EVEN MORE

fun with luametatex and context

Table of contents

1 Introduction

After five collections of 'articles' about the development of LuaTEX, ConTEXt MkIV, LuaMetaTEX and ConTEXt lmtx, there is even more to tell so here is number six. Wrapping up not only serves to inform the users but for me it is also a way to get things right: if you cannot write it down it's no good. It forces me to (re)consider interfaces and also test new code but of course it comes with no guarantees.

When writing this introduction I just finished the first chapter, about some new font stuff, as follow up on the (again) nice ConTEXt meeting in 2019. It's always inspiring to meet and talk with my TEX friends and see what they're doing. It keeps me going.

Some chapters end up in user group journals first so they will be added once they have been published and are available. The advantage is that these are then copy-edited. Many texts, also in previous development updates, got better because Karl Berry checked them thoroughly for TUGboat, for which I'm grateful.

Hopefully, this document serves a purpose.

Hans Hagen
PRAGMA ADE, Hasselt NL
Started in October 2019

2 TEX and Pi

This is a short status report[1] on Pi, not the famous version number of TEX (among other things), but the small machine, meant for education but nowadays also used for Internet Of Things projects, process control and toy projects. While the majority of TEX installations run on an Intel processor, the Raspberry Pi has an arm central processing unit. In fact, its main chip has the same foundation as those found in settop boxes all around the world. It's made for entertainment, not for number crunching.

At the ConTEXt meetings, it has become tradition to play with electronic gadgets. Every year we are curious what Harald König might bring this time. The last couple of meetings we also had talks about using TEX and MetaPost for designing (home-scale, automated) railroad systems, using LuaTEX for running domotica applications, using MetaPost for rendering high quality graphics from data from appliances, presenting TEX at computer and electronics bootcamps, and more. This year Frans Goddijn also brought back memories of low speed modem sounds, from the early days of TEX support. It is these things that make the meetings fun.

This year the meeting was in Belgium, close to the border of the Netherlands, and on the way there Mojca Miklavec traveled via my home, where the contextgarden compile farm runs on a server with plenty of cores, lots of memory and big disks. But the farm also has an old Mac connected as well as a tiny underpowered Raspberry Pi 2 for arm binaries that we had to fix: the small micro ssd card in it had finally given up. This is no surprise if you realize that it does a daily compilation of the whole TEX Live setup and also compiles LuaTEX, LuaMetaTEX and pplib when changes occur. Replacing the card worked out but nevertheless we decided to take the small machine with us to the meeting. We also took an external (2.5 inch) ssd box with us. The idea was to order a Raspberry Pi 4 on location, the much praised successor of the older models, the one with 4 GB of memory, real usb 3 ports and proper Ethernet.

At the meeting Harald showed us that he had version 1, 3 and 4 machines with him because he was looking into an energy control setup based on Zigbee devices. So we had the full range of Pi's there to play with.

This is a long introduction but the message is that we are dealing with a small but popular device with up to now four generations, using an architecture supported in TEX distributions. So how does that relate to ConTEXt? One of the reasons for LuaMetaTEX going lean and mean is that computers are no longer getting much faster and 'multiple small' energy-wise has more appeal than 'one large'. So then the question is: how can we make TEX run fast on small instead of gambling on big becoming even bigger (which does not seem to be happening anyway).

At the meeting Harald gave a talk "Which Raspberry Pi is the best for ConTEXt?" and I will use his data to give an overview: see Table ??rpispec.

model	**1**	**2**	**3**	**4**
chipset	BCM2835	BCM2835	BCM2835	BCM2835
CPU core	v6l rev 7	v7l rev 5	v7l rev 4	v7l rev 3
cores	1	4	4	4
free mem	443080	948308	948304	3999784
idlemips	997.08	38.40	38.40	108.00
bogomips	997.08	57.60	76.80	270.00
read SD	23.0 MB/s	23.2 MB/s	23.2 MB/s	45.1 MB/s
read USB		30.0 MB/s	30.0 MB/s	320.0 MB/s

[1] This chapter appeared in TugBoat 40:3. Thanks to Karl Berry for corrections.

After some discussion at the presentation we decided to discard the (absurd) bogomips value for the tiny Pi 1 computing board and not take the values for the others too seriously. But it will be clear that, especially when we consider the external drive that things have improved. The table doesn't mention Ethernet speed but because the 4 now has real support for it (instead of sharing the usb bus) we get close to 1 GB/s there.

The real performance test is of course processing a TEX document and what better to test than the TEX book. The processing time in seconds, after initial caching of files and fonts is:

model	**1**	**2**	**3**	**4**
TEXbook	13.649	7.023	4.553	1.694
`context --make`		19.949	11.796	6.034
`context --make TL`	89.454	46.578	29.256	14.146

The test of making the ConTEXt format using LuaTEX gives an indication of how well the io performs: it loads the file database, some 460 Lua modules and 355 TEX source files. On my laptop with Intel i7-3840QM with 16GB memory and decent ssd it takes 3.5 seconds (and 1 second less for LuaMetaTEX because there we don't compress the format file). Somehow a regular TEXLive installation performs much worse than the one from the contextgarden.

We didn't test real ConTEXt documents at the meeting but when I came home the Pi 4 was bound again to the compile farm. Harald and Mojca had prepared the machine to boot from the internal micro ssd and use the external disk for the rest. So, when we could compile LuaMetaTEX again, I made an arm installer for lmtx, and after that could not resist doing a simple test. First of course came generating the format. It took 6.3 seconds to make one, which is a bit more than Harald measured. I see a hiccup at the end so I guess that it has to do with the (external) disk or maybe there is some throttling going on because the machine sits on top of a (warm) server.

More interesting was testing a real document: the upcoming LuaMetaTEX manual. It has 226 pages, uses 21 font files, processes 225 MetaPost graphics, and in order to get it LuaMetaTEX does more than 50% of the work in Lua, including all font and backend-related operations. On my laptop it needs 9.5 seconds and on the Pi 4 it uses 33 seconds. Of course, if I take a more modern machine than this 8-year-old workhorse, I probably need half the time, but still the performance of the Raspberry Pi 4 is quite impressive. It uses hardly any energy and can probably compete rather well with a virtual machine on a heavily loaded machine. It means that when we ever have to upgrade the server, I can consider replacement by an Ethernet switch, with power over Ethernet, connected to a bunch of small Raspberries, also because normally one would connect to some shared storage medium.

Because I was curious how the dedicated small Fitlet that I use for controlling my lights and heating performs I also processed the manual there. After making the format, which takes 6 seconds, processing the manual took a little less than 30 seconds. In that respect it performs the same as a Raspberry Pi 4. But, inside that small (way more expensive) computer is an dual core AMD A10 Micro-6700T APU (with AMD Radeon R6 Graphics), running a recent 64-bit Ubuntu. It does some 2400 bogomips (compare that to the values of the Pi). I was a bit surprised that it didn't outperform the Raspberry because the (fast ssd) disk is connected to the main board and it has more memory and horsepower. It might be that in the end an arm processor is simply better suited for the kind of byte juggling that TEX does, where special cpu features and multiple cores don't contribute much. It definitely demonstrates that we cannot neglect this platform.

3 Modern Type 3 fonts

Support for Type3 fonts has been on my agenda for a couple of years now. Here I will take a look at them from the perspective of LuaMetaTEX.[2] The reason is that they might be useful for embedding (for instance) runtime graphics (such as symbols) in an efficient way. In TEX systems Type3 fonts are normally used for bitmap fonts, the pk output that comes via METAFONT. Where for instance Type1 fonts are defined using a set of font specific rendering operators, a Type3 font can contain arbitrary code, in pdf files these are pdf (graphic and text) operators.

A program like LuaTEX supports embedding of several font formats natively. A quick summary of relevant formats is the following:[3]

- **Type1:** these are outline fonts using `cff` descriptions, a compact format for storing outlines. Normally up to 256 characters are accessible but a font can have many more (as Latin Modern and TEX Gyre demonstrate).
- **OpenType:** these also use the `cff` format. As with Type1 the outlines are mostly cubic Bezier curves. Because there is no bounding box data stored in the format the engine has to pseudo-render the glyphs to get that information. When embedding a subset the backend code has to flatten the subroutine calls, which is another reason the `cff` blob has to be disassembled.
- **TrueType:** these use the `ttf` format which uses quadratic B-splines. The font can have a separate kerning table and stores information about the bounding box (which is then used by TEX to get the right heights and depths of glyphs). Of course those details never make it into the pdf file as such.
- **Type3:** as mentioned this format is (traditionally) used to store bitmap fonts but as we will see it can do more. It is actually the easiest format to deal with.

In LuaTEX any font can be a "wide" font, therefore in ConTEXt a Type1 font is not treated differently than an OpenType font. The LuaTEX backend can even disguise a Type1 font as an OpenType font. In the end, as not that much information ends up in the pdf file, the differences are not that large for the first three types. The content of a Type3 font is less predictable but even then it can have for instance a `ToUnicode` vector so it has no real disadvantages in, say, accessibility. In ConTEXt lmtx, which uses LuaMetaTEX without any backend, all is dealt with in Lua: loading, tweaking, applying and embedding.

The difference between OpenType and TrueType is mostly in the kind of curves and specific data tables. Both formats are nowadays covered by the OpenType specification. If you Google for the difference between these formats you can easily end up with rather bad (or even nonsense) descriptions. The best references are `https://en.wikipedia.org/wiki/Bézier_curve` and the ever-improving `https://docs.microsoft.com/en-us/typography` website.

Support for so-called variable fonts is mostly demanding of the front-end because in the backend it is just an instance of an OpenType or TrueType font being embedded. In this case the instance is generated by the ConTEXt font machinery which interprets the `cff` and `ttf` binary formats in doing so. This feature is not widely used but has been present from the moment these fonts showed up.

Type3 fonts don't have a particularly good reputation, which is mainly due to the fact that viewers pay less attention in displaying them, at least that was the case in the past. If they describe outlines, then

[2] This chapter appeared in TugBoat 40:3. Thanks to Karl Berry for corrections.
[3] Technically one can embed anything in the pdf file.

all is okay, apart from the fact that there is no anti-aliasing or hinting but on modern computers that is hardly an issue. For bitmaps the quality depends on the resolution and traditionally TEX bitmap fonts are generated for a specific device, but if you use a decent resolution (say 1200 dpi) then all should be okay. The main drawback is that viewers will render such a font and cache the (then available) bitmap which in some cases can have a speed penalty.

Using Type3 fonts in a pdf backend is not something new. Already in the pdfTEX era we were playing with so-called pdf glyph containers. In practice that worked okay but not so much for MetaPost output from METAFONT fonts. As a side note: it might actually work better now that in MetaFun we have some extensions for rendering the kind of paths used in fonts. But glyph containers were dropped long ago already and Type3 was limited to traditional TEX bitmap inclusion. However, in LuaMetaTEX it is easier to mess around with fonts because we no longer need to worry about side effects of patching font related inclusion (embedding) for other macro packages. All is now under Lua control: there is no backend included and therefore no awareness of something built-in as Type3.

So, as a prelude to the 2019 ConTEXt meeting, I picked up this thread and turned some earlier experiments into production code. Originally I meant to provide support for MetaPost graphics but that is still locked in experiments. I do have an idea for its interface, now that we have more control over user interfaces in MetaFun.

In addition to 'just graphics' there is another candidate for Type3 fonts — extensions to OpenType fonts:

1. Color fonts where stacked glyphs are used (a nice method).
2. Fonts where svg images are used.
3. Fonts that come with bitmap representations in png format.

It will be no surprise that we're talking of emoji fonts here although the second category is now also used for regular text fonts. When these fonts showed up support for them was not that hard to implement and (as often) we could make TEX be among the first to support them in print (often such fonts are meant for the web).

For category one, the stacked shapes, the approach was to define a virtual font where glyphs are flushed while backtracking over the width in order to get the overlay. Of course color directives have to be injected too. The whole lot is wrapped in a container that tells a pdf handler what character actually is represented. Due to the way virtual fonts work, every reference to a character results in the same sequence of glyph references, (negative) kern operations and color directives plus the wrapper in the page stream. This is not really an issue for emoji because these are seldom used and even then in small quantities. But it can explode a pdf page stream for a color text font. All happens at runtime and because we use virtual fonts, the commands are assembled beforehand for each glyph.

For the second category, svg images, we used a different approach. Each symbol was converted to pdf using Inkscape and cached for later use. Instead of injecting a glyph reference, a reference to a so-called `XForm` is injected, again with a wrapper to indicate what character we deal with. Here the overhead is not that large but still present as we need the so-called 'actual text' wrapper.

The third category is done in a similar way but this time we use GraphicsMagick to convert the images beforehand. The drawbacks are the same.

In ConTEXt lmtx a different approach is followed. The pdf stream that stacks the glyphs of category one makes a perfect stream for a Type3 character. Apart from some juggling to relate a Type3 font to an OpenType font, the page stream just contains references to glyphs (with the proper related Unicode slot). The overhead is minimal.

For the second category ConTEXt lmtx uses its built-in svg converter. The xml code of the shape is converted to (surprise): MetaPost. We could go directly to pdf but the MetaPost route is cheap and we can then get support for color spaces, transformations, efficient paths and high quality all for free. It also opens up the possibility for future manipulations. The Type3 font eventually has a sequence of drawing operations, mixed with transformations and color switches, but only once. Most of the embedded code is shared with the other categories (a plug-in model is used).

The third category follows a similar route but this time we use the built-in png inclusion code. Just like the other categories, the page stream only contains references to glyphs.

It was interesting to find that most of the time related to the inclusion went into figuring out why viewers don't like these fonts. For instance, in Acrobat there needs to be a glyph at index zero and all viewers seem to be able to handle at most 255 additional characters in a font. But once that, and a few more tricks, had become clear, it worked out quite well. It also helps to set the font bounding box to all zero values so that no rendering optimizations kick in. Also, some dimensions can are best used consistently. With svg there were some issues with reference points and bounding boxes but these could be dealt with. A later implementation followed a slightly different route anyway.

The implementation is reasonably efficient because most work is delayed till a glyph (shape) is actually injected (and most shapes in these fonts aren't used at all). The viewers that I have installed, Acrobat Reader, Acrobat X, and the mupdf-based Sumatrapdf viewer can all handle the current implementation.

An example of a category one font is Microsoft's `seguiemj`. I have no clue about the result in the future because some of these emoji fonts change every now and then, depending also on social developments. This is a category one font which not only has emoji symbols but also normal glyphs:

```
\definefontfeature[colored][default][colr=yes]
\definefont[TestA][file:seguiemj.ttf*colored]
\definesymbol[bug1][\getglyphdirect{file:seguiemj.ttf*colored} {\char"1F41C}]
\definesymbol[bug2][\getglyphdirect{file:seguiemj.ttf*colored} {\char"1F41B}]
```

The example below demonstrates this by showing the graphic glyph surrounded by the `x` from the emoji font, and from a regular text font.

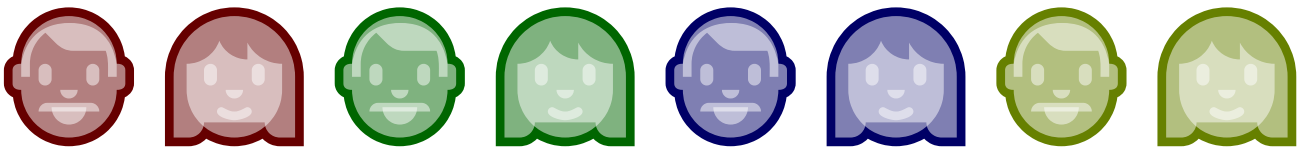
```
{\TestA x\char"1F41C x\char"1F41B x}%
\quad
{x\symbol[bug1]x\symbol[bug2]x}%
\quad
{\showglyphs x\symbol[bug1]x\symbol[bug2]x}%
```

In this mix we don't use a Type3 font for the characters that don't need stacked (colorful) glyphs, which is more efficient. So the `x` characters are references to a regular (embedded) OpenType font.

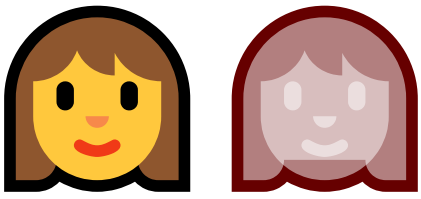
The next example comes from a manual and demonstrates that we can (still) manipulate colors as we wish.

```
\definecolor[emoji-red]   [r=.4]
\definecolor[emoji-blue]  [b=.4]
\definecolor[emoji-green] [g=.4]
```

```
\definecolor[emoji-yellow][r=.4,g=.5]
\definecolor[emoji-gray]  [s=1,t=.5,a=1]

\definefontcolorpalette
  [emoji-red]
  [emoji-red,emoji-gray]

\definefontcolorpalette
  [emoji-green]
  [emoji-green,emoji-gray]

\definefontcolorpalette
  [emoji-blue]
  [emoji-blue,emoji-gray]

\definefontcolorpalette
  [emoji-yellow]
  [emoji-yellow,emoji-gray]

\definefontfeature[seguiemj-r][default][ccmp=yes,dist=yes,colr=emoji-red]
\definefontfeature[seguiemj-g][default][ccmp=yes,dist=yes,colr=emoji-green]
\definefontfeature[seguiemj-b][default][ccmp=yes,dist=yes,colr=emoji-blue]
\definefontfeature[seguiemj-y][default][ccmp=yes,dist=yes,colr=emoji-yellow]

\definefont[MyColoredEmojiR][seguiemj*seguiemj-r]
\definefont[MyColoredEmojiG][seguiemj*seguiemj-g]
\definefont[MyColoredEmojiB][seguiemj*seguiemj-b]
\definefont[MyColoredEmojiY][seguiemj*seguiemj-y]
```



Let's look in more detail at the woman emoji. On the left we see the default colors, and on the right we see our own:



The multi-color variant in ConTEXt MkIV ends up as follows in the page stream:

```
/Span << /ActualText <feffD83DDC69> >> BDC
q
0.000 g
BT
/F8 11.955168 Tf
1 0 0 1 0 2.51596 Tm [<045B>]TJ
0.557 0.337 0.180 rg
1 0 0 1 0 2.51596 Tm [<045C>]TJ
```

```
1.000 0.784 0.239 rg
1 0 0 1 0 2.51596 Tm [<045D>]TJ
0.000 g
1 0 0 1 0 2.51596 Tm [<045E>]TJ
0.969 0.537 0.290 rg
1 0 0 1 0 2.51596 Tm [<045F>]TJ
0.941 0.227 0.090 rg
1 0 0 1 0 2.51596 Tm [<0460>]TJ
ET
Q
EMC
```

and the reddish one becomes:

```
/Span << /ActualText <feffD83DDC69> >> BDC
q
0.400 0 0 rg 0.400 0 0 RG
BT
/F8 11.955168 Tf
1 0 0 1 0 2.51596 Tm [<045B>]TJ
1 g 1 G /Tr1 gs
1 0 0 1 0 2.51596 Tm [<045C>1373<045D>1373<045E>1373<045F>1373<0460>]TJ
ET
Q
EMC
```

Each time this shape is typeset these sequences are injected. In ConTEXt lmtx we get this in the page stream:

```
BT
/F2 11.955168 Tf
1 0 0 1 0 2.515956 Tm [<C8>] TJ
ET
```

This time the composed shape ends up in the Type3 character procedure. In the colorful case (reformatted because it actually is a one-liner):

```
2812 0 d0
0.000 g              BT /V8 1 Tf [<045B>] TJ ET
0.557 0.337 0.180 rg BT /V8 1 Tf [<045C>] TJ ET
1.000 0.784 0.239 rg BT /V8 1 Tf [<045D>] TJ ET
0.000 g              BT /V8 1 Tf [<045E>] TJ ET
0.969 0.537 0.290 rg BT /V8 1 Tf [<045F>] TJ ET
0.941 0.227 0.090 rg BT /V8 1 Tf [<0460>] TJ ET
```

and in the reddish case (where we have a gray transparent color):

```
2812 0 d0
0.400 0 0 rg 0.400 0 0 RG
BT /V8 1 Tf [<045B>] TJ ET
1 g 1 G /Tr1 gs
BT /V8 1 Tf [<045C>] TJ ET
```

```
BT /V8 1 Tf [<045D>] TJ ET
BT /V8 1 Tf [<045E>] TJ ET
BT /V8 1 Tf [<045F>] TJ ET
BT /V8 1 Tf [<0460>] TJ ET
```

but this time we only get these verbose compositions once in the pdf file. We could optimize the last variant by a sequence of indices and negative kerns but it hardly pays off. There is no need for `ActualText` here because we have an entry in the `ToUnicode` vector:

```
<C8> <D83DDC69>
```

To be clear, the `/V8` is a sort of local reference to a font which can have an `/F8` counterpart elsewhere. These Type3 fonts have their own resource references and I found it more clear to use a different prefix in that case. If we only use a few characters of this kind, of course the overhead of extra fonts has a penalty but as soon we refer to more characters we gain a lot.

When we have svg fonts, the gain is a bit less. The pdf resulting from the MetaPost run can of course be large but they are included only once. In MkIV these would be (shared) so-called `XForm`s. In the page stream we then see a simple reference to such an `XForm` but again wrapped into an `ActualText`. In lmtx we get just a reference to a Type3 character plus of course an extra font.

The `emojionecolor-svginot` font is somewhat problematic (although maybe in the meantime it has become obsolete). As usual with new functionality, specifications are not always available or complete (especially when they are application specs turned into standards). This is also true with for instance svg fonts. The current documentation on the Microsoft website is reasonable and explains how to deal with the viewport but when I first implemented support for svg it was more trial and error. The original implementation in ConTEXt used Inkscape to generate pdf files with a tight bounding box and mix that with information from the font (in MkIV and the generic loader we still use this method). This results in a reasonable placement for emoji (that often sit on top of the baseline) but not for characters. More accurate treatment, using the origin and bounding box, fail for graphics with bad viewports and strange transform attributes. Now one can of course argue that I read the specs wrong, but inconsistencies are hard to deal with. Even worse is that successive versions of a font can demand different hacks. (I would not be surprised if some programs have built-in heuristics for some fonts that apply fixes.) Here is an example:

```
<svg transform="translate(0 -1788) scale(2.048)" viewBox="0 0 64 64" ...>
    <path d="... all within the viewBox ..." ... />
</svg>
```

It is no problem to scale up the image to normal dimensions, often 1000, or 2048 but I've also seen 512. The 2.048 suggests a 2048 unit approach, as does the 1788 shift. Now, we could scale up all dimensions by 1000/64 and then multiply by 2.048 and eventually shift over 1788, but why not scale the 1788 by 2.048 or scale 64 by 2.048? Even if we can read the standard to suit this specification it's just a bit too messy for my taste. In fact I tried all reasonable combinations and didn't (yet) get the right result. In that case it's easier to just discard the font. If a user complains that it (kind of) worked in the past, a counter-argument can be that other (more recent) fonts don't work otherwise. In the end we ended up with an option: when the `svg` feature value is `fixdepth` the vertical position will be fixed.

```
\definefontfeature[whatever][default][color=yes,svg=fixdepth]
\definefont[TestB][file:emojionecolor-svginot.ttf*whatever]
```

The newer `emojionecolor` font doesn't need this because it has a `viewBox` of `0 54.4 64 64` which fixes the baseline.

```
\definefontfeature[whatever][default][color=yes,svg=yes]
\definefont[TestB][file:emojionecolor.otf*whatever]
```

X  X

Another example of a pitfall is running into category one glyphs made from several pieces that all have the same color. If that color is black, one starts to wonder what is wrong. In the end the ConTEXt code was doing the right thing after all, and I would not be surprised if that glyph gets colors in an update of the font. For that reason it makes no sense to include them as examples here. After all, we can hardly complain about free fonts (and I'm also guilty of imposing bugs on users). By the way, for the font referred to here (`twemojimozilla`), another pitfall was that all shapes in my copy had a zero bounding box which means that although a width is specified, rendering in documents can give weird side effects. This can be corrected by the `dimensions` feature that forces the ascender and descender values to be used.

```
\definefontfeature[colored:x][default][colr=yes]
\definefontfeature[colored:y][default][colr=yes,dimensions={1,max,max}]
\definefont[TestC][file:twemojimozilla.ttf*colored:x]
\definefont[TestD][file:twemojimozilla.ttf*colored:y]
```



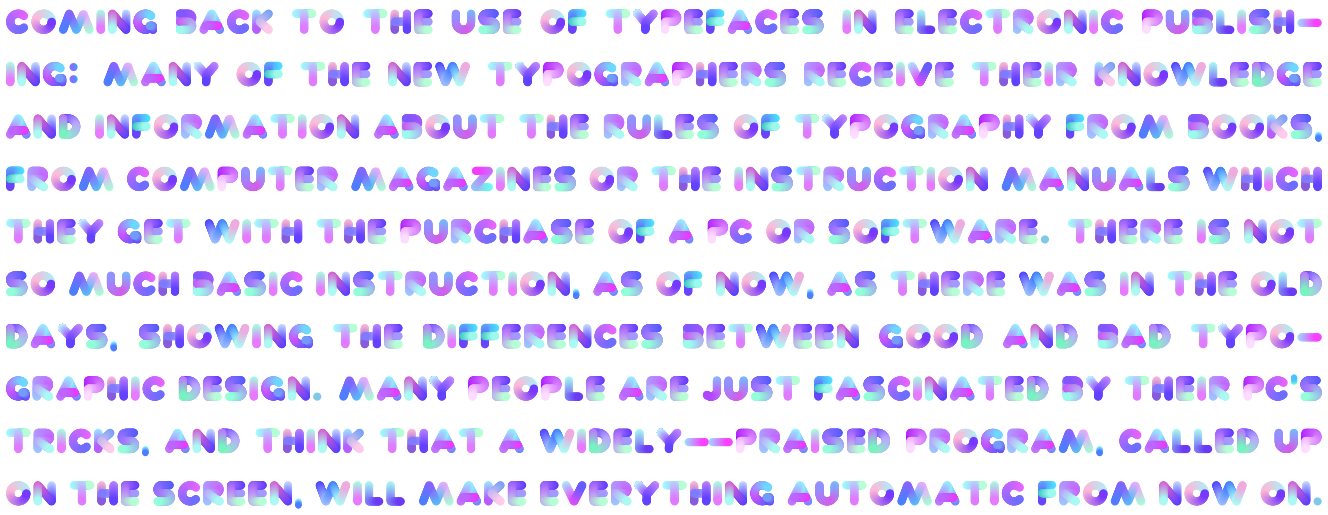

An example of a png-enhanced font is the `applecoloremoji` font. The bitmaps are relatively large for the quality they provide and some look like scans.

```
\definefontfeature[sbix][default][sbix=yes]
\definefont[TestE][file:applecoloremoji.ttc*sbix at 10bp]
```

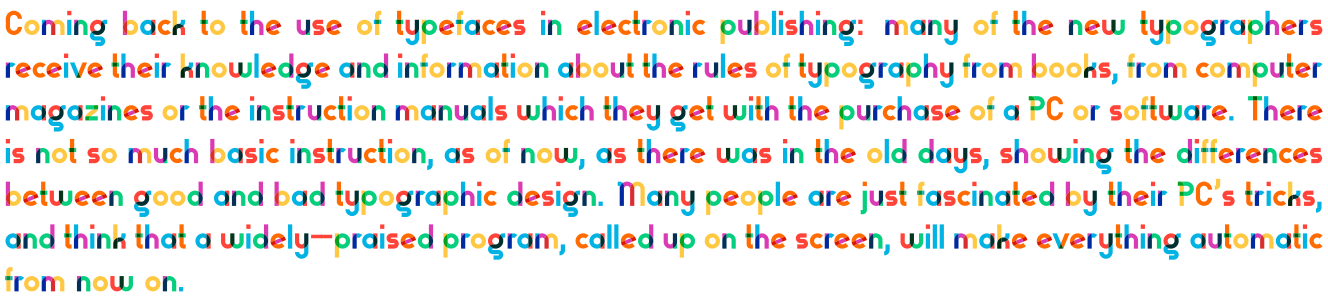
As mentioned above, there are fonts that color characters other than emoji. We give two examples (sometimes fonts are mentioned on the ConTEXt mailing list).

```
\definefontfeature
  [whatever]
  [default,color:svg]
  [script=latn,
   language=dflt]

\definefont[TestF][file:Abelone-FREE.otf*whatever          @ 13bp]
\definefont[TestG][file:Gilbert-ColorBoldPreview5*whatever @ 13bp]
\definefont[TestH][file:ColorTube-Regular*whatever         @ 13bp]
```

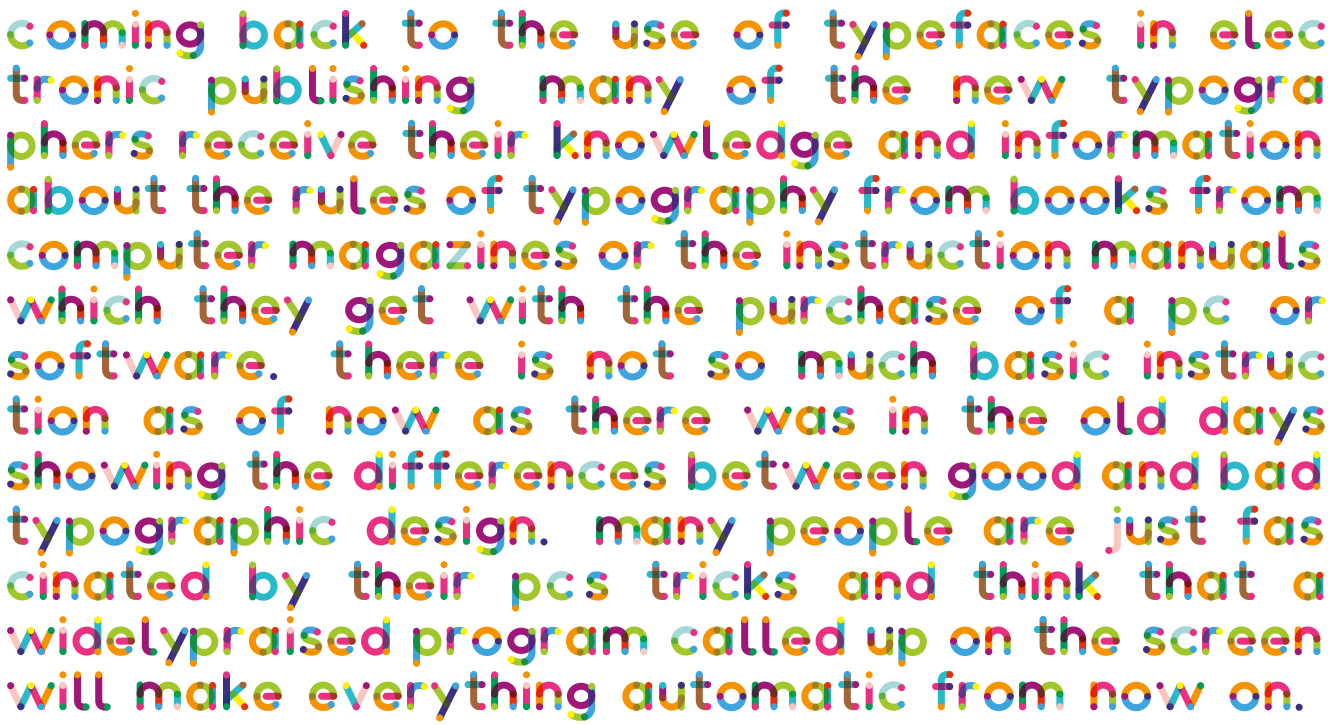
Of course one can wonder about the readability of these fonts and unless one used random colors (which bloats the file immensely) it might become boring, but typically such fonts are only meant for special purposes.



The previous font is the largest and as a consequence also puts some strain on the viewer, especially when zooming in. But, because viewers cache Type3 shapes it's a one-time penalty.



This font is rather lightweight. Contrary to what one might expect, there is no transparency used (but of course we do support that when needed).



This third example is again rather lightweight. Such fonts normally have a rather limited repertoire although there are some accented characters included. But they are not really meant for novels anyway. If you look closely you will also notice that some characters are missing and kerning is suboptimal.

I considered testing some more fonts but when trying to download some interesting looking ones I got a popup asking me for my email address in order to subscribe me to something: a definite no-go.

I already mentioned that we have a built-in converter that goes from svg to MetaPost. Although this article deals with fonts, the following code demonstrates that we can also access such graphics in MetaFun itself. The nice thing is that because we get pictures, they can be manipulated.

```
\startMPcode{doublefun}
    picture p ; p := lmt_svg [ filename = "mozilla-svg-001.svg" ] ;
    numeric w ; w := bbwidth(p) ;
    draw p ;
    draw p xscaled -1 shifted (2.5*w,0);
    draw p rotatedaround(center p,45) shifted (3.0*w,0) ;
    draw image (
        for i within p : if filled i :
            draw pathpart i withcolor green ;
        fi endfor ;
    ) shifted (4.5*w,0);
    draw image (
        for i within p : if filled i :
            fill pathpart i withcolor red withtransparency (1,.25) ;
        fi endfor ;
    ) shifted (6*w,0);
\stopMPcode
```

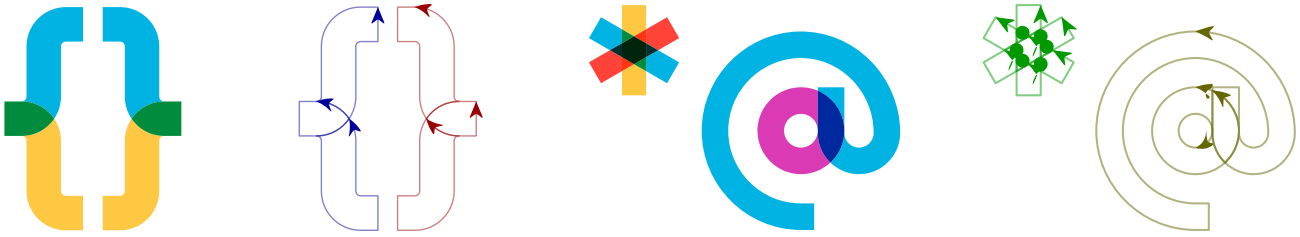
This graphic is a copy from a glyph from an emoji font, so we have plenty of such images to play with. The above manipulations result in:

Now that we're working with MetaPost we may as well see if we can also load a specific glyph, and indeed this is possible.

```
\startMPcode{doublefun}
    picture lb, rb ;
    lb := lmt_svg [ fontname = "Gilbert-ColorBoldPreview5", unicode = 123 ] ;
    rb := lmt_svg [ fontname = "Gilbert-ColorBoldPreview5", unicode = 125 ] ;
    numeric dx ; dx := 1.25 * bbwidth(lb) ;
    draw lb ;
    draw rb shifted (dx,0) ;
    pickup pencircle scaled 2mm ;
    for i within lb :
        draw lmt_arrow [
            path        = pathpart i,
            pen         = "auto",
            alternative = "curved",
            penscale    = 8
        ]
```

```
            shifted (3dx,0)
            withcolor "darkblue"
            withtransparency (1,.5)
        ;
    endfor ;
    for i within rb :
        draw lmt_arrow [
            path        = pathpart i,
            pen         = "auto",
            alternative = "curved",
            penscale    = 8
        ]
            shifted (4dx,0)
            withcolor "darkred"
            withtransparency (1,.5)
        ;
    endfor ;
\stopMPcode
```

Here we first load two character shapes from a font. The Unicode slots (which here are the same as the ascii slots) might look familiar: they indicate the curly brace characters. We get two pictures and use the `within` loop to run over the paths within these shapes. Each shape is made from three curves. As a bonus a few more characters are shown.



It is not hard to imagine that a collection of such graphics could be made into a font (at runtime). One only needs to find the motivation. Of course one can always use a font editor (or Inkscape) and tweak there, which probably makes more sense, but sometimes a bit of MetaPost hackery is a nice distraction. Editing the svg code directly is not that much fun. The overall structure often doesn't look that bad (apart from often rather redundant grouping):

```
<svg xmlns="http://www.w3.org/2000/svg">
    <path fill="#d87512" d="..."/>
    <g fill="#bc600d">
        <path d="..."/>
    </g>
    <g fill="#d87512">
        <path d="..."/>
        <path d="..."/>
    </g>
    <g fill="#bc600d">
        <path d="..."/>
    </g>
    ...
</svg>
```

In addition to `path`s there can be `line`, `circle` and similar elements but often fonts just use the `path` element. This element has a `d` attribute that holds a sequence of one character commands that each can be followed by numbers. Here are the start characters of four such attributes:

```
M11.585 43.742s.387 1.248.104 3.05c0 0 2.045-.466 1.898-2.27 ...
M53.33 39.37c0-4.484-35.622-4.484-35.622 0 0 10.16.05 ...
M42.645 56.04c1.688 2.02 9.275.043 10.504-2.28 5.01-9.482-.006 ...
M54.2 41.496s-.336 4.246-4.657 9.573c0 0 4.38-1.7 5.808-4.3 ...
```

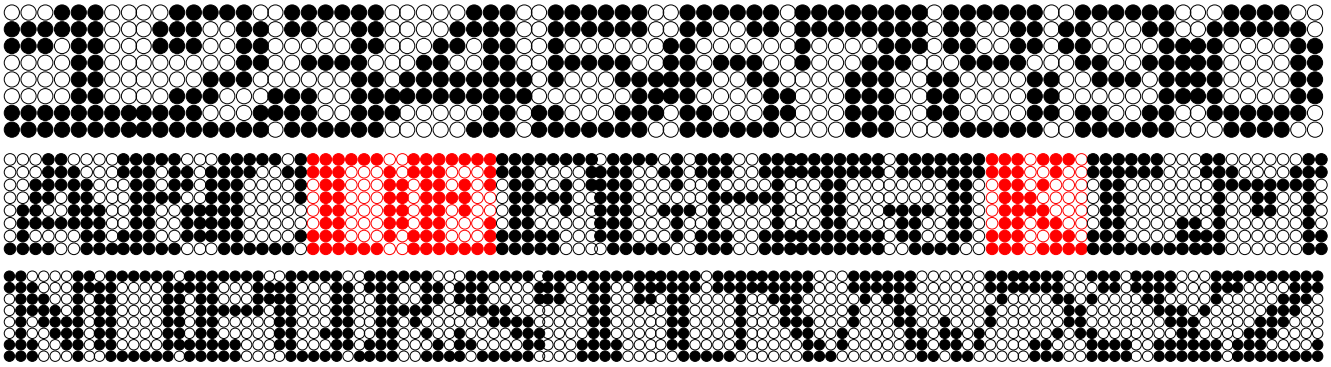
Indeed, numbers can be pasted together, also with the operators, which doesn't help with seeing at a glance what happens. Probably the best reference can be found at `https://developer.mozilla.org/en-US/docs/Web/SVG` where it is explained that a path can have move, line, curve, arc and other operators, as well use absolute and relative coordinates. How that works is for another article.

How would the TEX world look like today if Don Knuth had made METAFONT support colors? Of course one can argue that it is a bitmap font generator, but in our case using high resolution bitmaps might even work out better. In the example above the first text fragment uses a font that is very inefficient: it overlays many circles in different colors with slight displacements. Here a bitmap font would not only give similar effects but probably also be more efficient in terms of storage as well as rendering. The MetaPost successor does support color and with mplib in LuaTEX we can keep up quite well . . . as hopefully has been demonstrated here.

4 ThreeSix, Don Knuths first colorfont?

In the process of reaching completion and perfection Don Knuth occasionally posts links to upcoming parts of the TAOCP series on his web pages. Now, I admit that much is way beyond me but I do understand (and like) the graphics and I know that Don uses MetaPost. The next example code is just a proof of concept but might eventually become a decent module (with helpers) for making (runtime) fonts. After all, we need to adapt to current developments and TEXies always are willing to adapt and experiment. This chapter was written at the same time as the previous one on Type3 fonts so you might want to read that first.

The font explored here is `FONT36`, used in "A potpourri of puzzles" and flagged as "a special font designed for dissection puzzles" (in fascicle 9b for Volume 4). Playing with and visualizing for me often works better than formulas, which then distracts me from the original purpose, but let's have a closer look anyway.



The font has a fixed maximum height of 8 quantities. There is no depth in the characters. Some characters are wider. In this example we use a tight bounding box. In ConTEXt speak this font is just a regular font but with a special feature enabled.

```
\definefontfeature
  [fontthreesix]
  [default]
  [metapost=fontthreesix]

\definefont[DEKFontA][Serif*fontthreesix]
```

After this the `\DEKFontA` command will set this font as current font. The definition mentions `Serif` as font name. In ConTEXt this name will resolve in the currently defined Serif, so when your document uses Latin Modern that will be the one. The `fontthreesix` will make this instance use that feature set, and the feature definition has the defaults as parent (so we get kerning, ligatures, etc.) but as extra feature also `metapost`. This means that the new glyphs that are about to be defined will actually be injected in the `Serif`! We will replace characters in that instance. So, the following:

```
This font is used in \quotation {The Art Of Computer Programming} by
Don Knuth, not in volume~1, 2 or~3, but in number~4!
```

comes out as:

> This font is used in "The Art Of Computer Programming" by Don Knuth, not in volume 1, 2 or 3, but in number 4!

But that doesn't look too good, so we will tweak the font a bit:

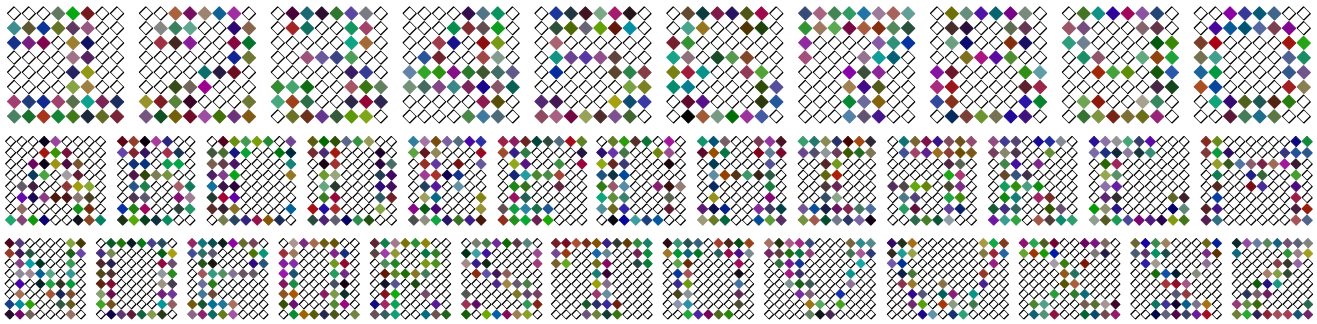
```
\definefontfeature
  [fontthreesix-color]
  [default]
  [metapost={category=fontthreesix,spread=.1}]

\definefont[DEKFontD][Serif*fontthreesix]
```

The `spread` (multiplied by the font unit, which is 12 basepoints here) will add a bit more spacing around the blob:

> ■his font is used in "■he ■rt ■f ■omputer ■rogramming" by ■on ■nuth, not in volume ■, ■ or ■, but in number ■!

Now, keep in mind that we're talking of a real font here. You can cut and paste these characters. It's just the default uppercase Latin alphabet plus digits.

Before we go and look at some of the code needed to render this, a few more examples will be given.



In the above example we not only use color, but also a different shape and random colors (that is: random per TEX job). The feature definition for this is:

```
\definefontfeature
  [fontthreesix-color]
  [default]
  [metapost={%
    category=fontthreesix,shape=diamond,%
    color=random,pen=fancy,spread=.1%
  }]
```

Possible shapes are `circle`, `diamond` and `square` and instead of a random color one can give a known color name. Using transparency makes no sense in this font.

A nice usage for this font are initials:

```
\setupinitial[font=Serif*fontthreesix-initial sa 5]
{\DEKFontB \placeinitial \input zapf\par}
```

The initial feature is defined as:

```
\definefontfeature
  [fontthreesix-initial]
  [metapost={category=fontthreesix,color=random,shape=circle}]
```

We use this in quoting Hermann Zapf, one that for sure is very applicable in a case like this:

oming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a or software. here is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. any people are just fascinated by their 's tricks, and think that a widely--praised program, called up on the screen, will make everything automatic from now on.

Some combinations of sub-features are shown in figure 4.1. We blow up the diamond with fancy pen example in figure 4.2. Alas, the TEX logo doesn't look that good in such a font. Using it for acronyms is not a good idea anyway, but maybe you can figure out figure 4.3.

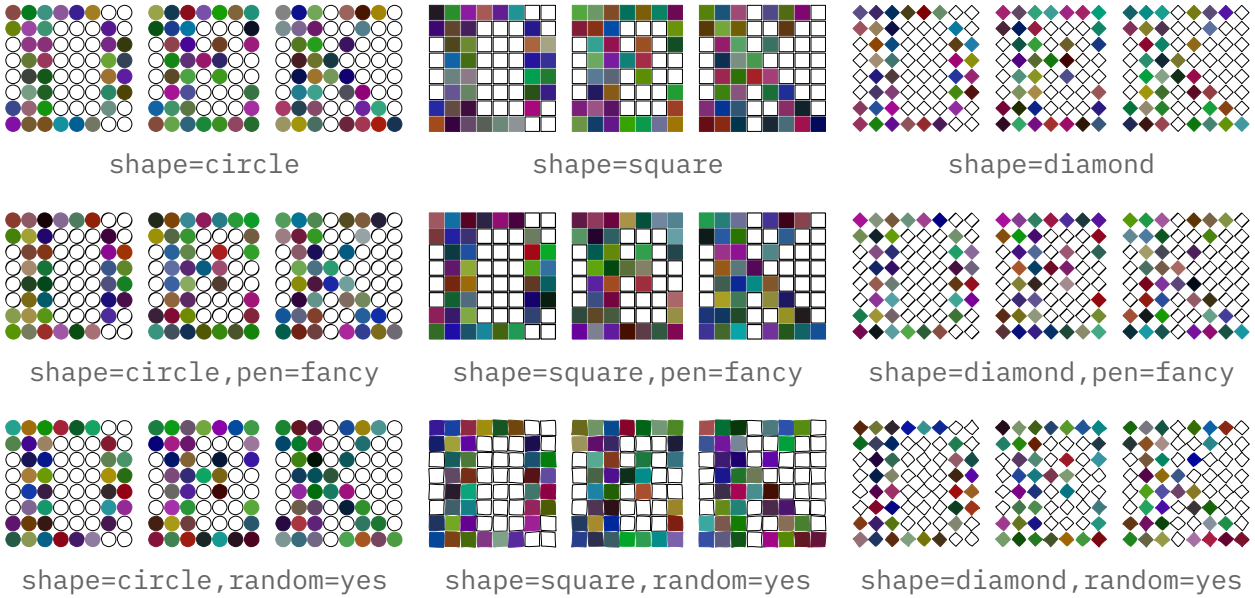


Figure 4.1

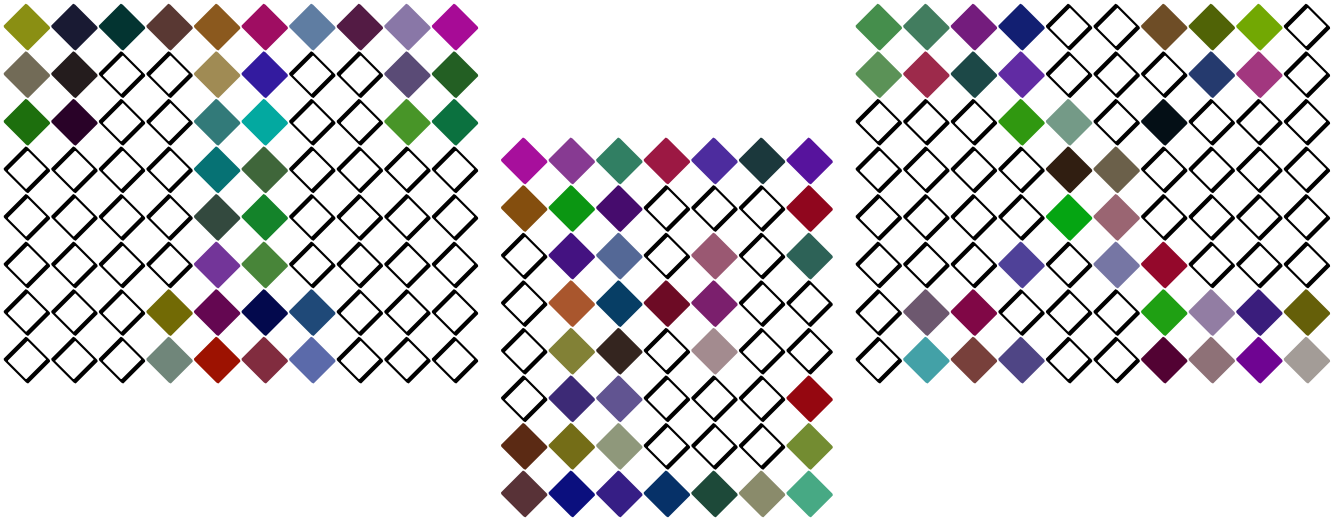


Figure 4.2

You can quit reading now or expose yourself to how this is coded. We use a combination of Lua and MetaPost, but different solutions are possible. The shapes are entered (or course) with zeros and ones.

```
\startluacode
local font36 = {
    ["0"] = "00111100 01111110 11000011 11000011 11000011 ...",
```

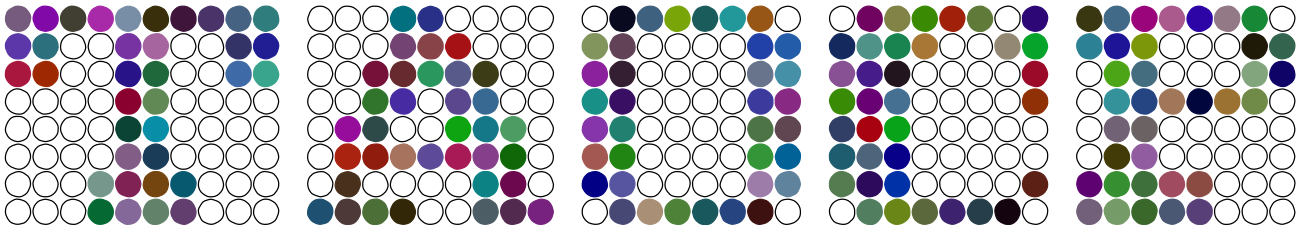


Figure 4.3

```
    ["1"] = "00011100 11111100 11101100 00001100 00001100 ...",
    .....
    ["D"] = "11111100 11100010 01100011 01100011 01100011 ...",
    ["E"] = "1111111 1110001 0110101 0111100 0110100 0110001 ...",
    .....
    ["K"] = "11101110 11100100 01101000 01110000 01111000 ...",
    .....
}
\stopluacode
```

We also use Lua to register this font. The actual code looks slightly different because it uses some helpers from the ConTEXt Lua libraries. We remap the bits pattern onto MetaPost macro calls.

```
\startluacode
local replace = {
    ["0"] = "N;",
    ["1"] = "Y;",
    [" "] = "L;",
}

function MP.registerthreesix(name)
    fonts.dropins.registerglyphs {
        name     = name,
        units    = 12,
        usecolor = true,
    }
    for u, v in table.sortedhash(font36) do
        local ny     = 8
        local nx     = (#v - ny + 1) // ny
        local height = ny * 1.1 - 0.1
        local width  = nx * 1.1 - 0.1
        local code   = string.gsub(v,".",replace)
        fonts.dropins.registerglyph {
            category = name,
            unicode  = utf.byte(u),
            width    = width,
            height   = height,
            code     = string.format("ThreeSix(%s);",code),
        }
    end
end

MP.registerthreesix("fontthreesix")
```

```
\stopluacode
```

So, after this the font `fontthreesix` is known to the system but we still need to provide MetaPost code to generate it. The glyphs themselves are now just sequences of `N`, `Y` and `L` with some wrapper code around it. The definitions are put in the `MP` namespace simply because a first version initialized in MetaPost, and there could create variants, but in the end I settled on the parameter interface at the TEX end.

The next definition looks a bit complex but normally such a macro is stepwise constructed. Notice how we can query the sub features. In order to make that possible the regular MetaFun parameter handling code is used. We just push the sub-features into to `mpsfont` namespace.

```
\startMPcalculation{simplefun}

def InitializeThreeSix =
    save Y, N, L, S ; save dx, dy, nx, ny ; save currentpen ;
    save shape, fillcolor, mypen, random, spread, hoffset ;
    string shape, fillcolor, mypen ; boolean random ;
    pen currentpen ;
    dx :=   11/10 ;
    dy := - 11/10 ;
    nx := - dx ;
    ny :=   0 ;
    shape      := getparameterdefault "mpsfont" "shape" "circle" ;
    random     := hasoption           "mpsfont" "random" "true" ;
    fillcolor  := getparameterdefault "mpsfont" "color" "" ;
    mypen      := getparameterdefault "mpsfont" "pen" "" ;
    spread     := getparameterdefault "mpsfont" "spread" 0 ;
    hoffset    := 12 * spread / 2 ;
    currentpen := pencircle
        if mypen = "fancy" :
            xscaled 1/20 yscaled 2/20 rotated 45
        else :
            scaled 1/20
        fi ;
    if shape == "square" :
        def S =
            unitsquare if random : randomized 1/10 fi
            shifted (nx,ny)
        enddef ;
    elseif shape = "diamond" :
        def S =
            unitdiamond if random : randomized 1/10 fi
            shifted (nx,ny)
        enddef ;
    else :
        def S =
            unitcircle if random : randomizedcontrols 1/20 fi
            shifted (nx,ny)
        enddef ;
    fi ;
    def N =
```

```
        nx := nx + dx ;
        draw S ;
    enddef ;
    if fillcolor = "random" :
        def Y =
            nx := nx + dx ;
            fillup S withcolor white randomized (2/3,2/3,2/3) ;
        enddef ;
    elseif fillcolor = "" :
        def Y =
            nx := nx + dx ;
            fillup S ;
        enddef ;
    else :
        def Y =
            nx := nx + dx ;
            fillup S withcolor fillcolor ;
        enddef ;
    fi ;
    def L =
        nx := - dx ;
        ny := ny + dy ;
    enddef ;
enddef ;

vardef ThreeSix (text code) =
    InitializeThreeSix ; % todo: once per instance run
    draw image (code) shifted (hoffset,-ny) ;
enddef ;

\stopMPcalculation
```

This code is not that efficient in the sense that there's quite some MetaPost-Lua-MetaPost traffic going on, for instance each parameter check involves this, but in practice performance is quite okay, certainly for such a small font. There will be an initializer option some day soon. The `simplefun` is a reference to an mplib instance that does load MetaFun but only the modules that make no sense for this kind of usage. It also enforces double mode. The calculations wrapper just executes the code and does not place some (otherwise empty) graphic.

Those who have seen (and/or read) "Concrete Mathematics" will have noticed the use of inline images, like dice. Dice are also used in "pre-fascicle 5a" (I need a few more lives to grasp that, so I stick to the images for now!). So, to compensate the somewhat complex code above, I will show how to accomplish that. This time we do all in MetaPost:

This is not that hard to follow. We define some shapes first. These could have been assigned to the `code` parameter directly but this is nicer. Next we register the font itself and after that we set glyphs. We also set the official Unicode slots. So, copying a dice can either result in a digit or in a Unicode slot for a dice. In the example below we switch to a color which demonstrates that our dice can be colored at the TEX end. It's either that or coloring at the MetaPost end as both demand a different kind of Type3 embedding trickery.

We actually predefine three features. The digits one will map regular digit in the input to dice. We accomplish that via a font feature:

```
\startluacode
fonts.handlers.otf.addfeature("dice:digits", {
    type      = "substitution",
    order     = { "dice:digits" },
    nocheck   = true,
    data      = {
        [0x30] = "invaliddice",
        [0x31] = 0x2680,
        [0x32] = 0x2681,
        [0x33] = 0x2682,
        [0x34] = 0x2683,
        [0x35] = 0x2684,
        [0x36] = 0x2685,
        [0x37] = "invaliddice",
        [0x38] = "invaliddice",
        [0x39] = "invaliddice",
    },
} )
\stopluacode
```

This kind of trickery is part of the font machinery used in ConTEXt and permits runtime adaption of fonts, so we just use the same mechanism. The `nocheck` is needed to avoid this feature not kicking in due to lack of (at the time of checking) yet undefined dice.

```
\definefontfeature
  [dice:normal]
  [default]
  [metapost={category=dice}]
\definefontfeature
  [dice:reverse]
  [default]
  [metapost={category=dice,option=reverse}]
\definefontfeature
  [dice:digits]
  [dice:digits=yes]

\definefont[DiceN] [Serif*dice:normal]
\definefont[DiceD] [Serif*dice:normal,dice:digits]
\definefont[DiceR] [Serif*dice:reverse,dice:digits]

{\DiceD Does 1 it 4 work? And {\darkgreen 3} too?} {\DiceR And how about
{\darkred 3} then? But 8 should sort of fail!}
```

Does ⚀ it ⚃ work? And ⚂ too? And how about ⚅ then? But ☒ should sort of fail!

The six digits and Unicode characters come out the same:

```
\red  \DiceD \dostepwiserecurse   {`1}   {`6}{1}{\char#1\quad}%
```

```
\blue \DiceN \dostepwiserecurse{"2680}{"2685}{1}{\char#1\quad}%
```

It is tempting to implement for instance 7 as two dice (a one to multi mapping in OpenType speak) but then one has to decide what combination is taken. One can also implement ligatures so that for instance 12 results in two six dice. But I think that's over the top and only showing TEX muscles. It is anyway not that hard to do as we have an interface for that already.

So why not do the dominos as well? Because there are so many dominos we predefine the shapes and then register the lot in a loop. We have horizontal and vertical variants. Being lazy I just made two helpers while one could have done but with some rotation and shifting of the horizontal one. The loop could be a macro but we don't save much code that way.

```
\startMPcalculation{simplefun}

picture Dominos[] ;

Dominos[0] := image() ;
Dominos[1] := image(draw(4,4);) ;
Dominos[2] := image(draw(2,6);draw(6,2););
Dominos[3] := image(draw(2,6);draw(4,4);draw(6,2););
Dominos[4] := image(draw(2,6);draw(6,6);draw(2,2);draw(6,2););
Dominos[5] := image(draw(2,6);draw(6,6);draw(4,4);draw(2,2);draw(6,2););
Dominos[6] := image(draw(2,6);draw(4,6);draw(6,6);draw(2,2);draw(4,2);draw(6,2););

lmt_registerglyphs [
    name     = "dominos",
    units    = 12,
    width    = 16,
    height   = 8,
    depth    = 0,
    usecolor = true,
] ;

def DrawDominoH(expr a, b) =
    draw image (
        pickup pencircle scaled 1/2 ;
        if (getparameterdefault "mpsfont" "color" "") = "black" :
            fillup unitsquare xyscaled (16,8) ;
            draw (8,.5) -- (8,7.5) withcolor white ;
            pickup pencircle scaled 3/2 ;
            draw Dominos[a]
                withpen currentpen
                withcolor white ;
            draw Dominos[b] shifted (8,0)
                withpen currentpen
                withcolor white ;
        else :
            draw unitsquare xyscaled (16,8) ;
```

```
            draw (8,0) -- (8,8) ;
            pickup pencircle scaled 3/2 ;
            draw Dominos[a]
                withpen currentpen ;
            draw Dominos[b] shifted (8,0)
                withpen currentpen ;
        fi ;
    ) ;
enddef ;

def DrawDominoV(expr a, b) = % is H rotated and shifted
    draw image (
        pickup pencircle scaled 1/2 ;
        if (getparameterdefault "mpsfont" "color" "") = "black" :
            fillup unitsquare xyscaled (8,16) ;
            draw (.5,8) -- (7.5,8) withcolor white ;
            pickup pencircle scaled 3/2 ;
            draw Dominos[a] rotatedaround(center Dominos[a],90)
                withpen currentpen
                withcolor white  ;
            draw Dominos[b] rotatedaround(center Dominos[b],90) shifted (0,8)
                withpen currentpen
                withcolor white  ;
        else :
            draw unitsquare xyscaled (8,16) ;
            draw (0,8) -- (8,8) ;
            pickup pencircle scaled 3/2 ;
            draw Dominos[a] rotatedaround(center Dominos[a],90)
                withpen currentpen ;
            draw Dominos[b] rotatedaround(center Dominos[b],90) shifted (0,8)
                withpen currentpen ;
        fi ;
    ) ;
enddef ;

begingroup
    save unicode ; numeric unicode ; unicode := 127025 ; % 1F031

    for i=0 upto 6 :
        for j=0 upto 6 :
            lmt_registerglyph [
                category = "dominos",
                unicode  = unicode,
                code     = "DrawDominoH(" & decimal i & "," & decimal j & ");",
                width    = 16,
                height   = 8,
            ] ;
            unicode := unicode + 1 ;
        endfor ;
    endfor ;
```

```
    save unicode ; numeric unicode ; unicode := 127075 ;

    for i=0 upto 6 :
        for j=0 upto 6 :
            lmt_registerglyph [
                category = "dominos",
                unicode  = unicode,
                code     = "DrawDominoV(" & decimal i & "," & decimal j & ");",
                width    = 8,
                height   = 16,
            ] ;
            unicode := unicode + 1 ;
        endfor ;
    endfor ;
endgroup ;

\stopMPcalculation
```

Again we predefine a couple of features:

```
\definefontfeature
  [dominos:white]
  [default]
  [metapost={category=dominos}]

\definefontfeature
  [dominos:black]
  [default]
  [metapost={category=dominos,color=black}]

\definefontfeature
  [dominos:digits]
  [dominos:digits=yes]
```

This last feature is yet to be defined. We could deal with the invalid dominos with some substitution trickery but let's keep it simple.

```
\startluacode
local ligatures = { }
local unicode   = 127025

for i=0x30,0x36 do
    for j=0x30,0x36 do
        ligatures[unicode] = { i, j }
        unicode = unicode + 1 ;
    end
end

fonts.handlers.otf.addfeature("dominos:digits", {
    type      = "ligature",
    order     = { "dominos:digits" },
```

```
    nocheck   = true,
    data      = ligatures,
} )
\stopluacode
```

That leaves showing an example. We define a few fonts and again we just extend the Serif:

```
\definefont[DominoW][Serif*dominos:white]
\definefont[DominoB][Serif*dominos:black]
\definefont[DominoD][Serif*dominos:white,dominos:digits]
```

The example is:

```
\DominoW
    \char"1F043\quad \quad
    \char"1F052\quad \quad
    \char"1F038\quad \quad
    \darkgreen\char"1F049\quad \char"1F07B\quad
\DominoB
    \char"1F087\quad
    \char"1F088\quad
    \char"1F089\quad
\DominoD
    \darkred 12\quad56\quad64
```

Watch the ligatures in action:

To what extent the usage of symbols like dice and dominos as characters in the mentioned book are responsible for them being in Unicode, I don't know. Now with all these emoji showing up one can wonder about graphics in such a standard anyway. But for sure, even after more than three decades, Don still makes nice fonts.

A treasure of tiny graphics can be found in "pre-fascicle 5c" and many are in color! In fact, it has dominos too. It must have been a lot of fun writing this! I'm thinking of turning the pentominoes into a font where a `GPOS` feature can deal with the inter-pentomino kerning (which mighty work out okay for example 36. The windmill dominos also make a nice example for a font where ligatures will boil down to the base form and the (one or more) blades are laid over. It's definitely an inspiring read.

5 Normalization

What I describe here was long due. I delayed it because when enabled it had best also be used and I need to (check and) adapt some code to it in order to profit from it. So, if used at all, it will take some time to have an effect on the ConTEXt code base. But first some background information.

When TEX builds a paragraph it splits the current text stream (that makes up the paragraph) into lines where each line becomes an horizontal box. In LuaTEX, this process is split into distinctive steps, contrary to regular TEX where the splitting is combined with hyphenation, ligature construction and font kerning. But what all engines have in common is that after the decision is made about what a line is, the result gets packages into the horizontal box.

The decision making is influenced by quite some factors, like:

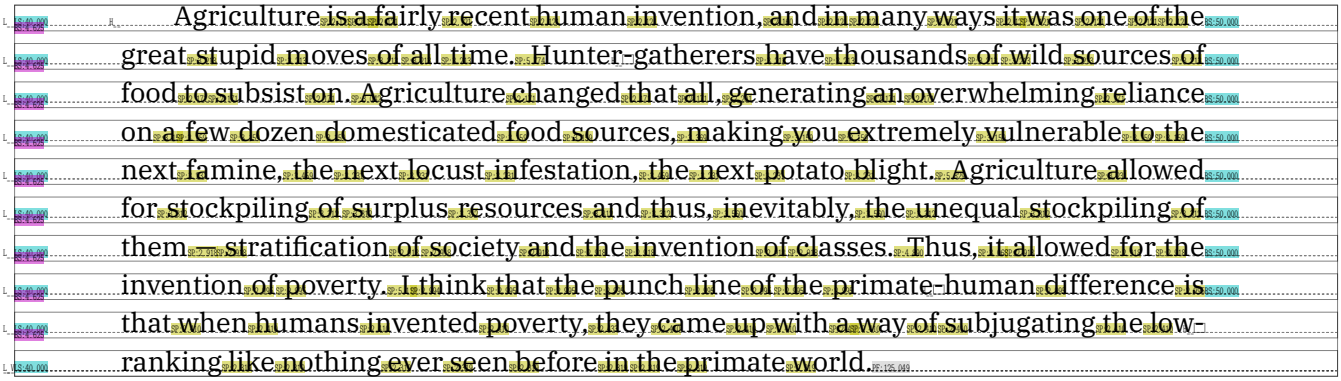
- The indentation of the first line, driven by the presence of a box of with a certain width and no height and depth (its always there, also when the indentation is zero).
- Hanging indentation, which can happen at each corner of the paragraph, or alternatively a specific parshape.
- Left and/or right margins, aka left skip and right skip. A right skip is always present, even when zero.
- The way the last line gets aligns, aka parfill skip.
- Directional changes that need to be carries over to the next line.
- Optional protrusion of characters to the left of right of the line, something that is sensitive for directional changes.
- Expansion of characters in order to get better inter-word spacing and/or to prevent lines being too bad. There can be stretch as well as shrink but on a per line basis. Inter-character kerns can also get that treatment.
- The penalties associated to hyphenation: the pre-last line, the last two lines, a list of penalties (ε-TEX), specific penalties bound to hyphenation pints (LuaTEX).
- The wish to have more or less lines than optimal, aka looseness. I have to admit that I never use that feature.

In traditional TEX it doesn't really matter how the resulting boxes look like, as long as the following steps can handle them, and those steps don't look into those boxes. In fact, unless you unpack a box, only the backend deals with the content. But in LuaTEX we have callbacks that hook into several stages and *can* look into the constructed boxes. In LuaTEX these boxes also have embedded directional information (needed by the backend) and (although that is seldom used) left or right boxed material, a features inherited from Aleph/Omega. And when messing around with the content of boxes one has to know what can be seen there. In principle the code can be reorganized a it but adding additional functionality is not that trivial because we want to stay close to the original implementation, even if it has been messed up a bit by successive additions. Eventually I might give it a try to integrate all these features a bit better, but on the other hand: it works.

In the next examples we show how the result of typesetting a paragraph looks like. We use the Sapolsky quote from the distribution. The cyan glue nodes are the left and right skip nodes, and the gray one at the end of the last line represents the parfill skip. The magenta ones at the edge are baseline skips. An indentation is shown in gray too. As experiment we have four normalization levels but in the end only the highest level makes sense, simply because normalization makes no sense unless one consistently normalizes all. We just keep the granularity because it makes it possible to explain what gets done.

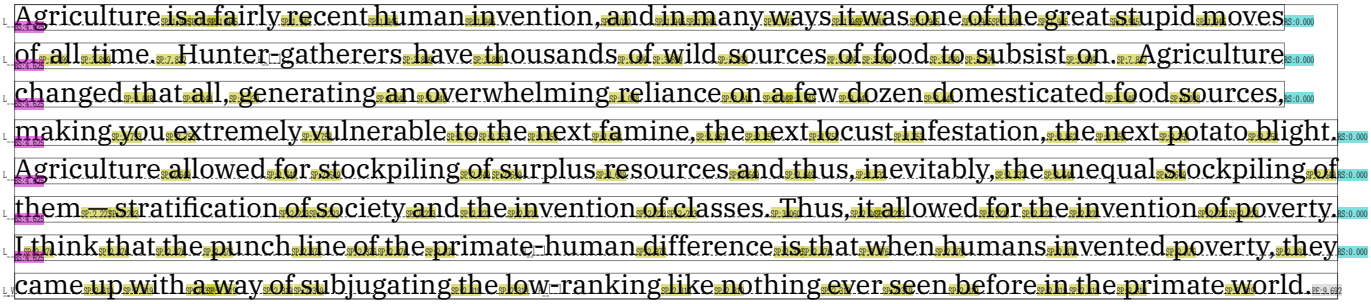
normalization 0, sample-1

```
\parindent  = 20pt
\leftskip   = 40pt
\rightskip  = 50pt
\hangindent =  0pt
\hangafter  =  0
```



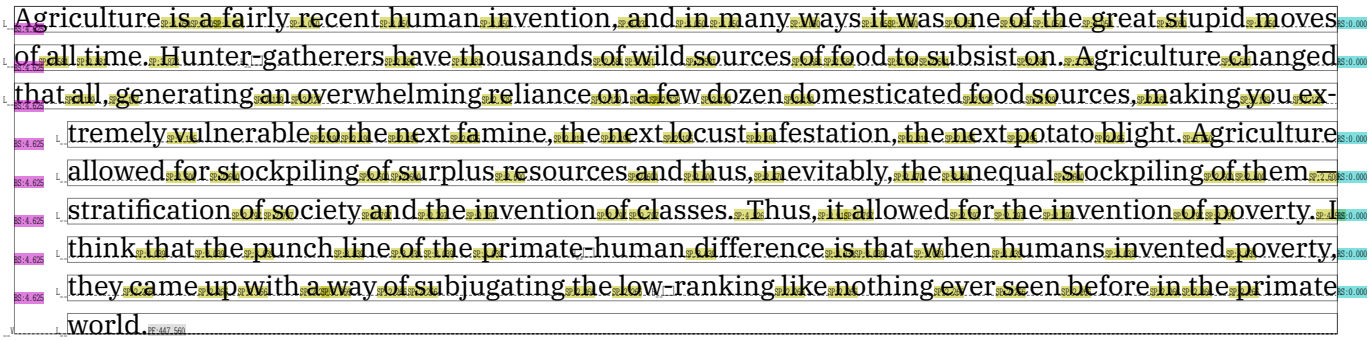
normalization 0, sample-2

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent =-20pt
\hangafter  = -3
```



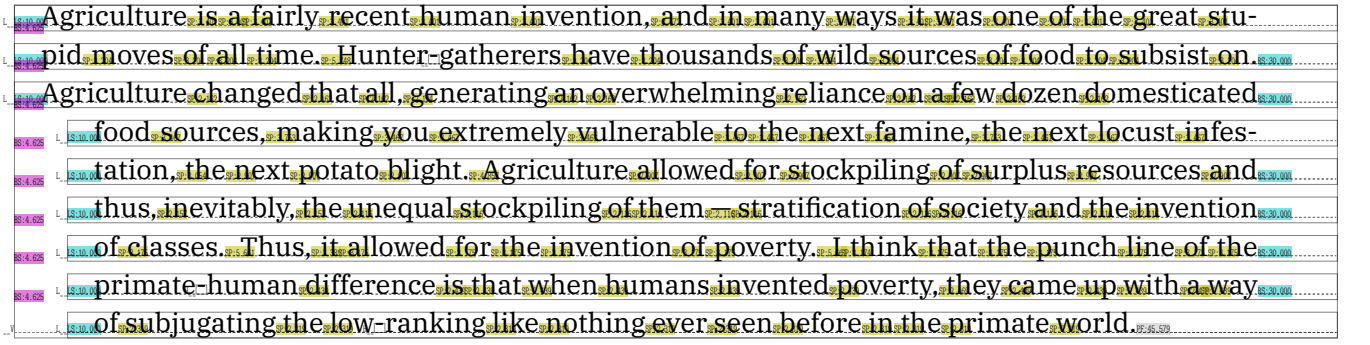
normalization 0, sample-3

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent = 20pt
\hangafter  =  3
```



normalization 0, sample-4

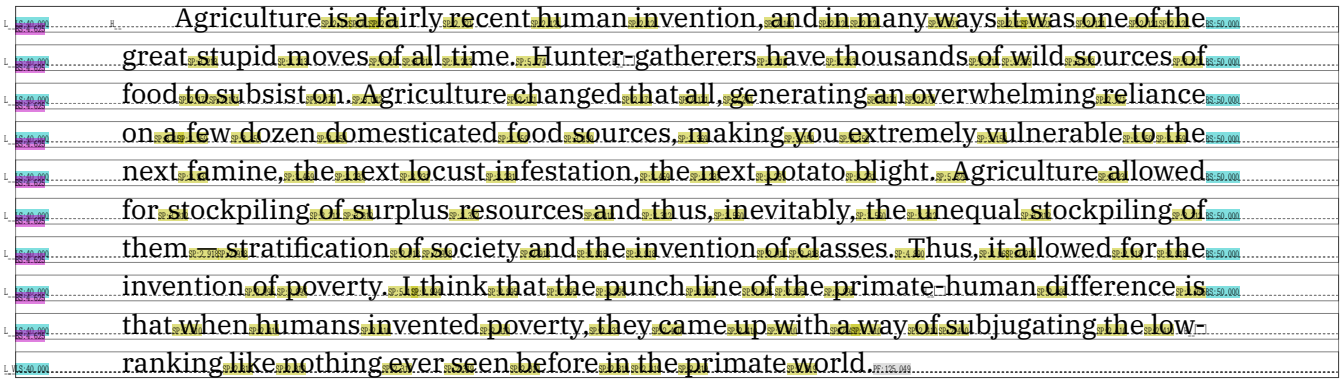
```
\parindent  =  0pt
\leftskip   = 10pt
\rightskip  = 30pt
\hangindent = 20pt
\hangafter  =  3
```



You might have noticed that the right skip is always there but the left skip is absent when it is zero. As said, as long as the result is okay, it does not really matter. But ... in LuaTEX (and therefore ConTEXt) it can have consequences because there we can kick in a callback that does something with lines. Such a callback often has to deal with these specific glues and them being optional makes for more testing. The more predictable the order is, the better. Although we can easily normalize lines (in a callback) to always have a left skip too it is also an option in the engine.

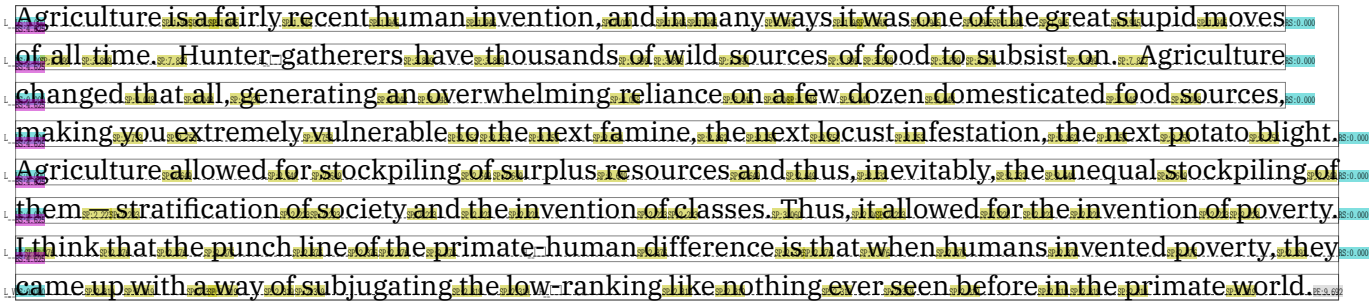
normalization 1, sample-1

```
\parindent  = 20pt
\leftskip   = 40pt
\rightskip  = 50pt
\hangindent =  0pt
\hangafter  =  0
```



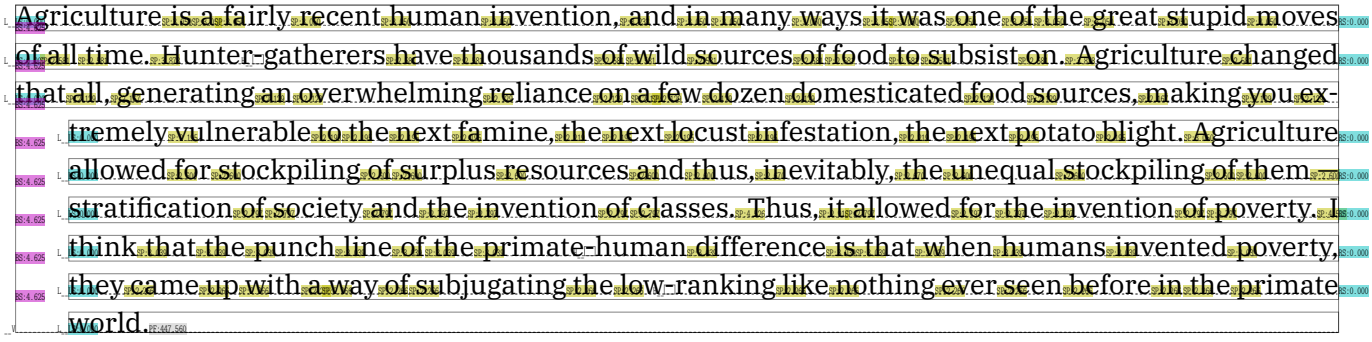
normalization 1, sample-2

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent =-20pt
\hangafter  = -3
```



normalization 1, sample-3

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent = 20pt
\hangafter  =  3
```



normalization 1, sample-4

```
\parindent  =  0pt
\leftskip   = 10pt
```

```
\rightskip  = 30pt
\hangindent = 20pt
\hangafter  =  3
```

Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them—stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world.

In the previous examples there are always left skips as well as right skips. It makes no sense to have an option to omit both zero left and right skips, because that again is unpredictable. But we can go further.

normalization 2, sample-1

```
\parindent  = 20pt
\leftskip   = 40pt
\rightskip  = 50pt
\hangindent =  0pt
\hangafter  =  0
```

Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them—stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world.

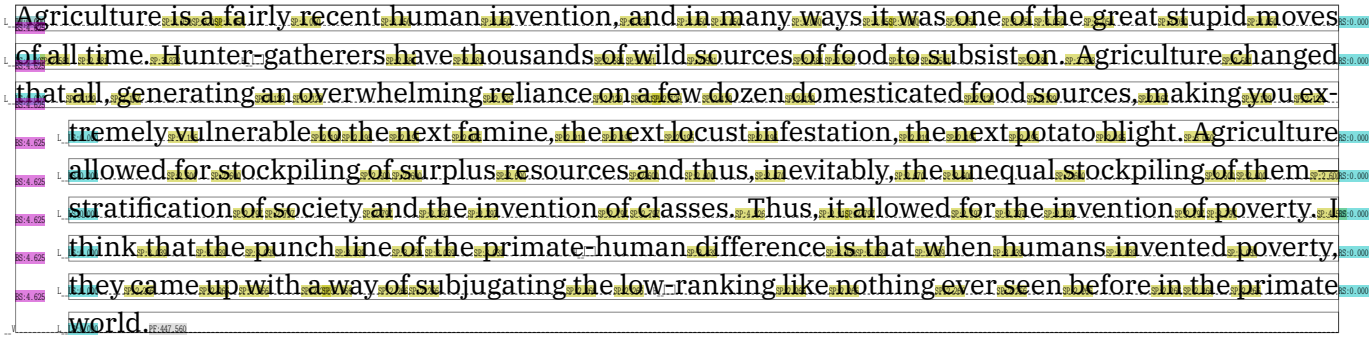
normalization 2, sample-2

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent =-20pt
\hangafter  = -3
```

Agriculture is a fairly recent human invention, and in many ways it was one of the great stupid moves of all time. Hunter-gatherers have thousands of wild sources of food to subsist on. Agriculture changed that all, generating an overwhelming reliance on a few dozen domesticated food sources, making you extremely vulnerable to the next famine, the next locust infestation, the next potato blight. Agriculture allowed for stockpiling of surplus resources and thus, inevitably, the unequal stockpiling of them—stratification of society and the invention of classes. Thus, it allowed for the invention of poverty. I think that the punch line of the primate-human difference is that when humans invented poverty, they came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world.

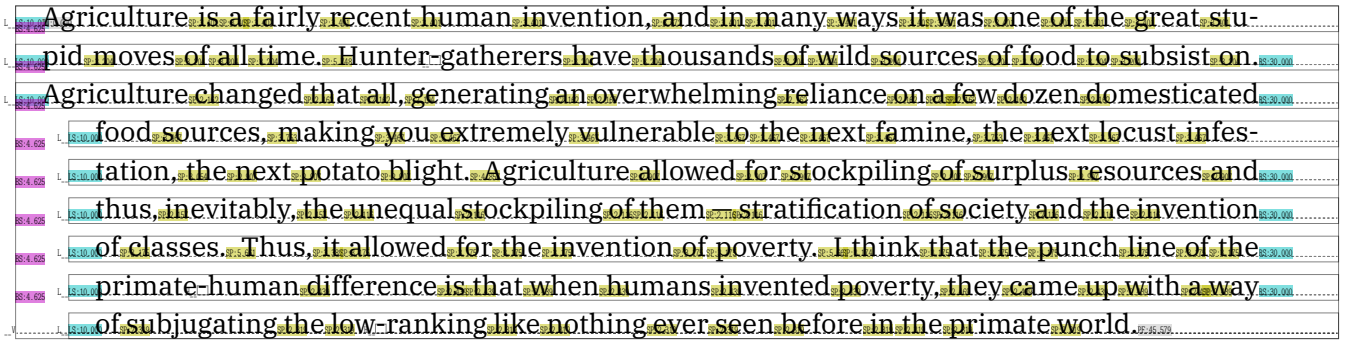
normalization 2, sample-3

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent = 20pt
\hangafter  =  3
```



normalization 2, sample-4

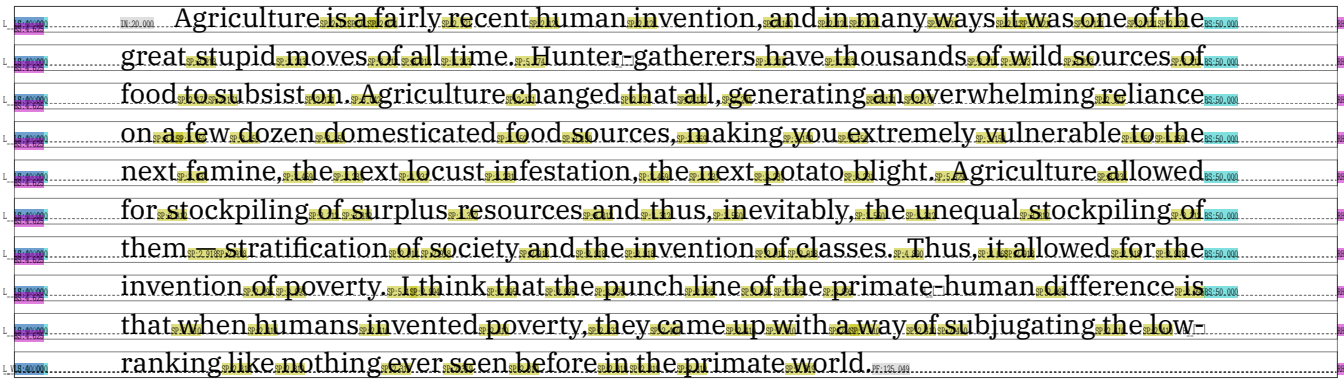
```
\parindent  =  0pt
\leftskip   = 10pt
\rightskip  = 30pt
\hangindent = 20pt
\hangafter  =  3
```



In these examples the indentation has been turned into a glue as well (actually it is more a kern but using a glue makes more sense). The hanging indentation however is not seen here: it is not represented by glue but instead sort of hidden in the width of the box and a shift of its content.

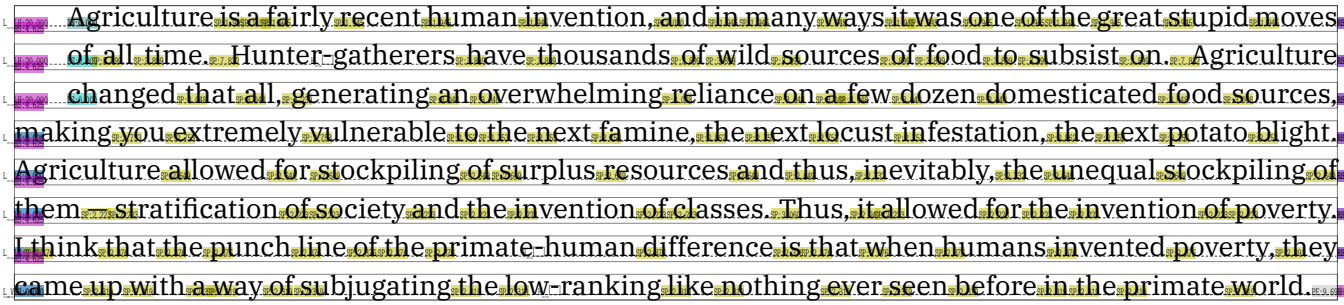
normalization 3, sample-1

```
\parindent  = 20pt
\leftskip   = 40pt
\rightskip  = 50pt
\hangindent =  0pt
\hangafter  =  0
```



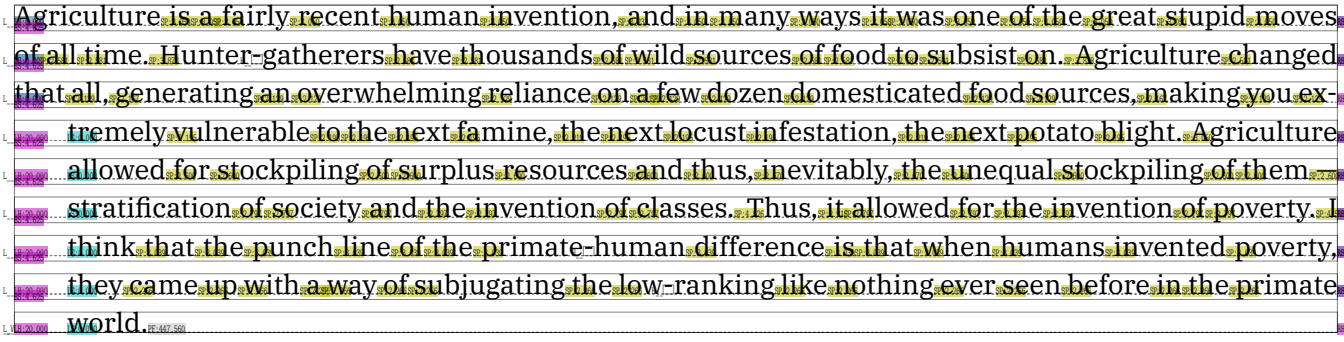
normalization 3, sample-2

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent =-20pt
\hangafter  = -3
```



normalization 3, sample-3

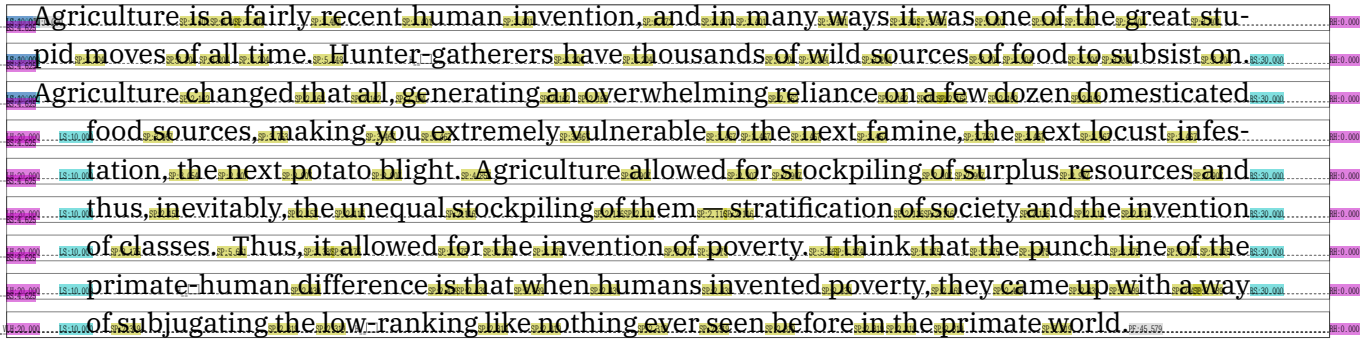
```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent = 20pt
\hangafter  =  3
```



normalization 3, sample-4

```
\parindent  =  0pt
\leftskip   = 10pt
```

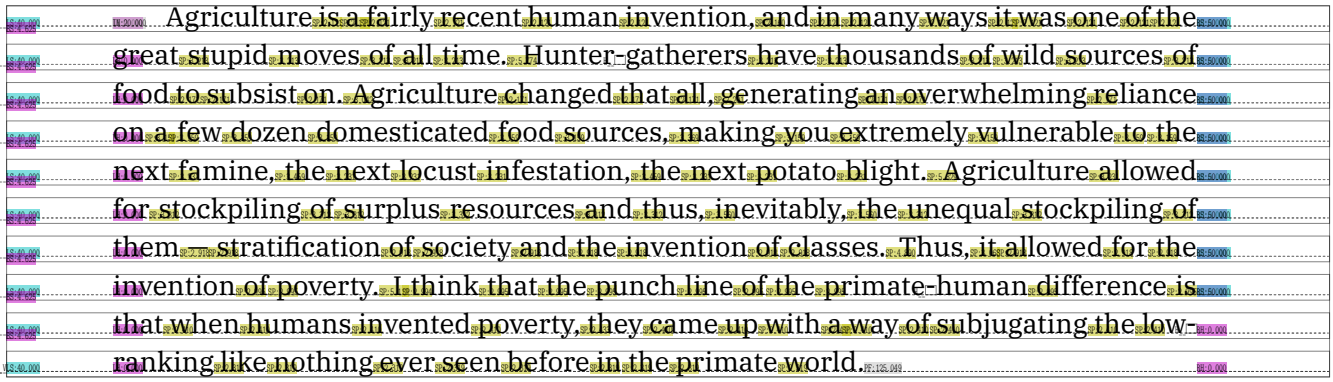
```
\rightskip  = 30pt
\hangindent = 20pt
\hangafter  =  3
```



In the previous examples the hanging indentation is turned into left and right hang skips. These cannot be set at the TEX end, but are injected when we instruct the normalizer to do so.

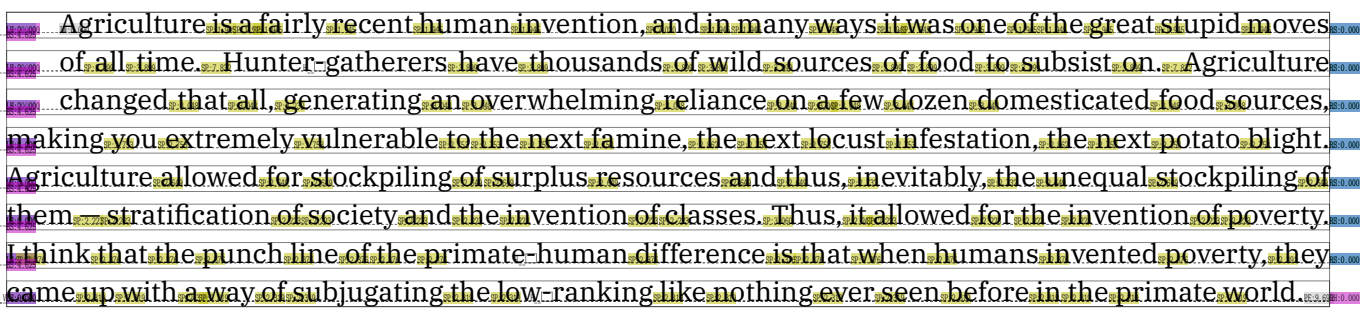
normalization 4, sample-1

```
\parindent  = 20pt
\leftskip   = 40pt
\rightskip  = 50pt
\hangindent =  0pt
\hangafter  =  0
```



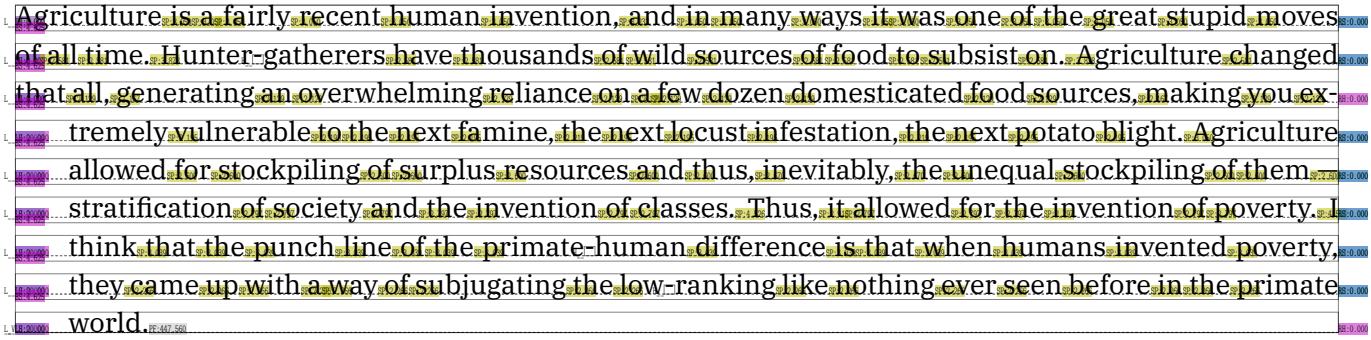
normalization 4, sample-2

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent =-20pt
\hangafter  = -3
```



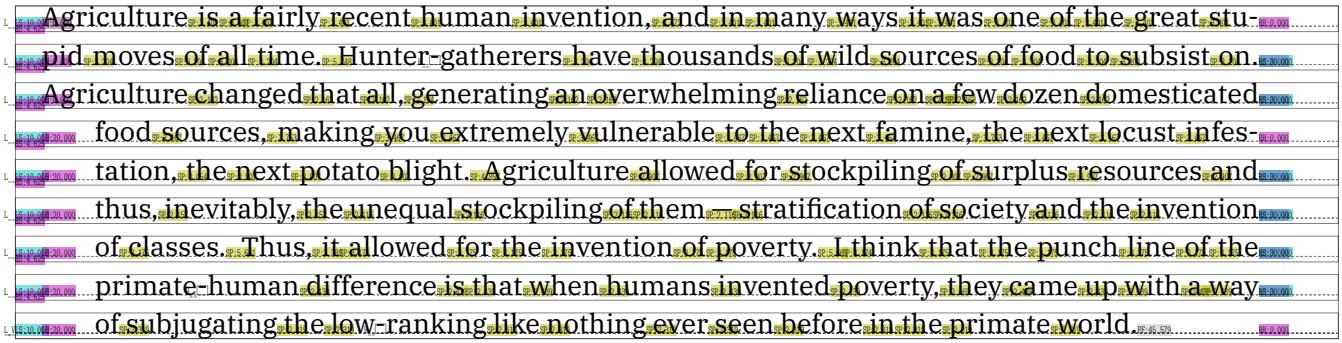
normalization 4, sample-3

```
\parindent  =  0pt
\leftskip   =  0pt
\rightskip  =  0pt
\hangindent = 20pt
\hangafter  =  3
```



normalization 4, sample-4

```
\parindent  =  0pt
\leftskip   = 10pt
\rightskip  = 30pt
\hangindent = 20pt
\hangafter  =  3
```

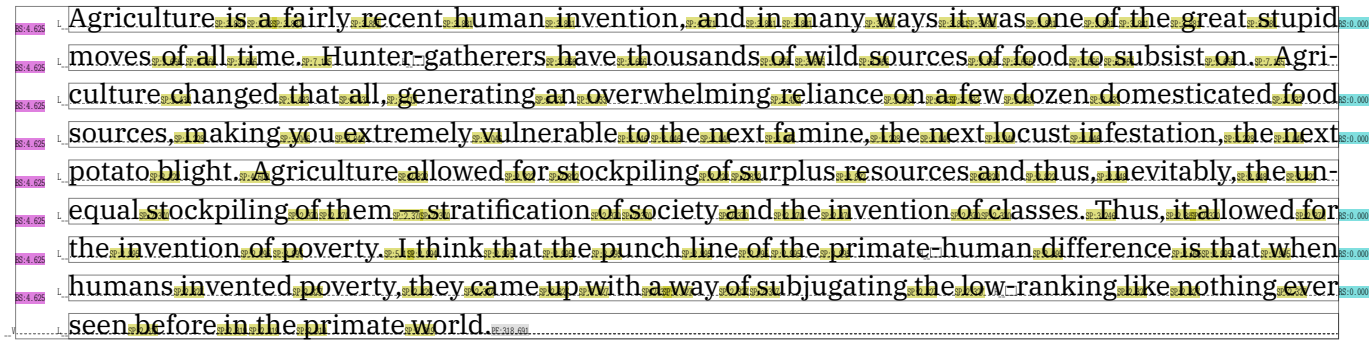


The previous examples differ from the previous set in that they push these hang related glue nodes before the left and after the right skip. As I couldn't make up my mind yet, I let LuaMetaTEX just provide both variants.

The option to keep hang related information explicitly in the line has some consequences. First of all, we now have glue and not some shift/width combination. Second, we have introduced an incompatibility: the lines now always have the proper width. You might have noticed that but we can show it more explicitly. We use two parameter sets

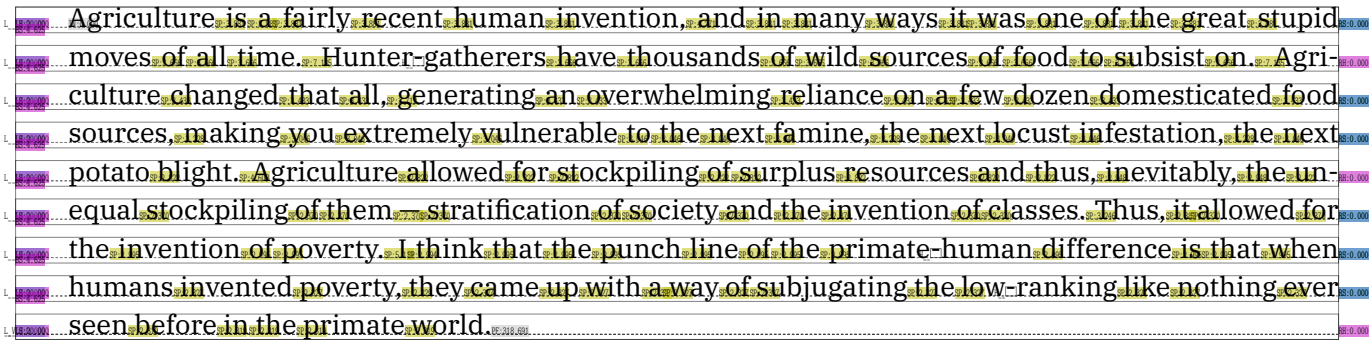
normalization 0, sample-5

```
\hangindent = 20pt
\hangafter  =  0
```



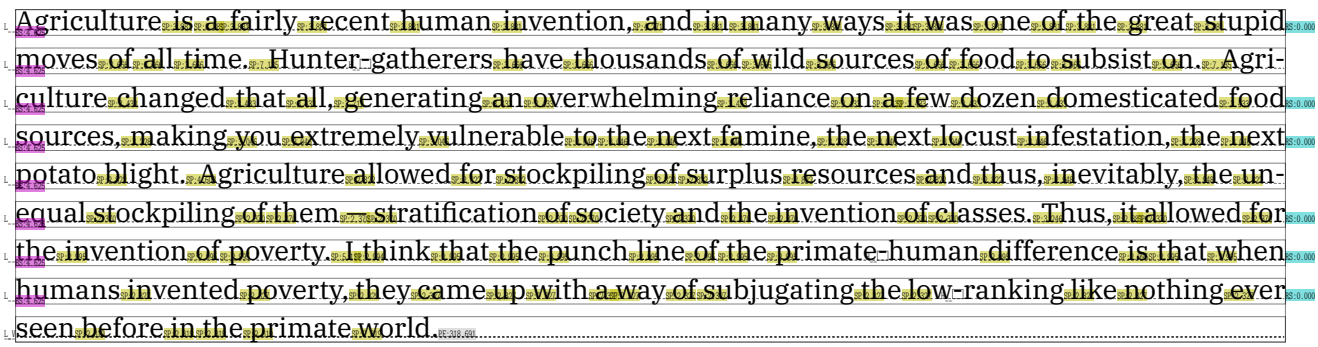
normalization 4, sample-5

```
\hangindent = 20pt
\hangafter  =  0
```



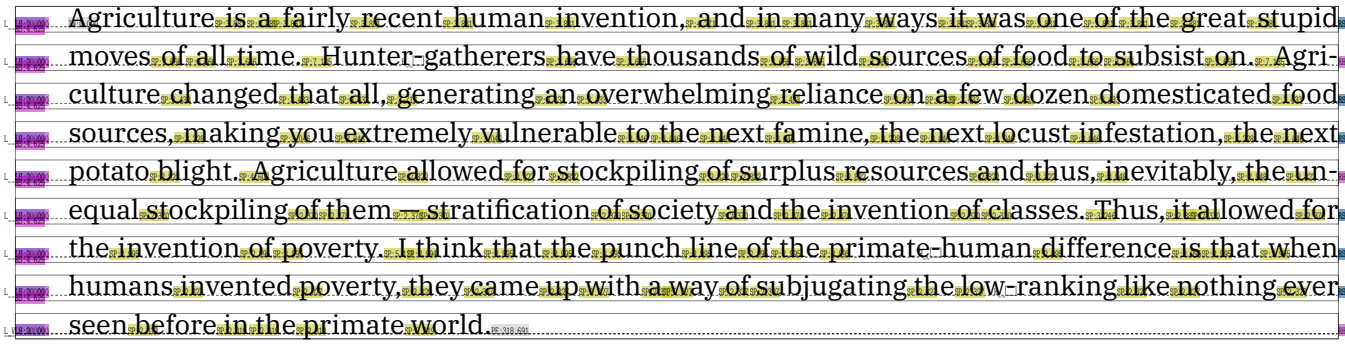
normalization 0, sample-6

```
\hangindent =-20pt
\hangafter  =  0
```



normalization 4, sample-6

```
\hangindent =-20pt
\hangafter  =  0
```



A not yet mention part of the normalization is that, because they are no longer of relevance, the special local par nodes have been removed. The one that starts a paragraph is turned into a normal directional node if needed, so that we get properly balanced pairs of directional nodes. It must been said that the code that does all this is a bit of a mess. We want to stay close to the original code, but we also need to deal with all these extensions, like directions, protrusion, extra boxes, etc.

Not shown here is that there is a fifth mode of operation. When we enable that level an overfull box will get a correction skip so that the right skip etc are properly aligned. How useful this is: we'll see.

Now, when I decide to keep this feature, which can be set at the Lua end to do the previously mentioned tasks, depending on a feature level ranging from zero to four, I also need to check the impact on existing ConTEXt code, which (currently) is complicated by the fact that most is shared between MkIV and lmtx, and only LuaMetaTEX has this normalization feature. I will probably enable it for a while locally in order to see if there are side effects. Then, when the code base gets adapted, we have to assume that normalization happens, so there is no way back.

6 Expansion

Character expansion was introduced in pdfTEX a couple of decades ago. It is a mechanism that scales glyphs horizontally in order to reduce excessive whitespace that is needed to properly justify a paragraph. I must admit that I never use it myself but there are users who do. Although this mechanism evolved a bit, and in LuaTEX is implemented a bit different, the basics remained the same. If you have no clue what this is about, you can just quite reading here.

A font can be set up to expand characters by a certain amount: they can shrink or stretch. This is driven by three parameters: `step`, `stretch` and `shrink`. The values are in thousands because TEX has no float quantity. Originally these values were percentages of the width of a glyph, later they became related to the em width but in LuaTEX we went back to the former definition.

In ConTEXt MkIV we have an interface that works as follows:

```
\startluacode
    local classes = fonts.expansions.classes

    classes.qualitya = {
        vector  = "default",
        factor  = 1,
        stretch = 4,
        shrink  = 2,
        step    = .5,
    }

    classes.qualityb = {
        vector  = "default",
        factor  = 1,
        stretch = 8,
        shrink  = 4,
        step    = .5,
    }

\stopluacode
```

The default vector looks like this:

```
vectors['default'] = {
    [0x0041] = 0.5, -- A
    [0x0042] = 0.7, -- B
    -- and some more
}
```

The values that we pass to the engine are stretch 40, shrink 20, and step 5 for `qualitya` and stretch 80, shrink 40, and step 5 for `qualityb`, so we multiply by 10. In the engine the step is limited to 100, the stretch to 1000 and the shrink to 500. But these extremes produce quite bad results.

The expansion class is set with the `expansion` feature, as in:

```
\definefontfeature [basea] [default] [expansion=qualitya]
\definefontfeature [baseb] [default] [expansion=qualityb]
```

```
\definefont [FontA] [Serif*basea]
\definefont [FontB] [Serif*baseb]
```

In figure 6.1 we see this in action, using the following code:

```
\setupalign[verytolerant,stretch,hz] % hz triggers expansion
\dorecurse {30} {%
    {\FontB \darkred test me #1,} \FontA \dorecurse{#1}{test ##1, }%
}\par
```

test me 1, test 1, test me 2, test 1, test 2, test me 3, test 1, test 2, test 3, test me 4, test 1, test 2, test 3, test 4, test me 5, test 1, test 2, test 3, test 4, test 5, test me 6, test 1, test 2, test 3, test 4, test 5, test 6, test me 7, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test me 8, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test me 9, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test me 10, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test me 11, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test me 12, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test me 13, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test me 14, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test me 15, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test me 16, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test me 17, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test me 18, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test me 19, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test me 20, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test me 21, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test me 22, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test me 23, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test me 24, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test me 25, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test me 26, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test me 27, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test 27, test me 28, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test 27, test 28, test me 29, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test 27, test 28, test 29, test me 30, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test 27, test 28, test 29, test 30,

Figure 6.1

There is one drawback with this method, although so far I never heard a user complain, which can be an indication of how this mechanism is used: you cannot mix fonts with different step, stretch and/or shrink. As we just did this in the example, this statement is not really true in LuaMetaTEX: there we only need to keep the step the same. This is compatible in the sense that otherwise we would quit the run, so at least

we carry on: the smallest stretch and shrink is taken. But, we do issue a warning (once) because there can be side effects! This is not that pretty a solution anyway because it depends on what font is used first.

It is for this reason that we have another option: in ConTEXt lmtx you can define a specific expansion:

```
\defineexpansion
  [myexpansion]
  [step=1, % default
   stretch=50,
   shrink=20]
```

There is no need to have a different step than 1. In pdfTEX instances are created per step used, but in LuaTEX this is more fluid. There is no gain in using different steps. We just keep it for compatibility reasons. This specific expansion is enables with:

```
\setexpansion[myexpansion]
```

and the result is shown in figure 6.2. This time the set expansion wins over the one set in the font. All fonts that have the expansion feature set are treated the same. By using this method you can locally have different values.

Deep down we use some new primitives:

```
\adjustspacingstep
\adjustspacingstretch
\adjustspacingshrink
```

The step is limited to 100 (10%) and the stretch and shrink to 500 (50%) and the stretch to 1000 (100%) but these extremes are only useful for examples.

test me 1, test 1, test me 2, test 1, test 2, test me 3, test 1, test 2, test 3, test me 4, test 1, test 2, test 3, test 4, test me 5, test 1, test 2, test 3, test 4, test 5, test me 6, test 1, test 2, test 3, test 4, test 5, test 6, test me 7, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test me 8, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test me 9, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test me 10, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test me 11, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test me 12, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test me 13, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test me 14, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test me 15, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test me 16, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test me 17, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test me 18, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test me 19, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test me 20, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test me 21, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test me 22, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test me 23, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test me 24, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test me 25, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test me 26, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test me 27, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test 27, test me 28, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test 27, test 28, test me 29, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test 27, test 28, test 29, test me 30, test 1, test 2, test 3, test 4, test 5, test 6, test 7, test 8, test 9, test 10, test 11, test 12, test 13, test 14, test 15, test 16, test 17, test 18, test 19, test 20, test 21, test 22, test 23, test 24, test 25, test 26, test 27, test 28, test 29, test 30,

Figure 6.2

7 Macros

In a rather large macro package like ConTEXt a user can't know all the commands. Even I often only use a handful of them. One danger luring is that commands can get redefined and that deep down such a command is used and that the new definition is not doing the expected.

A command like \framed can have companions, like \setupframed. These for the user visible commands are implemented using commands with less nice names, often in some namespace, using underscores and therefore also much longer. There is not that much change that a user spoils those. When a user uses \defineframed, a new command gets defined and when doing that a user should know what has already been defined.

In addition to these commands, we also have entities like character definitions and math symbols, as well as a while lot of registers (counters, dimensions etc.), and let's not forget primitives: the build in commands.

We can go paranoid and try to protect all these commands from redefinitions, but one of the prominent properties of TEX is that you *can* redefine commands. It therefore makes no sense to prohibit this. Also, in practice there are seldom problems, at least I haven't heard of them. In fact, too much protection can also bite us because sometimes redefinition is handy or even needed. Take for instance the \NC command used in tables: it adapts itself to the circumstances.

At the engine level there is not that much that we can (currently) do. In LuaMetaTEX we can define frozen macros. The question is: will we use that feature?

```
\frozen\def\foo{foo}
```

After this definition, one cannot easily redefine \foo. It is however possible (unless these primitives are redefined) to do this:

```
\unletfrozen\foo
```

and make \foo available again. It's companion is:

```
\letfrozen\foo
```

Because messing with frozen macros can issue an error message, we cannot demonstrate it on paper. But we can show a few companion primitives because just to be complete, one can also unprotect and protect existing macros:

```
          \def\crap{crap} \edef\morecrap{\crap} \meaning\morecrap
  \letprotected\crap      \edef\morecrap{\crap} \meaning\morecrap
\unletprotected\crap      \edef\morecrap{\crap} \meaning\morecrap
```

Protection prevents expansion of macros in some cases. One can define a macro by using the prefix \protected although in ConTEXt this doesn't work as expected. The protected was already taken when this primitive shows up, so we use \unexpanded or \normalprotected instead. And yes, that one also clashes with a primitive that showed up later, but as it's not used often, the longer \normalunexpanded will do.

```
macro:->crap
macro:->\crap
```

```
macro:->crap
```

Back to freezing commands: the question is, will and if so, how will we use this in ConTEXt? This is one of the decisions we have to make in 2020.

8 Libraries

8.1 Introduction

The LuaMetaTEX binary comes with a couple of libraries built in. These normally provide enough functionality to get a TEX job done. But take the case where need to manipulate (or convert) an image before we can include it? It would be nice if ConTEXt does that for you so having some features in the binary that handle it make sense. However, given that such a conversion only happens once it makes more sense to just call an external program and let that deal with it. It is for that reason that the ConTEXt code base has hardly any library related code: most of what one wants to do can be done by calling a program. Some callers are built in, others can be dealt with using the Adityas filter module. The most significant runtime exception is probably accessing sql databases where it might be more efficient to use a library call instead of calling a client. And even then the main reason for that interface being present is the simple fact that I (ab)use the engine to serve requests that need some kind of database access. Another example of where we need some external program is in generating barcodes. Here one can argue that it does make sense to do that runtime, for instance because they change or because one doesn't like to have dozens of cached barcode images on disk.

In this chapter I will explain how we deal with libraries in LuaMetaTEX. Because libraries create a dependency an approach is chosen that tries to avoid bloating the source tree with additional header and source files. This is made easy by the fact that we don't need full blown interfaces to libraries where all methods are exposed. We know what we need and most of these tasks somehow relate to typesetting which is a limited application with known demands in terms of input, output and performance. We don't need to serve every possible scenario.

8.2 Using LUA libraries

One approach is to use a Lua library that sits between the embedded Lua instance and the external library. Say that one does this:

```
local mylib = require("mylib")
```

This can locate and load the file `mylib.lua` which implements a bunch of (Lua) functions. But, it can also load a library, for instance `mylib.dll`, a binary that provides functions that themselves can call external ones. Often such a library is also responsible for some resource management which is then done via user-data objects. Such a connector library on the one hand refers to Lua library methods (like `const char * str = lua_tostring (L, 1);` for fetching a Lua string variable from the argument list) and on the other hand to those in the external library (like passing that string `str` to a function and passing the result back to Lua with `lua_pushstring (L, result);`). If we would follow that approach in LuaMetaTEX it means that in addition to the main binary (on MS Windows that is `luametatex.exe`) there is also an extra intermediate binary (on MS Windows that is `mylib.dll`) plus the external library (on MS Windows that could be `foolib.dll`) which itself can depend on other libraries.

In this approach we need to compile the extra intermediate libraries alongside the main LuaMetaTEX binary. Quite likely we then need access to the header files of the external libraries too. We might even decide to put the dependencies in our source tree. But, this is not what we like to do: it adds extra work, we need to keep an eye on updates and operating specific patches, we complicate the compilation, etc. This all contradicts the fact that we want LuaMetaTEX to be simple. There is no need to complicate the setup just because a very few users want to use some library. Add to this the fact that quite likely we need

to provide a version of LuaMetaTEX that exposes its Lua related symbols which makes for a larger binary. So, this approach is not really an option because at the same time we like to keep the binary (and memory footprint) as small as possible (think of running in a container or on a low energy device).

8.3 A variant

There are a few issues when you use Lua libraries from elsewhere. First of all, you need to get hold of one that matches the version of Lua that you use. There are not that many and some only can be set up as part of a larger framework. Also, you can find plenty of modules that seem not to be maintained (or maybe they are just very stable and I'm wrong here). Also, not all platforms are supported equally well. Then there is the question to what extend libraries are to stay. What is considered to be the standard today might not be tomorrow. Even in the rather stable TEX ecosystem we see them come and go. These are all reasons to avoid hard coded dependencies. Ideally we like users to be able to compile LuaMetaTEX in the future without too must hassle.

A couple of years after we started the LuaTEX project, a solution for using libraries was implemented, called SwigLib, because it uses the swig infrastructure. It was an attempt to come up with a more or less standard approach, a rather one-to-one mapping so that basically any library could be interfaced. But, probably because no one really needs libraries, it never catched on. In MkIV we still support loading libraries made that way but in lmtx that code has been removed.

As a side note: the code that deals with this in MkIV also deals with version specific loading. When we were playing with for instance MySQL libs we found out that it made sense to be able to support different apis, but in the end, given the rare usage of libraries, that made no sense either. Therefore in lmtx locating libraries has version support removes and as a consequence is much simpler (code-wise).

8.4 Foreign function interfaces

Then there is a ffi interface, first introduced in LuajitTEX as it is part of LuaJIT, and later a similar library was built-in LuaTEX. But LuaJIT doesn't conceptually follow Lua upgrades and its future is unsure so in LuaMetaTEX there is no jit variant (the jit part was never used anyway as it only slowed down a run; we just used the ffi part plus the fact that the restricted virtual machine performs better). The ffi library used in LuaTEX also comes from elsewhere and it doesn't seem to be maintained any longer, so that code is to be kept working in the perspective of LuaTEX. Both technologies hook into the processor architecture and are somewhat complex so when their maintenance becomes unsure we have to reconsider using them. Not all hardware platforms are supported[4] and the functionality can differ in details per platform. To some extend we can keep using ffi in LuaTEX because Luigi takes care of it, but who knows when it becomes too problematic. Does it make sense to adopt a library that needs tweaks depending on architectures? For now we're good for LuaTEX, so for a while we're also okay (in MkIV).

The nice thing about ffi is that one can define the interface at runtime. Of course this interface has to fit the current version of the library api, but that is doable. It is up to a user of a library to determine where it comes from. It can be put in the TEX tree but also being taken from wherever the operating system put it in the path. Of course that can then be a bit of an issue when there are different versions because programs can ship their own variants, but when you use a library you probably are aware of that and know what you're doing. A drawback of ffi is that it opens up the whole machinery pretty low level, which can be considered a risk. Some can consider that to be a security threat. It for these reasons that LuaMetaTEX doesn't provide the ffi feature; users who depend on it can of course use MkIV with LuaTEX.

[4] As I write this only Intel works while ARM doesn't and only on MS Windows, linux and os-x I can compile without alignment warnings

8.5 So how to proceed?

When a library and its Lua interface are kept external the main binary has to be compiled in a way that permits loading libraries (read: symbols need to be known). When we use ffi that is not needed. And when a library is internal we have the disadvantage that we mentioned at the start of this chapter.

So, how do we combine the advantages of ffi (runtime binding), external libraries (no need to have all that code in the code base) and internal libraries (no loading issues)? At some point it stroke me that we actually can do that with not that much effort. The solution was probably subconsciously implanted by noticing the fact that the LuaMetaTEX machinery uses function pointers in some places and the fact that when a Lua library is loaded by Lua itself, a specific initialization function is called to initialize it: by combining these concepts we can delay the binding till when a library is needed.

In LuaMetaTEX we can therefore have some optional libraries that offer a minimal interface because after all we can do a lot at the Lua end. Optional libraries register themselves in the global `optional` table. We're talking of a couple of hundred lines of C for a simple binding. The functions in an optional library table can be used (accessed) without loading the library and then just do nothing useful. So, before using them you need to load the third party library but we can safely assume that the Lua wrapper code calls an initializer when it needs some feature. That initializer, which by the way is located at the Lua end, loads the external library, and when that is successful the needed helpers are bound by resolving function pointers. There is no dependency when nothing is used: the main binary stays lean and mean because the binding normally only adds a few KB. Users can compile without dependencies and when used performance is quite okay (no ffi overhead).

The LuaMetaTEX distribution only ships a few such bindings but these can serve as example. What is shipped has a proper Lua companion file and these are then the standard one used in the ConTEXt distribution. Think of MySQL and SQLite (for databases), zint (for barcodes), simple Curl (for fetching stuff), Ghostscript and GraphicsMagick (for some conversions) bindings . When compiled into LuaMetaTEX these will add some interfacing code to the main binary but that gets compensated by the removal of the ffi library. The Lua interfaces provide just enough to get us going. At some point we can consider providing libraries as optional part of an installation because we can generate them using the buildbot infrastructure managed by Mojca, but the core distribution (source code) is kept clean.

9 Is LUAMETATEX still TEX?

9.1 Introduction

Is LuaMetaTEX really a TEX (compatible) engine? The answer to that depends on how you define TEX. If you think of the program with the same name, the answer is definitely "no", simply because a program that is not exactly behaving like "TEX The Program" cannot be called TEX. This is why derived programs have `tex` in their name but also some addition that indicates that it isn't the original: `e`, `pdf`, `lua`. Don't confuse that with macro package names that have `tex` in their name. If you find such binaries that they are likely some stub to an engine (binary) that preloads a format file (a memory dump) with the same name.

When you mean "TEX The Macro Language" the answer is a bit more nuanced especially when the results are pretty close to identical. In the next sections I will discuss this in more detail from the perspective of how ConTEXt evolved and what engines it has used.

9.2 Multiple engines

When we started with ConTEXt there was not that much choice in engines. Basically one just used original TEX, but although we used the version that came with the book, pretty soon we switched to emTEX, a version that gave more memory; later a real huge version showed up. The fonts used were bitmaps and the viewer was a dvi bitmap viewer. However, when our new printer could not be set up properly we decided to move on to PostScript fonts. That also meant using a different backend driver (dvipsone). And then of course we also started using a previewer that could handle outline fonts. Once you start along that route graphics come into play, color shows up and hyperlinks become an option. A couple of years later the pdf document rendering format was introduced. This paragraph already mentions a lot of different programs and adaptations, but we're still talking good old TEX here and ConTEXt was set up in such a way that it adapted itself to whatever ecosystem made sense. When looking at TEX one has to consider the front as well as the backend, and both have related primitives and features. Extensions to the frontend have been driven by the demands of macro packages (beyond the original ideas) and those of the backend relate to what the evolving rendering demands impose.

A couple of decades ago the ε-TEX project started. It's objective was to extend stable TEX with a couple of more primitives and features: it is a superset and therefore still TEX, but as it really is an extension the name was extended too (with the bit unusual character ε). At that point the main reason for ConTEXt was convenience because the new features were already kind of present in the code base (think of emulated behavior). Again the macro package adapted itself at runtime.

Then pdfTEX came around which had some impact. It introduced the concept of a built-in backend that avoided additional programs. The ε-TEX extensions were merged into this program so that basically meant that it replaced its predecessors. For a user pdfTEX was just TEX. For some reason the narrative became that ConTEXt depended on pdfTEX, probably because it was always quick in using its features, a side effect of being close to the development.

The ConTEXt package was an early adopter of MetaPost and that graphic subsystem, although still external, was integrated in such a way that users could think of it being embedded. This was made possible by the fact that right from the start ConTEXt came with an infrastructure that handled processing including subruns as needed for MetaPost. This is why, years later, adding a MetaPost library to LuaTEX was a logical step. As ConTEXt came with a lot of scripts (for all kind of tasks related to typesetting and managing a TEX ecosystem) adding a scripting language (like Lua) was not that strange either.

In parallel to pdfTEX the experimental Omega program was on its way and although at some point a stable Aleph variant was there, it never was robust enough for production. Its main contribution (that survived) was the introduction if directional typesetting. There were ConTEXt users using it but for very specific applications. It's also why the bidirectional model of Omega inspired LuaTEX more than the simpler model that ε-TEX used.

9.3 The merge

We now move forward to LuaTEX and more precisely LuaMetaTEX because that is for ConTEXt the engine of choice now. To what extend is it TEX or not? The naive answer is "no" because some primitives are not present and/or are implemented using Lua. However, these primitives fall into categories. Some relate to the backend and in LuaMetaTEX the backend is not built-in and as a consequence a macro package has to provide the primitives as part of its implementation of a backend. This is no big deal because the backend related primitives in TEX The Program are actually examples of extensions and implemented as such. Handling them happens in kind of isolated code. Take `\special`: it is basically a no-op when the dvi driver doesn't interpret what is passed to the dvi file.[5] [6]

A more drastic change is the lack of font loaders and that no fonts can be stored in the format. Again this relates to the simple fact that todays fonts are more demanding so we need to extend the machinery and as we do that via Lua extensions we can as well do all that way. Less drastic, but it could have side effects, is that the machinery has to be able to deal with OpenType math. And of course all is Unicode aware so additional primitives cope with that. But in principle the old stuff should still work. Hyphenation is also expanded: patterns are loaded runtime and the hyphenation, ligature building and kerning stages are split, which actually it a good thing.

The LuaMetaTEX code base is a follow up on LuaTEX, that combines good old TEX (but adapted with respect to fonts, languages and math as mentioned), parts of ε-TEX (so it provides more primitives), bits of pdfTEX (like protrusion and expansion, although adapted), and rudiments of Omega (Aleph). And of course there's a lot of new stuff too, primitives as well as ways to plug in Lua code plus some helpers at the Lua end.

As an example of progression, by now the ε-TEX extensions that we kept are integrated more naturally in existing subsystems. A nice detail is that there are no longer any version numbers that relate to ε-TEX; for a while they were kept but suddenly I realized that it makes no sense to waste (four) command codes on something that is of not much use: there has never been a real ε-TEX follow up after its stable release so testing for a version makes no sense. No backend means no pdfTEX version info too and Omega version numbers serve no purpose either. If a macro package needs to know what functionality is there, testing for the LuaTEX version number, revision and maybe functionality level makes enough sense. By the way, one reason for a clean up related to ε-TEX was that where ε-TEX uses change files to replace or extend good old TEX code, LuaTEX has one integrated code base.

9.4 The verdict

So in the end the answer is that LuaMetaTEX is mostly TEX but that due to developments like for instance Unicode, OpenType fonts and math, as well as the wish to use images, color, runtime graphics, directionality, features beyond what the engine has built, etc. in the end it hopefully meets the demands to today. In

[5] ConTEXt MkII has a bunch of backend drivers, TEX code, that targets specific postprocessors and they hook into primitives like `\special` or the additional `\pdf ...` ones in pdfTEX.

[6] We need to keep in mind that by the time pdfTEX and later LuaTEX were developed memory constraints were lifted so these engines didn't have to work around the limitations that for instance ε-TEX and Omega had to cope with.

its core the same code is still there although extensions and hooks got mixed in more naturally. When in documents (or talks) I speak of TEX I basically refer to a concept (materialized in the set of core primitives and related functionality) but once extensions come into play I try to talk of LuaTEX or LuaMetaTEX. This happens kind of automatic because I know what got added but I can imagine that users who entered the game later don't always see what was added (and when).

10 Numbers

A few decades of programming in the TEX language can make one wish for certain features. It will therefore be no surprise that in LuaTEX (and even more in LuaMetaTEX) we have some additional functionality. However, I have to admit that some of these are not used that much in ConTEXt MkIV and lmtx. The reason is that some wishes date from MkII times and because we now have Lua we actually don't need that many new fancy features. Also, it makes no sense to rewrite mechanisms that are already working well. However, in order to fully exploit the possibilities that Lua gives, there are some additions that relate to the way this language can communicate with TEX. Of course there's also the issue of a potentially enhanced performance, but there is not much to gain in that department.

A side effect of adding features, of which some are just there to complete the picture, or, as mentioned, because they were supposed to make sense, is that I make examples. Here I show the result of one of these experiments. I have no clue how useful this is, but I've learned not to underestimate users in their demands and creativity.

Internally, TEX does all in 32 bit integers. When you say:

```
\scratchcounter 123
\scratchdimen   123pt
```

the `123` gets assigned to a count register and the `123pt` is assigned to a dimen register but actually that is then also an integer: the internal unit of a dimen is a scaled point (sp) and only when its value is shown to the user, a real number can show up, followed by the `pt` unit. The precision is limited, so you can expect about four decimal positions precision here. There is no concept of a floating point number in TEX, and the only case where a float gets used is in the final calculations of glue and even that only comes into play in the backend.

So, although I don't really have an application for it in ConTEXt (otherwise I'd already added a float data type to the engine), it sounded like a good idea to see if we could emulate float support. In the following examples the numbers are managed in Lua and therefore they are global. I could make a local variant but why complicate matters. These macros start with `\lua` to make clear that they are not managed by TEX.

```
\luacardinal bar  123
\luainteger  bar -456
\luafloat    bar  123.456E-3
```

We define `bar` three times. Each type gets its own hash, so from the perspective of Lua its nature is kept: integer or double.

```
\the\luacardinal bar \quad
\the\luainteger  bar \quad
\the\luafloat    bar
```

123 -456 0.123455999999999996297184168270177906379103660583496093 75

Instead of decimal values, you can also use hexadecimal values (watch the `p` for exponents):

```
\luacardinal bar  0x123
\luainteger  bar -0x456
\luafloat    bar  0x123.456p-3
```

So, now we get:

```
291   -1110   36.40887451171875
```

From these examples you see two kind of usage: setting a value, and using it. It is that property that makes them special. Because the macros are implemented using Lua calls it means that at the Lua end we know what usage is expected. And it is that dualistic property that I wanted to explore but that in the end only makes sense it a very few cases, but sometimes those few are important. We could of course two macros, a setter and a getter, but using one kind of its in.

The setters accept an optional equal sign, as in:

```
\luainteger gnu=  123456   \luafloat gnu=  123.456e12
\luainteger gnu = 123456   \luafloat gnu = 123.456e12
\luainteger gnu  =123456   \luafloat gnu  =123.456e12
```

Although Lua is involved in picking up the value, storing it someplace, and retrieving it on demand, performance is pretty good. You probably won't notice the overhead anyway.

The values that `\the` returns are serialized numbers. However, sometimes you want what TEX sees as a numeric token, For that we have these variants

```
\luadimen test 100pt
\scratchdimen = .25 \luadimen test
\the\scratchdimen
```

Which produces the expected value: `25.0pt`, something that depends on the fact that the dimension is not a serialized. Talking of serialization, there are several ways that Lua can do that so let's give some examples. We start with some definitions. Beware, floats and cardinals are stored independently!

```
\luacardinal x = -123
\luacardinal y =  456

\luafloat    x =  123.123
\luafloat    y = -456.456
```

We have a macro `\luaexpression` (not to be confused with `\luaexpr`) that takes an optional keyword:

```
- : \luaexpression          {n.x + 2*n.y}
f : \luaexpression float    {n.x + 2*n.y}
i : \luaexpression integer  {n.x + 2*n.y}
c : \luaexpression cardinal {n.x + 2*n.y}
b : \luaexpression boolean  {n.x + 2*n.y}
l : \luaexpression lua      {n.x + 2*n.y}
```

The serialization can be different for these cases:

```
- : -789.789
f : -789.78899999999998726707417517900466918945312 5
i : -790
c : 790
b : 1
l : -0x1.8ae4fdf3b645ap+9
```

The numbers namespace resolves to a float, integer or cardinal (in that order) and calculations take place as in Lua. If you only use integers then normally Lua will also serialize them as such.

Here is another teaser. Say that we set the scratchdimen register to a value:

```
\scratchdimen 123.456pt
```

We now introduce the \nodimen macro, that can be used this way:

```
[\the\scratchdimen] [\the\nodimen\scratchdimen]
```

[123.456pt] [123.456pt]

which is not that spectacular. Nor is this:

```
\nodimen\scratchdimen = 654.321pt
```

But how about this:

```
\the \nodimen bp \scratchdimen  651.876462bp
\the \nodimen cc \scratchdimen  50.959168cc
\the \nodimen cm \scratchdimen  22.996753cm
\the \nodimen dd \scratchdimen  611.510013dd
\the \nodimen in \scratchdimen  9.05384in
\the \nodimen mm \scratchdimen  229.96753mm
\the \nodimen nc \scratchdimen  51.103896nc
\the \nodimen nd \scratchdimen  613.246746nd
\the \nodimen pt \scratchdimen  654.320999pt
\the \nodimen sp \scratchdimen  42881581sp
```

So here we have a curious mix of setter and getter. The setting part is not that interesting but we just provide it as convenience (and demo). Of course we can have 10 specific macros instead. Keep in mind that this is a low level macro, so it doesn't use the normal ConTEXt user interface.

A bit more complex are one or two dimensional arrays. Again this is an example implementation where users can come up with more ideas.

```
\newarray name integers   type integer   nx 2 ny 2
\newarray name booleans   type boolean   nx 2 ny 2
\newarray name floats     type float     nx 2 ny 2
\newarray name dimensions type dimension nx 4
```

Here we define three two-dimensional assays and one one-dimensional array. The type determines the initialization as well as the scanner and serializer. Values can be set as follows:

```
\arrayvalue integers   1 2 4        \arrayvalue integers   2 1 8
\arrayvalue booleans   1 2 true     \arrayvalue booleans   2 1 true
\arrayvalue floats     1 2 12.34    \arrayvalue floats     2 1 34.12
\arrayvalue dimensions 1   12.34pt  \arrayvalue dimensions 3   34.12pt
```

If you want to check an array on the console, you can say:

```
\showarray integers
```

We now access some values. Apart from the float these are (sort of) native data types.

```
[\the\arrayvalue integers   1 2]
[\the\arrayvalue booleans   1 2]
[\the\arrayvalue floats     1 2]
[\the\arrayvalue dimensions 1  ]\crlf
[\the\arrayvalue integers   2 1]
[\the\arrayvalue booleans   2 1]
[\the\arrayvalue floats     2 1]
[\the\arrayvalue dimensions   3]
```

This produces:

[4] [1] [12.339999999999999857891452847979962825775146484375] [12.34pt]
[8] [1] [34.119999999999997442046151263639330863952636718750] [34.12pt]

You can of course use these values in many ways:

```
\dostepwiserecurse{1}{4}{1}{
    [\the\arrayvalue dimensions #1 :
     \luaexpression dimen {math.sind(30) * a.dimensions[#1]}]
}
```

This gives:

[12.34pt: 6.17pt] [0.0pt: 0pt] [34.12pt: 17.06pt] [0.0pt: 0pt]

In addition to the already seen integer and dimension variables fed back into TEX, we also have booleans. These are just integers with the value zero or one. In order to make their use easier there is a new `\ifboolean` primitive that takes such a bit:

```
slot 1 is \ifboolean\arrayequals dimensions 1 0pt zero \else not zero \fi
slot 2 is \ifboolean\arrayequals dimensions 2 0pt zero \else not zero \fi
```

We get:

slot 1 is not zero
slot 2 is zero

A variant is a comparison macro. Of course we can use the dimen comparison conditional instead:

```
slot 1: \ifcase\arraycompare dimensions 1 3pt lt \or eq \else gt \fi zero
slot 2: \ifcase\arraycompare dimensions 2 3pt lt \or eq \else gt \fi zero
slot 3: \ifcase\arraycompare dimensions 3 3pt lt \or eq \else gt \fi zero
slot 4: \ifcase\arraycompare dimensions 4 3pt lt \or eq \else gt \fi zero

slot 1: \ifcmpdim\arrayvalue dimensions 1 3pt lt \or eq \else gt \fi zero
slot 2: \ifcmpdim\arrayvalue dimensions 2 3pt lt \or eq \else gt \fi zero
slot 3: \ifcmpdim\arrayvalue dimensions 3 3pt lt \or eq \else gt \fi zero
slot 4: \ifcmpdim\arrayvalue dimensions 4 3pt lt \or eq \else gt \fi zero
```

We get:

slot 1: gt zero
slot 2: lt zero

slot 3: gt zero
slot 4: lt zero

slot 1: gt zero
slot 2: lt zero
slot 3: gt zero
slot 4: lt zero

Anyway, the question is: do we need this kind of trickery, and if so, what more is needed? But beware: we do have Lua anyway, so there is no need for a complex user interface at the TEX end just for the sake of it looking more TEX. The above shows a bit what is possible.

It is too soon to discuss the low level interface because it still evolves. After some initial experiments, I decided to follow a slightly different route, and often the third implementation starts to look what I like more.

11 Parameters

When TEX reads input it either does something directly, like setting a register, loading a font, turning a character into a glyph node, packaging a box, or it sort of collects tokens and stores them somehow, in a macro (definition), in a token register, or someplace temporary to inject them into the input later. Here we'll be discussing macros, which have a special token list containing the preamble defining the arguments and a body doing the real work. For instance when you say:

```
\def\foo#1#2{#1 + #2 + #1 + #2}
```

the macro `\foo` is stored in such a way that it knows how to pick up the two arguments and when expanding the body, it will inject the collected arguments each time a reference like `#1` or `#2` is seen. In fact, quite often, TEX pushes a list of tokens (like an argument) in the input stream and then detours in taking tokens from that list. Because TEX does all its memory management itself the price of all that copying is not that high, although during a long and more complex run the individual tokens that make the forward linked list of tokens get scattered in token memory and memory access is still the bottleneck in processing.

A somewhat simplified view of how a macro like this gets stored is the following:

```
hash entry "foo" with property "macro call" =>

    match (# property stored)
    match (# property stored)
    end of match

    match reference 1
    other character +
    match reference 2
    other character +
    match reference 1
    other character +
    match reference 2
```

When a macro gets expanded, the scanner first collects all the passed arguments and then pushes those (in this case two) token lists on the parameter stack. Keep in mind that due to nesting many kinds of stacks play a role. When the body gets expanded and a reference is seen, the argument that it refers to gets injected into the input, so imagine that we have this definition:

```
\foo#1#2{\ifdim\dimen0=0pt #1\else #2\fi}
```

and we say:

```
\foo{yes}{no}
```

then it's as if we had typed:

```
\ifdim\dimen0=0pt yes\else no\fi
```

So, you'd better not have something in the arguments that messes up the condition parser! From the perspective of an expansion machine it all makes sense. But it also means that when arguments are not used, they still get parsed and stored. Imagine using this one:

```
\def\foo#1{\iffalse#1\oof#1\oof#1\oof#1\oof#1\fi}
```

When TEX sees that the condition is false it will enter a fast scanning mode where it only looks at condition related tokens, so even if `\oof` is not defined this will work ok:

```
\foo{!}
```

But when we say this:

```
\foo{\else}
```

It will bark! This is because each `#1` reference will be resolved, so we effectively have

```
\def\foo#1{\iffalse\else\oof\else\oof\else\oof\else\oof\else\fi}
```

which is not good. On the other hand, since expansion takes place in quick parsing mode, this will work:

```
\def\oof{\else}
\foo\oof
```

which actually is:

```
\def\foo#1{\iffalse\oof\oof\oof\oof\oof\oof\oof\oof\oof\fi}
```

So, a reference to an argument effectively is just a replacement. As long as you keep that in mind, and realize that while TEX is skipping 'if' branches nothing gets expanded, you're okay.

Most users will associate the `#` character with macro arguments or preambles in low level alignments, but since most macro packages provide a higher level set of table macros the latter is less well known. But, as often with characters in TEX, you can do magic things:

```
\catcode`?=\catcode`#

\def\foo #1#2?3{?1?2?3} \meaning\foo\space=>\foo{1}{2}{3}\par
\def\foo ?1#2?3{?1?2?3} \meaning\foo\space=>\foo{1}{2}{3}\par
\def\foo ?1?2#3{?1?2?3} \meaning\foo\space=>\foo{1}{2}{3}\par
```

Here the question mark also indicates a macro argument. However, when expanded we see this as result:

```
macro:#1#2?3->?1?2?3 =>123
macro:?1#2?3->?1?2?3 =>123
macro:?1?2#3->#1#2#3 =>123
```

The last used argument signal character (officially called a match character, here we have two that fit that category, `#` and `?`) is used in the serialization! Now, there is an interesting aspect here. When TEX stores the preamble, as in our first example:

```
match (# property stored)
match (# property stored)
end of match
```

the property is stored, so in the later example we get:

```
match (# property stored)
match (# property stored)
```

```
match (? property stored)
end of match
```

But in the macro body the number is stored instead, because we need it as reference to the parameter, so when that bit gets serialized TEX (or more accurately: LuaTEX, which is what we're using here) doesn't know what specific signal was used. When the preamble is serialized it does keep track of the last so-called match character. This is why we see this inconsistency in rendering.

A simple solution would be to store the used signal for the match argument, which probably only takes a few lines of extra code (using a nine integer array instead of a single integer), and use that instead. I'm willing to see that as a bug in LuaTEX but when I ran into it I was playing with something else: adding the ability to prevent storing unused arguments. But the resulting confusion can make one wonder why we do not always serialize the match character as `#`.

It was then that I noticed that the preamble stored the match tokens and not the number and that TEX in fact assumes that no mixture is used. And, after prototyping that in itself trivial change I decided that in order to properly serialize this new feature it also made sense to always serialize the match token as `#`. I simply prefer consistency over confusion and so I caught two flies in one stroke. The new feature is indicated with a `#0` parameter:

```
\bgroup
\catcode`?=\catcode`#

\def\foo ?1?0?3{?1?2?3} \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1#0?3{?1?2?3} \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo #1#2?3{?1?2?3} \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1#2?3{?1?2?3} \meaning\foo\space=>\foo{1}{2}{3}\crlf
\def\foo ?1?2#3{?1?2?3} \meaning\foo\space=>\foo{1}{2}{3}\crlf
\egroup
```

```
macro:#1#0#3->#1#2#3 =>13
macro:#1#0#3->#1#2#3 =>13
macro:#1#2#3->#1#2#3 =>123
macro:#1#2#3->#1#2#3 =>123
macro:#1#2#3->#1#2#3 =>123
```

So, what is the rationale behind this new `#0` variant? Quite often you don't want to do something with an argument at all. This happens when a macro acts upon for instance a first argument and then expands another macro that follows up but only deals with one of many arguments and discards the rest. Then it makes no sense to store unused arguments. Keep in mind that in order to use it more than once an argument does need to be stored, because the parser only looks forward. In principle there could be some optimization in case the tokens come from macros but we leave that for now. So, when we don't need an argument, we can avoid storing it and just skip over it. Consider the following:

```
\def\foo     #1{\ifnum#1=1 \expandafter\fooone\else\expandafter\footwo\fi}
\def\fooone#1#0{#1}
\def\footwo#0#2{#2}
\foo{1}{yes}{no}
\foo{0}{yes}{no}
```

We get:

yes no

Just for the record, tracing of a macro shows that indeed there is no argument stored:

```
\def\foo#1#0#3{....}
\foo{11}{22}{33}
\foo #1#0#3->....
#1<-11
#2<-
#3<-33
```

Now, you can argue, what is the benefit of not storing tokens? As mentioned above, the TEX engines do their own memory management.[7] This has large benefits in performance especially when one keeps in mind that tokens get allocated and are recycled constantly (take only lookahead and push back).

However, even if this means that storing a couple of unused arguments doesn't put much of a dent in performance, it does mean that a token sits somewhere in memory and that this bit of memory needs to get accessed. Again, this is no big deal on a computer where a TEX job can take one core and basically is the only process fighting for cpu cache usage. But less memory access might be more relevant in a scenario of multiple virtual machines running on the same hardware or multiple TEX processes on one machine. I didn't carefully measure that so I might be wrong here. Anyway, it's always good to avoid moving around data when there is no need for it.

Just to temper expectations with respect to performance, here are some examples:

```
\catcode`!=9 % ignore this character
\firstoftwoarguments
  {!!!!!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!!!!!}
\secondoftwoarguments
  {!!!!!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!!!!!}
\secondoffourarguments
  {!!!!!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!!!!!}
  {!!!!!!!!!!!!!!!!!!}{!!!!!!!!!!!!!!!!!!}
```

In ConTEXt we define these macros as follows:

```
\def\firstoftwoarguments       #1#2{#1}
\def\secondoftwoarguments      #1#2{#2}
\def\secondoffourarguments#1#2#3#4{#2}
```

The performance of 2 million expansions is the following (probably half or less on a more modern machine):

macro	total	step
\firstoftwoarguments	0.245	0.000000123
\secondoftwoarguments	0.251	0.000000126
\secondoffourarguments	0.390	0.000000195

But we could use this instead:

```
\def\firstoftwoarguments       #1#0{#1}
\def\secondoftwoarguments      #0#2{#2}
\def\secondoffourarguments#0#2#0#0{#2}
```

[7] An added benefit is that dumping and undumping is relatively efficient too.

which gives:

macro	total	step
\firstoftwoarguments	0.229	0.000000115
\secondoftwoarguments	0.236	0.000000118
\secondoffourarguments	0.323	0.000000162

So, no impressive differences, especially when one considers that when that many expansions happen in a run, getting the document itself rendered plus expanding real arguments (not something defined to be ignored) will take way more time compared to this. I always test an extension like this on the test suite[8] as well as the LuaMetaTEX manual (which takes about 11 seconds) and although one can notice a little gain, it makes more sense not to play music on the same machine as we run the TEX job, if gaining milliseconds is that important. But, as said, it's more about unnecessary memory access than about cpu cycles.

This extension is downward compatible and its overhead can be neglected. Okay, the serialization now always uses # but it was inconsistent before, so I'm willing to sacrifice that (and I'm pretty sure no ConTEXt user cares or will even notice). Also, it's only in LuaMetaTEX (for now) so that other macro packages don't suffer from this patch. The few cases where ConTEXt can benefit from it are easy to isolate for MkIV and lmtx so we can support LuaTEX and LuaMetaTEX.

I mentioned LuaTEX and how it serializes, but for the record, let's see how pdfTEX, which is very close to original TEX in terms of source code, does it. If we have this input:

```
\catcode`D=\catcode`#
\catcode`O=\catcode`#
\catcode`N=\catcode`#
\catcode`-=\catcode`#
\catcode`K=\catcode`#
\catcode`N=\catcode`#
\catcode`U=\catcode`#
\catcode`T=\catcode`#
\catcode`H=\catcode`#

\def\dek D1O2N3-4K5N6U7T8H9{#1#2#3 #4#6#7#8#9}

{\meaning\dek \tracingall \dek don{}knuth}
```

The meaning gets typeset as:

```
macro:D1O2N3-4K5N6U7T8H9->H1H2H3 H4H6H7H8H9don nuth
```

while the tracing reports:

```
\dek D1O2N3-4K5N6U7T8H9->H1H2H3 H5H6H7H8H9
D1<-d
O2<-o
N3<-n
-4<-
K5<-k
```

[8] Currently some 1600 files that take 24 minutes plus or minus 30 seconds to process on a high end 2013 laptop. The 260 page manual with lots of tables, verbatim and MetaPost images takes around 11 seconds. A few milliseconds more or less don't really show here. I only time these runs because I want to make sure that there are no dramatic consequences.

```
N6<-n
U7<-u
T8<-t
H9<-h
```

The reason for the difference, as mentioned, is that the tracing uses the template and therefore uses the stored match token, while the meaning uses the reference match tokens that carry the number and at that time has no access to the original match token. Keeping track of that for the sake of tracing would not make sense anyway. So, traditional TEX, which is what pdfTEX is very close to, uses the last used match token, the H. Maybe this example can convince you that dropping that bit of log related compatibility is not that much of a problem. I just tell myself that I turned an unwanted side effect into a new feature.

A few side notes

The fact that characters can be given a special meaning is one of the charming properties of TEX. Take these two cases:

```
\bgroup\catcode`\&=5 &\egroup
\bgroup\catcode`\!=5 !\egroup
```

In both lines there is now an alignment character used outside an alignment. And, in both cases the error message is similar:

```
! Misplaced alignment tab character &
! Misplaced alignment tab character !
```

So, indeed the right character is shown in the message. But, as soon as you ask for help, there is a difference: in the first case the help is specific for a tab character, but in the second case a more generic explanation is given. Just try it.

The reason is an explicit check for the ampersand being used as tab character. Such is the charm of TEX. I'll probably opt for a trivial change to be consistent here, although in ConTEXt the ampersand is just an ampersand so no user will notice.

There are a few more places where, although in principle any character can serve any purpose, there are hard coded assumptions, like $ being used for math, so a missing dollar is reported, even if math started with another character being used to enter math mode. This makes sense because there is no urgent need to keep track of what specific character was used for entering math mode. An even stronger argument could be that TEXies expect dollars to be used for that purpose. Of course this works fine:

```
\catcode`€=\catcode`$
€ \sqrt{x^3} €
```

But when we forget an € we get messages like:

```
! Missing $ inserted
```

or more generic:

```
! Extra }, or forgotten $
```

which is definitely a confirmation of "America first". Of course we can compromise in display math because this is quite okay:

```
\catcode`€=\catcode`$
$€ \sqrt{x^3} €$
```

unless of course we forget the last dollar in which case we are told that

```
! Display math should end with $$
```

so no matter what, the dollar wins. Given how ugly the Euro sign looks I can live with this, although I always wonder what character would have been taken if TEX was developed in another country.

12 Parsing

The macro mechanism is TEX is quite powerful and once you understand the concept of mixing parameters and delimiters you can do a lot with it. I assume that you know what we're talking about, otherwise quit reading. When grabbing arguments, there are a few catches.

- When they are used, delimiters are mandate: TEX will go on reading an argument till the (current) delimiter condition is met. This means that when you forget one you end up with way more in the argument than expected or even run out of input.
- Because specified arguments and delimiters are mandate, when you want to parse input, you often need multi-step macros that first pick up the to be parsed input, and then piecewise fetch snippets. Bogus delimiters have to be appended to the original in order to catch a run away argument and checking has to be done to get rid of them when all is ok.

The first item can be illustrated as follows:

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

When `\foo` gets expanded TEX first looks for a `[` and then starts collecting tokens for parameter `#1`. It stops doing that when aa `]` is seen. So,

```
\starttext
    \foo[whatever
\stoptext
```

will for sure give an error. When collecting tokens, TEX doesn't expand them so the `\stoptext` is just turned into a token that gets appended.

The second item is harder to explain (or grasp):

```
\def\foo[#1=#2]{(#1/#2)}
```

Here we expect a key and a value, so these will work:

```
\foo[key=value]
\foo[key=]
```

while these will fail:

```
\foo[key]
\foo[]
```

unless we have:

```
\foo[key]=]
\foo[]=]
```

But, when processing the result, we then need to analyze the found arguments and correct for them being wrong. For instance, argument `#1` can become `]` or here `key]`. When indeed a valid key/value combination is given we need to get rid of the two 'fixup' tokens `=]`. Normally we will have multiple key/value pairs separated by a comma, and in practice we only need to catch the missing equal because we can ignore

empty cases. There are plenty of examples (rather old old code but also more modern variants) in the ConTEXt code base.

I will now show some new magic that is available in LuaMetaTEX as experimental code. It will be tested in lmtx for a while and might evolve in the process.

```
\def\foo#1=#2,{(#1/#2)}

\foo 1=2,\ignorearguments
\foo 1=2\ignorearguments
\foo 1\ignorearguments
\foo \ignorearguments
```

Here we pick up a key and value separated by an equal sign. We end the input with a special signal command: `\ignorearguments`. This tells the parser to quit scanning. So, we get this, without any warning with respect to a missing delimiter of running away:

(1/2)(1/2)(1/)(/)

The implementation is actually fairly simple and adds not much overhead. Alternatives (and I pondered a few) are just too messy, would remind me too much of those awful expression syntaxes, and definitely impact performance of macro expansion, therefore: a no-go.

Using this new feature, we can implement a key value parser that does a sequence. The prototypes used to get here made only use of this one new feature and therefore still had to do some testing of the results. But, after looking at the code, I decided that a few more helpers could make better looking code. So this is what I ended up with:

```
\def\grabparameter#1=#2,%
  {\ifarguments\or\or
     % (\whatever/#1/#2)\par%
     \expandafter\def\csname\namespace#1\endcsname{#2}%
     \expandafter\grabnextparameter
   \fi}

\def\grabnextparameter
  {\expandafterspaces\grabparameter}

\def\grabparameters[#1]#2[#3]%
  {\def\namespace{#1}%
   \expandafterspaces\grabparameter#3\ignorearguments\ignorearguments}
```

Now, this one actually does what the ConTEXt `\getparameters` command does: setting variables in a namespace. Being a parameter driven macro package this kind of macros have been part of ConTEXt since the beginning. There are some variants and we also need to deal with the multilingual interface. Actually, MkIV (and therefore lmtx) do things a bit different, but the same principles apply.

The `\ignorearguments` quits the scanning. Here we need two because we actually quit twice. The `\expandafterspaces` can be implemented in traditional TEX macros but I though it is nice to have it this way; the fact that I only now added it has more to do with cosmetics. One could use the already somewhat older extension `\futureexpandis` (which expands the second or third token depending seeing the first, in this variant ignoring spaces) or a bunch of good old primitives to do the same. The new conditional `\ifarguments` can be used to act upon the number of arguments given. It reflects the most recently expanded macro. There is also a `\lastarguments` primitive (that provides the number of arguments.

So, what are the benefits? You might think that it is about performance, but in practice there are not that many parameter settings going on. When I process the LuaMetaTEX manual, only some 5000 times one or more parameters are set. And even in a way more complex document that I asked my colleague to run I was a bit disappointed that only some 30.000 cases were reported. I know of users who have documents with hundreds of thousands of cases, but compared to the rest of processing this is not where the performance bottleneck is.[9] This means that a change in implementation like the above is not paying off in significantly better runtime: all these low level mechanisms in ConTEXt have been very well optimized over the years. And faster machines made old bottlenecks go away anyway. Take this use case:

```
\grabparameters
  [foo]
  [key0=value0,
   key1=value1,
   key2=value2,
   key3=value3]
```

After this, parameters can be accessed with:

```
\def\getvalue#1#2{\csname#1#2\endcsname}
```

used as:

```
\getvalue{foo}{key2}
```

which takes care of characters normally not permitted in macro names, like the digits in this example. Of course some namespace protection can be added, like adding a colon between the namespace and the key, but let's take just this one.

Some 10.000 expansions of the grabber take on my machine 0.045 seconds while the original `\getparameters` takes 0.090 so although for this case we're twice as fast, the 0.045 difference will not be noticed on a real run. After all, when these parameters are set some action will take place. Also, we don't actually use this macro for collecting settings with the `\setupsomething` commands, so the additional overhead that is involved adds a baseline to performance that can turn any gain into noise. But some users might notice some gain. Of course this observation might change once we apply this trickery in more places than parameter parsing, because I have to admit that there might be other places in the support macros where we can benefit: less code, double performance, but these are all support macros that made sense in MkII and not that much in MkIV or lmtx and are kept just for convenience and backward compatibility. Think of some list processing macros. So, as a kind of nostalgic trip I decided to rewrite some low level macros anyway, if only to see what is no longer used and/or to make the code base somewhat (c)leaner.

Elsewhere I introduce the `#0` argument indicator. That one will just gobbles the argument and does not store a token list on the stack. It saves some memory access and token recycling when arguments are not used. Another special indicator is `#+`. That one will flag an argument to be passed as-is. The `#-` variant will simply discard an argument and move on. The following examples demonstrate this:

```
\def\foo     [#1]{\detokenize{#1}}
\def\ofo     [#0]{\detokenize{#1}}
\def\oof     [#+]{\detokenize{#1}}
\def\fof[#1#-#2]{\detokenize{#1#2}}
```

[9] Think of thousands of pages of tables with cell settings applied.

```
\def\fff[#1#0#3]{\detokenize{#1#3}}

\meaning\foo\ : <\foo[{123}]> \crlf
\meaning\ofo\ : <\ofo[{123}]> \crlf
\meaning\oof\ : <\oof[{123}]> \crlf
\meaning\fof\ : <\fof[123]>   \crlf
\meaning\fff\ : <\fof[123]>   \crlf
```

This gives:

```
macro:[#1]->\detokenize {#1} : <123>
macro:[#0]->\detokenize {#1} : <>
macro:[#+]->\detokenize {#1} : <{123}>
macro:[#1#-#2]->\detokenize {#1#2} : <13>
macro:[#1#0#3]->\detokenize {#1#3} : <13>
```

When playing with new features like the one described here, it makes sense to use them in existing macros so that they get well tested. Some of the low level system files come in different versions: for MkII, MkIV and lmtx. The MkII files often also have the older implementations, so they are also good for looking at the history. The lmtx files can be leaner and meaner than the MkIV files because they use the latest features.[10]

When I was rewriting some of these low level MkIV macros using the newer features, at some point I wondered why I still had to jump through some hoops. Why not just add some more primitives to deal with that? After all, LuaTEX and LuaMetaTEX already have more primitives that are helpful in parsing, so a few dozen more lines don't hurt. As long as these primitives are generic and not that specific. In this particular case we talk about two new conditionals (in addition to the already present comparison primitives):

```
\ifhastok    <token>       {<token list>}
\ifhastoks  {<token list>} {<token list>}
\ifhasxtoks {<token list>} {<token list>}
```

You can probably guess what they do from their names. The last one is the expandable variant of the second one. The first one is the fast one. When playing with these I decided to redo the set checker. In MkII that one is done in good old TEX, in MkIV we use Lua. So, how about going back to TEX?

```
\ifhasxtoks {cd} {abcdef}
```

This check is true. But that doesn't work well with a comma separated list, but there is a way out:

```
\ifhasxtoks {,cd,} {,ab,cd,ef,}
```

However, when I applied that a user reported that it didn't handle optional spaces before commas. So how do we deal with such optional characters tokens?

```
\def\setcontains#1#2{\ifhasxtoks{,#1,}{,#2,}}

\ifcondition\setcontains{cd}{ab,cd,ef}YES \else NO \fi
\ifcondition\setcontains{cd}{ab, cd, ef}YES \else NO \fi
```

[10] Some 70 primitives present in LuaTEX are not in LuaMetaTEX. On the other hand there are also about 70 new primitives. Of those gone, most concerned the backend, fonts or no longer relevant features from other engines. Of those new, some are really new primitives (conditionals, expansion magic), some control previously hardwired behaviour, some give access to properties of for instance boxes, and some are just variants of existing ones but with options for control.

We get:

YES NO

The \ifcondition is an old one. When nested in a condition it will be seen as an \if... by the fast skipping scanner, but when expanded it will go on and a following macro has to expand to a proper condition. That said, we can take care of the optional space by entering some new territory. Look at this:

```
\def\setcontains#1#2{\ifhasxtoks{,\expandtoken 9 "20 #1,}{,#2,}}

\ifcondition\setcontains{cd}{ab,cd,ef}YES \else NO \fi
\ifcondition\setcontains{cd}{ab, cd, ef}YES \else NO \fi
```

We get:

YES YES

So how does that work? The \expandtoken injects a space token with catcode 9 which means that it is in the to be ignored category. When a to be ignored token is seen, and the to be checked token is a character (letter, other, space or ignored) then the character code will be compared. When they match, we move on, otherwise we just skip over the ignored token (here the space).

In the ConTEXt code base there are already files that are specific for MkIV and lmtx. The most visible difference is that we use the \orelse primitive to construct nicer test trees, and we also use some of the additional \future... and \expandafter... features. The extensions discussed here make for the most recent differences (we're talking end May 2020).

After implementing this trick I decided to look at the macro definition mechanism one more time and see if I could also use this there. Before I demonstrate another next feature, I will again show the argument extensions, this time with a fourth variant:

```
\def\TestA#1#2#3{{(#1)(#2)(#3)}}
\def\TestB#1#0#3{(#1)(#2)(#3)}
\def\TestC#1#+#3{(#1)(#2)(#3)}
\def\TestD#1#-#2{(#1)(#2)}
```

The last one specifies a to be thrashed argument: #-. It goes further than the second one (#0) which still keeps a reference. This is why in this last case the third argument gets number 2. The meanings of these four are:

```
macro:#1#2#3->{(#1)(#2)(#3)}
macro:#1#0#3->(#1)(#2)(#3)
macro:#1#+#3->(#1)(#2)(#3)
macro:#1#-#2->(#1)(#2)
```

There are some subtle differences between these variants, as you can see from the following examples:

```
\TestA1{\red 2}3
\TestB1{\red 2}3
\TestC1{\red 2}3
\TestD1{\red 2}3
```

Here you also see the side effect of keeping the braces. The zero argument (#0) is ignored, and the thrashed argument (#-) can't even be accessed.

```
(1)(2)(3)
(1)()(3)
(1)(2)(3)
(1)(3)
```

In the next example we see two delimiters being used, a comma and a space, but they have catcode 9 which flags them as ignored. This is a signal for the parser that both the comma and the space can be skipped. The zero arguments are still on the parameter stack, but the thrashed ones result in a smaller stack, not that the later matters much on today's machines.

```
\normalexpanded {
    \def\noexpand\foo
    \expandtoken 9 "2C % comma
    \expandtoken 9 "20 % space
    #1=#2]%
}{(#1)(#2)}
```

This means that the next tree expansions won't bark:

```
\foo,key=value]
\foo, key=value]
\foo  key=value]
```

or expanded:

```
(key)(value)
(key)(value)
(key)(value)
```

Now, why didn't I add these primitives long ago already? After all, I already added dozens of new primitives over the years. To quote Andrew Cuomo, what follows now are opinions, not facts.

Decades ago, when TEX showed up, there was no Internet. I remember that I got my first copy on floppy disks. Computers were slow and memory was limited. The TEXbook was the main resource and writing macros was a kind of art. One could not look up solutions, so trial and error was a valid way to go. Figuring out what was efficient in terms of memory consumption and runtime was often needed too. I remember meetings where one was not taken serious when not talking in the right 'token', 'node', 'stomach' and 'mouth' speak. Suggesting extensions could end up in being told that there was no need because all could be done in macros or even arguments of the "who needs that". I must admit that nowadays I wonder to what extend that was related to extensions taking away some of the craftmanship and showing off. In a way it is no surprise that (even trivial to implement) extensions never surfaced. Of course then the question is: will extensions that once were considered not of importance be used today? We'll see.

Let's end by saying that, as with other experiments, I might port some of the new features in LuaMetaTEX to LuaTEX, but only after they have become stable and have been tested in lmtx for quite a while.

13 Tokens

This is mostly a wrapup of some developments, and definitely not a tutorial.

Talking deep down TEX is talking about tokens and nodes. Roughly spoken, from the perspective of the user, tokens are what goes in and stays in (as macro, token list of whatever) and nodes is what get produced and eventually results in output. A character in the input becomes one token (before expansion) and a control sequence like `\foo` also is turned into a token. Tokens can be linked into lists. This actually means that in the engine we can talk of tokens in two ways: the single item with properties that trigger actions, or as compound item with that item and a pointer to the next token (called link). In Lua speak token memory can be seen as:

```
fixmem = {
    { info, link },
    { info, link },
    { info, link },
    { info, link },
    ....
}
```

Both are 32 bit integers. The `info` is a combination of a command code (an operator) and a so called chr code (operand) and these determine its behaviour. For instance the command code can indicate an integer register and the chr code then indicates the number of that register. So, like:

```
fixmem = {
    { { cmd, chr}, index_into_fixmem },
    { { cmd, chr}, index_into_fixmem },
    { { cmd, chr}, index_into_fixmem },
    { { cmd, chr}, index_into_fixmem },
    ....
}
```

In the following line the characters that make three words are tokens (letters), so are the space (spacer), the curly braces (begin- and endgroup token) and the bold face switch (which becomes one token which resolves to a token list of tokens that trigger actions (in this case switching to a bolder font).

```
foo {\bf bar} foo
```

When TEX reads a line of input tokens are expanded immediately but a sequence can also become part fo a macro body or token list. Here we have $3_{\texttt{foo}} + 1 + 1_{\texttt{\{}} + 1_{\texttt{\bf}} + 3_{\texttt{bar}} + 1_{\texttt{\}}} + 1 + 3_{\texttt{foo}} = 14$ tokens.

A control sequence normally starts with a backslash. Some are built in, these are called primitives, and others are defined by the macro package or the user. There is a lookup table that relates the tokenized control sequence to some action. For instance:

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

creates an entry that leads (directly or following a hash chain) to the three letter token list. Every time the input sees `\foo` it gets resolved to that list via a hash lookup. However, once internalized and part of a token list, it is a direct reference. On the other hand,

```
\the\count0
```

triggers the `\the` action that relates to this control sequence, which then reads a next token and operates on that. That next token itself expects a number as follow up. In the end the value of `\count0` is found and that one is also in the so called equivalent lookup table, in what TEX calls specific regions.

```
equivalents = {
    { level, type, value },
    { level, type, value },
    { level, type, value },
    ...
}
```

The value is in most cases similar to the info (cmd & chr) field in fixmem, but one difference is that counters, dimensions etc directly store their value, which is why we sometimes need the type separately, for instance in order to reclaim memory for glue or node specifications. It sound complicated and it is, but as long as you get a rough idea we can continue. Just keep in mind that tokens sometimes get expanded on the fly, and sometimes just get stored.

There are a lot of primitives and each has a unique info. The same is true for characters (each category has its own command code, so regular letters can be distinguished from other tokens, comment signs, math triggers etc). All important basic bits are in table of equivalents: macros as well as registers although the meaning of a macro and content of token lists lives in the fixmem table and the content of boxes in so called node lists (nodes have their own memory).

In traditional TEX the lookup table for primitives, registers and macros is as compact as can be: it is an array of so called 32 bit memory words. These can be divided into halfs and quarters, so in the source you find terms like `halfword` and `quarterword`. The lookup table is a hybrid:

```
[level 8] [type 8] [value 16] | [equivalent 32]
[level 8] [type 8] [value 16] | [equivalent 32]
[level 8] [type 8] [value 16] | [equivalent 32]
...
```

The mentioned counters and such are directly encoded in an equivalent and the rest is a combination of level, type and value. The level is used for the grouping, and in for instance pdfTEX there can therefore be at most 255 levels. In LuaTEX we use a wider model. There we have 64 bit memory words which means that we have way more levels and don't need to have this dual nature:

```
[level 16] [type 16] [value 32]
[level 16] [type 16] [value 32]
[level 16] [type 16] [value 32]
...
```

We already showed a Lua representation. The type in this table is what a command code is in an 'info' field. In such a token the integer encodes the command as well as a value (called chr). In the lookup table the type is the command code. When TEX is dealing with a control sequences it looks at the type, otherwise it filters the command from the token integer. This means that a token cannot store an integer (or dimension), but the lookup table actually can do that. However, commands can limit the range, for instance characters are bound by what Unicode permits.

Internally, LuaTEX still uses these ranges of fast accessible registers, like counters, dimensions and attributes. However, we saw that in LuaTEX they don't overlap with the level and type. In LuaTEX, at

least till version 1.13 we still have the shadow array for levels but in LuaMetaTEX we just use those in the equivalents lookup table. If you look in the Pascal source you will notice that arrays run from `[somemin ... somemax]` which in the C source would mean using offsets. Actually, the shadow array starts at zero so we waste the part that doesn't need shadowing. It is good to remind ourselves that traditional TEX is 8 bit character based.

The equivalents lookup table has all kind of special ranges (combined into regions of similar nature, in TEX speak), like those for lowercase mapping, specific catcode mappings, etc. but we're still talking of $n \times 256$ entries. In LuaTEX all these mappings are in dedicated sparse hash tables because we need to support the full Unicode repertoire. This means that, while on the one hand LuaTEX uses more memory for the lookup table the number of slots can be less. But still there was the waste of the shadow level table: I didn't calculate the exact saving of ditching that one, but I bet it came close to what was available as total memory for programs and data on the first machines that I used for running TEX. But ... after more than a decade of LuaTEX we now reclaimed that space in LuaMetaTEX.[11]

Now, in case you're interested (and actually I just write it down because I don't want to forget it myself) the lookup table in LuaMetaTEX is layout as follows

the hash table	
some frozen primitives	
current and defined fonts	one slot + many pointers
undefined control sequence	one slot
internal and register glue	pointer to node
internal and register muglue	pointer to node
internal and register toks	pointer to token list
internal and register boxes	pointer to node list
internal and register counts	value in token
internal and register attributes	value in token
internal and register dimens	value in token
some special data structures	pointer to node list
the (runtime) extended hash table	

Normally a user doesn't need to know anything about these specific properties of the engine and it might comfort you to know that for a long time I could stay away from these details. One difference with the other engines is that we have internal variables and registers split more explicitly. The special data structures have their own slots and are not just put somewhere (semi random). The initialization is bit more granular in that we properly set the types (cmd codes) for registers which in turn is possible because for instance we're able to distinguish glue types. This is all part of coming up with a bit more consistent interface to tokens from the Lua end. It also permits diagnostics.

Anyway, we now are ready for some more details about tokens. You don't need to understand all of it in order to define decent macros. But when you are using LuaTEX and do want to mess around here is some insight. Assume we have defined these macros:

```
\def\MacroA{a} \def\MacroB{b}
\def\macroa{a} \def\macrob{b}
\def\MACROa{a} \def\MACROb{b}
```

How does that end up internally?

[11] Don't expect a gain in performance, although using less memory might pay back on a virtual machine or when TEX has to share the cpu cache.

	cmd	name	chr	cs	rawchr
\scratchcounterone	80	register_int	260	75231	459190
\scratchcountertwo	80	register_int	261	75279	459191
\scratchdimen	84	register_dimen	257	2162	590283
\scratchtoks	78	register_toks	257	655603	327989
\scratchcounter	80	register_int	257	9310	459187
\letterpercent	128	call	0	101735	2708
\everypar	79	internal_toks	1	27262	327720
\%	74	char_given	37	39	
\pagegoal	93	set_page_property	0	27028	
\pagetotal	93	set_page_property	1	54424	
\hangindent	85	internal_dimen	17	105414	590020
\hangafter	81	internal_int	42	52526	458849
\dimdim	116	undefined_cs	0	196624	
\relax	0	relax	0	3380	1114112
\dimen	102	register	2	3190	
\stoptext	129	protected_call	0	29178	57963
\MacroA	128	call	0	5553	672102
\MacroB	128	call	0	667022	667248
\MacroC	116	undefined_cs	0	196624	
\macroa	128	call	0	6609	671560
\macrob	128	call	0	667023	665323
\macroc	116	undefined_cs	0	196624	
\MACROa	128	call	0	4625	661596
\MACROb	128	call	0	4626	671446
\MACROc	116	undefined_cs	0	196624	

We show the raw chr value but in the Lua interface these are normalized to for instance proper register indices. This is because the raw numbers can for instance be indices into memory or some Unicode reference with catcode specific bits set. But, while these indices are real and stable, these offsets can actually change when the implementation changes. For that reason, in LuaMetaTEX we can better talk of command codes as main indicator and:

subcommand	for tokens that have variants, like `\ifnum`
register indices	for the 64K register banks, like `\count 0`
internal indices	for internal variables like `\parindent`
characters	specific Unicode slots combined with catcode
pointers	to token lists, macros, Lua functions, nodes

This so called `cs` number is a pointer into the table of equivalents. That number results comes from the hash table. A macro name, when scanned the first time, is still a sequence of bytes. This sequence is used to compute a hash number, which is a pointer to a slot in the lower part of the hash (lookup) table. That slot points to a string and a next hash entry in the higher end. A lookup goes as follows:

1. compute the index into the hash table from the string
2. goto the slot with that index and compare the `string` field
3. when there is no match goto the slot indicated by the `next` field
4. compare again and keep following `next` fields till there is no follow up
5. optionally create a new entry
6. use the index of that entry as index in the table of equivalents

So, in Lua speak, we have:

```
hashtable = {
    -- lower part, accessed via the calculated hash number
    { stringpointer, nextindex },
    { stringpointer, nextindex },
    ...
    -- higher part, accessed by following nextindex
    { stringpointer, nextindex },
    { stringpointer, nextindex },
    ...
}
```

Eventually, after following a lookup chain in the hash tabl;e, we end up at pointer to the equivalents lookup table that we already discussed. From then on we're talking tokens. When you're lucky, the list is small and you have a quick match. The maximum initial hash index is not that large, around 64K (double that in LuaMetaTEX), so in practice there will often be some indirect (multi-compare) match but increasing the lower end of the hash table might result in less string comparisons later on, but also increases the time to calculate the initial hash needed for accessing the lower part. Here you can sort of see that:

```
\dostepwiserecurse{`a}{`z}{1}{
    \expandafter\def\csname whatever\Uchar#1\endcsname
      {}
}
\dostepwiserecurse{`a}{`z}{1}{
    \expandafter\let\csname somemore\Uchar#1\expandafter\endcsname
        \csname whatever\Uchar#1\endcsname
}
```

	cmd	name	chr	cs	rawchr
\whatevera	128	call	0	667029	671318
\somemorea	128	call	0	667043	671318
\whateverb	128	call	0	667030	665501
\somemoreb	128	call	0	57246	665501
\whateverc	128	call	0	56989	671829
\somemorec	128	call	0	667044	671829
\whateverd	128	call	0	667031	672028
\somemored	128	call	0	667045	672028
\whatevere	128	call	0	667032	671421
\somemoree	128	call	0	667046	671421
\whateverf	128	call	0	667033	671328
\somemoref	128	call	0	57250	671328
\whateverg	128	call	0	56993	671449
\somemoreg	128	call	0	667047	671449
\whateverh	128	call	0	667034	665496
\somemoreh	128	call	0	667048	665496
\whateveri	128	call	0	56995	671345
\somemorei	128	call	0	57253	671345
\whateverj	128	call	0	667035	665486
\somemorej	128	call	0	57254	665486

```
\whateverk   128  call     0   56997  665483
\somemorek   128  call     0   57255  665483
\whateverl   128  call     0   56998  666046
\somemorel   128  call     0  667049  666046
\whateverm   128  call     0   56999  671424
\somemorem   128  call     0   57257  671424
\whatevern   128  call     0   57000  671293
\somemoren   128  call     0   57258  671293
\whatevero   128  call     0   57001  665550
\somemoreo   128  call     0   57259  665550
\whateverp   128  call     0  667036  665494
\somemorep   128  call     0   57260  665494
\whateverq   128  call     0   57003  671398
\somemoreq   128  call     0   57261  671398
\whateverr   128  call     0  667037  665493
\somemorer   128  call     0  667050  665493
\whatevers   128  call     0  667038  667294
\somemores   128  call     0  667051  667294
\whatevert   128  call     0  667039  671332
\somemoret   128  call     0   57264  671332
\whateveru   128  call     0  667040  671242
\somemoreu   128  call     0   57265  671242
\whateverv   128  call     0  667041  672085
\somemorev   128  call     0   57266  672085
\whateverw   128  call     0   57009  672091
\somemorew   128  call     0   57267  672091
\whateverx   128  call     0   57010  666064
\somemorex   128  call     0   57268  666064
\whatevery   128  call     0   57011  666646
\somemorey   128  call     0   57269  666646
\whateverz   128  call     0  667042  603553
\somemorez   128  call     0   57270  603553
```

The command code indicates a macro and the action related to it is an expandable call. We have no sub command[12] so that column shows zeros. The fifth column is the hash entry which can bring us back to the verbose name as needed in reporting while the last column is the index to into token memory (watch the duplicates for `\let` macros: a ref count is kept in order to be able to manage such shared references). When you look a the cs column you will notice that some numbers are close which (I think) in this case indicates some closeness in the calculated hash name and followed chain.

It will be clear that it is best to not make any assumptions with respect to the numbers which is why, in LuaMetaTEX we sort of normalize them when accessing properties.

field	meaning
command	operator
cmdname	internal name of operator
index	sanitized operand
mode	original operand

[12] We cheat a little here because chr actually is an index into token memory but we don't show them as such.

csname	associated name
id	the index in token memory (a virtual address)
tok	the integer representation

active	true when an active character
expandable	true when expandable command
protected	true when a protected command
frozen	true when a frozen command
user	true when a user defined command

When a control sequence is an alias to an existing primitive, for instance made by `\let`, the operand (chr) picked up from its meaning. Take this:

```
\newif\ifmyconditionone
\newif\ifmyconditiontwo

                    \meaning\ifmyconditionone    \crlf
                    \meaning\ifmyconditiontwo    \crlf
                    \meaning\myconditiononetrue  \crlf
                    \meaning\myconditiontwofalse \crlf
\myconditiononetrue \meaning\ifmyconditionone    \crlf
\myconditiontwofalse\meaning\ifmyconditiontwo    \crlf
```

```
\iffalse
\iffalse
macro:->\let \ifmyconditionone \iftrue
macro:->\let \ifmyconditiontwo \iffalse
\iftrue
\iffalse
```

Internally this is:

```
                  cmd  name     chr      cs
\ifmyconditionone  123  if_test   23  667054
\ifmyconditiontwo  123  if_test   24   19025
\iftrue            123  if_test   23    6713
\iffalse           123  if_test   24   13157
```

The whole list of available commands is given below. Once they are stable the LuaMetaTEX manual will document the accessors. In this chapter we use:

```
kind, min, max, fixedvalue token.get_range("primitive")
cmd, chr, cs  = token.get_cmdchrcs("primitive")
```

The kind of command is given in the first column, which can have the following values:

0	no	not accessible
1	regular	possibly with subcommand
2	character	the Unicode slot is encodes in the the token
3	register	this is an indexed register (zero upto 64K)
4	internal	this is an internal register (range given)
5	reference	this is a reference to a node, Lua function, etc.

6 data a general data entry (kind of private)
7 token a token reference (that can have a followup)

	cmd	name	min	max	default or subcommands
2	0	relax	0	0x10FFFF	0x110000
2	1	left_brace	0	0x10FFFF	
2	2	right_brace	0	0x10FFFF	
2	3	math_shift	0	0x10FFFF	
2	4	tab_mark	0	0x10FFFF	
2	5	car_ret	0	0x10FFFF	
2	6	mac_param	0	0x10FFFF	
2	7	sup_mark	0	0x10FFFF	
2	8	sub_mark	0	0x10FFFF	
2	9	ignore	0	0x10FFFF	
2	10	spacer	0	0x10FFFF	
2	11	letter	0	0x10FFFF	
2	12	other_char	0	0x10FFFF	
2	13	par_end	0	0x10FFFF	0x110000
1	14	stop	0	1	0=end 1=dump
1	15	delim_num	0	1	0=delimiter 1=Udelimiter
2	16	char_num	0	0x10FFFF	
1	17	math_char_num	0	3	0=mathchar 1=Umathchar 2=Umathcharnum 3=Umathclass
1	18	mark	0	2	0=mark 1=marks 2=clearmarks
8	19	node			
1	20	xray	0	6	0=show 1=showbox 2=showthe 3=showlists 4=showgroups 5=showtokens 6=showifs
1	21	make_box	0	7	0=box 1=copy 2=lastbox 3=vsplit 4=tpack 5=vpack 6=hpack 7=vtop
1	22	hmove	0	1	0=moveright 1=moveleft
1	23	vmove	0	1	0=lower 1=raise
1	24	un_hbox	0	1	0=unhbox 1=unhcopy
1	25	un_vbox	0	1	0=pagediscards 1=unvcopy
1	26	remove_item	0	2	0=unkern 1=unpenalty 2=unskip
1	27	hskip	0	4	0=hfil 1=hfill 2=hss 3=hfilneg 4=hskip
1	28	vskip	0	4	0=vfil 1=vfill 2=vss 3=vfilneg 4=vskip
1	29	mskip	0	0	0=mskip
1	30	kern	0	0	0=kern
1	31	mkern	0	0	0=mkern
1	32	leader_ship	0	5	0=shipout 1=<unavailable> 2=leaders 3=cleaders 4=xleaders 5=gleaders
1	33	halign	0	0	0=halign
1	34	valign	0	0	0=valign
1	35	no_align	0	0	0=noalign
1	36	vrule	0	1	0=vrule 1=novrule
1	37	hrule	0	1	0=hrule 1=nohrule
1	38	insert	0	0	0=insert
1	39	vadjust	0	0	0=vadjust
1	40	ignore_something	0	2	0=ignorespaces 1=ignorepars 2=ignorearguments
1	41	after_something	0	5	0=aftergroup 1=afterassignment 2=atendofgroup 3=aftergrouped 4=afterassigned 5=atendofgrouped
1	42	break_penalty	0	0	0=penalty
1	43	start_par	0	2	0=noindent 1=indent 2=quitvmode
1	44	ital_corr	0	0	0=/
1	45	accent	0	0	0=accent
1	46	math_accent	0	1	0=mathaccent 1=Umathaccent
1	47	discretionary	0	2	0=discretionary 1=- 2=automaticdiscretionary
1	48	eq_no	0	1	0=<unavailable> 1=<unavailable>
1	49	left_right	1	7	1=left 2=middle 3=right 4=Uvextensible 5=Uleft 6=Umiddle 7=Uright
1	50	math_comp	0	9	0=mathord 1=mathop 2=mathbin 3=mathrel 4=mathopen 5=mathclose 6=mathpunct 7=mathinner 8=underline 9=overline
1	51	limit_switch	0	3	0=displaylimits 1=limits 2=nolimits 3=ordlimits
1	52	above	0	7	0=Uskewedwithdelims 1=over 2=atop 3=Uskewed 4=Uabove 5=Uover 6=Uatop 7=UUskewed
1	53	math_style	0	7	0=displaystyle 1=crampeddisplaystyle 2=textstyle 3=crampedtextstyle 4=scriptstyle 5=crampedscriptstyle 6=scriptscriptstyle 7=cramped-scriptscriptstyle
1	54	math_choice	0	1	0=mathchoice 1=Ustack
1	55	non_script	0	0	0=nonscript
1	56	vcenter	0	0	0=vcenter
1	57	case_shift	0	1	0=lowercase 1=uppercase
1	58	message	0	1	0=message 1=errmessage
1	59	catcode_table	0	1	0=savecatcodetable 1=initcatcodetable

```
1    60  end_local                0         0  0=endlocalcontrol
5    61  lua_function_call        0  0x1FFFFF
5    62  lua_call                 0  0x1FFFFF
1    63  in_stream                0         1  0=closein 1=openin
1    64  begin_group              0         0  0=begingroup
1    65  end_group                0         0  0=endgroup
1    66  omit                     0         0  0=omit
1    67  ex_space                 0         0  0=<space>
1    68  boundary                 0         3  0=noboundary 1=boundary 2=protrusionboundary 3=wordboundary
1    69  radical                  0         7  0=radical 1=Uradical 2=Uroot 3=Uunderdelimiter 4=Uoverdelimiter 5=Udelim-
                                               iterunder 6=Udelimiterover 7=Uhextensible
1    70  super_sub_script         0         7  0=Usubscript 1=Usuperscript 2=Usuperprescript 3=Usubprescript 4=Unosubscript
                                               5=Unosuperscript 6=Unosubprescript 7=Unosuperprescript
1    71  math_shift_cs            0         3  0=Ustartmath 1=Ustopmath 2=Ustartdisplaymath 3=Ustopdisplaymath
1    72  end_cs_name              0         0  0=endcsname
1    73  set_local_box            0         1  0=localleftbox 1=localrightbox
2    74  char_given               0  0x10FFFF
2    75  math_given               0  0x10FFFF
2    76  math_xgiven              0  0x10FFFF
1    77  some_item                0        40  0=lastpenalty 1=lastkern 2=lastskip 3=lastnodetype 4=lastnodesubtype 5=in-
                                               putlineno 6=badness 7=luatexversion 8=luatexrevision 9=currentgrouplevel
                                               10=currentgrouptype 11=currentiflevel 12=currentiftype 13=currentifbranch
                                               14=gluestretchorder 15=glueshrinkorder 16=fontid 17=fontcharwd 18=fontcharht
                                               19=fontchardp 20=fontcharic 21=mathstyle 22=Umathcharclass 23=Umathcharfam
                                               24=Umathcharslot 25=lastarguments 26=luavaluefunction 27=insertht 28=left-
                                               marginkern 29=rightmarginkern 30=parshapelength 31=parshapeindent 32=par-
                                               shapedimen 33=gluestretch 34=glueshrink 35=mutoglue 36=gluetomu 37=numexpr
                                               38=dimexpr 39=glueexpr 40=muexpr
3    78  register_toks            0    0xFFFF
4    79  internal_toks            0        10  0=output 1=everypar 2=everymath 3=everydisplay 4=everyhbox 5=everyvbox
                                               6=everyjob 7=everycr 8=everytab 9=errhelp 10=everyeof
3    80  register_int             0    0xFFFF
4    81  internal_int             0       119  0=pretolerance 1=tolerance 2=linepenalty 3=hyphenpenalty 4=exhyphenpenalty
                                               5=clubpenalty 6=widowpenalty 7=displaywidowpenalty 8=brokenpenalty 9=binop-
                                               penalty 10=relpenalty 11=predisplaypenalty 12=postdisplaypenalty 13=inter-
                                               linepenalty 14=doublehyphendemerits 15=finalhyphendemerits 16=adjdemer-
                                               its 17=mag 18=delimiterfactor 19=looseness 20=time 21=day 22=month 23=year
                                               24=showboxbreadth 25=showboxdepth 26=shownodedetails 27=hbadness 28=vbad-
                                               ness 29=pausing 30=tracingonline 31=tracingmacros 32=tracingstats 33=trac-
                                               ingparagraphs 34=tracingpages 35=tracingoutput 36=tracinglostchars 37=trac-
                                               ingcommands 38=tracingrestores 39=uchyph 40=outputpenalty 41=maxdeadcycles
                                               42=hangafter 43=floatingpenalty 44=globaldefs 45=fam 46=escapechar 47=de-
                                               faulthyphenchar 48=defaultskewchar 49=endlinechar 50=newlinechar 51=language
                                               52=lefthyphenmin 53=righthyphenmin 54=holdinginserts 55=errorcontextlines
                                               56=localinterlinepenalty 57=localbrokenpenalty 58=noligs 59=nokerns 60=no-
                                               spaces 61=catcodetable 62=outputbox 63=setlanguage 64=exhyphenchar 65=ad-
                                               justspacing 66=adjustspacingstep 67=adjustspacingstretch 68=adjustspac-
                                               ingshrink 69=protrudechars 70=tracingfonts 71=tracingassigns 72=tracing-
                                               groups 73=tracingifs 74=tracingscantokens 75=tracingnesting 76=predisplay-
                                               direction 77=lastlinefit 78=savingvdiscards 79=savinghyphcodes 80=mathe-
                                               qnogapstep 81=mathdisplayskipmode 82=mathscriptsmode 83=mathnolimitsmode
                                               84=mathrulesmode 85=mathrulesfam 86=mathitalicsmode 87=shapemode 88=first-
                                               validlanguage 89=hyphenationbounds 90=mathsurroundmode 91=predisplaygap-
                                               factor 92=hyphenpenaltymode 93=automatichyphenpenalty 94=explicithyphen-
                                               penalty 95=automatichyphenmode 96=compoundhyphenmode 97=breakafterdirmode
                                               98=exceptionpenalty 99=prebinoppenalty 100=prerelpenalty 101=mathpenalties-
                                               mode 102=mathdelimitersmode 103=mathscriptboxmode 104=mathscriptcharmode
                                               105=mathrulethicknessmode 106=mathflattenmode 107=luacopyinputnodes 108=fix-
                                               upboxesmode 109=glyphdimensionsmode 110=internalcodesmode 111=supmarkmode
                                               112=glyphdatafield 113=glyphstatefield 114=glyphscriptfield 115=matholdmode
                                               116=pardirection 117=textdirection 118=mathdirection 119=linedirection
3    82  register_attr            0    0xFFFF
0    83  internal_attr
3    84  register_dimen           0    0xFFFF
4    85  internal_dimen           0        21  0=parindent 1=mathsurround 2=lineskiplimit 3=hsize 4=vsize 5=maxdepth
                                               6=splitmaxdepth 7=boxmaxdepth 8=hfuzz 9=vfuzz 10=delimitershortfall
                                               11=nulldelimiterspace 12=scriptspace 13=predisplaysize 14=displaywidth
                                               15=displayindent 16=overfullrule 17=hangindent 18=<unavailable> 19=<unavail-
                                               able> 20=emergencystretch 21=pxdimen
3    86  register_glue            0    0xFFFF
```

```
4    87  internal_glue            0        15  0=lineskip 1=baselineskip 2=parskip 3=abovedisplayskip 4=belowdisplayskip
                                               5=abovedisplayshortskip 6=belowdisplayshortskip 7=leftskip 8=rightskip
                                               9=topskip 10=splittopskip 11=tabskip 12=spaceskip 13=xspaceskip 14=parfill-
                                               skip 15=mathsurroundskip
3    88  register_mu_glue         0    0xFFFF
4    89  internal_mu_glue         1         3  1=thinmuskip 2=medmuskip 3=thickmuskip
5    90  lua_value                0  0x1FFFFF
1    91  set_font_property        0         5  0=hyphenchar 1=skewchar 2=lpcode 3=rpcode 4=efcode 5=fontdimen
1    92  set_aux                  0         2  0=spacefactor 1=prevdepth 2=prevgraf
1    93  set_page_property        0        10  0=pagegoal 1=pagetotal 2=pagestretch 3=pagefilstretch 4=pagefillstretch
                                               5=pagefilllstretch 6=pageshrink 7=pagedepth 8=deadcycles 9=insertpenalties
                                               10=interactionmode
1    94  set_box_property         0        10  0=wd 1=ht 2=dp 3=boxdirection 4=boxorientation 5=boxxoffset 6=boxyoffset
                                               7=boxxmove 8=boxymove 9=boxtotal 10=boxattr
7    95  set_specification
1    96  def_char_code            0         9  0=catcode 1=lccode 2=uccode 3=sfcode 4=mathcode 5=Umathcode 6=Umathcodenum
                                               7=delcode 8=Udelcode 9=Udelcodenum
1    97  def_family               0         2  0=textfont 1=scriptfont 2=scriptscriptfont
1    98  set_math_param           0       114  0=Umathquad 1=Umathaxis 2=Umathspacingmode 3=Umathoperatorsize 4=Umathover-
                                               barkern 5=Umathoverbarrule 6=Umathoverbarvgap 7=Umathunderbarkern 8=Umath-
                                               underbarrule 9=Umathunderbarvgap 10=Umathradicalkern 11=Umathradicalrule
                                               12=Umathradicalvgap 13=Umathradicaldegreebefore 14=Umathradicaldegreeafter
                                               15=Umathradicaldegreeraise 16=Umathstackvgap 17=Umathstacknumup 18=Umath-
                                               stackdenomdown 19=Umathfractionrule 20=Umathfractionnumvgap 21=Umathfrac-
                                               tionnumup 22=Umathfractiondenomvgap 23=Umathfractiondenomdown 24=Umathfrac-
                                               tiondelsize 25=Umathskewedfractionhgap 26=Umathskewedfractionvgap 27=Umath-
                                               limitabovevgap 28=Umathlimitabovebgap 29=Umathlimitabovekern 30=Umathlimit-
                                               belowvgap 31=Umathlimitbelowbgap 32=Umathlimitbelowkern 33=Umathnolimitsub-
                                               factor 34=Umathnolimitsupfactor 35=Umathunderdelimitervgap 36=Umathunderde-
                                               limiterbgap 37=Umathoverdelimitervgap 38=Umathoverdelimiterbgap 39=Umathsub-
                                               shiftdrop 40=Umathsupshiftdrop 41=Umathsubshiftdown 42=Umathsubsupshiftdown
                                               43=Umathsubtopmax 44=Umathsupshiftup 45=Umathsupbottommin 46=Umathsupsubbot-
                                               tommax 47=Umathsubsupvgap 48=Umathspacebeforescript 49=Umathspaceafterscript
                                               50=Umathconnectoroverlapmin 51=Umathordordspacing 52=Umathordopspacing
                                               53=Umathordbinspacing 54=Umathordrelspacing 55=Umathordopenspacing 56=Umath-
                                               ordclosespacing 57=Umathordpunctspacing 58=Umathordinnerspacing 59=Umath-
                                               opordspacing 60=Umathopopspacing 61=Umathopbinspacing 62=Umathoprelspac-
                                               ing 63=Umathopopenspacing 64=Umathopclosespacing 65=Umathoppunctspacing
                                               66=Umathopinnerspacing 67=Umathbinordspacing 68=Umathbinopspacing 69=Umath-
                                               binbinspacing 70=Umathbinrelspacing 71=Umathbinopenspacing 72=Umathbin-
                                               closespacing 73=Umathbinpunctspacing 74=Umathbininnerspacing 75=Umath-
                                               relordspacing 76=Umathrelopspacing 77=Umathrelbinspacing 78=Umathrelrelspac-
                                               ing 79=Umathrelopenspacing 80=Umathrelclosespacing 81=Umathrelpunctspac-
                                               ing 82=Umathrelinnerspacing 83=Umathopenordspacing 84=Umathopenopspac-
                                               ing 85=Umathopenbinspacing 86=Umathopenrelspacing 87=Umathopenopenspacing
                                               88=Umathopenclosespacing 89=Umathopenpunctspacing 90=Umathopeninnerspac-
                                               ing 91=Umathcloseordspacing 92=Umathcloseopspacing 93=Umathclosebinspacing
                                               94=Umathcloserelspacing 95=Umathcloseopenspacing 96=Umathcloseclosespacing
                                               97=Umathclosepunctspacing 98=Umathcloseinnerspacing 99=Umathpunctordspac-
                                               ing 100=Umathpunctopspacing 101=Umathpunctbinspacing 102=Umathpunctrelspac-
                                               ing 103=Umathpunctopenspacing 104=Umathpunctclosespacing 105=Umathpunct-
                                               punctspacing 106=Umathpunctinnerspacing 107=Umathinnerordspacing 108=Umath-
                                               inneropspacing 109=Umathinnerbinspacing 110=Umathinnerrelspacing 111=Umath-
                                               inneropenspacing 112=Umathinnerclosespacing 113=Umathinnerpunctspacing
                                               114=Umathinnerinnerspacing
7    99  set_font
7   100  def_font
6   101  data                     0  0x1FFFFF
1   102  register                 0         4  0=count 1=attribute 2=dimen 3=skip 4=muskip
1   103  combine_toks             0         7  0=toksapp 1=etoksapp 2=tokspre 3=etokspre 4=gtoksapp 5=xtoksapp 6=gtokspre
                                               7=xtokspre
1   104  advance                  0         0  0=advance
1   105  multiply                 0         0  0=multiply
1   106  divide                   0         0  0=divide
1   107  prefix                   0         2  0=global 1=protected 2=frozen
1   108  let                      0         9  0=glet 1=let 2=futurelet 3=futuredef 4=letcharcode 5=letfrozen 6=unletfrozen
                                               7=letprotected 8=unletprotected 9=letdatacode
1   109  shorthand_def            0        10  0=chardef 1=mathchardef 2=Umathchardef 3=Umathcharnumdef 4=countdef 5=attrib-
                                               utedef 6=dimendef 7=skipdef 8=muskipdef 9=toksdef 10=luadef
1   110  read_to_cs               0         1  0=read 1=readline
```

1	111	def	0	3	0=def 1=gdef 2=edef 3=xdef
1	112	set_box	0	0	0=setbox
1	113	hyph_data	0	7	0=hyphenation 1=patterns 2=prehyphenchar 3=posthyphenchar 4=preexhyphenchar 5=postexhyphenchar 6=hyphenationmin 7=hjcode
1	114	set_interaction	0	3	0=batchmode 1=nonstopmode 2=scrollmode 3=errorstopmode
1	115	set_font_id	0	0	0=setfontid
1	116	undefined_cs	0	0	0=<unavailable>
1	117	expand_after	0	9	0=expandafter 1=unless 2=futureexpand 3=futureexpandis 4=futureexpandisap 5=expandafterspaces 6=expandafterpars 7=expandtoken 8=expandcstoken 9=expand
1	118	no_expand	0	0	0=noexpand
1	119	input	0	3	0=input 1=endinput 2=scantokens 3=scantextokens
5	120	lua_expandable_call	0	0x1FFFFF	
5	121	lua_local_call	0	0x1FFFFF	
1	122	begin_local	0	0	0=beginlocalcontrol
1	123	if_test	2	48	2=fi 3=else 4=or 5=orelse 6=if 7=ifcat 8=ifabsnum 9=ifnum 10=ifabsdim 11=ifdim 12=ifodd 13=ifvmode 14=ifhmode 15=ifmmode 16=ifinner 17=ifvoid 18=ifhbox 19=ifvbox 20=iftok 21=ifcstok 22=ifx 23=iftrue 24=iffalse 25=ifchknum 26=ifnumval 27=ifcmpnum 28=ifchkdim 29=ifdimval 30=ifcmpdim 31=ifcase 32=ifdefined 33=ifcsname 34=ifincsname 35=iffontchar 36=ifcondition 37=ifeof 38=iffrozen 39=ifprotected 40=ifusercmd 41=ifempty 42=ifboolean 43=ifmathparameter 44=ifmathstyle 45=ifarguments 46=ifhastok 47=ifhastoks 48=ifhasxtoks
1	124	cs_name	0	2	0=csname 1=lastnamedcs 2=begincsname
1	125	convert	0	16	0=number 1=directlua 2=luafunction 3=luabytecode 4=expanded 5=immediateassignment 6=immediateassigned 7=string 8=csstring 9=romannumeral 10=meaning 11=Uchar 12=luaescapestring 13=fontname 14=jobname 15=formatname 16=luatexbanner
1	126	the	0	3	0=the 1=thewithoutunit 2=detokenize 3=unexpanded
1	127	top_bot_mark	0	9	0=topmark 1=firstmark 2=botmark 3=splitfirstmark 4=splitbotmark 5=topmarks 6=firstmarks 7=botmarks 8=splitfirstmarks 9=splitbotmarks
7	128	call			
7	129	protected_call			
7	130	frozen_call			
7	131	frozen_protected_call			
7	132	frozen_cs_end_template			
7	133	frozen_cs_dont_expand			
7	134	internal_glue_ref			
7	135	register_glue_ref			
7	136	internal_mu_glue_ref			
7	137	register_mu_glue_ref			
7	138	specification_ref			
7	139	box_ref			