that had a positive impact. But maybe it's just a way of thinking that hasn't evolved yet. Also, the fact that functions are used instead of a real language extension makes it less possible that there is a speedup involved.

5.4 Garbage collection

In the beginning I played with tuning the LUA garbage collector in order to improve performance. For some documents changing the step and multiplier worked out well, but for others it didn't, so I decided that one can best leave the values as they are. Turning the garbage collector off as expected gives a relative small speedup, and for the average run the extra memory used can be neglected. Just keep in mind that a TEX run are never persistent so memory can't keep filling. I did some tests with the in theory faster (experimental) generational mode of the garbage collector but it made runs significantly slower. For instance processing the `fonts-mkiv.pdf` went from 9 to 9.5 seconds.

5.5 Conclusion

So what is, given unpredictable performance hits of advertised optimizations, the best approach. It all starts by the LUA (and TEX) code: sloppy coding can have a price. Some of that can be disguised by clever interpreters but some can't. If the code is already fast, there is not much to gain. When going from MKII to MKIV more and more LUA got introduced and lots of approaches were benchmarked, so, I'm already rather confident that there is not that much to gain. It will never have the impressive performance of interactive games and that's something we have to live with. As long as LUA stays lean and mean, things can only get better over time.

6 Flash forward

6.1 Introduction

At the 2013 ConTEXt meeting in Breslov, Harald König has taken some of his gadgets with him and this time the target was to get ConTEXt running on small devices, most noticeably a mobile phone. You may wonder what purpose this serves, but with such devices becoming more powerful each year, and desktops and laptops getting less popular, we might see the small devices taking their place. Especially when we can dock them in a cradle and connect them to a proper monitor and keyboard we might end up with universal devices. Combine that with projection on our retinas and less tactile input and it will be clear that we should at least look into this from the perspective of TEX usage.

6.2 The tests

We used five tests for measuring basic performance. Of course we made sure that binaries and resources were cached.

Test 1 measures some basics, like typesetting a paragraph, flushing pages and loading a file. Because we do lots of pages we can also see if garbage collection is a problem.

```
\starttext
  \dorecurse{1000}{\input ward \par}
\stoptext
```

A normal ConTEXt run is triggered with:

```
context speed-1
```

but with

```
context --timing speed-1
```

memory consumption is measured and one can generate a visual representation of this afterwards.

```
context --extra=timing speed-1
```

We don’t show them here, simply because we saw nothing exciting in the ones for these tests.

The second test is rather stupid but it gives an indication of how efficient the base pagebuilder is:

```
\starttext
  \dorecurse{1000}{test \page}
\stoptext
```

The numbers are normally 10 to 20 times more impressive than those for regular runs.

Test three is a variation on test one but this time we avoid the file being read in many times, so we inline `ward.tex`. We also add no page breaks so we get less pages but with more content.

```
\starttext
  \dorecurse{1000}{
    The Earth, as a habitat for animal life, is in old age and
    has a fatal illness. Several, in fact. It would be happening
    whether humans had ever evolved or not. But our presence is
    like the effect of an old|-|age patient who smokes many packs
    of cigarettes per day |=| and we humans are the cigarettes.
    \par
  }
\stoptext
```

The fourth test draws a few MetaPost graphics, which themselves use a bit of typeset text.

```
\starttext

\dorecurse{10} {
  \startMPcode
     draw fullcircle scaled 1cm withpen pencircle scaled 1mm ;
     draw textext("X") ;
  \stopMPcode
}

\stoptext
```

The last test, number five, is more demanding. Here we use some colors (which stresses the backend) and a dynamic switch to smallcaps, which puts a bit of a burden on the OpenType handler.

```
\setupbodyfont[pagella]

\starttext

\dorecurse {100} {
  \input ward \par
  \dorecurse{100} {
    \dontleavehmode
    {\green this is green}
    {\red \smallcaps this is red}
    {\blue \bf this is blue}
  }
  \par
}
```

```
\stoptext
```

6.3 Regular laptops

We started measuring on Haralds laptop, a Lenovo X201i, and got the following timings (that matched our expectations). The second column shows the runtime, the last column the pages per second.

speed-1	5.8	17.1
speed-2	3.6	275.6
speed-3	5.1	19.8
speed-4	0.6	1.8
speed-5	11.9	10.6

Just for comparison, as I'm wrapping this up in 2016, on my current Dell 7600 I get these timings (the last two columns are with LuajitTeX):

speed-1	4.6	21.9	3.0	33.5
speed-2	3.6	278.2	2.8	357.7
speed-3	4.2	23.6	2,7	37.0
speed-4	0.8	1.3	0.6	1.7
speed-5	6.2	20.3	4.0	31.9

These tests were run with a LuaTeX 0.98 and the most recent ConTeXt OpenType font processor. As we do more in Lua that a few years back, one can't expect a much faster run, even when the Dell has a faster processor than the Lenovo. However, what gets noticed is that the fifth speed test runs about twice as fast which is mostly due to improvements in the handling of OpenType features.

6.4 The Nexus IV

This mobile phone has a quad-core arm processor running at 1.5 GHz. With 2 Gb memory this should be sufficient for running TeX. The operating system is Android, which means that some effort is needed to put TeX with its resources on the internal flash disk. Access was remote from a laptop.

speed-1	41.9	2.4
speed-2	27.5	36.4
speed-3	38.7	2.6
speed-4	3.4	3.0
speed-5	87.9	1.4

So it looks like the phone runs these tests about five times slower than the laptop. The fifth test is most stressful on the hardware but as noted, a more recent ConTeXt will give better times there due to improvements in feature processing.

6.5 The Raspbery Pi

The Pi (we're talking of the first model here) has an extension bus and can be used to control whatever device, it has more the properties (and build) of a media player and indeed there are dedicated installations for that. But as this popular small device can host any LINUX distribution this is what was done. The distribution of choice was OpenSuse. The setup was really experimental with an unboxed Pi, an unframed LCD panel, a keyboard and mouse, a power supply and some wires to connect this all. With an ethernet cable running directly to the router a distribution could be fetched and installed.

This device has a single core arm processor running at 700 Mhz with half a gigabyte of memory. Persistent memory is a flash card, not that fast but acceptable. The maximum read speed was some 20 MB per second. It was no real surprise that the set of tests ran much slower than on the phone.

It took a bit of experimenting but a 200 Mhz overclock of the CPU combined with overclocked memory made performance jump up. In fact, we got a speed that we could somehow relate to the phone that has a more modern CPU and runs at 1.5 times that speed.

Being a regular LINUX setup, installation was more straightforward than on the phone but of course it took a while before all was in place. The default clock timings are:

speed-1	95.841	1.043
speed-2	76.817	13.018
speed-3	84.890	1.178
speed-4	13.241	0.076
speed-5	192.288	0.660

Again, the main conclusion here is that documents that need lots of OpenType feature juggling, this is not the best platform.

6.6 Summary

We see small devices gaining more performance each iteration than larger machines. Their screens and input method also evolve at a higher speed. The question is if arm will keep dominating this segment, but at least it is clear that they are useable for TEX processing. Keep in mind that we used LuaTEX, which means that we also have Lua with its garbage collector. Add ConTEXt to that, which is not that small and preloads quite some resources, and it will be clear that these devices actually perform quite well, given slower memory, slower disks, small caches etc. With down-scaled intel chips showing up it can only get better. Keep in mind that we only need one core, so the speed of one core matters more than having multiple cores available, although the other cores can be wasted on keeping up with your social demands on such a device in parallel with the TEX run.

A runtime five to ten times slower than a decent laptop is not something that we look forward to in a production environment, but when you're on the road it is quite okay, especially if it can replace a somewhat heavy portable workstation like we do. Okay, how much TEX processing do you need when mobile, but still. As vendors of server hardware are looking into high density servers with lots of small fast processors, we might at some point actually use TEX on such hardware. By then performance might be en par with virtual machines running on average loaded machines.

We are pretty sure that on following ConTEXt meetings more such experiments will be done so we'll keep you posted.

7 Font expansion

7.1 Introduction

A lot in LuaTEX is not new. It started as a mix of pdfTEX (which itself is built on top of original TEX and ε-TEX) and the directional bits of Aleph (which is a variant of Omega). Of course large portions have been changed in the meantime, most noticeably the input encoding (Unicode), fonts with a more generic fontloader and Lua based processing, Unicode math and related font rendering, and many subsystems can be overloaded or extended. But at the time I write this (end of January 2013) the parbuilder still has the pdfTEX font expansion code.

This code is the result of a research project by Hàn Thế Thành. By selectively widening shapes a better greyness of the paragraph can be achieved. This trick is inspired by the work of Hermann Zapf and therefore, instead of expansion, we often talk of *hz* optimization.

It started with (runtime) generated METAFONT bitmap fonts and as a consequence we ended up with many more font instances. However, when eventually bitmap support was dropped and outlines became the norm, the implementation didn't change much. Also some of the real work was delegated to the backend and as it goes then: never change a working system if there's no reason.

When I played with the Lua based par builder I quickly realized that this implementation was far from efficient. It was already known that enabling it slowed down par building and I saw that this was largely due to many redundant calculations, generating auxiliary fonts, and the interaction between front- and backend. And, as I seldom hesitate to reimplement something that can be done better (one reason why ConTEXt is never finished) I came to an alternative implementation. That was 2010. What helped was that by that time Hartmut Henkel already had made the backend part cleaner, in the sense that instead of including multiple instances of the same font (but with different glyph widths) the base font was transformed in-line. This made me realize that we could use just one font in the frontend and pass the scale with the glyph node to the backend. And so, an extra field was added to glyphs nodes in order to make experiments possible.

More than two years later (January 2013) I finally took up this pet project and figured out how to adapt the backend. Interestingly a few lines of extra code we all that was needed. At the same time the frontend part became much simpler, that is, in the Lua parbuilder. But eventually it will be retrofitted into the core engine, if only because that's much faster.

7.2 The changes

The most important changes are the following. Instead of multiple font instances, only one is used. This way less memory is used, no extra font instances need to be created (and those OpenType fonts can be large).

Because less calculations are needed the code looks less complex and more elegant. Okay, the parbuilder code will never really look easy, if only because much more is involved.

The glyph related factors are related to the emwidth. This makes not much sense so in ConTEXt we define them in fractions of the character width, map them onto emwidths, and in the parbuilder need to go to glyph related widths again. If we can get rid of these emwidths, we have less complex code.

Probably for reasons of efficiency an expanded font carries a definition that tells how much stretch and shrink is permitted and how large the steps are. So, for instance a font can be widened 5% and narrowed 3% in steps of 1% which gives at most 8 extra instances. There is no real reason why this should be a font property and the parbuilder cannot deal with fonts with different steps anyway, so it makes more sense to make it a property of the paragraph and treat all fonts alike. In the Lua based variant we can even have more granularity but we leave that for now. In any case this will lift the limitation of mixed font usage that is present in the original mechanism.

The front- and backend code with repect to expansion gets clearly separated. In fact, the backend doesn't need to do any calculations other than applying the factor that is carried with the glyph. This and previously mentioned simplifications make the mechanism more efficient.

It is debatable if expansion needs to be applied to font kerns, as is the case in the old mechanism. So, at least it should be an option. Removing this feature would again made the code nicer. If we keep it, we should keep in mind that expansion doesn't work well with complex fonts (say Arabic) but I will look into this later. It might be feasible when using the Lua based variant because then we can use some of the information that is carried around with the related mechanisms. Of course this then related to the Lua based font builder.

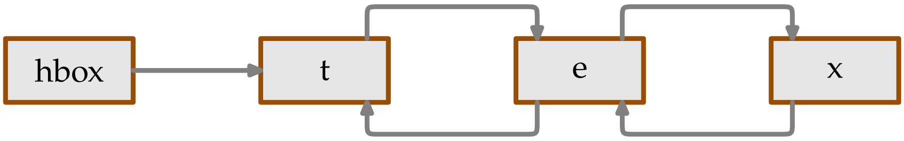
8 Juggling nodes

8.1 Introduction

When you use TEX, join the community, follow mailing lists, read manuals, and/or attend meetings, there will come a moment when you run into the word 'node'. But, as a regular user, even if you write macros, you can happily ignore them because in practice you will never really see them. They are hidden deep down in TEX.

Some expert TEXies love to talk about TEX's mouth, stomach, gut and other presumed bodily elements. Maybe it is seen as proof of the deeper understanding of this program as Don Knuth uses these analogies in his books about TEX when he discusses how TEX reads the input, translates it and digests it into a something that can be printed or viewed. No matter how your input gets digested, at some point we get nodes. However, as users have no real access to the internals, nodes never show themselves to the user. They have no bodily analogy either.

A character that is read from the input can become a character node. Multiple characters can become a linked list of nodes. Such a list can contain other kind of nodes as well, for instance spaced become glue. There can also be penalties that steer the machinery. And kerns too: fixed displacements. Such a list can be wrapped in a box. In the process hyphenation is applied, characters become glyphs and intermediate math nodes becomes a combination of regular glyphs, kerns and glue, wrapped into boxes. So, an hbox that contains the three glyphs `tex` can be represented as follows:



Eventually a long sequence of nodes can become a paragraph of lines and each line is a box. The lines together make a page which is also a box. There are many kind of nodes but some are rather special and don't translate directly to some visible result. When dealing with TEX as user we can forget about nodes: we never really see them.

In this example we see an hlist (hbox) node. Such a node has properties like width, height, depth, shift etc. The characters become glyph nodes that have (among other properties) a reference to a font, character, language.

Because TEX is also about math, and because math is somewhat special, we have noads, some intermediate kind of node that makes up a math list, that eventually gets transformed into a list of nodes. And, as proof of extensibility, Knuth came up with a special node that is more or less ignored by the machinery but travels with the list and can be dealt with in special backend code. Their name indicates what it's about: they are called whatsits (which sounds better that whatevers). In LuaTEX some whatsits are used in the frontend, for instance directional information is stored in whatsits.

The LuaTEX engine not only opens up the Unicode and OpenType universes, but also the traditional TEX engine. It gives us access to nodes. And this permits us to go beyond what was possible before and therefore on mailing lists like the ConTEXt list, the word node will pop up more frequently. If you look into the Lua files that ship with ConTEXt you cannot avoid seeing them. And, when you use the cld interface you might even want to manipulate them. A nice side effect is that you can sound like an expert without having to refer to bodily aspects of TEX: you just see them as some kind of Lua userdata variable. And you access them like tables: they are abstracts units with properties.

8.2 Basics

Nodes are kind of special in the sense that you need to keep an eye on creation and destruction. In TEX itself this is mostly hidden:

```
\setbox0\hbox{some text}
```

If we look *into* this box we get a list of glyphs (see figure 8.1).

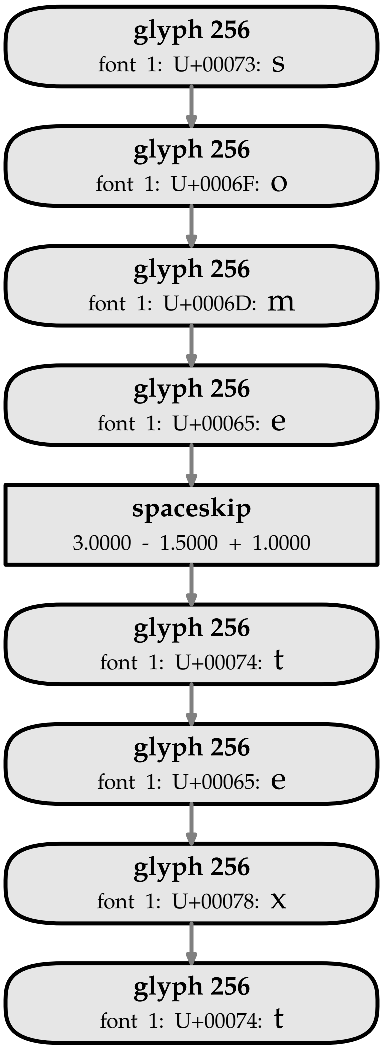


Figure 8.1

In TEX you can flush such a box using `\box0` or copy it using `\copy0`. You can also flush the contents i.e. omit the wrapper using `\unhbox0` and `\unhcopy0`. The possibilities for disassembling the content of a box (or any list for that matter) are limited. In practice you can consider disassembling to be absent.

This is different at the LUA end: there we can really start at the beginning of a list, loop over it and see what's in there as well as change, add and remove nodes. The magic starts with:

```
local box = tex.box[0]
```

Now we have a variable that has a so called `hlist` node. This node has not only properties like `width`, `height`, `depth` and `shift`, but also a pointer to the content: `list`.

```
local list = box.list
```

Now, when we start messing with this list, we need to keep into account that the nodes are in fact userdata objects, that is: they are efficient TEX data structures that have a LUA interface. At the TEX end the repertoire of commands that we can use to flush boxes is rather limited and as we cannot mess with the content we have no memory management issues. However, at the LUA end this is different. Nodes can have pointers to other nodes and they can even have special properties that relate to other resources in the program.

Take this example:

```
\setbox0\hbox{some text}
\directlua{node.write(tex.box[0])}
```

At the TEX end we wrap something in a box. Then we can at the LUA end access that box and print it back into the input. However, as TEX is no longer in control it cannot know that we already flushed the list. Keep in mind that this is a simple example, but imagine more complex content, that contains hyperlinks or so. Now take this:

```
\setbox0\hbox{some text 1}
\setbox0\hbox{some text 2}
```

Here TEX knows that the box has content and it will free the memory beforehand and forget the first text. Or this:

```
\setbox0\hbox{some text}
\box0 \box0
```

The box will be used and after that it's empty so the second flush is basically a harmless null operation: nothing gets inserted. But this:

```
\setbox0\hbox{some text}
\directlua{node.write(tex.box[0])}
\directlua{node.write(tex.box[0])}
```

will definitely fail. The first call flushes the box and the second one sees no box content and will bark. The best solution is to use a copy:

```
\setbox0\hbox{some text}
```

```
\directlua{node.write(node.copy_list(tex.box[0]))}
```

That way TEX doesn't see a change in the box and will free it when needed: when it gets flushed, reassigned, at the end of a group, wherever.

In ConTEXt a somewhat shorter way of printing back to TEX is the following and we will use that:

```
\setbox0\hbox{some text}
\ctxlua{context(node.copy_list(tex.box[0])}
```

or shortcut into ConTEXt:

```
\setbox0\hbox{some text}
\cldcontext{node.copy_list(tex.box[0])}
```

As we've now arrived at the Lua end, we have more possibilities with nodes. In the next sections we will explore some of these.

8.3 Management

The most important thing to keep in mind is that each node is unique in the sense that it can be used only once. If you don't need it and don't flush it, you should free it. If you need it more than once, you need to make a copy. But let's first start with creating a node.

```
local g = node.new("glyph")
```

This node has some properties that need to be set. The most important are the font and the character. You can find more in the LuaTEX manual.

```
g.font = font.current()
g.char = utf.byte("a")
```

After this we can write it to the TEX input:

```
context(g)
```

This node is automatically freed afterwards. As we're talking Lua you can use all kind of commands that are defined in ConTEXt. Take fonts:

```
\startluacode
local g1 = node.new("glyph")
local g2 = node.new("glyph")

g1.font = fonts.definers.internal {
    name = "dejavuserif",
    size = "60pt",
}

g2.font = fonts.definers.internal {
    name = "dejavusansmono",
    size = "60pt",
}

g1.char = utf.byte("a")
g2.char = utf.byte("a")

context(g1)
context(g2)
\stopluacode
```

We get: aa, but there is one pitfall: the nodes have to be flushed in horizontal mode, so either put `\dontleavehmode` in front or add `context.dontleavehmode()`. If you get error messages like `this can't happen` you probably forgot to enter horizontal mode.

In ConTEXt you have some helpers, for instance:

```
\startluacode
local id = fonts.definers.internal { name = "dejavuserif" }

context(nodes.pool.glyph(id,utf.byte("a")))
context(nodes.pool.glyph(id,utf.byte("b")))
context(nodes.pool.glyph(id,utf.byte("c")))
\stopluacode
```

or, when we need these functions a lot and want to save some typing:

```
\startluacode
local getfont  = fonts.definers.internal
local newglyph = nodes.pool.glyph
local utfbyte  = utf.byte

local id = getfont { name = "dejavuserif" }

context(newglyph(id,utfbyte("a")))
context(newglyph(id,utfbyte("b")))
context(newglyph(id,utfbyte("c")))
\stopluacode
```

This renders as: abc. We can make copies of nodes too:

```
\startluacode
local id = fonts.definers.internal { name = "dejavuserif" }
local a  = nodes.pool.glyph(id,utf.byte("a"))

for i=1,10 do
    context(node.copy(a))
end

node.free(a)
```

```
\stopluacode
```

This gives: aaaaaaaaaa. Watch how afterwards we free the node. If we have not one node but a list (for instance because we use box content) you need to use the alternatives `node.copy_list` and `node.free_list` instead.

In ConTEXt there is a convenient helper to create a list of text nodes:

```
\startluacode
context(nodes.typesetters.tonodes("this works okay"))
\stopluacode
```

And indeed, this works okay, even when we use spaces. Of course it makes more sense (and it is also more efficient) to do this:

In this case the list is constructed at the TEX end. We have now learned enough to start using some convenient operations, so these are introduced next. Instead of the longer `tonodes` call we will use the shorter one:

```
local head, tail = string.tonodes("this also works"))
```

As you see, this constructor returns the head as well as the tail of the constructed list.

8.4 Operations

If you are familiar with Lua you will recognize this kind of code:

```
local str = "time: " .. os.time()
```

Here a string `str` is created that is built out if two concatinated snippets. And, Lua is clever enough to see that it has to convert the number to a string.

In ConTEXt we can do the same with nodes:

```
\startluacode
local foo = string.tonodes("foo")
local bar = string.tonodes("bar")
local amp = string.tonodes(" & ")

context(foo .. amp .. bar)
\stopluacode
```

This will append the two node lists: foo & bar.

```
\startluacode
local l = string.tonodes("l")
local m = string.tonodes(" ")
local r = string.tonodes("r")

context(5 * l .. m .. r * 5)
\stopluacode
```

You can have the multiplier on either side of the node: lllll rrrrr. Addition and subtraction is also supported but it comes in flavors:

```
\startluacode
local l1 = string.tonodes("aaaaaa")
local r1 = string.tonodes("bbbbbb")
local l2 = string.tonodes("cccccc")
local r2 = string.tonodes("dddddd")
local m  = string.tonodes(" + ")

context((l1 - r1) .. m .. (l2 + r2))
\stopluacode
```

In this case, as we have two node (lists) involved in the addition and subtraction, we get one of them injected into the other: after the first, or before the last node. This might sound weird but it happens.

aaaaabbbbbba + cddddddccccc

We can use these operators to take a slice of the given node list.

```
\startluacode
local l = string.tonodes("123456")
local r = string.tonodes("123456")
local m = string.tonodes("+ & +")

context((l - 3) .. (1 + m - 1).. (3 + r))
\stopluacode
```

So we get snippets that get appended: 123 & 456. The unary operator reverses the list:

```
\startluacode
local l = string.tonodes("123456")
local r = string.tonodes("123456")
local m = string.tonodes(" & ")

context(l .. m .. - r)
\stopluacode
```

This is probably not that useful, but it works as expected: 123456 & 654321.

We saw that `*` makes copies but sometimes that is not enough. Consider the following:

```
\startluacode
local n = string.tonodes("123456")

context((n - 2) .. (2 + n))
\stopluacode
```

Because the slicer frees the unused nodes, the value of `n` in the second case is undefined. It still points to a node but that one already has been freed. So you get an error message. But of course (as already demonstrated) this is valid:

```
\startluacode
local n = string.tonodes("123456")

context(2 + n - 2)
\stopluacode
```

We get the two middle characters: 34. So, how can we use a node (list) several times in an expression? Here is an example

```
\startluacode
local l = string.tonodes("123")
local m = string.tonodes(" & ")
local r = string.tonodes("456")

context((l^1 .. r^1)^2 .. m^1 .. r .. m .. l)
\stopluacode
```

Using `^` we create copies, so we can still use the original later on. You can best make sure that one reference to a node is not copied because otherwise we get a memory leak. When you write the above without copying LuaTeX most likely end up in a loop. The result of the above is:

123456123456 & 456 & 123

Let's repeat it once more time: keep in mind that we need to do the memory management ourselves. In practice we will seldom need more than the concatination, but if you make complex expressions be prepared to loose some memory when you copy and don't free them. As TeX runs are normally limited in time this is hardly an issue.

So what about the division. We needed some kind of escape and as with `lpeg` we use the `/` to apply additional operations.

```
\startluacode
local l = string.tonodes("123")
local m = string.tonodes(" & ")
local r = string.tonodes("456")

local function action(n)
    for g in node.traverse_id(node.id("glyph"),n) do
        g.char = string.byte("!")
    end
    return n
end

context(l .. m / action .. r)
\stopluacode
```

And indeed we the middle glyph gets replaced: 123 ! 456.

```
\startluacode
local l = string.tonodes("123")
local r = string.tonodes("456")

context(l .. nil .. r)
\stopluacode
```

When you construct lists programmatically it can happen that one of the components is nil and to some extend this is supported: so the above gives: 123456.

Here is a summary of the operators that are currently supported. Keep in mind that these are not built in LuaTeX but extensions in MkIV. After all, there are many ways to map operators on actions and this is just one.

`n1 .. n2`	append nodes (lists) `n1` and `n2`, no copies
`n * 5`	append 4 copies of node (list) `n` to `n`
`5 + n`	discard the first 5 nodes from list `n`
`n - 5`	discard the last 5 nodes from list `n`
`n1 + n2`	inject (list) `n2` after first of list `n1`
`n1 - n2`	inject (list) `n2` before last of list `n1`
`n^2`	make two copies of node (list) `n` and keep the orginal
`- n`	reverse node (list) `n`
`n / f`	apply function `f` to node (list) `n`

As mentioned, you can only use a node or list once, so when you need it more times, you need to make copies. For example:

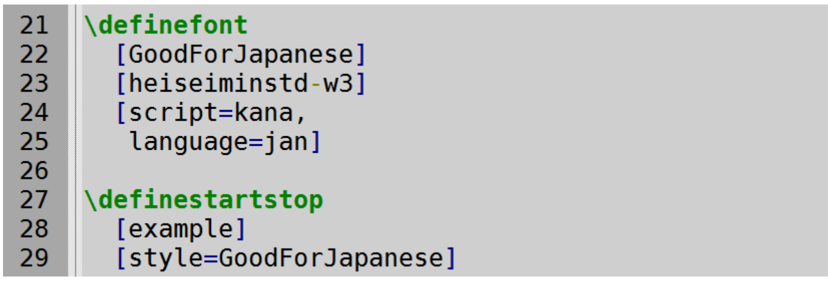
```
\startluacode
local l = string.tonodes(     -- maybe: nodes.maketext
    " 1 2 3 "
)
local r = nodes.tracers.rule( -- not really a user helper (spec might
change)
    string.todimen("1%"),     -- or maybe: nodes.makerule("1%",...)
    string.todimen("2ex"),
    string.todimen(".5ex"),
    "maincolor"
)

context(30 * (r^1 .. l) .. r)
\stopluacode
```

This gives a mix of glyphs, glue and rules: ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮ 1 2 3 ▮. Of course you can wonder how often this kind of juggling happens in use cases but at least in some core code the concatination (`..`) gives a bit more readable code and the overhead is quite acceptable.

9 Still Expanding

In the beginning of October 2013 Luigi figured out that LuajitTEX could actually deal with UTF identifiers. After we played a bit with this, a patch was made for stock LuaTEX to provide the same. In the process I found out that I needed to adapt the SciTE lexer a bit and that some more characters had to get catcode 11 (letter). In the following text screendumps from the editor will be used instead of verbatim code. This also demonstrates how SciTE deals with syntax highlighting.

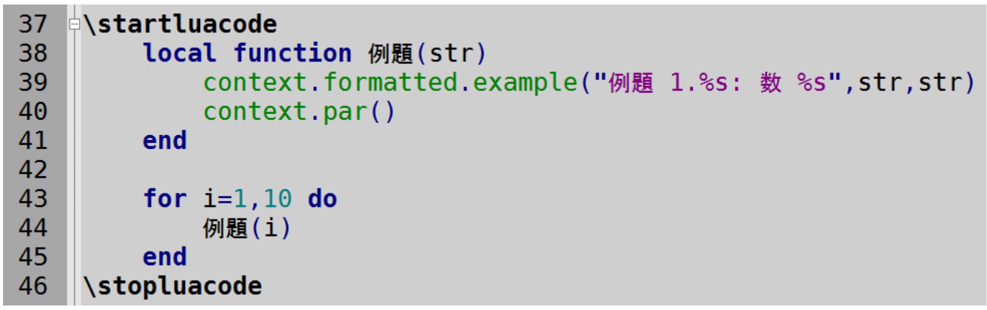
First we define a proper font for to deal with CJK characters and a helper macro that wraps an example using that font.



```
\definefont
  [GoodForJapanese]
  [heiseiminstd-w3]
  [script=kana,
   language=jan]

\definestartstop
  [example]
  [style=GoodForJapanese]
```

According to the Google translator, 例題 means example and 数 means number. It doesn't matter much as we only use these characters as demo. Of course one can wonder if it makes sense to define functions, variables and keys in a script other than basic Latin, but at least it looks kind of modern.

We only show the first three lines. Because using the formatter gives nicer source code we operate in that subnamespace.



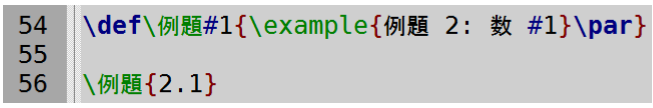
```
\startluacode
    local function 例題(str)
        context.formatted.example("例題 1.%s: 数 %s",str,str)
        context.par()
    end

    for i=1,10 do
        例題(i)
    end
\stopluacode
```

例題 1.1: 数 1

例題 1.2: 数 2

例題 1.3: 数 3

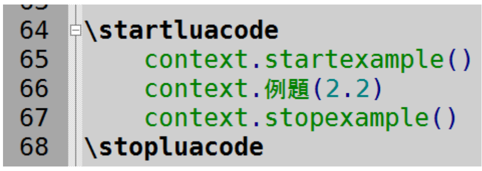
As ConTEXt is already UTF aware for a while you can define macros with such characters. It was a sort of coincidence that this specific range of characters had not yet gotten the proper catcodes, but that is something users don't need to worry about. If your script doesn't work, we just need to initialize a few more characters.



```
\def\例題#1{\example{例題 2: 数 #1}\par}

\例題{2.1}
```

例題 2: 数 2.1

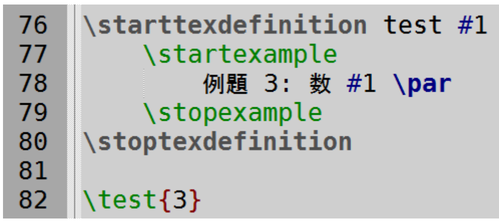
Of course this command is now also present at the Lua end:



```
\startluacode
    context.startexample()
    context.例題(2.2)
    context.stopexample()
\stopluacode
```

例題 2: 数 2.2

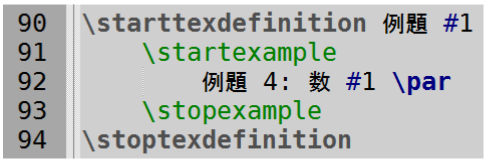
The `MKVI` parser has also been adapted to this phenomena as have the alternative ways of defining macros. We could already do this:



```
\starttexdefinition test #1
    \startexample
        例題 3: 数 #1 \par
    \stopexample
\stoptexdefinition

\test{3}
```

例題 3: 数 3

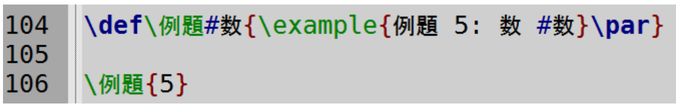
But now we can also do this:



```
\starttexdefinition 例題 #1
    \startexample
        例題 4: 数 #1 \par
    \stopexample
\stoptexdefinition
```

例題 2: 数 4

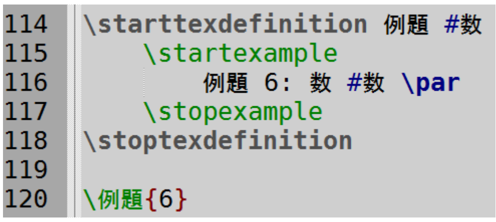
Named parameters support a wider range of characters too:



```
\def\例題#数{\example{例題 5: 数 #数}\par}

\例題{5}
```

例題 5: 数 5

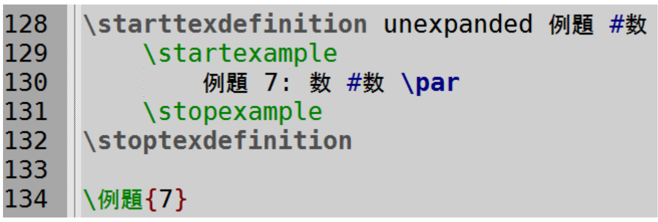
So, in the end we can have definitions like this:



```
\starttexdefinition 例題 #数
    \startexample
        例題 6: 数 #数 \par
    \stopexample
\stoptexdefinition

\例題{6}
```

例題 5: 数 6

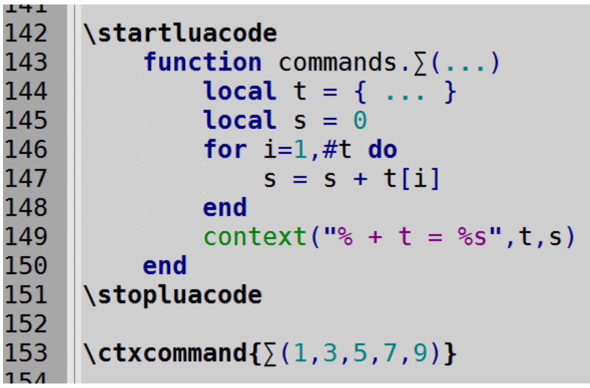
Of course the optional (first) arguments still are supported but these stay Latin.



```
\starttexdefinition unexpanded 例題 #数
    \startexample
        例題 7: 数 #数 \par
    \stopexample
\stoptexdefinition

\例題{7}
```

例題 5: 数 7

Finally Luigi wondered of we could use math symbols too and of course there is no reason why not:



```
\startluacode
    function commands.∑(...)
        local t = { ... }
        local s = 0
        for i=1,#t do
            s = s + t[i]
        end
        context("% + t = %s",t,s)
    end
\stopluacode

\ctxcommand{∑(1,3,5,7,9)}
```

1 + 3 + 5 + 7 + 9 = 25

The ConTeXt source code will of course stay ascii, although some of the multi lingual user interfaces already use characters other than that, for instance accented characters or completely different scripts (like Persian). We just went a step further and supported it at the Lua end which in turn introduced those characters into MkVI.

10 Going nuts

10.1 Introduction

This is not the first story about speed and it will probably not be the last one either. This time we discuss a substantial speedup: upto 50% with LuajitTeX. So, if you don't want to read further at least know that this speedup came at the cost of lots of testing and adapting code. Of course you could be one of those users who doesn't care about that and it may also be that your documents don't qualify at all.

Often when I see a kid playing a modern computer game, I wonder how it gets done: all that high speed rendering, complex environments, shading, lightning, inter–player communication, many frames per second, adapted story lines, ... Apart from clever programming, quite some of the work gets done by multiple cores working together, but above all the graphics and physics processors take much of the workload. The market has driven the development of this hardware and with success. In this perspective it's not that much of a surprise that complex TeX jobs still take some time to get finished: all the hard work has to be done by interpreted languages using rather traditional hardware. Of course all kind of clever tricks make processors perform better than years ago, but still: we don't get much help from specialized hardware.[12] We're sort of stuck: when I replaced my 6 year old laptop (when I buy one, I always buy the fastest one possible) for a new one (so again a fast one) the gain in speed of processing a document was less than twice. The many times faster graphic capabilities are not of much help there, not is twice the amount of cores.

So, if we ever want to go much faster, we need to improve the software. The reason for trying to speed up MkIV has been mentioned before, but let's summarize it here:

- There was a time when users complained about the speed of ConTeXt, especially compared to other macro packages. I'm not so sure if this is still a valid complaint, but I do my best to avoid bottlenecks and much time goes into testing efficiency.
- Computers don't get that much faster, at least we don't see an impressive boost each year any more. We might even see a slowdown when battery live dominates: more cores at a lower speed seems to be a trend and that doesn't suit current TeX engines well. Of course we assume that TeX will be around for some time.
- Especially in automated workflows where multiple products each demanding a couple of runs are produced speed pays back in terms of resources and response time. Of course the time invested in the speedup is never regained by ourselves, but we hope that users appreciate it.
- The more we do in Lua, read: the more demanding users get and the more functionality is enabled, the more we need to squeeze out of the processor. And we want to do more in Lua in order to get better typeset results.

---

[12] Apart from proper rendering on screen and printing on paper.

- Although Lua is pretty fast, future versions might be slower. So, the more efficient we are, the less we probably suffer from changes.

- Using more complex scripts and fonts is so demanding that the number of pages per second drops dramatically. Personally I consider a rate of 15 pps with LuaTEX or 20 pps with LuajitTEX reasonable minima on my laptop.[13]

- Among the reasons why LuaJIT jitting does not help us much is that (at least in ConTEXt) we don't use that many core functions that qualify for jitting. Also, as runs are limited in time and much code kicks in only a few times the analysis and compilation doesn't pay back in runtime. So we cannot simply sit down and wait till matters improve.

Luigi Scarso and I have been exploring several options, with LuaTEX as well as LuajitTEX. We observed that the virtual machine in LuajitTEX is much faster so that engine already gives a boots. The advertised jit feature can best be disabled as it slows down a run noticeably. We played with `ffi` as well, but there is additional overhead involved (`cdata`) as well as limited support for userdata, so we can forget about that too.[14] Nevertheless, the twice as fast virtual machine of LuaJIT is a real blessing, especially if you take into account that ConTEXt spends quite some time in Lua. We're also looking forward to the announced improved garbage collector of LuaJIT.

In the end we started looking at LuaTEX itself. What can be gained there, within the constraints of not having to completely redesign existing (ConTEXt) Lua code?[15]

10.2 Two access models

Because the ConTEXt code is reasonably well optimized already, the only option is to look into LuaTEX itself. We had played with the TEX–Lua interface already and came to the conclusion that some runtime could be gained there. On the long run it adds up but it's not too impressive; these extensions are awaiting integration. Tracing and bechmarking as well as some quick and dirty patches demonstrated that there were two bottlenecks in accessing fields in nodes: checking (comparing the metatables) and constructing results (userdata with metatable).

In case you're infamiliar with the concept this is how nodes work. There is an abstract object called node that is in Lua qualified as user data. This object contains a pointer to TEX's node memory.[16] As it is real user data (not so called light) it also carries a metatable. In the metatble methods are defined and one of them is the indexer. So when you say this:

---

[13] A Dell 6700 laptop with Core i7 3840QM, 16 GB memory and SSD, running 64 bit Windows 8.

[14] As we've now introduced getters we can construct a metatable at the Lua end as that is what `ffi` likes most. But even then, we don't expect much from it: the four times slow down that experiments showed will not magically become a large gain.

[15] In the end a substantial change was needed but only in accessing node properties. The nice thing about C is that there macros often provide a level of abstraction which means that a similar adaption of TEX source code would be more convenient.

[16] The traditional TEX node memory manager is used, but at some point we might change to regular C (de)allocation. This might be slower but has some advantages too.

```
local nn = n.next
```

given that `n` is a node (userdata) the `next` key is resolved up using the `__index` metatable value, in our case a function. So, in fact, there is no `next` field: it's kind of virtual. The index function that gets the relevant data from node memory is a fast operation: after determining the kind of node, the requested field is located. The return value can be a number, for instance when we ask for `width`, which is also fast to return. But it can also be a node, as is the case with `next`, an then we need to allocate a new userdata object (memory management overhead) and a metatable has to be associated. And that comes at a cost.

In a previous update we had already optimized the main `__index` function but felt that some more was possible. For instance we can avoid the lookup of the metatable for the returned node(s). And, if we don't use indexed access but a instead a function for frequently accessed fields we can sometimes gain a bit too.

A logical next step was to avoid some checking, which is okay given that one pays a bit attention to coding. So, we provided a special table with some accessors of frequently used fields. We actually implemented this as a so called 'fast' access model, and adapted part of the ConTEXt code to this, as we wanted to see if it made sense. We were able to gain 5 to 10% which is nice but still not impressive. In fact, we concluded that for the average run using fast was indeed faster but not enough to justify rewriting code to the (often) less nice looking faster access. A nice side effect of the recoding was that I can add more advanced profiling.

But, in the process we ran into another possibility: use accessors exclusively and avoiding userdata by passing around references to TEX node memory directly. As internally nodes can be represented by numbers, we ended up with numbers, but future versions might use light userdata instead to carry pointers around. Light userdata is cheap basic object with no garbage collection involved. We tagged this method 'direct' and one can best treat the values that gets passed around as abstract entities (in MkIV we call this special view on nodes 'nuts').

So let's summarize this in code. Say that we want to know the next node of `n`:

```
local nn = n.next
```

Here `__index` will be resolved and the associated function be called. We can avoid that lookup by applying the `__index` method directly (after all, that one assumes a userdata node):

```
local getfield = getmetatable(n).__index

local nn = getfield(n,"next") -- userdata
```

But this is not a recomended interface for regular users. A normal helper that does checking is as about fast as the indexed method:

```
local getfield = node.getfield
local nn = getfield(n,"next") -- userdata
```

So, we can use indexes as well as getters mixed and both perform more of less equal. A dedicated getter is somewhat more efficient:

```
local getnext = node.getnext

local nn = getnext(n) -- userdata
```

If we forget about checking, we can go fast, in fact the nicely interfaced `__index` is the fast one.

```
local getfield = node.fast.getfield

local nn = getfield(n,"next") -- userdata
```

Even more efficient is the following as that one knows already what to fetch:

```
local getnext = node.fast.getnext

local nn = getnext(n) -- userdata
```

The next step, away from userdata was:

```
local getfield = node.direct.getfield

local nn = getfield(n,"next") -- abstraction
```

and:

```
local getnext = node.direct.getnext

local nn = getnext(n) -- abstraction
```

Because we considered three variants a bit too much and because `fast` was only 5 to 10% faster in extreme cases, we decided to drop that experimental code and stick to providing accessors in the node namespace as well as direct variants for critical cases.

Before you start thinking: ‘should I rewrite all my code?’ think twice! First of all, `n.next` is quite fast and switching between the normal and direct model also has some cost. So, unless you also adapt all your personal helper code or provide two variants of each, it only makes sense to use direct mode in critical situations. Userdata mode is much more convenient when developing code and only when you have millions of access you can gain by direct mode. And even then, if the time spent in LUA is small compared to the time spent in TEX it might not even be noticeable. The main reason we made direct variants is that it does pay of in OPENTYPE font processing where complex scripts can result in many millions of calls indeed. And that code will be set up in such a way that it will use userdata by default and only in well controlled case (like MKIV) we will use direct mode.[17]

---

[17] When we are confident that `direct` node code is stable we can consider going direct in generic code as well, although we need to make sure that third party code keeps working.

Another thing to keep in mind is that when you provide hooks for users you should assume that they use the regular mode so you need to cast the plugins onto direct mode then. Because the idea is that one should be able to swap normal functions by direct ones (which of course is only possible when no indexes are used) all relevant function in the `node` namespace are available in `direct` as well. This means that the following code is rather neutral:

```
local x = node -- or: x = node.direct

for n in x.traverse(head) do
  if x.getid(n) == node.id("glyph") and x.getchar(n) == 0x123 then
    x.setfield(n,"char",0x456)
  end
end
```

Of course one needs to make sure that `head` fits the model. For this you can use the cast functions:

```
node.direct.todirect(node or direct)
node.direct.tonode(direct or node)
```

These helpers are flexible enough to deal with either model. Aliasing the functions to locals is of course more efficient when a large number of calls happens (when you use LuajitTeX it will do some of that for you automatically). Of course, normally we use a more natural variant, using an id traverser:

```
for n in node.traverse_id(head,node.id("glyph")) do
  if n.char == 0x123 then
    n.char = 0x456
  end
end
```

This is not that much slower, especially when it's only ran once. Just count the number of characters on a page (or in your document) and you will see that it's hard to come up with that many calls. Of course, processing many pages of Arabic using a mature font with many features enabled and contextual lookups, you do run into quantities. Tens of features times tens of contextual lookup passes can add up considerably. In Latin scripts you never reach such numbers, unless you use fonts like Zapfino.

10.3 The transition

After weeks of testing, rewriting, skyping, compiling and making decisions, we reached a more or less stable situation. At that point we were faced with a speedup that gave us a good feeling, but transition to the faster variant has a few consequences.

- We need to use an adapted code base: indexes are to be replaced by function calls. This is a tedious job that can endanger stability so it has to be done with care.[18]
- When using an old engine with the new MkIV code, this approach will result in a somewhat slower run. Most users will probably accept a temporary slowdown of 10%, so we might take this intermediate step.
- When the regular getters and setters become available we get back to normal. Keep in mind that these accessors do some checking on arguments so that slows down to the level of using indexes. On the other hand, the dedicated ones (like `getnext`) are more efficient so there we gain.
- As soon as direct becomes available we suddenly see a boost in speed. In documents of average complexity this is 10-20% and when we use more complex scripts and fonts it can go up to 40%. Here we assume that the macro package spends at least 50% of its time in Lua.

If we take the extremes: traditional indexed on the one hand versus optimized direct in LuajitTeX, a 50% gain compared to the old methods is feasible. Because we also retrofitted some fast code into the regular accessor, indexed mode should also be somewhat faster compared to the older engine.

In addition to the already provide helpers in the `node` namespace, we added the following:

`getnext`	this one is used a lot when analyzing and processing node lists
`getprev`	this one is used less often but fits in well (companion to `getnext`)
`getfield`	this is the general accessor, in userdata mode as fast as indexed
`getid`	one of the most frequent called getters when parsing node lists
`getsubtype`	especially in fonts handling this getter gets used
`getfont`	especially in complex font handling this is a favourite
`getchar`	as is this one
`getlist`	we often want to recurse into hlists and vlists and this helps
`getleader`	and also often need to check if glue has leader specification (like list)
`setfield`	we have just one setter as setting is less critical

As `getfield` and `setfield` are just variants on indexed access, you can also use them to access attributes. Just pass a number as key. In the `direct` namespace, helpers like `insert_before` also deal with direct nodes.

We currently only provide `setfield` because setting happens less than getting. Of course you can construct nodelists at the Lua end but it doesn't add up that fast and indexed access is then probably as efficient. One reason why setters are less an issue is that they don't return nodes so no userdata overhead is involved. We could (and might)

---

[18] The reverse is easier, as converting getters and setters to indexed is a rather simple conversion, while for instance changing type .next into a `getnext` needs more checking because that key is not unique to nodes.

provide `setnext` and `setprev`, although, when you construct lists at the Lua end you will probably use the type insert_after helper anyway.

10.4 Observations

So how do these variants perform? As we no longer have `fast` in the engine that I use for this text, we can only check `getfield` where we can simulate fast mode with calling the `__index` metamethod. In practice the `getnext` helper will be somewhat faster because no key has to be checked, although the `getfield` functions have been optimized according to the frequencies of accessed keys already.

node[*]	0.516
node.fast.getfield	0.616
node.getfield	0.494
node.direct.getfield	0.172

Here we simulate a dumb 20 times node count of 200 paragraphs `tufte.tex` with a little bit of overhead for wrapping in functions.[19] We encounter over three million nodes this way. We average a couple or runs.

```
local function check(current)
  local n = 0
  while current do
    n = n + 1
    current = getfield(current,"next") -- current = current.next
  end
  return n
end
```

What we see here is that indexed access is quite okay given the amount of nodes, but that direct is much faster. Of course we will never see that gain in practice because much more happens than counting and because we also spend time in TEX. The 300% speedup will eventually go down to one tenth of that.

Because ConTEXt avoids node list processing when possible the baseline performance is not influenced much.

```
\starttext \dorecurse{1000}{test\page} \stoptext
```

With LuaTEX we get some 575 pages per second and with LuajitTEX more than 610 pages per second.

```
\setupbodyfont[pagella]

\edef\zapf{\cldcontext
  {context(io.loaddata(resolvers.findfile("zapf.tex")))}}

\starttext \dorecurse{1000}{\zapf\par} \stoptext
```

---

[19] When typesetting Arabic or using complex fonts we quickly get a tenfold.

For this test LuaTeX needs 3.9 seconds and runs at 54 pages per second, while LuajitTeX needs only 2.3 seconds and gives us 93 pages per second.

Just for the record, if we run this:

```
\starttext
\stoptext
```

a LuaTeX runs takes 0.229 seconds and a LuajitTeX run 0.178 seconds. This includes initializing fonts. If we run just this:

```
\stoptext
```

LuaTeX needs 0.199 seconds and LuajitTeX only 0.082 seconds. So, in the meantime, we hardly spend any time on startup. Launching the binary and managing the job with `mtxrun` calling `mtx-context` adds 0.160 seconds overhead. Of course this is only true when you have already ran ConTeXt once as the operating system normally caches files (in our case format files and fonts). This means that by now an edit-preview cycle is quite convenient.[20]

As a more practical test we used the current version of `fonts-mkiv` (166 pages, using all kind of font tricks and tracing), `about` (60 pages, quite some traced math) and a torture test of Arabic text (61 pages dense text). The following measurements are from 2013-07-05 after adapting some 50 files to the new model. Keep in mind that the old binary can fake a fast getfield and setfield but that the other getters are wrapped functions. The more we have, the slower it gets. We used the mingw versions.

version	fonts	about	arabic
old mingw, indexed plus some functions	8.9	3.2	20.3
old mingw, fake functions	9.9	3.5	27.4
new mingw, node functions	9.0	3.1	20.8
new mingw, indexed plus some functions	8.6	3.1	19.6
new mingw, direct functions	7.5	2.6	14.4

The second row shows what happens when we use the adapted ConTeXt code with an older binary. We're slower. The last row is what we will have eventually. All documents show a nice gain in speed and future extensions to ConTeXt will no longer have the same impact as before. This is because what we here see also includes TeX activity. The 300% increase of speed of node access makes node processing less influential. On the average we gain 25% here and as on these documents LuajitTeX gives us some 40% gain on indexed access, it gives more than 50% on the direct function based variant.

In the fonts manual some 25 million getter accesses happen while the setters don't exceed one million. I lost the tracing files but at some point the Arabic test showed more than 100 millions accesses. So it's save to conclude that setters are sort of neglectable. In the fonts manual the amount of accesses to the previous node were less that 5000 while the id and next fields were the clear winners and list and leader fields also scored

---

[20] I use SciTE with dedicated lexers as editor and currently `sumatrapdf` as previewer.

high. Of course it all depends on the kind of document and features used, but we think that the current set of helpers is quite adequate. And because we decided to provide that for normal nodes as well, there is no need to go direct for more simple cases.

Maybe in the future further tracing might show that adding getters for width, height, depth and other properties of glyph, glue, kern, penalty, rule, hlist and vlist nodes can be of help, but quite probably only in direct mode combined with extensive list manipulations. We will definitely explore other getters but only after the current set has proven to be useful.

10.5 Nuts

So why going nuts and what are nuts? In Dutch 'node' sounds a bit like 'noot' and translates back to 'nut'. And as in ConTEXt I needed word for these direct nodes they became 'nuts'. It also suits this project: at some point we're going nuts because we could squeeze more out of LuajitTEX, so we start looking at other options. And we're sure some folks consider us being nuts anyway, because we spend time on speeding up. And adapting the LuaTEX and ConTEXt MkIV code mid–summer is also kind of nuts.

At the ConTEXt 2013 conference we will present this new magic and about that time we've done enough tests to see if it works our well. The LuaTEX engine will provide the new helpers but they will stay experimental for a while as one never knows where we messed up.

I end with another measurement set. Every now and and then I play with a Lua variant of the TEX par builder. At some point it will show up on MkIV but first I want to abstract it a bit more and provide some hooks. In order to test the performance I use the following tests:

```
\testfeatureonce{1000}{\setbox0\hbox{\tufte}}

\testfeatureonce{1000}{\setbox0\vbox{\tufte}}

\startparbuilder[basic]
  \testfeatureonce{1000}{\setbox0\vbox{\tufte}}
\stopparbuilder
```

We use a `\hbox` to determine the baseline performance. Then we break lines using the built-in parbuilder. Next we do the same but now with the Lua variant.[21]

	luatex		**luajittex**	
	total	**linebreak**	**total**	**linebreak**
223 pp nodes	5.67	2.25 flushing	3.64	1.58 flushing
hbox nodes	3.42		2.06	
vbox nodes	3.63	0.21 baseline	2.27	0.21 baseline

---

[21] If we also enable protrusion and hz the Lua variant suffers less because it implements this more efficient.

vbox lua nodes	7.38	3.96	3.95	1.89
223 pp nuts	4.07	1.62 flushing	2.36	1.11 flushing
hbox nuts	2.45		1.25	
vbox nuts	2.53	0.08 baseline	1.30	0.05 baseline
vbox lua nodes	6.16	3.71	3.03	1.78
vbox lua nuts	5.45	3.00	2.47	1.22

We see that on this test nuts have an advantage over nodes. In this case we mostly measure simple font processing and there is no markup involved. Even a 223 page document with only simple paragraphs needs to be broken across pages, wrapped in page ornaments and shipped out. The overhead tagged as 'flushed' indicates how much extra time would have been involved in that. These numbers demonstrate that with nuts the Lua parbuilder is performing 10% better so we gain some. In a regular document only part of the processing involves paragraph building so switching to a Lua variant has no big impact anyway, unless we have simple documents (like novels). When we bring hz into the picture performance will drop (and users occasionally report this) but here we already found out that this is mostly an implementation issue: the Lua variant suffers less so we will backport some of the improvements.[22]

10.6 LUA 5.3

When we were working on this the first working version of Lua 5.3 was announced. Apart from some minor changes that won't affect us, the most important change is the introduction of integers deep down. On the one hand we can benefit from this, given that we adapt the TeX-Lua interfaces a bit: the distinction between `to_number` and `to_integer` for instance. And, numbers are always somewhat special in TeX as it relates to reproduction on different architectures, also over time. There are some changes in conversion to string (needs attention) and maybe at some time also in the automated casting from strings to numbers (the last is no big deal for us).

On the one hand the integers might have a positive influence on performance especially as scaled points are integers and because fonts use them too (maybe there is some advantage in memory usage). But we also need a proper efficient round function (or operator) then. I'm wondering if mixed integer and float usage will be efficient, but on the the other hand we do not that many calculations so the benefits might outperform the drawbacks.

We noticed that 5.2 was somewhat faster but that the experimental generational garbage collecter makes runs slower. Let's hope that the garbage collector performance doesn't degrade. But the relative gain of node versus direct will probably stay.

Because we already have an experimental setup we will probably experiment a bit with this in the future. Of course the question then is how LuajitTeX will work out, because it is already not 5.2 compatible it has to be seen if it will support the next level. At least in ConTeXt MkIV we can prepare ourselves as we did with Lua 5.2 so that we're ready when we follow up.

---

[22] There are still some aspects that can be approved. For instance these tests still checks lists for `prev` fields, something that is not needed in future versions.

11 Lua strings

11.1 Introduction

In the crited project[23] we have to deal with large amounts of data. The sources are in TEI XML and processed directly in ConTEXt MkIV, and we have to filter content from different places in the XML tree. Processing relies on Lua a lot because we use Lua for dealing with the XML. We're talking about Latin and Greek texts so there is no demand for extensive font processing in Lua is moderate. But as critical editions have lots of line specific referencing and notes there are some more complex layout elements involved, and again these use Lua. There is also extensive use of bibliographies and it will be no surprise that Lua comes to help too.[24]

One secondary objective is to be able to process the complex documents at a speed of at least 20 pages per second on a modern 2014 workstation laptop. One way of achieving this is to use LuajitTEX which has a faster virtual Lua machine. However, we ran into several issues with the LuaJIT interpreter, which is fully Lua language 5.1 and partly 5.2 compatible but definitely has a different low level implementation. In the next sections I will discuss two issues that Luigi and I ran into and for which we could come up with reasonable workarounds.

11.2 The stacks

A TEX job is normally a multi-pass experience. One run can produce information that is used in a successive one. The reason is that something can happen on page 15 that influences the typesetting of page 9. There can even be a partial chain reaction: you typeset a document the first time the table of contents (and the pages it refers to) is not known yet but information is saved that makes it possible next time. That next run it gets included and it takes for instance 4 pages. This means that all page numbers shift up. This in turn will trigger a new run because all cross references might change too: two digit page numbers can become three digits, so paragraphs can run wider, and that again can trigger more pages. Normally an initial three runs is enough, and with minor updates of the source one or two runs are enough after that.

The multi-pass information is saved in tables in the so called utility file and loaded a next run. Common subtables are shared in the process. In order to determine if there has been crucial changes that demand an extra run, we have to make sure that random order in these tables is eliminated. Normally we already sort keys in tables when writing them to file but some tables come out in the order the traversing `next` function delivers them. In the more recent 5.2 versions Lua has added some randomness to the order in which hashed tables are organized, so while in previous versions we could assume that for a specific binary the order was the same each time, we cannot rely on that any longer. This is not that important for normal cases, but we compare previous

---

[23] This is a project by Thomas Schmitz, Alan Braslau, Luigi Scarso and Hans Hagen funded by the Institut für Klassische und Romanische Philologie Universität Bonn.

[24] One of the objectives of the project is to update and enhance the bibliographic subsystem.

and current versions of the utility file and pack shared tables in them as well, which means that we are sensitive for a change in order. But, this could be dealt with at the cost of some extra sorting.[25]

Anyway, this kind of changes in the Lua machinery is harmless apart from taking some time to adapt to it. It is also the reason why we cannot simply push a new update of Lua into LuaTEX because low level changes can have an (yet unknown) impact. Of course performance is the biggest issue here: we don't want a slower LuaTEX.

In the past we already reported on the benefits of LuajitTEX, especially its faster virtual machine. We don't benefit from jitting; on the contrary it slows us down. One reason is that we cross the Lua–C boundary often and hardly use any of the optimized functions. Part of the speed is achieved by a different implementation deep down and one of them is a different virtual machine instruction set. While Lua can go real big in terms of memory and table construction, LuaJIT limits us to at most 2G memory and poses some 64K limitations in functions and table constructors. The memory is not so much the issue in the crited project but the (nested) table constructor is. When we have a few tens of thousands of cross references, index entries and/or list entries we simply cannot load the multi-pass data. A few days of playing with splitting up nested tables didn't help much: it made the code look horrible and eventually we again ran into a maximum of 64K someplace as a `dofile` effectively makes a function that gets run and LuaJIT doesn't like that size. For the record: we don't have such issues with large font tables probably because they are just one big table. The reason why we cannot use that approach is that serializing the potentially very large tables in the utility file also has limitations.

Eventually this could be solved by assuming only forward referencing for certain registers. That way we only used the index entries collected in memory during the run and as long as we don't put a register before it's entries are defined we're okay. So here we have a typical case where one can set an option to circumvent an engine limitation.[26] Explaining this in a user manual is a challenge, because an error message like the following is not that helpful:

```
main function has more than 65536 constants
```

But, once we could generate these indices again by posing some limitations, LuajitTEX had other issues. This time we got excessive runtime and we spent quite some time sorting that one out. More on that in the next section.

11.3 Hashing

One of the reasons why (text processing with) Lua is rather fast is that it hashes its strings so that a test for equality is real fast. This means that for each string that enters Lua a hash value is calculated and that hash is used in comparisons. Of course hashing takes time, but especially when you work with lots of tables the advantage of a simple

---

[25] In ConTEXt we also pack font tables which saves lots of memory and also some load time).

[26] A decade ago similar tricks had to be used to support hundreds of thousands of hyperlinks in TEX engines with at that time limited memory capabilities.

hash compare outweighs this one–time hashing. On the other hand, if you work with files and process lines, and maybe split these in words, you might end up with a lot of unneeded hashing. But, in LuaTEX and therefore MkIV we benefit from hashing a lot. In Lua 5.2 the hash function was adapted so that only strings upto than (default) 40 characters get hashed. In practice we're not affected much by this, as most keywords we use are shorter than this boundary. And in ConTEXt we do quite some keyword checking.

So, when we were conducting tests with these large registers, we were surprised that LuajitTEX performed significantly slower (ten times or more) that stock LuaTEX, while until then we had observed that a LuajitTEX run was normally some 20 to 40% faster.

The first impression was that it related to the large amount of strings that are written from Lua to TEX. After index entries are collected, they are sorted and the index is flushed to TEX. This happens in one go, and TEX code ends up in the TEX input stack. Some actions are delayed and create callbacks to Lua, so some wrapping in functions happens too. That means that some (Lua) strings are only freed later on, but that proved not to be the main problem.

When the entries are typeset, an interactive cross reference is kept track of and these exist till the document is closed and the referencing information is written to the pdf file. Of course we could tweak this but once you start along that path there is no end to writing ugly hacks.

Eventually we found that the slowdown relates to hashing, especially because that is not the first area where you look. Why is this? The specific register concerned lots of small greek words, pointing to locations in a text, where locations looked like `1.2.3`. In case you wonder why greek is mentioned: in multi-byte utf sequences there is a lot of repetition:

word	unicode	bytes
βον	`3B2 1F77 3BF 3BD`	`CE B2 E1 BD B7 CE BF CE BD`
βου	`3B2 1F77 3BF 3C5`	`CE B2 E1 BD B7 CE BF CF 85`
βιος	`3B2 3B9 3BF 1F7A 3C2`	`CE B2 CE B9 CE BF E1 BD BA CF 82`
βουλν	`3B2 3BF 3C5 3BB 1F74 3BD`	`CE B2 CE BF CF 85 CE BB E1 BD B4 CE BD`
βουλς	`3B2 3BF 3C5 3BB 1FC6 3C2`	`CE B2 CE BF CF 85 CE BB E1 BF 86 CF 82`

When cross referencing these index entries with their origin, you end up with reference identifiers like `foo:1.2.3` or, because ConTEXt has automated internal references (which are rather efficient in the resulting pdf), we get `aut:1`, `aut:2` upto in this case some 30.000 of them.

The problem with hashing is as follows. When we write commands to TEX or use data with a repetitive property, the similarity of these strings can be hard on the hasher as it can produce similar hash keys in which case collisions need to be dealt with. I'm no expert on hashing but looking at the code shows that in LuaJIT (at least in the version we're talking about) the string is seen as chunks of 4 bytes. The first, last, middle and halfway middle chunks are consulted and after some bit juggling we get a hash value.

In the case of strings like the following it is clear that the beginning and end look quite the same:

```
foo:000001  foo:010001  foo:100001
```

or:

```
foo:1.2.12  foo:1.3.12  foo:1.4.12  foo:1.5.12
```

It seems that the used method of hashing is somewhat arbitrary and maybe tuned for specific applications. In order to see what the impact is of hashing quite similar strings, some experiments were conducted: with LuaTeX 0.73 using Lua 5.2 hashing, with LuajitTeX 0.73, and with the same LuajitTeX but using the hash variant of native Lua 5.1. For each variant we ran tests where strings of increasing length were combined with a number (running from one to one million).

none	<string>
right	<string> <number>
left	<number> <string>
center	<string> <number> <string>
edges	<number> <string> <number>

The differences between engines can be seen in tables in the next page. In the fourth table we summarize which engine performs best. Keep in mind that LuajitTeX has the advantage of the faster virtual machine so it has an additional speed advantage.

We show three tables with measurements. The `none` column shows the baseline of the test:

```
local t = { }
for i=1,1000000 do
    t[i] = i
end
```

The column tagged 'right' does this:

```
local t = { }
for i=1,1000000 do
    t[i] = text .. i
end
```

And 'left' does:

```
local t = { }
for i=1,1000000 do
    t[i] = i .. text
end
```

That leaves 'center':

```
local t = { }
```

```
for i=1,1000000 do
    t[i] = text .. i .. text
end
```

and ‘edges’:

```
local t = { }
for i=1,1000000 do
    t[i] = i .. text .. i
end
```

Of course there is also the loop and the concatenation involved so the last two variants have some more overhead. We show some measurements in tables 11.1, 11.2 and 11.3. So, there we have strings like:

```
2abc
222abc
22222abc
abc222222
222222abc222222
222222abc222222
abc2222abc
```

and so on. Of course a million such strings makes not much sense in practice but it serves our purpose of testing.

In these tables you can see some extremes. On the average Lua 5.2 performs quite okay as does standard LuaJIT. However, when we bring the 5.1 hash variant into LuajitTeX we get a more predictable average performance as it deals better with some of the extreme cases that make LuajitTeX crawl compared to LuaTeX. We have done more tests and interesting is to see that in the 5.1 (and derived 5,2) method there are sometimes cases where odd lengths perform much worse than even lengths. Red values are larger than two times the average, blue values larger than average while green values indicate a less than half average value.

In table 11.4 we show which method performs best relative to each other. Of course in many applications there will be no such extreme cases, but we happen to ran into them. But, even if `JIT20` is a winner in most cases, the fact that it has extreme slow exceptions makes it a bit of a gamble.

The 5.1 hasher runs over the string with a step that depends on the length of the string. We’ve seen that in 5.2 it doesn’t hash strings larger than 40 characters. The step is calculated by shifting the length (by default) over 5 bits. This means that for strings of size 32 and more the step becomes 2 which is why we see this odd/even timing issue in the tables. Basically we hash at most 32 characters of the 40. The next table shows that the less characters we take into account (first column) the less unique keys we get (second column).

n	**unique**	**text**
3	22	`/Border [ 0 0 0 ] /F 4 /Subtype /Link /A * 0 R`

	none	right	left	center	edges	text
1	0.016	1.190	1.143	1.188	1.701	`a`
2	0.025	1.177	1.141	1.175	1.685	`ab`
3	0.025	1.183	1.142	1.179	1.691	`abc`
4	0.025	1.183	1.147	1.187	1.692	`abcd`
5	0.025	1.194	1.156	1.209	1.705	`abcde`
6	0.025	1.201	1.161	1.215	1.714	`abcdef`
7	0.027	1.203	1.164	1.222	1.714	`abcdefg`
8	0.026	1.202	1.162	1.215	1.715	`abcdefgh`
9	0.025	1.206	1.171	1.209	1.698	`abcdefghi`
10	0.025	1.210	1.161	1.207	1.707	`abcdefghij`
11	0.025	1.213	1.165	1.228	1.708	`abcdefghijk`
12	0.025	1.205	1.165	1.224	1.708	`abcdefghijkl`
13	0.025	1.215	1.162	3.586	1.705	`abcdefghijklm`
14	0.025	1.207	1.175	5.056	1.708	`abcdefghijklmn`
15	0.025	1.215	1.177	3.965	1.712	`abcdefghijklmno`
16	0.025	1.210	1.177	5.097	1.725	`abcdefghijklmnop`
17	0.024	1.213	1.180	3.982	1.724	`abcdefghijklmnopq`
18	0.025	1.219	1.182	5.195	1.714	`abcdefghijklmnopqr`
19	0.025	1.217	1.184	4.016	1.722	`abcdefghijklmnopqrs`
20	0.025	1.221	1.182	5.199	5.623	`abcdefghijklmnopqrst`
21	0.025	1.244	1.191	4.056	1.815	`abcdefghijklmnopqrstu`
22	0.025	1.247	1.193	1.082	5.637	`abcdefghijklmnopqrstuv`
23	0.025	1.251	1.220	1.085	1.827	`abcdefghijklmnopqrstuvw`
24	0.025	1.244	1.205	1.071	5.580	`abcdefghijklmnopqrstuvwx`
25	0.025	1.247	1.195	1.070	1.821	`abcdefghijklmnopqrstuvwxy`
26	0.025	5.240	5.094	1.088	5.514	`abcdefghijklmnopqrstuvwxyz`
27	0.025	5.257	44.874	1.069	1.838	`abcdefghijklmnopqrstuvwxyzA`
28	0.025	5.231	5.412	1.075	5.577	`abcdefghijklmnopqrstuvwxyzAB`
29	0.025	5.208	45.411	1.081	1.841	`abcdefghijklmnopqrstuvwxyzABC`
30	0.026	5.248	5.536	1.091	5.643	`abcdefghijklmnopqrstuvwxyzABCD`
31	0.024	5.351	45.540	1.084	1.844	`abcdefghijklmnopqrstuvwxyzABCDE`
32	0.025	5.376	5.550	1.078	5.657	`abcdefghijklmnopqrstuvwxyzABCDEF`
33	0.025	5.422	45.903	1.077	1.831	`abcdefghijklmnopqrstuvwxyzABCDEFG`
34	0.025	5.266	5.525	1.082	5.710	`abcdefghijklmnopqrstuvwxyzABCDEFGH`
35	0.025	5.223	48.141	1.076	1.848	`abcdefghijklmnopqrstuvwxyzABCDEFGHI`
36	0.025	5.260	5.427	1.083	6.241	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJ`
37	0.025	5.310	45.596	1.080	1.590	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJK`
38	0.025	5.233	5.950	1.080	1.579	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL`
39	0.025	5.314	45.252	1.088	1.567	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM`
40	0.024	5.489	5.531	1.074	1.570	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN`
41	0.025	5.598	45.903	1.074	1.574	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO`
42	0.025	5.657	6.033	1.081	1.569	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP`
43	0.025	1.115	1.296	1.069	1.568	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ`
44	0.025	1.080	1.048	1.080	1.572	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR`
45	0.025	1.083	1.051	1.085	1.566	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS`
46	0.025	1.083	1.046	1.090	1.573	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST`
47	0.024	1.082	1.052	1.088	1.576	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU`
48	0.025	1.080	1.048	1.085	1.570	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV`
49	0.025	1.085	1.049	1.080	1.571	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW`
50	0.025	1.083	1.037	1.077	1.568	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX`
	0.025	2.594	9.092	1.719	2.406	

Table 11.1 `context test.tex`

	none	right	left	center	edges	text
1	0.016	1.125	1.125	1.125	1.656	`a`
2	0.000	1.141	1.109	1.110	1.594	`ab`
3	0.016	1.109	1.094	1.124	1.651	`abc`
4	0.009	1.149	1.147	1.014	1.653	`abcd`
5	0.008	1.153	1.142	1.008	1.646	`abcde`
6	0.009	1.140	1.157	1.014	1.652	`abcdef`
7	0.008	1.214	1.155	1.006	1.652	`abcdefg`
8	0.008	1.169	1.256	1.170	1.642	`abcdefgh`
9	0.008	2.557	1.216	1.169	1.644	`abcdefghi`
10	0.009	2.048	1.296	1.172	1.636	`abcdefghij`
11	0.008	2.621	2.841	1.172	1.639	`abcdefghijk`
12	0.009	1.977	1.761	1.196	1.638	`abcdefghijkl`
13	0.008	2.560	1.589	1.201	1.635	`abcdefghijklm`
14	0.008	1.983	1.592	1.194	1.634	`abcdefghijklmn`
15	0.009	2.537	2.722	1.200	1.637	`abcdefghijklmno`
16	0.008	1.955	2.279	1.221	1.639	`abcdefghijklmnop`
17	0.009	2.511	1.889	1.219	1.639	`abcdefghijklmnopq`
18	0.008	2.035	1.157	1.202	1.652	`abcdefghijklmnopqr`
19	0.009	2.583	1.486	1.203	1.635	`abcdefghijklmnopqrs`
20	0.008	2.012	1.404	1.224	1.643	`abcdefghijklmnopqrst`
21	0.009	2.560	1.056	1.224	1.639	`abcdefghijklmnopqrstu`
22	0.009	2.008	1.111	1.223	1.648	`abcdefghijklmnopqrstuv`
23	0.009	2.555	1.084	1.226	1.648	`abcdefghijklmnopqrstuvw`
24	0.009	1.951	1.071	1.239	1.645	`abcdefghijklmnopqrstuvwx`
25	0.008	2.518	1.048	1.239	1.645	`abcdefghijklmnopqrstuvwxy`
26	0.009	2.069	1.062	1.234	1.635	`abcdefghijklmnopqrstuvwxyz`
27	0.009	2.616	1.076	1.236	1.636	`abcdefghijklmnopqrstuvwxyzA`
28	0.008	2.065	1.085	1.260	1.639	`abcdefghijklmnopqrstuvwxyzAB`
29	0.009	2.671	1.060	1.270	1.651	`abcdefghijklmnopqrstuvwxyzABC`
30	0.010	2.075	1.117	1.274	1.648	`abcdefghijklmnopqrstuvwxyzABCD`
31	0.008	2.631	1.056	1.270	1.652	`abcdefghijklmnopqrstuvwxyzABCDE`
32	0.008	2.048	1.090	1.294	1.656	`abcdefghijklmnopqrstuvwxyzABCDEF`
33	0.009	2.548	1.079	1.301	1.647	`abcdefghijklmnopqrstuvwxyzABCDEFG`
34	0.008	2.043	1.060	1.301	1.653	`abcdefghijklmnopqrstuvwxyzABCDEFGH`
35	0.008	2.618	1.053	1.347	1.649	`abcdefghijklmnopqrstuvwxyzABCDEFGHI`
36	0.008	2.018	1.086	1.388	1.643	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJ`
37	0.009	2.535	1.034	1.417	1.667	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJK`
38	0.008	2.018	1.163	1.430	1.639	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL`
39	0.008	2.548	1.051	1.454	1.643	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM`
40	0.008	1.980	1.117	1.489	1.639	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN`
41	0.008	2.510	1.051	1.495	1.637	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO`
42	0.009	2.069	1.052	1.498	1.642	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP`
43	0.009	2.643	1.084	1.502	1.642	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ`
44	0.009	2.052	1.172	1.524	1.641	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR`
45	0.008	2.610	1.064	1.523	1.649	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS`
46	0.008	2.040	1.193	1.522	1.640	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST`
47	0.009	2.557	1.029	1.509	1.640	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU`
48	0.009	2.038	1.172	1.533	1.642	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV`
49	0.008	2.586	1.078	1.541	1.645	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW`
50	0.008	2.107	1.114	1.535	1.643	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX`
	0.009	2.117	1.260	1.285	1.643	

Table 11.2 `context --jit --jithash=luajit20 test.tex`

	none	right	left	center	edges	text
1	0.000	1.157	1.094	1.110	1.625	`a`
2	0.000	1.125	1.111	1.133	1.659	`ab`
3	0.009	1.128	1.122	1.133	1.659	`abc`
4	0.008	1.128	1.119	1.132	1.668	`abcd`
5	0.008	1.131	1.122	1.141	1.661	`abcde`
6	0.009	1.134	1.121	1.141	1.660	`abcdef`
7	0.009	1.129	1.120	1.140	1.667	`abcdefg`
8	0.009	1.128	1.116	1.140	1.662	`abcdefgh`
9	0.008	1.124	1.112	1.137	1.660	`abcdefghi`
10	0.008	1.127	1.116	1.146	1.659	`abcdefghij`
11	0.009	1.132	1.121	1.150	1.664	`abcdefghijk`
12	0.009	1.135	1.122	1.168	1.674	`abcdefghijkl`
13	0.009	1.139	1.128	3.021	1.677	`abcdefghijklm`
14	0.009	1.142	1.129	3.952	1.676	`abcdefghijklmn`
15	0.009	1.138	1.124	3.309	1.673	`abcdefghijklmno`
16	0.009	1.134	1.121	3.999	1.680	`abcdefghijklmnop`
17	0.008	1.144	1.130	3.405	1.678	`abcdefghijklmnopq`
18	0.008	1.142	1.134	4.034	1.686	`abcdefghijklmnopqr`
19	0.009	1.145	1.133	3.998	1.690	`abcdefghijklmnopqrs`
20	0.009	1.148	1.133	4.145	4.488	`abcdefghijklmnopqrst`
21	0.008	1.152	1.138	4.095	1.759	`abcdefghijklmnopqrstu`
22	0.008	1.154	1.144	4.238	4.466	`abcdefghijklmnopqrstuv`
23	0.009	1.154	1.141	4.441	1.743	`abcdefghijklmnopqrstuvw`
24	0.009	1.163	1.153	4.404	4.455	`abcdefghijklmnopqrstuvwx`
25	0.008	1.162	1.151	4.531	1.747	`abcdefghijklmnopqrstuvwxy`
26	0.009	4.392	3.902	4.585	4.466	`abcdefghijklmnopqrstuvwxyz`
27	0.008	4.341	33.170	4.851	1.727	`abcdefghijklmnopqrstuvwxyzA`
28	0.009	4.642	4.508	5.002	4.959	`abcdefghijklmnopqrstuvwxyzAB`
29	0.009	4.650	32.597	36.952	1.747	`abcdefghijklmnopqrstuvwxyzABC`
30	0.009	4.617	4.613	59.268	5.001	`abcdefghijklmnopqrstuvwxyzABCD`
31	0.008	4.696	33.058	42.982	1.747	`abcdefghijklmnopqrstuvwxyzABCDE`
32	0.009	4.936	4.438	39.540	4.953	`abcdefghijklmnopqrstuvwxyzABCDEF`
33	0.009	4.874	32.999	69.576	1.738	`abcdefghijklmnopqrstuvwxyzABCDEFG`
34	0.008	4.975	4.840	43.781	4.961	`abcdefghijklmnopqrstuvwxyzABCDEFGH`
35	0.009	4.994	33.765	40.142	1.744	`abcdefghijklmnopqrstuvwxyzABCDEFGHI`
36	0.009	5.213	4.780	70.239	5.114	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJ`
37	0.008	5.117	32.366	46.930	1.742	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJK`
38	0.008	5.230	4.573	43.434	5.150	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL`
39	0.008	5.312	32.632	76.315	1.752	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM`
40	0.008	5.483	4.573	51.809	5.195	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN`
41	0.008	5.595	32.400	46.811	1.772	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO`
42	0.009	5.527	4.961	87.013	5.141	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP`
43	0.009	5.624	32.732	55.775	1.780	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ`
44	0.009	5.893	5.046	49.956	5.552	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR`
45	0.009	5.897	32.684	495.147	1.819	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS`
46	0.008	5.984	4.982	542.566	5.482	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST`
47	0.009	5.834	32.420	66.082	1.835	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU`
48	0.009	6.172	5.057	97.620	5.619	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV`
49	0.009	6.180	32.873	531.977	1.863	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW`
50	0.009	6.306	5.420	576.093	5.626	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX`
	0.008	3.220	9.671	64.994	2.776	

Table 11.3 `context --jit --jithash=lua51 test.tex`

	right	left	center	edges
1				
2				
3				
4			JIT20	
5			JIT20	
6			JIT20	
7	JIT51		JIT20	
8		JIT51		
9	JIT51	JIT51		
10	JIT51	JIT51		
11	JIT51	JIT51		
12	JIT51	JIT51		
13	JIT51	JIT51	JIT20	
14	JIT51	JIT51	JIT20	
15	JIT51	JIT51	JIT20	
16	JIT51	JIT51	JIT20	
17	JIT51	JIT51	JIT20	
18	JIT51		JIT20	
19	JIT51	JIT51	JIT20	
20	JIT51	JIT51	JIT20	JIT20
21	JIT51	JIT20	JIT20	JIT20
22	JIT51		JIT20	JIT20
23	JIT51		JIT20	JIT20
24	JIT51	JIT20	JIT20	JIT20
25	JIT51	JIT20	JIT20	JIT20
26	JIT20	JIT20	JIT20	JIT20
27	JIT20	JIT20	JIT20	JIT20
28	JIT20	JIT20	JIT20	JIT20
29	JIT20	JIT20	JIT20	JIT20
30	JIT20	JIT20	JIT20	JIT20
31	JIT20	JIT20	JIT20	JIT20
32	JIT20	JIT20	JIT20	JIT20
33	JIT20	JIT20	JIT20	JIT20
34	JIT20	JIT20	JIT20	JIT20
35	JIT20	JIT20	JIT20	JIT20
36	JIT20	JIT20	JIT20	JIT20
37	JIT20	JIT20	JIT20	
38	JIT20	JIT20	JIT20	JIT20
39	JIT20	JIT20	JIT20	JIT20
40	JIT20	JIT20	JIT20	JIT20
41	JIT20	JIT20	JIT20	JIT20
42	JIT20	JIT20	JIT20	JIT20
43	JIT20	JIT20	JIT20	JIT20
44	JIT20	JIT20	JIT20	JIT20
45	JIT20	JIT20	JIT20	JIT20
46	JIT20	JIT20	JIT20	JIT20
47	JIT20	JIT20	JIT20	JIT20
48	JIT20	JIT20	JIT20	JIT20
49	JIT20	JIT20	JIT20	JIT20
50	JIT20	JIT20	JIT20	JIT20

LuajitTeX only

	right	left	center	edges
1	JIT20	JIT51	JIT51	JIT51
2	JIT51	JIT20	JIT20	JIT20
3	JIT20	JIT20	JIT20	JIT20
4	JIT51	JIT51	JIT20	JIT20
5	JIT51	JIT51	JIT20	JIT20
6	JIT51	JIT51	JIT20	JIT20
7	JIT51	JIT51	JIT20	JIT20
8	JIT51	JIT51	JIT51	JIT20
9	JIT51	JIT51	JIT51	JIT20
10	JIT51	JIT51	JIT51	JIT20
11	JIT51	JIT51	JIT51	JIT20
12	JIT51	JIT51	JIT51	JIT20
13	JIT51	JIT51	JIT20	JIT20
14	JIT51	JIT51	JIT20	JIT20
15	JIT51	JIT51	JIT20	JIT20
16	JIT51	JIT51	JIT20	JIT20
17	JIT51	JIT51	JIT20	JIT20
18	JIT51	JIT51	JIT20	JIT20
19	JIT51	JIT51	JIT20	JIT20
20	JIT51	JIT51	JIT20	JIT20
21	JIT51	JIT20	JIT20	JIT20
22	JIT51	JIT20	LUA52	JIT20
23	JIT51	JIT20	LUA52	JIT20
24	JIT51	JIT20	LUA52	JIT20
25	JIT51	JIT20	LUA52	JIT20
26	JIT20	JIT20	LUA52	JIT20
27	JIT20	JIT20	LUA52	JIT20
28	JIT20	JIT20	LUA52	JIT20
29	JIT20	JIT20	LUA52	JIT20
30	JIT20	JIT20	LUA52	JIT20
31	JIT20	JIT20	LUA52	JIT20
32	JIT20	JIT20	LUA52	JIT20
33	JIT20	JIT20	LUA52	JIT20
34	JIT20	JIT20	LUA52	JIT20
35	JIT20	JIT20	LUA52	JIT20
36	JIT20	JIT20	LUA52	JIT20
37	JIT20	JIT20	LUA52	LUA52
38	JIT20	JIT20	LUA52	LUA52
39	JIT20	JIT20	LUA52	LUA52
40	JIT20	JIT20	LUA52	LUA52
41	JIT20	JIT20	LUA52	LUA52
42	JIT20	JIT20	LUA52	LUA52
43	LUA52	JIT20	LUA52	LUA52
44	LUA52	LUA52	LUA52	LUA52
45	LUA52	LUA52	LUA52	LUA52
46	LUA52	LUA52	LUA52	LUA52
47	LUA52	JIT20	LUA52	LUA52
48	LUA52	LUA52	LUA52	LUA52
49	LUA52	LUA52	LUA52	LUA52
50	LUA52	LUA52	LUA52	LUA52

Both engines.

Table 11.4 The best performances per engine and hasher.

3	31	`<< /D [ * 0 R /Fit ] /S /GoTo >>`
4	43	`/Border [ 0 0 0 ] /F 4 /Subtype /Link /A * 0 R`
4	51	`<< /D [ * 0 R /Fit ] /S /GoTo >>`
5	410	`/Border [ 0 0 0 ] /F 4 /Subtype /Link /A * 0 R`
5	210	`<< /D [ * 0 R /Fit ] /S /GoTo >>`
6	29947	`/Border [ 0 0 0 ] /F 4 /Subtype /Link /A * 0 R`
6	29823	`<< /D [ * 0 R /Fit ] /S /GoTo >>`

In the next table we show a few cases. The characters that are taken into account are colored red.[27]

n	text	consulted
3	`<< /D [ 8 0 R /Fit ] /S /GoTo >>`	`<< /D [ 8 0 R /Fit ] /S /GoTo >>`
3	`<< /D [ 9 0 R /Fit ] /S /GoTo >>`	`<< /D [ 9 0 R /Fit ] /S /GoTo >>`
3	`<< /D [ 10 0 R /Fit ] /S /GoTo >>`	`<< /D [ 10 0 R /Fit ] /S /GoTo >>`
3	`<< /D [ 11 0 R /Fit ] /S /GoTo >>`	`<< /D [ 11 0 R /Fit ] /S /GoTo >>`
3	`<< /D [ 12 0 R /Fit ] /S /GoTo >>`	`<< /D [ 12 0 R /Fit ] /S /GoTo >>`
4	`<< /D [ 8 0 R /Fit ] /S /GoTo >>`	`<< /D [ 8 0 R /Fit ] /S /GoTo >>`
4	`<< /D [ 9 0 R /Fit ] /S /GoTo >>`	`<< /D [ 9 0 R /Fit ] /S /GoTo >>`
4	`<< /D [ 10 0 R /Fit ] /S /GoTo >>`	`<< /D [ 10 0 R /Fit ] /S /GoTo >>`
4	`<< /D [ 11 0 R /Fit ] /S /GoTo >>`	`<< /D [ 11 0 R /Fit ] /S /GoTo >>`
4	`<< /D [ 12 0 R /Fit ] /S /GoTo >>`	`<< /D [ 12 0 R /Fit ] /S /GoTo >>`

Of course, in practice, in LUA 5.2 the longer string exceeds 40 characters so is never hashed anyway. Apart from this maximum, the LUA hash code looks like this:

```
/* Lua will use at most ~(2^LUAI_HASHLIMIT) bytes from
a string to compute its hash */
...
h = cast(unsigned int,len) ;
step = (len>>LUAI_HASHLIMIT) + 1 ;
for (l1=len; l1>=step; l1-=step) {
   h = h ^ ((h<<5) + (h>>2) + cast(unsigned char,str[l1-1])) ;
}
...
```

This translates in verbose LUA function as follows:

```
function string.luahash(str,shift)
    local len  = #str
    local hash = len
    local step = bit32.rshift(len,shift or 5) + 1
    for i=len,1,-step do
        hash = bit32.bxor(hash, (
            bit32.lshift(hash,5) +
            bit32.rshift(hash,2) +
            string.byte(string.sub(str,i,i))
```

---

[27] Again the first column indicates the shift applied to the length in order to determine the step.

```
        ) )
    end
    return hash
end
```

The reader can argue that the following string would perform better:

```
/Subtype/Link/Border[0 0 0]/F 4/A 12 0 R
```

but this is not the case. Also, here we use PDF code, but similar cases can happen if we flush TEX commands:

```
\dothisorthat{1}
\dothisorthat{101}
\dothisorthat{10101}
```

And in the case of UTF strings, it remains a fact that when characters need two bytes a sequence can end up with each odd or even byte being the same. This is one more reason to support upto 64 byte (or 40 in practice) hashing.

Because of this we decided to experiment with a value of 64 instead.[28] We can do the same when we use the LUA 5.1 method in LUAJIT. In table 11.5 and 11.6 we show the timings. Interesting is that we lost the extremes now. The performance of the default settings are compared with the higher values in table 11.7. Of course the numbers are just indications and there might be small differences between test runs. Therefore we use a threshold of 5% when we compare two methods.

So how does this affect us in document production? It is not that hard to get a processing rate of a few dozen pages per second on a modern machine, even with somewhat complex documents, where XML turns into PDF. However, interactivity comes somehow with a price when we use LUAJITTEX. In CONTEXT MKIV we do all PDF annotations in LUA and that involves assembling dictionaries. Here are two examples, a destination:

```
<< /D [ 15 0 R /Fit ] /S /GoTo >>
```

and a reference:

```
/Subtype /Link /Border [ 0 0 0 ] /F 4 /A 16 0 R
```

These strings are build with small variations and at some point end up in the PDF file. The same string can end up in the file several times, although sometimes we can create a reusable object. In the last case we keep them at the LUA end as reference to such a shareable object, a key in an object reference hash. Now imagine that we have some 30K of such references and/or destinations, which indeed happens in crited documents. In the next two lines we use a `*` to show where the differences are:

```
<< /D [ * 0 R /Fit ] /S /GoTo >>
/Subtype /Link /Border [ 0 0 0 ] /F 4 /A * 0 R
```

---

[28] Of course, in LUATEX, the length limit kicks in before we get to 64.

	none	right	left	center	edges	text
1	0.026	1.202	1.154	1.198	1.723	`a`
2	0.026	1.199	1.156	1.202	1.728	`ab`
3	0.026	1.203	1.174	1.210	1.731	`abc`
4	0.026	1.207	1.177	1.216	1.743	`abcd`
5	0.026	1.210	1.180	1.221	1.738	`abcde`
6	0.027	1.219	1.209	1.256	1.758	`abcdef`
7	0.027	1.234	1.196	1.236	1.741	`abcdefg`
8	0.026	1.218	1.187	1.230	1.742	`abcdefgh`
9	0.026	1.215	1.188	1.217	1.744	`abcdefghi`
10	0.026	1.210	1.193	1.227	1.734	`abcdefghij`
11	0.025	1.214	1.196	1.225	1.732	`abcdefghijk`
12	0.025	1.213	1.180	1.229	1.734	`abcdefghijkl`
13	0.026	1.218	1.186	1.241	1.733	`abcdefghijklm`
14	0.026	1.219	1.191	1.249	1.736	`abcdefghijklmn`
15	0.026	1.236	1.187	1.261	1.748	`abcdefghijklmno`
16	0.026	1.230	1.192	1.256	1.745	`abcdefghijklmnop`
17	0.026	1.226	1.195	1.259	1.743	`abcdefghijklmnopq`
18	0.026	1.225	1.192	1.056	1.740	`abcdefghijklmnopqr`
19	0.025	1.223	1.186	1.057	1.741	`abcdefghijklmnopqrs`
20	0.025	1.230	1.194	1.062	1.751	`abcdefghijklmnopqrst`
21	0.026	1.231	1.197	1.069	1.756	`abcdefghijklmnopqrstu`
22	0.025	1.231	1.208	1.087	1.756	`abcdefghijklmnopqrstuv`
23	0.025	1.234	1.198	1.072	1.760	`abcdefghijklmnopqrstuvw`
24	0.026	1.232	1.195	1.063	1.759	`abcdefghijklmnopqrstuvwx`
25	0.026	1.235	1.199	1.066	1.764	`abcdefghijklmnopqrstuvwxy`
26	0.026	1.248	1.248	1.062	1.762	`abcdefghijklmnopqrstuvwxyz`
27	0.026	1.247	1.216	1.070	1.772	`abcdefghijklmnopqrstuvwxyzA`
28	0.027	1.264	1.223	1.070	1.770	`abcdefghijklmnopqrstuvwxyzAB`
29	0.026	1.248	1.211	1.073	1.586	`abcdefghijklmnopqrstuvwxyzABC`
30	0.026	1.252	1.220	1.075	1.584	`abcdefghijklmnopqrstuvwxyzABCD`
31	0.026	1.255	1.218	1.105	1.593	`abcdefghijklmnopqrstuvwxyzABCDE`
32	0.025	1.256	1.219	1.109	1.594	`abcdefghijklmnopqrstuvwxyzABCDEF`
33	0.025	1.257	1.223	1.122	1.589	`abcdefghijklmnopqrstuvwxyzABCDEFG`
34	0.026	1.253	1.220	1.129	1.596	`abcdefghijklmnopqrstuvwxyzABCDEFGH`
35	0.025	1.077	1.046	1.141	1.590	`abcdefghijklmnopqrstuvwxyzABCDEFGHI`
36	0.026	1.080	1.033	1.159	1.599	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJ`
37	0.026	1.060	1.034	1.162	1.595	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJK`
38	0.025	1.060	1.040	1.171	1.599	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL`
39	0.025	1.063	1.033	1.178	1.600	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM`
40	0.026	1.061	1.029	1.137	1.602	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN`
41	0.025	1.060	1.032	1.138	1.604	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO`
42	0.025	1.064	1.032	1.151	1.622	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP`
43	0.026	1.068	1.039	1.151	1.635	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ`
44	0.026	1.069	1.039	1.149	1.633	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR`
45	0.025	1.067	1.041	1.160	1.642	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS`
46	0.026	1.071	1.040	1.155	1.651	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST`
47	0.025	1.073	1.042	1.155	1.664	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU`
48	0.026	1.088	1.059	1.146	1.668	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV`
49	0.026	1.099	1.067	1.173	1.673	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW`
50	0.025	1.102	1.063	1.140	1.669	`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX`
	0.026	1.179	1.148	1.155	1.689	

Table 11.5 `context test.tex` with len<=40 and hash<=64

	none	right	left	center	edges	text
1	0.016	1.078	1.078	1.094	1.684	a
2	0.008	1.112	1.098	1.116	1.657	ab
3	0.008	1.108	1.091	1.109	1.646	abc
4	0.009	1.108	1.095	1.126	1.653	abcd
5	0.009	1.104	1.099	1.131	1.647	abcde
6	0.009	1.110	1.102	1.135	1.648	abcdef
7	0.009	1.113	1.099	1.130	1.650	abcdefg
8	0.009	1.116	1.108	1.123	1.640	abcdefgh
9	0.009	1.115	1.107	1.127	1.646	abcdefghi
10	0.009	1.120	1.114	1.132	1.645	abcdefghij
11	0.009	1.126	1.121	1.137	1.646	abcdefghijk
12	0.009	1.121	1.115	1.144	1.646	abcdefghijkl
13	0.008	1.128	1.117	1.158	1.648	abcdefghijklm
14	0.010	1.129	1.121	1.168	1.655	abcdefghijklmn
15	0.009	1.132	1.120	1.174	1.657	abcdefghijklmno
16	0.009	1.127	1.118	1.205	1.650	abcdefghijklmnop
17	0.009	1.129	1.115	1.232	1.655	abcdefghijklmnopq
18	0.009	1.134	1.079	1.263	1.660	abcdefghijklmnopqr
19	0.008	1.134	1.138	1.273	1.656	abcdefghijklmnopqrs
20	0.009	1.134	1.123	1.306	1.659	abcdefghijklmnopqrst
21	0.009	1.137	1.124	1.331	1.663	abcdefghijklmnopqrstu
22	0.009	1.150	1.135	1.346	1.677	abcdefghijklmnopqrstuv
23	0.009	1.151	1.137	1.349	1.682	abcdefghijklmnopqrstuvw
24	0.008	1.131	1.120	1.326	1.662	abcdefghijklmnopqrstuvwx
25	0.009	1.134	1.120	1.326	1.677	abcdefghijklmnopqrstuvwxy
26	0.009	1.136	1.122	1.329	1.689	abcdefghijklmnopqrstuvwxyz
27	0.009	1.147	1.126	1.328	1.706	abcdefghijklmnopqrstuvwxyzA
28	0.009	1.145	1.130	1.329	1.722	abcdefghijklmnopqrstuvwxyzAB
29	0.008	1.155	1.140	4.739	1.758	abcdefghijklmnopqrstuvwxyzABC
30	0.009	1.169	1.147	5.212	1.778	abcdefghijklmnopqrstuvwxyzABCD
31	0.009	1.195	1.173	5.438	1.784	abcdefghijklmnopqrstuvwxyzABCDE
32	0.009	1.200	1.175	5.288	1.782	abcdefghijklmnopqrstuvwxyzABCDEF
33	0.008	1.201	1.181	5.698	1.797	abcdefghijklmnopqrstuvwxyzABCDEFG
34	0.009	1.218	1.201	5.676	1.805	abcdefghijklmnopqrstuvwxyzABCDEFGH
35	0.008	1.230	1.215	5.933	1.822	abcdefghijklmnopqrstuvwxyzABCDEFGHI
36	0.009	1.251	1.230	5.795	1.830	abcdefghijklmnopqrstuvwxyzABCDEFGHIJ
37	0.008	1.257	1.234	5.933	1.842	abcdefghijklmnopqrstuvwxyzABCDEFGHIJK
38	0.008	1.273	1.251	5.953	1.849	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL
39	0.009	1.289	1.260	6.297	1.845	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM
40	0.009	1.295	1.273	6.005	1.841	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN
41	0.009	1.312	1.285	6.303	1.843	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNO
42	0.008	1.325	1.309	6.110	1.852	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
43	0.009	1.337	1.319	6.672	1.871	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
44	0.009	1.330	1.305	6.417	1.838	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
45	0.008	1.328	1.303	6.690	1.843	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS
46	0.009	1.330	1.310	6.400	1.852	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST
47	0.009	1.330	1.312	7.058	1.853	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
48	0.008	1.331	1.308	6.736	1.847	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV
49	0.009	1.326	1.305	7.123	1.850	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW
50	0.009	1.331	1.305	6.893	1.848	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
	0.009	1.190	1.174	3.366	1.735	

Table 11.6 `context --jit test.tex` with hash<=64

	right	left	center	edges
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13			40 / 64	
14			40 / 64	
15			40 / 64	
16			40 / 64	
17			40 / 64	
18			40 / 64	
19			40 / 64	
20			40 / 64	40 / 64
21			40 / 64	
22				40 / 64
23				
24				40 / 64
25				
26	40 / 64	40 / 64		40 / 64
27	40 / 64	40 / 64		
28	40 / 64	40 / 64		40 / 64
29	40 / 64	40 / 64		40 / 64
30	40 / 64	40 / 64		40 / 64
31	40 / 64	40 / 64		40 / 64
32	40 / 64	40 / 64		40 / 64
33	40 / 64	40 / 64		40 / 64
34	40 / 64	40 / 64		40 / 64
35	40 / 64	40 / 64	40 / 32	40 / 64
36	40 / 64	40 / 64	40 / 32	40 / 64
37	40 / 64	40 / 64	40 / 32	
38	40 / 64	40 / 64	40 / 32	
39	40 / 64	40 / 64	40 / 32	
40	40 / 64	40 / 64	40 / 32	
41	40 / 64	40 / 64	40 / 32	
42	40 / 64	40 / 64	40 / 32	
43		40 / 64	40 / 32	
44			40 / 32	
45			40 / 32	
46			40 / 32	
47			40 / 32	40 / 32
48			40 / 32	40 / 32
49			40 / 32	40 / 32
50			40 / 32	40 / 32

LuaTeX (size limit 40)

	right	left	center	edges
1	40 / 64			
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13			40 / 64	
14			40 / 64	
15			40 / 64	
16			40 / 64	
17			40 / 64	
18			40 / 64	
19			40 / 64	
20			40 / 64	40 / 64
21			40 / 64	40 / 64
22			40 / 64	40 / 64
23			40 / 64	
24			40 / 64	40 / 64
25			40 / 64	
26	40 / 64	40 / 64	40 / 64	40 / 64
27	40 / 64	40 / 64	40 / 64	
28	40 / 64	40 / 64	40 / 64	40 / 64
29	40 / 64	40 / 64	40 / 64	
30	40 / 64	40 / 64	40 / 64	40 / 64
31	40 / 64	40 / 64	40 / 64	
32	40 / 64	40 / 64	40 / 64	40 / 64
33	40 / 64	40 / 64	40 / 64	
34	40 / 64	40 / 64	40 / 64	40 / 64
35	40 / 64	40 / 64	40 / 64	
36	40 / 64	40 / 64	40 / 64	40 / 64
37	40 / 64	40 / 64	40 / 64	40 / 32
38	40 / 64	40 / 64	40 / 64	40 / 64
39	40 / 64	40 / 64	40 / 64	40 / 32
40	40 / 64	40 / 64	40 / 64	40 / 64
41	40 / 64	40 / 64	40 / 64	
42	40 / 64	40 / 64	40 / 64	40 / 64
43	40 / 64	40 / 64	40 / 64	
44	40 / 64	40 / 64	40 / 64	40 / 64
45	40 / 64	40 / 64	40 / 64	
46	40 / 64	40 / 64	40 / 64	40 / 64
47	40 / 64	40 / 64	40 / 64	
48	40 / 64	40 / 64	40 / 64	40 / 64
49	40 / 64	40 / 64	40 / 64	
50	40 / 64	40 / 64	40 / 64	40 / 64

LuajitTeX (no size limit)

Table 11.7 More than 5% difference between 32 byte or 64 byte hashing.

If we replace these `*` by a number, there are big differences between the engines with respect to the time needed. This is summarized in the next table.[29]

Lua 5.2	**LuaJIT 2.0**	**LuaJIT 2.0+5.1**	
0.096	0.046	0.047	`<< /D [ * 0 R /Fit ] /S /GoTo >>`
0.054	6.017	0.055	`/Subtype /Link /Border [ 0 0 0 ] /F 4 /A * 0 R`

Especially the second case behaves bad in LuaJIT. Say that a result comes out as:

```
/Subtype /Link /Border [ 0 0 0 ] /F 4 /A 12 0 R
/Subtype /Link /Border [ 0 0 0 ] /F 4 /A 123 0 R
/Subtype /Link /Border [ 0 0 0 ] /F 4 /A 1234 0 R
```

The LuaJIT hasher (more or less) looks at the first 4, last 4, middle 4 and somewhere a quarter along the string, and uses these sequences for the calculation, so you can imagine that there are clashes. The Lua 5.1 hasher runs over part of the string and sees more of the difference. The 5.2 hasher has a threshold and doesn't hash at all when the length exceeds (by default) 40 characters, which is the case with the second string. Looking at only specific parts of a string is somewhat arbitrary and what works for one kind of application is not always good for another.

After these tests we decided that it makes sense to replace the LuaJIT hash calculation by the traditional Lua one (or at least give users a choice at startup. The choice of hash is a runtime option:

```
mtxrunjit --script context --jithash=lua51    ......
mtxrunjit --script context --jithash=luajit20 ......
```

For the moment we default to the traditional Lua 5.1 hashing method. Although it can behave real bad on some large strings we think that chances are low that this will happen in practice. An overall good performance on strings like the hyperlink examples is more important. Using the Lua 5.2 method would be even better but it required a change in the virtual machine and that is not something we have in mind.

---

[29] The numbers concern 30K hash creations. The time shown is the average over 30 runs.

12 Properties

12.1 Introduction

Attributes are a nice extension to TEX as they permits us to let information travel with nodes. Internally they are represented as a linked list that travels with a node. Because often a sequence of nodes has the same attributes, this mechanism is quite efficient. Access is relatively fast too. Attributes have a number and a value (also a number) which is fine. Of course one could wish for them to be anything, but imagine the amount of management needed in the engine if that were the case. Not only does saving and restoring (due to grouping) at the TEX end has no LUA equivalent, an overload of the LUA registry (the most natural interface for this) is not what we want. Of course it is also not acceptable that (future) extensions slow down a run. In fact, leaner and meaner should be the main objective.

At some point I thought that packing crucial information in a node using a bitset would help to speed up some critical mechanisms (mostly fonts) but although managing some 32 or 64 on–off states is possible in a more closed macro package, in practice it would lead to conflicts in use. Also, an experimental implementation of this idea was not faster than using attributes due to the fact that manipulating bits also involves function calls that deal with setting, resetting, masking and more. It also makes nodes larger and increases the memory footprint.

So, when I discarded that idea, I moved to another one, which is associating a LUA table with each node (that makes sense). Again, an implementation where some way a reference to a table is carried with a node, is non–trivial because it has to go via the LUA registry and will not be too efficient in terms of speed. Also, when dealing with such information one wants to stay at the LUA end and not cross the C–boundary too often.

Therefore a different approach was taken which involves a LUA table. The main issue with carrying information with a node is not to associate that information, but to make sure that it gets cleaned up when a node is freed and copied when a node is copied. All nodes that have attributes, also get properties.

12.2 The implementation

The implementation is rather minimalistic. This is because hard codes solutions don't fit in the LUATEX design philosophy. Also, there are many ways to use such a mechanism so too much hard coded behaviour only complicates usage.

When a node is copied, we also copy the associated property entry. Normally its type is `nil` or `table`. Depending on how you enabled this mechanism, the table copy is shallow (just a reference to the same table), or we assign en empty table with the original as metatable index. The second approach as some more overhead.

When a new node is assigned, nothing extra is done with the properties. The overhead is zero. This means that when you want to assign properties at the LUA end, you also have to check if a node property already has a table and if not, create one. The same is true for querying properties: you have to test if there are properties at all.

When you use the 'direct' node model, you can directly access the property table. But, with direct as well as wrapped nodes, you can also use setters and getters. The property table has no metatable so you can add your own one for alternative access if needed. In ConTEXt you can best stay away from such hacks and use the provided mechanisms because otherwise you get a performance hit.

12.3 The LUA interface

The interface (in regular nodes as well as direct ones) is quite simple and provides five functions:

```
set_properties_mode(boolean,boolean)
flush_properties_table()
get_properties_table()
getproperty(node_id)
setproperty(node_id,value)
```

By default this mechanism is disabled so that when it's not used, there is no overhead involved. With `set_properties_mode` the first argument determines if you enable or disable this mechanism. The properties themselves are untouched. When the second argument is `true` copied properties create a new table with a metatable pointing to the original. You can flush all properties with `flush_properties_table`.

You can access and set properties with `getproperty` and `setproperty`. Instead you can also use the table approach, where you can reach the table with `get_properties_table`. Keep in mind that the normal and direct calls to this function return a different table.

12.4 A few examples

The following examples use ConTEXt but apart from the calls to the `context` namespace, they are rather generic. We have enabled the property mechanism with:

```
set_properties_mode(true)
```

We fill a box:

```
\newbox\MyPropertyBox

\setbox\MyPropertyBox=\hbox{test}

local list = tex.getbox("MyPropertyBox").list

local function start()
    context.starttabulate { "||||" }
    context.HL()
end
```

```
local function stop()
    context.HL()
    context.stoptabulate()
end

local function row(n,p)
    context.NC() context(tostring(n==p))
    context.NC() context(tostring(n))
    context.NC() context(tostring(p))
    context.NC() context.NR()
end
```

We will demonstrate the four access models. First regular properties using functions:

```
for n in node.traverse(list) do
    node.setproperty(n,{ vif = n })
end
start()
for n in node.traverse(list) do
    row(n,node.getproperty(n).vif)
end
stop()
```

```
true  <node nil < 805343 > 805633 : glyph 256>     <node nil < 805343 > 805633 : glyph 256>
true  <node 805343 < 805633 > 805337 : glyph 256>  <node 805343 < 805633 > 805337 : glyph 256>
true  <node 805633 < 805337 > 700091 : glyph 256>  <node 805633 < 805337 > 700091 : glyph 256>
true  <node 805337 < 700091 > 805331 : kern 1>     <node 805337 < 700091 > 805331 : kern 1>
true  <node 700091 < 805331 > nil : glyph 256>     <node 700091 < 805331 > nil : glyph 256>
```

We can use a table instead (in fact, we can use both approaches mixed:

```
local n_properties = node.get_properties_table()

for n in node.traverse(list) do
    n_properties[n] = { vit = n }
    node.direct.setproperty(n,{ vdf = n })
end
start()
for n in node.traverse(list) do
    row(n,n_properties[n].vit)
end
stop()
```

```
true  <node nil < 805343 > 805633 : glyph 256>     <node nil < 805343 > 805633 : glyph 256>
true  <node 805343 < 805633 > 805337 : glyph 256>  <node 805343 < 805633 > 805337 : glyph 256>
true  <node 805633 < 805337 > 700091 : glyph 256>  <node 805633 < 805337 > 700091 : glyph 256>
true  <node 805337 < 700091 > 805331 : kern 1>     <node 805337 < 700091 > 805331 : kern 1>
true  <node 700091 < 805331 > nil : glyph 256>     <node 700091 < 805331 > nil : glyph 256>
```

The direct method looks the same, apart from a cast to direct:

```
for n in node.direct.traverse(node.direct.todirect(list)) do
```

```
    node.direct.setproperty(n,{ vdf = n })
end
start()
for n in node.direct.traverse(node.direct.todirect(list)) do
    row(n,node.direct.getproperty(n).vdf)
end
stop()
```

```
true   805343   805343
true   805633   805633
true   805337   805337
true   700091   700091
true   805331   805331
```

Again, we can use the table approach:

```
local d_properties = node.direct.get_properties_table()

for n in node.direct.traverse(node.direct.todirect(list)) do
    d_properties[n] = { vdt = n }
end
start()
for n in node.direct.traverse(node.direct.todirect(list)) do
    row(n,d_properties[n].vdt)
end
stop()
```

```
true   805343   805343
true   805633   805633
true   805337   805337
true   700091   700091
true   805331   805331
```

13 Functions

13.1 Introduction

As part of the crited project Luigi and I also tried to identity weak spots in the engine and although we found some issues not all were dealt with because complicating the machinery makes no sense. However just like the new `properties` mechanism provides a real simple way to associate extra Lua data to a node without bothering about freeing it when a node is flushed, the next `luafunctions` mechanism provides an additional and fast way to cross the TeX–Lua boundary.

13.2 Callbacks

In LuaTeX we can create more functionality by using Lua which means that we end up (at least in ConTeXt) with a constant switching between TeX macro expansion and Lua code interpretation. The magic word in this process is `callback` and there are two variants:

- At well defined moments in processing its input and node lists, TeX will check if a specific callback is defined and if so, it will run that code.
- As part of the input you can have a `\directlua` command and that one gets expanded and processed. It can print back content into the current input buffer.[30]

The first type is call a 'direct' callback because TeX calls it directly, and the second one is an 'indirect' one (even if the command is `\directlua`). It has a deferred cousin `\latelua` that results in a node being inserted that will become a Lua call during shipout, when the page is turned into a PDF stream.

A callback of the first category is pretty fast because the code is already translated in Lua bytecode. Checking if a callback has been assigned at all is fast too. The second variant is slower because each time the input has to be interpreted and checked on validity. Then there is of course some overhead in making the call itself.

There is a subtle aspect there. If you have a document that needs say ten calls like:

```
\directlua{tex.print("[x]")}
```

and you have these calls inlined, you end up with ten times conversion into tokens (TeX's internal view) and ten times conversion back to a string that gets fed into Lua. On the other hand,

```
\def\MyCall{\directlua{tex.print("[x]")}}
```

where we call `\MyCall` ten times is more efficient because we have already tokenized the `\directlua`. If we have

---

[30] Currently this process is somewhat more complex than needed, which is a side effect of supporting multiple Lua states in the first versions of LuaTeX. We will clean up this mechanism at some point.

```
foo foo foo \directlua{tex.print("[1]")} ...
bar bar bar \directlua{tex.print("[2]")} ...
```

It makes sense to wrap this into a definition:

```
\def\MyCall#1{\directlua{tex.print("[#1]")}}
```

and use:

```
foo foo foo \MyCall{1} bar bar bar \MyCall{1} ...
```

Of course this is not unique for `\directlua` and to be honest, apart from convenience (read: less input) the gain often can be neglected. Because a macro package wraps functionality in (indeed) macros we already save us the tokenization step. We can save some time by wrapping more in a function at the Lua end:

```
\startluacode
function MyFloat(f)
    tex.print(string.format("%0.5f",f))
end
\stopluacode

\def\MyFloat#1%
  {\directlua{MyFloat(#1)}}
```

This is somewhat more efficient than:

```
\def\MyFloat#1%
  {\directlua{tex.print(string.format("\letterpercent0.5f",#1))}}
```

Of course this is only true when we call this macro a lot of times.

13.3 Shortcuts

When we talk of 'often' or 'a lot' we mean many thousands of calls. There are some places in ConTEXt where this is indeed the case, for instance when we process large registers in critical editions: a few hundred pages of references generated in Lua is no exception there. Think of the following:

```
\startluacode
function GetTitle(n)
    tex.print(Entries[n].title)
end
\stopluacode

\def\GetTitle#1%
  {\directlua{GetTitle(#1)}}
```

If we call `\GetTitle` ourselves it's the same as the `\MyFloat` example, but how about this:

```
\def\GetTitle#1%
  {{\bf \directlua{GetTitle(#1)}}}

\startluacode
function GetTitle(n)
    tex.print(Entries[n].title)
end

function GetEntry(n)
    if Entries[n] then
        tex.print("\\directlua{GetTitle(",n,")}")
        -- some more action
    end
end
\stopluacode
```

Here we have two calls where one is delayed till a later time. This delay results in a tokenization and transation to Lua so it will cost time. A way out is this:

```
\def\GetTitle#1%
  {{\bf \luafunction#1}}

\startluacode
local functions = tex.get_functions_table()

function GetTitle(n)
    tex.print(Entries[n].title)
end

function GetEntry(n)
    if Entries[n] then
        local m = #functions+1
        functions[m] = function() GetTitle(n) end
        tex.print("\\GetTitle{",m,"}")
        -- some more action
    end
end
\stopluacode
```

We define a function at the Lua end and just print a macro call. That call itself calls the defined function using `\luafunction`. For a large number of calls this is more efficient but it will be clear that you need to make sure that used functions are cleaned up. A simple way is to start again at slot one after (say) 100.000 functions, another method is to reset used functions and keep counting.

```
\startluacode
local functions = tex.get_functions_table()

function GetTitle(n)
```

```
    tex.print(Entries[n].title)
end

function GetEntry(n)
    if Entries[n] then
        local m = #functions+1
        functions[m] = function(slot) -- the slot number is always
            GetTitle(n)               -- passed as argument so that
            functions[slot] = nil     -- we can reset easily
        end
        tex.print("\\GetTitle{",m,"}")
        -- some more action
    end
end
\stopluacode
```

As you can expect, in ConTEXt users are not expect to deal directly with functions at all. Already for years you can so this:

```
\def\GetTitle#1%
  {{\bf#1}}

\startluacode
function GetEntry(n)
    if Entries[n] then
        context(function() context.GetTitle(Entries[n].title) end)
        -- some more action
    end
end
\stopluacode
```

Upto LuaTEX 0.78 we had a ConTEXt specific implementation of functions and from 0.79 onwards we use this new mechanism but users won't see that in practice. In the `cld-mkiv.pdf` manual you can find more about accessing ConTEXt from the Lua end.

Keep in mind that `\luafunction` is not that clever: it doesn't pick up arguments. That will be part of future more extensive token handling but of course that will then also be a real slow downer because a mix of TEX tokenization and serialization is subtoptimal (we already did extensive tests with that).

13.4 Helpers

The above mechanism demands some orchestration in the macro package. For instance freeing slots should be consistent and therefore user should not mess directly with the functions table. If you really want to use this feature you can best do this:

```
\startctxfunction MyFunctionA
    context(" A1 ")
```

```
\stopctxfunction

\startctxfunctiondefinition MyFunctionB
    context(" B2 ")
\stopctxfunctiondefinition

\starttext
    \dorecurse{10000}{\ctxfunction{MyFunctionA}}   \page
    \dorecurse{10000}{\MyFunctionB}                \page
    \dorecurse{10000}{\ctxlua{context(" C3 ")}}    \page
    \dorecurse{10000}{\ctxlua{tex.sprint(" D4 ")}} \page
\stoptext
```

In case you're curious about performance, here are timing. Given that we have 10.000 calls the gain is rather neglectable especially because the whole run takes 2.328 seconds for 52 processed pages resulting in 22.4 pages per second. The real gain is in more complex calls with more tokens involved and in ConTEXt we have some placed where we run into the hundreds of thousands. A similar situation occurs when your input comes from databases and is fetched stepwise.

A	**B**	**C**	**D**
0.053	0.044	0.081	0.081

So, we can save 50% runtime but on a simple document like this a few percent is not that much. Of course many such few percentages can add up, and it's one of the reasons why ConTEXt MkIV is pretty fast in spite of all the switching between TEX and Lua. One objective is that an average complex document should be processed with a rate of at least 20 pages per second and in most cases we succeed. This fast function accessing can of course trigger new features in ConTEXt, ones we didn't consider useful because of overhead.

Keep in mind that in most cases, especially when programming in Lua directly the `context` command already does all kind of housekeeping for you. For instance it also keeps track of so called trial typesetting runs and can inject nodes in the current stream as well. So, be warned: there is no real need to complicate your code with this kind of hackery if some high level subsystem provides the functionality already.

14 LUA in METAPOST

14.1 Introduction

Already for some years I have been wondering how it would be if we could escape to Lua inside MetaPost, or in practice, in mplib in LuaTEX. The idea is simple: embed Lua code in a MetaPost file that gets run as soon as it's seen. In case you wonder why Lua code makes sense, imagine generating graphics using external data. The capabilities of Lua to deal with that is more flexible and advanced than in MetaPost. Of course we could generate a MetaPost definition of a graphic from data but it often makes more sense to do the reverse. I finally found time and reason to look into this and in the following sections I will describe how it's done.

14.2 The basics

The approach is comparable to LuaTEX's `\directlua`. That primitive can be used to execute Lua code and in combination with `tex.print` we can pipe strings back into the TEX input stream. A complication is that we have to be able to operate under different so called catcode regimes: the meaning of characters can differ per regime. We also have to deal with line endings in special ways as they relate to paragraphs and such. In MetaPost we don't have that complication so getting back input into the MetaPost input, we can do so with simple strings. For that a mechanism similar to `scantokens` can be used. That way we can return anything (including nothing) as long as MetaPost can interpret it and as long as it fulfils the expectations.

```
numeric n ; n := scantokens("123.456") ;
```

A script is run as follows:

```
numeric n ; n := runscript("return '123.456'") ;
```

This primitive doesn't have the word `lua` in its name so in principle any wrapper around the library can use it as a hook. In the case of LuaTEX the script language is of course Lua. At the MetaPost end we only expect a string. How that string is constructed is completely up to the Lua script. In fact, the user is completely free to implement the runner any way she or he wants, like:

```
local function scriptrunner(code)
    local f = loadstring(code)
    if f then
        return tostring(f())
    else
        return ""
    end
end
```

This is hooked into an instance as follows:

```
local m = mplib.new {
    ...
    run_script = scriptrunner,
    ...
}
```

Now, beware, this is not the ConTeXt way. We provide print functions and other helpers, which we will explain in the next section.

14.3 Helpers

After I got this feature up and running I played a bit with possible interfaces at the ConTeXt (read: MetaFun) end and ended up with a bit more advanced runner where no return value is used. The runner is wrapped in the `lua` macro.

```
numeric n ; n := lua("mp.print(12.34567)") ;
draw textext(n) xsized 4cm withcolor maincolor ;
```

This renders as:

12.34567

In case you wonder how efficient calling Lua is, don't worry: it's fast enough, especially if you consider suboptimal Lua code and the fact that we switch between machineries.

```
draw image (
    lua("statistics.starttiming()") ;
    for i=1 upto 10000 : draw
        lua("mp.pair(math.random(-200,200),math.random(-50,50))") ;
    endfor ;
    setbounds currentpicture to fullsquare xyscaled (400,100) ;
    lua("statistics.stoptiming()") ;
    draw textext(lua("mp.print(statistics.elapsedtime())"))
        ysized 50 ;
) withcolor maincolor withpen pencircle scaled 1 ;
```

Here the line:

```
draw lua("mp.pair(math.random(-200,200),math.random(-50,50))") ;
```

effectively becomes (for instance):

```
draw scantokens "(25,40)" ;
```

which in turn becomes:

```
draw scantokens (25,40) ;
```

The same happens with this:

```
draw textext(lua("mp.print(statistics.elapsedtime())")) ...
```

This becomes for instance:

```
draw textext(scantokens "1.23") ...
```

and therefore:

```
draw textext(1.23) ...
```

We can use `mp.print` here because the `textext` macro can deal with numbers. The following also works:

```
draw textext(lua("mp.quoted(statistics.elapsedtime())")) ...
```

Now we get (in MetaPost speak):

```
draw textext(scantokens (ditto & "1.23" & ditto) ...
```

Here `ditto` represents the double quotes that mark a string. Of course, because we pass the strings directly to `scantokens`, there are no outer quotes at all, but this is how it can be simulated. In the end we have:

```
draw textext("1.23") ...
```

What print variant you use, `mp.print` or `mp.quoted`, depends on what the expected code is: an assignment to a numeric can best be a number or an expression resulting in a number.

This graphic becomes:

The runtime on my current machine is some 0.25 seconds without and 0.12 seconds with caching. But to be honest, speed is not really a concern here as the amount of complex MetaPost graphics can be neglected compared to extensive node list manipulation. Generating the graphic with LuajitTeX takes 15% less time.[31]

The three print command accumulate their arguments:

```
numeric n ; n := lua("mp.print(1) mp.print('+') mp.print(2)") ;
draw textext(n) xsized 1cm withcolor maincolor ;
```

As expected we get:

---

[31] Processing a small 8 page document like this takes about one second, which includes loading a bunch of fonts.

3

Equally valid is:

```
numeric n ; n := lua("mp.print(1,'+',2)") ;
draw textext(n) xsized 1cm withcolor maincolor ;
```

This gives the same result:

3

Of course all kind of action can happen between the prints. It is also legal to have nothing returned as could be seen in the 10.000 dot example: there the timer related code returns nothing, so effectively we have `scantokens("")`. Another helper is `mp.quoted`, as in:

```
draw
    textext(lua("mp.quoted('@0.3f'," & decimal n & ")"))
    withcolor maincolor ;
```

This typesets 3.000. Note the `@`. When no percent character is found in the format specifier, we assume that an `@` is used instead.

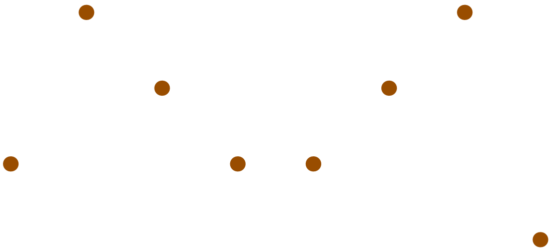
But, the real benefit of embedded Lua is when we deal with data that is stored at the Lua end. First we define a small dataset:

```
\startluacode
table.save("demo-data.lua",
    {
        { 1, 2 }, { 2, 4 }, { 3, 3 }, { 4, 2 },
        { 5, 2 }, { 6, 3 }, { 7, 4 }, { 8, 1 },
    }
)
\stopluacode
```

There are several ways to deal with this table. I will show clumsy as well as better looking ways.

```
lua("MP = { } MP.data = table.load('demo-data.lua')") ;
numeric n ;
lua("mp.print('n := ',\#MP.data)") ;
for i=1 upto n :
    drawdot
        lua("mp.pair(MP.data[" & decimal i & "])") scaled cm
        withpen pencircle scaled 2mm
        withcolor maincolor ;
endfor ;
```

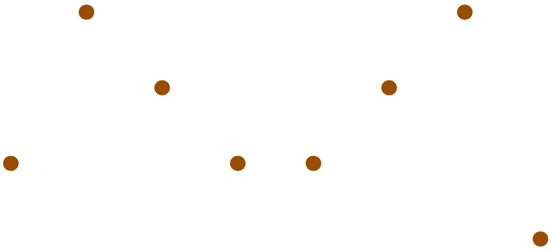
Here we load a Lua table and assign the size to a MetaPost numeric. Next we loop over the table entries and draw the coordinates.



We will stepwise improve this code. In the previous examples we omitted wrapper code but here we show it:

```
\startluacode
    MP.data = table.load('demo-data.lua')
    function MP.n()
        mp.print(#MP.data)
    end
    function MP.dot(i)
        mp.pair(MP.data[i])
    end
\stopluacode

\startMPcode
    numeric n ; n := lua("MP.n()") ;
    for i=1 upto n :
        drawdot
            lua("MP.dot(" & decimal i & ")") scaled cm
            withpen pencircle scaled 2mm
            withcolor maincolor ;
    endfor ;
\stopMPcode
```

So, we create a few helpers in the `MP` table. This table is predefined so normally you don't need to define it. You may however decide to wipe it clean.



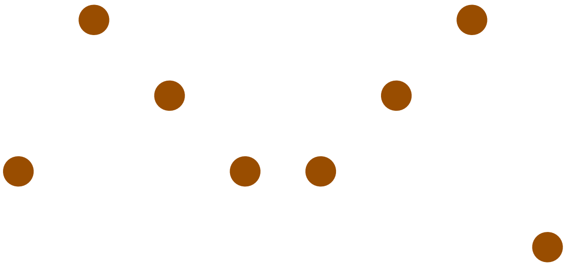
You can decide to hide the data:

```
\startluacode
    local data = { }
    function MP.load(name)
        data = table.load(name)
    end
    function MP.n()
        mp.print(#data)
    end
    function MP.dot(i)
        mp.pair(data[i])
    end
\stopluacode
```

It is possible to use less Lua, for instance in:

```
\startluacode
    local data = { }
    function MP.loaded(name)
        data = table.load(name)
        mp.print(#data)
    end
    function MP.dot(i)
        mp.pair(data[i])
    end
\stopluacode

\startMPcode
    for i=1 upto lua("MP.loaded('demo-data.lua')") :
        drawdot
            lua("MP.dot(",i,")") scaled cm
            withpen pencircle scaled 4mm
            withcolor maincolor ;
    endfor ;
\stopMPcode
```

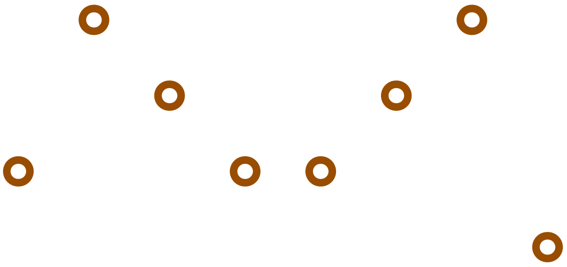
Here we also omit the `decimal` because the `lua` macro is clever enough to recognize it as a number.



By using some MetaPost magic we can even go a step further in readability:

```
\startMPcode{doublefun}
    lua.MP.load("demo-data.lua") ;

    for i=1 upto lua.MP.n() :
        drawdot
            lua.MP.dot(i) scaled cm
            withpen pencircle scaled 4mm
            withcolor maincolor ;
    endfor ;

    for i=1 upto MP.n() :
        drawdot
            MP.dot(i) scaled cm
            withpen pencircle scaled 2mm
            withcolor white ;
    endfor ;
\stopMPcode
```

Here we demonstrate that it also works well in `double` mode, which makes much sense when processing data from other sources. Note how we omit the type lua. prefix: the `MP` macro will deal with that.



So in the end we can simplify the code that we started with to:

```
\startMPcode{doublefun}
    for i=1 upto MP.loaded("demo-data.lua") :
        drawdot
            MP.dot(i) scaled cm
            withpen pencircle scaled 2mm
            withcolor maincolor ;
    endfor ;
\stopMPcode
```

14.4 Access to variables

The question with such mechanisms is always: how far should we go. Although MetaPost is a macro language, it has properties of procedural languages. It also has more introspective features at the user end. For instance, one can loop over the resulting picture and manipulate it. This means that we don't need full access to MetaPost internals. However, it makes sense to provide access to basic variables: `numeric`, `string`, and `boolean`.

```
draw textext(lua("mp.quoted('@0.15f',mp.get.numeric('pi')-math.pi)"))
    ysized 1cm
    withcolor maincolor ;
```

In double mode you will get zero printed but in scaled mode we definitely get a different results:

-0.000006349878856

In the next example we use `mp.quoted` to make sure that indeed we pass a string. The `textext` macro can deal with numbers, but an unquoted `yes` or `no` is asking for problems.

```
boolean b ; b := true ;
draw textext(lua("mp.quoted(mp.get.boolean('b') and 'yes' or 'no')"))
    ysized 1cm
    withcolor maincolor ;
```

Especially when more text is involved it makes sense to predefine a helper in the `MP` namespace, if only because MetaPost (currently) doesn't like newlines in the middle of a string, so a `lua` call has to be on one line.

yes

Here is an example where Lua does something that would be close to impossible, especially if more complex text is involved.

```
string s ; s := "TEX" ; % ""
draw textext(lua("mp.quoted(characters.lower(mp.get.string('s')))"))
    ysized 1cm
    withcolor maincolor ;
```

As you can see here, the whole repertoire of helper functions can be used in a MetaFun definition.

τεχ

14.5 The library

In ConTEXt we have a dedicated runner, but for the record we mention the low level constructor:

```
local m = mplib.new {
    ...
    script_runner = function(s) return loadstring(s)() end,
    script_error  = function(s) print(s) end,
    ...,
}
```

An instance (in this case `m`) has a few extra methods. Instead you can use the helpers in the library.

`m:get_numeric(name)`	returns a numeric (double)
`m:get_boolean(name)`	returns a boolean (`true` or `false`)
`m:get_string (name)`	returns a string
`mplib.get_numeric(m,name)`	returns a numeric (double)
`mplib.get_boolean(m,name)`	returns a boolean (`true` or `false`)
`mplib.get_string (m,name)`	returns a string

In ConTEXt the instances are hidden and wrapped in high level macros, so there you cannot use these commands.

14.6 CONTEXT helpers

The `mp` namespace provides the following helpers:

`print(...)`	returns one or more values
`pair(x,y) pair(t)`	returns a proper pair
`triplet(x,y,z) triplet(t)`	returns an RGB color
`quadruple(w,x,y,z) quadruple(t)`	returns an CMYK color
`format(fmt,...)`	returns a formatted string
`quoted(fmt,...) quoted(s)`	returns a (formatted) quoted string
`path(t[,connect][,close])`	returns a connected (closed) path

The `mp.get` namespace provides the following helpers:

`numeric(name)`	gets a numeric from MetaPost
`boolean(name)`	gets a boolean from MetaPost
`string(name)`	gets a string from MetaPost

14.7 Paths

In the meantime we got several questions on the ConTEXt mailing list about turning coordinates into paths. Now imagine that we have this dataset:

```
10 20 20 20 -- sample 1
30 40 40 60
50 10

10 10 20 30 % sample 2
30 50 40 50
50 20

10 20 20 10 # sample 3
30 40 40 20
50 10
```

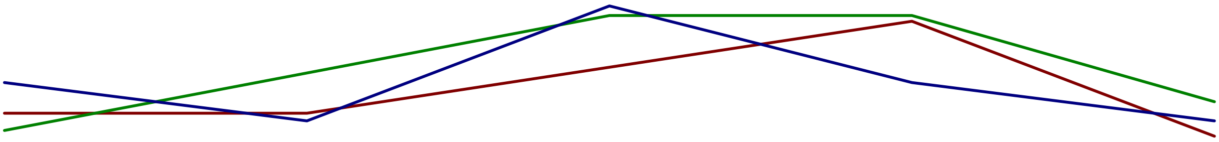
In this case I have put the data in a buffer, so that it can be shown here, as well as used in a demo. Look how we can add comments. The following code converts this into a table with three subtables.

```
\startluacode
  MP.myset = mp.dataset(buffers.getcontent("dataset"))
\stopluacode
```

We use the `MP` (user) namespace to store the table. Next we turn these subtables into paths:

```
\startMPcode
  for i=1 upto lua("mp.print(mp.n(MP.myset))") :
    draw
      lua("mp.path(MP.myset[" & decimal i & "])")
      xysized (HSize,10ExHeight)
      withpen pencircle scaled .25ExHeight
      withcolor basiccolors[i]/2 ;
  endfor ;
\stopMPcode
```

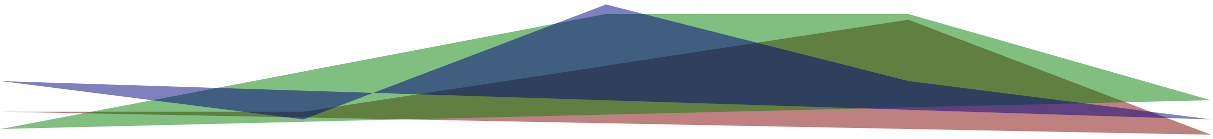
This gives:



Instead we can fill the path, in which case we will also need to close it. The `true` argument deals with that:

```
\startMPcode
  for i=1 upto lua("mp.print(mp.n(MP.myset))") :
    path p ; p :=
      lua("mp.path(MP.myset[" & decimal i & "],true)")
      xysized (HSize,10ExHeight) ;
    fill p
      withcolor basiccolors[i]/2
      withtransparency (1,.5) ;
  endfor ;
\stopMPcode
```

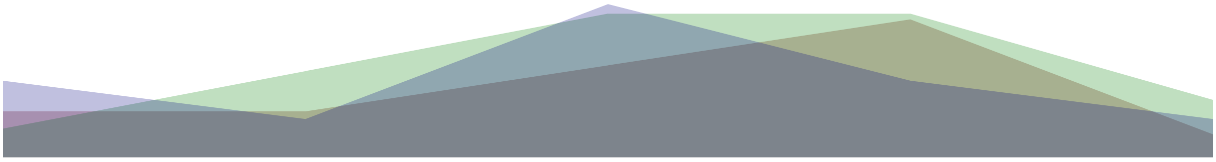
We get:



The following makes more sense:

```
\startMPcode
  for i=1 upto lua("mp.print(mp.n(MP.myset))") :
    path p ; p :=
      lua("mp.path(MP.myset[" & decimal i & "])")
      xysized (HSize,10ExHeight) ;
    p :=
      (xpart llcorner boundingbox p,0) --
      p --
      (xpart lrcorner boundingbox p,0) --
      cycle ;
    fill p
      withcolor basiccolors[i]/2
      withtransparency (1,.25) ;
  endfor ;
\stopMPcode
```

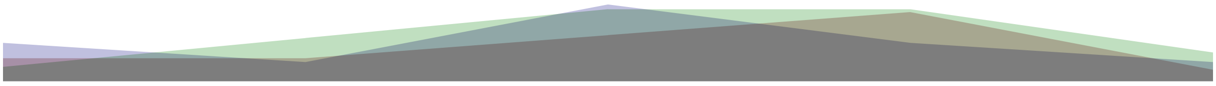
So this gives:



This (area) fill is so common, that we have a helper for it:

```
\startMPcode
  for i=1 upto lua("mp.size(MP.myset)") :
    fill area
      lua("mp.path(MP.myset[" & decimal i & "])")
      xysized (HSize,5ExHeight)
      withcolor basiccolors[i]/2
      withtransparency (2,.25) ;
  endfor ;
\stopMPcode
```

So this gives:



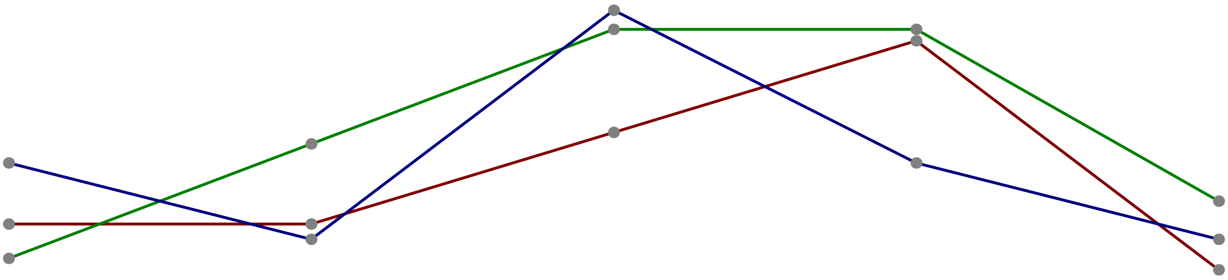
This snippet of MetaPost code still looks kind of horrible, so how can we make it look better? Here is an attempt. First we define a bit more Lua:

```
\startluacode
local data = mp.dataset(buffers.getcontent("dataset"))

MP.dataset = {
  Line = function(n) mp.path(data[n]) end,
  Size = function()  mp.size(data)    end,
}
\stopluacode
```

We can now make the MetaPost look more natural. Of course, this is possible because in MetaFun the `lua` macro does some extra work.

```
\startMPcode
  for i=1 upto lua.MP.dataset.Size() :
    path p ; p :=
      lua.MP.dataset.Line(i)
      xysized (HSize,20ExHeight) ;
    draw
      p
      withpen pencircle scaled .25ExHeight
      withcolor basiccolors[i]/2 ;
    drawpoints
      p
      withpen pencircle scaled ExHeight
      withcolor .5white ;
  endfor ;
\stopMPcode
```

As expected, we get the desired result:



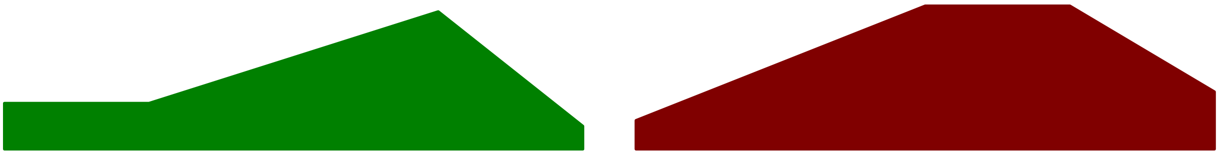
Once we start making things look nicer and more convenient, we quickly end up with helpers like those in the next example. First we save some demo data in files:

```
\startluacode
  io.savedata("foo.tmp","10 20 20 20 30 40 40 60 50 10")
  io.savedata("bar.tmp","10 10 20 30 30 50 40 50 50 20")
\stopluacode
```

We load the data in datasets:

```
\startMPcode
  lua.mp.datasets.load("foo","foo.tmp") ;
  lua.mp.datasets.load("bar","bar.tmp") ;
  fill area
    lua.mp.datasets.foo.Line()
    xysized (HSize/2-EmWidth,10ExHeight)
    withpen pencircle scaled .25ExHeight
    withcolor green/2 ;
  fill area
    lua.mp.datasets.bar.Line()
    xysized (HSize/2-EmWidth,10ExHeight)
    shifted (HSize/2+EmWidth,0)
    withpen pencircle scaled .25ExHeight
    withcolor red/2 ;
\stopMPcode
```

Because the datasets are stored by name, we can use them without worrying about them being forgotten:



If no tag is given, the filename (without suffix) is used as a tag, so the following is valid:

```
\startMPcode
  lua.mp.datasets.load("foo.tmp") ;
  lua.mp.datasets.load("bar.tmp") ;
\stopMPcode
```

The following methods are defined for a dataset:

`method`	usage
`Size`	the number of subsets in a dataset
`Line`	the joined pairs in a dataset making a non-closed path
`Data`	the table containing the data (in subsets, so there is always at least one subset)

Due to limitations in MetaPost suffix handling the methods start with an uppercase character.

14.8 Remark

The features described here are currently still experimental but the interface will not change. There might be a few more accessors and for sure more Lua helpers will be provided. As usual I need some time to play with it before I make up my mind. It is also possible to optimize the MetaPost–Lua script call a bit, but I might do that later.

When we played with this interface we ran into problems with loop variables and macro arguments. These are internally kind of anonymous. Take this:

```
for i=1 upto 100 : draw(i,i) endfor ;
```

The `i` is not really a variable with name `i` but becomes an object (capsule) when the condition is scanned, and a reference to that object when the body is scanned. The body of the for loop gets expanded for each step, but at that time there is no longer a variable `i`. The same is true for variables in:

```
def foo(expr x, y, delta) = draw (x+delta,y+delta) enddef ;
```

We are still trying to get this right with the LUA interface. Interesting is that when we were exploring this, we ran into quite some cases where we could make METAPOST abort due some memory or stack overflow. Some are just bugs in the new code (due to the new number model) while others come with the design of the system: border cases that never seem to happen in interactive use while the library use assumes no interaction in case of errors.

In CONTEXT there are more features and helpers than shown here but these are discussed in the METAFUN manual.

15 LUATEX 0.79

15.1 Introduction

To some it might look as if not much has been done in LuaTEX development but this is not true. First of all, the 2013 versions (0.75-0.77) are quite stable and can be used for production so there is not much buzz about new things. ConTEXt users normally won't even notice changes because most is encapsulated in functionality that itself won't change. The binaries on the `contextgarden.net` are always the latest so an update results in binaries that are in sync with the Lua and TEX code. Okay, behaviour might become better but that could also be the side effect of better coding. Of course some more fundamental changes can result in temporary bugs but those are normally easy to solve.

Here I will only mention the most important work done. I'll leave out the technical details as they can be found in the manual and in articles that were written during development. The version discussed is 0.79.

15.2 Speed

One of the things we spent a lot of time on is speed. This is of course of more importance for a system like ConTEXt that can spend more than half its time in Lua, but eventually we all benefit from it. For the average user it doesn't matter much if a run takes a few seconds but in automated workflows these accumulate and if a process has to produce 5 documents of 20 pages (each demanding a few runs) or a few documents of several hundreds of pages, it might make a difference. In the CritEd project we aim for complex documents produced from XML at a rate of 20 pages per second, at least for stock LuaTEX.[32] In an edit-preview cycle it feels better if we don't use more than half a second for a couple of pages: loading the TEX format, initializing the Lua modules, loading fonts, typesetting and producing a proper PDF file. We also want to be prepared for the ultra portable computers where multiple cores compensate the lower frequency, which harms TEX as sequential processor using one core only.

An important aspect of speedup is that it must not obscure the code. This is why the easiest way to achieve it is to use a faster variant of Lua, and LuaJIT with its faster virtual machine, is a solution for that. We are aware of the fact that processors not necessarily become faster, but that on the other hand memory becomes larger. Disk speed also got better with the arrival of flash based storage. Because LuaTEX should run smoothly on future portable devices, the more we can gain now, the better it gets in the future. A decent basic performance is possible and we don't have to focus too much on memory and disk access and mostly need to keep an eye on basic CPU cycles. Although we have some ideas about improving performance, tests demonstrate that LuaTEX is not doing

---

[32] This might look slow but a lot is happening there. A simple 100 page document with one word per page processes at more that 500 pages per second but this is hard to match with more realistic documents. When processing data from bases using the CLD interface getting 50 pages per seconds is no problem.

that bad and we don't have to change it's internals. In fact, if we do it might as well result in a drastic slowdown!

One interesting performance factor is console output. Because TEX outputs immediately with hardly any buffering, it depends a lot on the speed of console output. This itself depends on what console is used. UNIX consoles normally have some buffering and refresh delay built in. There the speed depends on what fonts are used and to what extend the output gets interpreted (escape sequences are an example). I've run into cases where a run took seconds more because of a bad choice of fonts. On WINDOWS it's more complicated since there the standard console (like TEX) is unbuffered. The good news is that there are several alternatives that perform quite well, like console2 and conemu. These alternatives buffer output and have refresh delays. But still, on a very high res screen, with a large console window logging has impact. Interesting is that when I run from the editor (SciTE) output is pretty fast, so normally I never notice much of a slowdown. Of course these kind of performance issues can hit you more when you work in a remote terminal.

The reason why I mention this is that in order to provide a user feedback about issues, there has to be some logging and depending on the kind of use, more or less is needed. This means that on the CONTEXT mailing list we sometimes get complaints about the amount of logging. It is for this reason that much logging is optional and all logging can be disabled as well. Because we go through LUA we have some control over efficiency too. In the current LUATEX release most logging can now be intercepted, including error messages.

Talking of a slowdown, in the CRITED project we have to deal with real large indices (tens of thousands of entries) and we found out that in the case of interactive variants (register entry to text and back) the use of LUAJITTEX could bring down a run to a grinding halt. In the end, after much testing we figured out that a suboptimal string hashing function was the culprit and we did extensive tests with both the LUAJIT, LUA 5.1 and LUA 5.2 variant. We ended up by replacing the LUAJIT hash function by the the LUA 5.1 one which is a relative easy operation. Because LUAJIT can address less memory than regular LUA it will always be a matter of testing if LUAJITTEX can be used instead of LUATEX. Standard document processing (reports and such) is normally no problem but processing large amounts of data from databases can be an issue.

In the process of cleaning up the code base for sure we will also find ways to make things run even smoother. But, in any case, version 0.80 is already a good benchmark for what can be achieved.

15.3 Nodes

One of the bottlenecks in the hybrid approach is crossing the so called C boundary. This is not really a bottleneck, unless we're talking of many millions of function calls. In practice this only happens in for instance more extreme font handling (Devanagari or sometimes Arabic). If performance is really an issue one can fallback on a more direct node access model. Of course the overhead of access should be compared to other related activities: one can gain .25 seconds on a run in using the direct access model, but if the whole runs takes 25 seconds, it can be neglected. If the price paid for it is less readable code it should only be done deep down a macro package where no user even sees the code. We use this access model in the ConTEXt core modules and so far it looks quite okay, especially for more extensive manipulations. The gain in speed is quite noticeable if you use the more advanced features of ConTEXt.

There can be some changes in the node model but not that drastic as the current model is quite ok and also stays close to original TEX so that existing documentation still applies. One of the changes will be that glue spec (sub)nodes will disappear and glue nodes will carry that information. Direction whatsits will become first class nodes as they are part of the concept (whatsits normally relate to extensions) and the same might happen with image nodes. As a side effect we can restructure the code so that it becomes more readable. Some experimental PDFTEX functionality will be removed as it can be done better with callbacks.

15.4 The parbuilder and HZ

As we started from PDFTEX we inherit also its experimental code and character. One of the objectives is to separate font- and backend as good as possible. We have already achieved a lot and apart from bringing consistency in the code, the biggest change has been a partial rewrite of the hz code, especially the way fonts are managed. Instead of making copies of fonts with different properties, we now carry information in the relevant nodes. The backend code already got away from multiple fonts by using transformation of the base font instead of additional font instances, so this was a natural adaptation. This was actually triggered by the fact that a LUA based par builder demonstrated that this made sense. The new approach uses less memory and is a bit faster (at least in theory).

In callbacks it makes life easier when a node list has a predictable structure. For instance, the result of a paragraph broken into lines still has discretionary nodes. Is that really needed? Lines can have left- or rightskip nodes, depending on the fact if they were set. Math nodes can disappear as part of a cleanup in the line break code, but this is unfortunate when one expects them to be somewhere in the list in a callback. All this will be made consistent. These are issues we will look into on the way to version 1.0.

I occasionally play with the LUA based par builder and it is quite compatible even if we take the floating point LUA aspect into account. However when using hz the outcome is different: sometimes better, sometimes worse. Personally I don’t care too much as long as it’s consistent. Features like hz are for special cases anyway and can never be stable over years if only because fonts evolve. And we’re talking of bordercase typesetting: narrow columns that no matter what method is used will never look okay.[33]

15.5 The backend

The separation of front- and backend is more a pet project. There is some experimental code that will get removed or integrated. We try to make the backend consistent from

---

[33] Some people don’t like larger spaces, others don’t like stretched glyphs.

the TEX as well as LUA end and some is reflected in additional features and callbacks.

Some of the variables that can be set (the LUA counterparts of the `\pdf..` token registers at the TEX end) are now consistent with each other and avoid going via pseudo tokenization. Typical aspects of a backend that only a few users will notice but nevertheless needed work.

The merge of engines also resulted in inconsistencies in function names, like using `pdf_` in function names where nothing `PDF` is involved.

15.6 Backlinks

In callbacks we mostly deal with node lists. At the TEX end of course we also have these lists but there it is quite clear what gets done with them. This means that there is no need for double linked lists. It also means that what is known as the head of a list can in fact be in the middle. The for TEX characteristic nesting model has resulted in stacks and current pointers. The code uses so called temp nodes to point at the head node.

As a consequence in LUATEX, where we present a double linked list, before the current version one could run into cases where for instance a head node had a prev pointer, even one that made no sense. As said, no big deal in TEX but in the hands of a user who manipulates the node list it can be dramatic. The current version has cleaned head nodes as well as consistent backlinks, but of course we keep the internals mostly unchanged because we stay close to the Knuthian original when possible.[34]

15.7 Properties

Sometimes you want to associate additional information to a node. A natural way to do this is attributes. These can be set at the TEX and LUA end and accessed at the LUA end. At the LUA end one can have tables with nodes as indices and store extra information but that has the disadvantage that one has no clue if such information is current: nodes come and go and are recycled.

For this reason we now have a global properties table where each allocated node can have a table with whatever information users might like to store. This itself is not special, but the nice thing is that when a node is freed, that information is also freed. So, you cannot run into old data. When nodes are copied its properties are also copied. The overhead, when not used, is close to zero, which is always an objective when extending the core engine.

Of course this model demands that macro package somehow controls consistent use but that is not different from what already has to be done. Also, simple extensions like this avoid hard codes solutions, which is also something we want to avoid.

---

[34] Even with extensions the original documentation still covers most of what happens.

15.8 LUA calls

We have so called user nodes that can carry a number, string, token list or node list. We now have added Lua to this repertoire. In fact, we now could use only a Lua variable and we might have done so in retrospect, but for the moment we we stick to the current model of several basic types. The Lua variable can be anything and it is up to the user (in some callback) to deal with them.

User nodes are not to be confused with late Lua nodes. You can store a function call in a user node but that's about it. You can at a later moment decide to call that function but it's still an explicit action. The value of a late Lua node on the other hand is dealt with automatically during shipout. When the value is a string it gets interpreted as Lua, but new is that when the value is a function it will get called. At that moment we have access to some of the current backend properties, like locations.

15.9 Artefacts

Because LuaTEX took code from pdfTEX, that is built upon ε-TEX, which in turn is an extension to TEX, and Omega, that also extends TEX, there is code that no longer makes sense for us. Combine that with the fact that some code carries signatures of translated Pascal to C, we have some cleanup to do as follow up on the not to be underestimated move to C. This is an ongoing process but also fun doing. Luigi and I spend many hours exploring venues and have interesting Skype sessions that can easily sidetrack, and with Taco getting more time for LuaTEX we expect to get most of our (still growing) todo list done.

Because LuaTEX started out as an experiment, there is some old code around. For instance, we used to have multiple instances and this still shows in some places. We can simplify the Lua to TEX interface a bit and clean up the Lua global state handling, but we're not in a big hurry with this. Experiments have been done with some extensions to the writer code but they are hold back to after the cleanup.

In a similar fashion we have sped up the way Lua keyword and values get resolved. Already early in the development we did this for critical code like passing Lua font tables to TEX, followed by accessing nodes, but now we have done that for most code. There is still some to do but it has the side effect of not only consistency but also of helping to document the interface. Of course we learn a lot about the Lua internals too. The C macro system is of great help here, although the mentioned pascal conversion (web2c) and merged engines have resulted in some inconsistency that needs to be cleaned up before we start documenting more of the internals (another subproject we want to finish before retirement).

15.10 Callbacks

There are a few more callbacks and most of them come from the tracker. The backend now has page related callbacks, the Lua error handler can be intercepted. Error messages that consist of multiple pieces are handled better too. When a file is opened and closed a callback is now possible. Technically we could have combined this with the already present callbacks but as in TEX synchronization matters these new callbacks relate to current message callbacks that show `[]`, `{}`, `<>` and/or `<<>>` fenced filenames, where the later were introduced in successive backend code.

15.11 LUA

We currently use LUA 5.2 but a next version will show up soon. Because LUA 5.3 introduces a hybrid number model, this will be one of the next things to play with. It could work out well, because TEX is internally integer based (scaled points) but you never know. It could be that we need to check existing code for serialization and printing issues but normally that will not lead to compatibility issues. We could even decide to stick to LUA 5.2 or at least wait till all has stabilized. There is some basic support for UTF in 5.3 but in CONTEXT we don't depend on that. In practice hardly any processing takes place that assumes that UTF is more than a sequence of bytes and LUA can handle bytes quite well.

15.12 CONTEXT

Of course the development of LUATEX has consequences for CONTEXT. For instance, existing code is used to test alternative solutions and sometimes these make it into the core. Some new features are used immediately, like the more consistent control over PDF properties, but others have to wait till the new binary is more widespread.[35]

Some of the improvement in the code base directly relate to CONTEXT activities. For instance the CRITED project (complex critical editions) uncovered some hashing issues with LUAJIT that have been taken care of now. The (small) additions to the PDF backend resulted in a partial cleanup of relatively old CONTEXT backend code.

Although some more complex mechanisms, like multi-columns are being reworked, it is still needed to open up a bit more of the TEX internals, so we have some work to do. As usual, version 0.80 doesn't mean that only 0.20 has to be done to get to 1.00, as development is not a linear process. The jump from 0.77 to 0.79 for instance involved a lot of work (exploration as well as testing). But as long as it's fun to do, time doesn't matter much. As we've said before: we're in no hurry.

---

[35] Normally dissemination is rather fast because the contextgarden provides recent binaries. The new windows binaries often show up within hours after the repository has been updated.