

# Hagen



# Content

<b>Introduction</b>	<b>4</b>
<b>1 The first decade</b>	<b>6</b>
<b>2 Plug mode, an application of ffi</b>	<b>8</b>
<b>3 Variable fonts</b>	<b>22</b>
<b>4 Emoji again</b>	<b>34</b>
<b>5 Performance</b>	<b>50</b>
<b>6 Editing</b>	<b>68</b>
<b>7 Tricky fences</b>	<b>74</b>
<b>8 The state of PDF</b>	<b>84</b>
<b>9 From LUA 5.2 to 5.3</b>	<b>90</b>
<b>10 Executing TeX</b>	<b>96</b>
<b>11 Modern Latin</b>	<b>104</b>
<b>12 More (new) expansion trickery</b>	<b>140</b>
<b>13 Amputating code</b>	<b>146</b>
<b>14 Getting there, version 1.10</b>	<b>156</b>



# Introduction

With LUATEX version 1.0 being released it's not time to move on to a next stage in the development. The first four stages were discussed in 'mk', 'hybrid', 'about' and 'still'. Much in there ended up as article in user group journals. some was just a wrap-up of something I ran into or played with. Also, some of it could be seen as a kind of manual for a specific aspect of LUATEX and/or CONTEXT.

In this document we continue this kind of reporting. Maybe it's useful for others to read about it but in the first place it serves me to wrap up experiences occasionally.

Some chapters were meant for publications in user groups journals so they are made public afterwards. I like to thank Karl Berry for correcting many of my mistakes and improving the content. Because Luigi Scarso and I spend quite some time on LUATEX development, we also share many of the experiences described in this document. Without his patience with me this would not be possible.

Hans Hagen  
Hasselt NL  
2016 onward

<http://www.luaotex.org>  
<http://www.pragma-ade.com>



# 1 The first decade

When writing this it's hard to believe that we're already a decade working on LUATEX and about the same time on MKIV. The question is, did we achieve the objectives? The answer can easily be "yes" because we didn't start with objectives, just with some experiments with a LUA extension interface. However, it quickly became clear that this was the way to go. Already in an early stage we took a stand in what direction we had to move.

How did we end up with LUA and not one of the other popular scripting languages? The CONTEXT macro package always came with a runner. Not only did the runner manage the (often) multiple runs, it also took care of sorting the index and other inter-job activities. Additional helpers were written for installing fonts, managing (and converting) images, job control, etc. First they were binaries (written in MODULA 2), but successive implementations used PERL and RUBY. When I found out that the SCITE editor I switched to had an extension mechanism using LUA, I immediately liked that language. It's clean, not bloated, relatively stable, evolves in an academic environment and is not driven by commerce and/or short term success, and above all, the syntax makes the code look good. So, it was the most natural candidate for extending TeX.

Already for along time, TeX is a stable program and whatever we do with it, we should not break it. There has been frontend extensions, like  $\varepsilon$ -TeX, and backend extensions, like PDFTeX, and experiments like OMEGA and ALEPH and we could start from there. So, basically we took PDFTeX, after all, that was what we used for the first experiments, and merged some ALEPH directional code in it. A tremendous effort was undertaken (thanks to funding by the Oriental TeX project) to convert the code base from PASCAL to C.

It is hard to get an agreement over what needs to be added and it's a real waste of time to enter that route by endless discussions: every TeX user has different demands and macro packages differ in philosophy. So, in the spirit of the extension language LUA we stuck to concept of "If you want it better, write it in LUA". As a consequence we had to provide access to the internals with efficient and convenient methods, something that happened stepwise. We did extend the engine with a few features that make live easier but tried to limit ourselves. On the other hand, due to developments with fonts and languages we generalized these concepts so that extending and controlling them is easier. And, due to developments in math font technology we also added alternative code paths to the math renderer.

All these matters have been presented and discussed at meetings, in user group journals and in documents that are part of the CONTEXT suite. And during this decade the CONTEXT users have been patient testers of whatever we threw at them in the MKIV version of this macro package.

It's kind of interesting to note that in the TeX community it takes a while before version 1 of programs becomes available. Some programs never (seem to) reach that state.

However, for us version 1.0 marks the moment that we consider the interfaces to be stable. Of course we move on so a version 2.0 can divert and provide more or even less interfaces, provide new functionality or drop obsolete features. The intermediate versions (up to version one) were always quite useable in production. In 2005 the first prototype of LUATEX was demonstrated at the TUG conference, and in 2007 at the TUG conference we had a whole day on LUATEX. At that time CONTEXT MKIV evolved fast and we already had decent OPENTYPE support as part of the oriental TEX project. It was in those years that the major reorganization of the code base took place but in successive years many subsystems were opened and cleaned up. There were some occasions where an interface was changed for the better but adapting was not that hard. It might have helped that much of CONTEXT MKIV is written in LUA. What also helped is that most CONTEXT users quickly switched to MKIV, if only because MKII was frozen. And, thanks to those users, we were able to root out bugs and bottlenecks. It was interesting to see that the approach of mixing TEX, METAPOST and LUA catched on quite well.

By the end of September 2016, at the 10<sup>th</sup> CONTEXT meeting we released what we call the first long term stable version of LUATEX. This version performs quite well but we might still add a few things here and there and the code will be further cleaned up and documented. In the meantime LUATEX is also used in other macro packages. It will not replace PDFTEX (at least not soon) because that engine does the job for most of the publications done in TEX: articles. As they are mostly in English and use traditional fonts, there is no need to switch to the more flexible but somewhat slower LUATEX. In a similar fashion XETEX serves those who want the benefits of PDFTEX, hard-coded font support and token juggling at the TEX level. We will support those engines with MKII but as mentioned, we will not develop new code for. We strongly advice CONTEXT users to use LUATEX but there the advertisements stop. Personally I haven't used PDFTEX (which made TEX survive in the evolving world of electronic documents) for a decade and I never really used XETEX (which opened up the TEX world to modern fonts). At least for the coming decade I hope that LUATEX can serve us well.

## 2 Plug mode, an application of ffi

A while ago, at an NTG meeting, Kai Eigner and Ivo Geraarts demonstrated how to use the Harfbuzz (hb) library for processing OPENTYPE fonts. The main motivation for them playing with that was that it provides a way to compare the LUA based font machinery with other methods. They also assumed that it would give a better performance for complex fonts and/or scripts.

One of the guiding principles of LUATEX development is that we don't provide hard coded solutions. For that reason we opened up the internals so that one can provide solutions written in pure LUA, but, of course, one can cooperate with libraries via LUA code as well. Hard coding solutions makes no sense as there are often several solutions possible, depending on one's need. Although development is closely related to CONTEXT, the development of the LUATEX engine is generic. We try to be macro package agnostic. Already in an early stage we made sure that the CONTEXT font handler could be used in other packages as well, but one can easily dream up light weight variants for specific purposes. The standard TeX font handling was kept and is called **base** mode in CONTEXT. The LUA variant is tagged **node** mode because it operates on the node list. Later we will refer to these modes.

With the output of XeTEX for comparison, the first motive mentioned for looking into support for such a library is not that strong. And when we want to test against the standard, we can use MS-Word. A minimal CONTEXT MkIV installation one only has the LUATEX engine. Maintaining several renderers simultaneously might give rise to unwanted dependencies.

The second motive could be more valid for users because, for complex fonts, there is—or at least was—a performance hit with the LUA variant. Some fonts use many lookup steps or are inefficient even in using their own features. It must be said that till now I haven't heard CONTEXT users complain about speed. In fact, the font handling became many times faster the last few years, and probably no one even noticed. Also, when using alternatives to the built in methods, in the end, you will loose functionality and/or interactions with other mechanisms that are built into the current font system. Any possible gain in speed is lost, or even becomes negative, when a user wants to use additional functionality that requires additional processing.<sup>1</sup>

Just kicking in some alternative machinery is not the whole story. We still need to deal with the way TeX sees text, and that, in practice, is as a sequence of glyph nodes—mixed with discretionaryaries for languages that hyphenate, glue, kern, boxes, math, and more. It's the discretionary part that makes it a bit complex. In contextual analysis as well as positioning one needs to process up to three additional cases: the pre, post and replace

---

<sup>1</sup> In general we try to stay away from libraries. For instance, graphics can be manipulated with external programs, and caching the result is much more efficient than recreating it. Apart from SQL support, where integration makes sense, I never felt the need for libraries. And even SQL can efficiently be dealt with via intermediate files.

texts—either or not linked backward and forward. And as applied features accumulate one ends up winding and unwinding these snippets. In the process one also needs to keep an eye on spaces as they can be involved in lookups. Also, when injecting or removing glyphs one needs to deal with attributes associated with nodes. Of course something hard codes in the engine might help a little, but then one ends up with the situation where macro packages have different demands (and possible interactions) and no solution is the right one. Using LUA as glue is a way to avoid that problem. In fact, once we go along that route, it starts making sense to come up with a stripped down LUATEX that might suit CONTEXT better, but it's not a route we are eager to follow right now.

Kai and Ivo are plain T<sub>E</sub>X users so they use a font definition and switching environment that is quite different from CONTEXT. In an average CONTEXT run the time spent on font processing is measurable but not the main bottleneck because other time consuming things happen. Sometimes the load on the font subsystem can be higher because we provide additional features normally not found in OPENTYPE. Add to that a more dynamic font model and it will be clear that comparing performance between situations that use different macro packages is not that trivial (or relevant).

More reasons why we follow a LUA route are that we: support (run time generated) virtual fonts, are able to kick in additional features, can let the font mechanism cooperate with other functionality, and so on. In the upcoming years more trickery will be provided in the current mechanisms. Because we had to figure out a lot of these OPENTYPE things a decade ago when standards were fuzzy quite some tracing and visualization is available. Below we will see some timings, It's important to keep in mind that in CONTEXT the OPENTYPE font handler can do a bit more if requested to do so, which comes with a bit of overhead when the handler is used in CONTEXT—something we can live with.

Some time after Kai's presentation he produced an article, and that was the moment I looked into the code and tried to replicate his experiments. Because we're talking libraries, one can understand that this is not entirely trivial, especially because I'm on another platform than he is—Windows instead of OSX. The first thing that I did was rewrite the code that glues the library to T<sub>E</sub>X in a way that is more suitable for CONTEXT. Mixing with existing modes (`base` or `node` mode) makes no sense and is asking for unwanted interferences, so instead a new `plug` mode was introduced. A sort of general text filtering mechanism was derived from the original code so that we can plug in whatever we want. After all, stability is not the strongest point of today's software development, so when we depend on a library, we need to be prepared for other (library based) solutions—for instance, if I understood correctly, XETEX switched a few times.

After redoing the code the next step was to get the library running and I decided that the `ffi` route made most sense.<sup>2</sup> Due to some expected functions not being supported, my

---

<sup>2</sup> One can think of an intermediate layer but I'm pretty sure that I have different demands than others, but `ffi` sort of frees us from endless discussions.

efforts in using the library failed. At that time I thought it was a matter of interfacing, but I could get around it by piping into the command line tools that come with the library, and that was good enough for testing. Of course it was dead slow, but the main objective was comparison of rendering so it doesn't matter that much. After that I just quit and moved on to something else.

At some point Kai's article came close to publishing, and I tried the old code again, and, surprise, after some messing around, the library worked. On my system the one shipped with Inkscape is used, which is okay as it frees me from bothering about installations. As already mentioned, we have no real reason in CONTEXT for using fonts libraries, but the interesting part was that it permitted me to play with this so called `ffi`. At that moment it was only available in LUAJITTEXBecause that creates a nasty dependency, after a while, Luigi Scarso and I managed to get a similar library working in stock LUATEX which is of course the reference. So, I decided to give it a second try, and in the process I rewrote the interfacing code. After all, there is no reason not to be nice for libraries and optimize the interface where possible.

Now, after a decade of writing LUA code, I dare to claim that I know a bit about how to write relatively fast code. I was surprised to see that where Kai claimed that the library was faster than the LUA code.I saw that it really depends on the font. Sometimes the library approach is actually slower, which is not what one expects. But remember that one argument for using a library is for complex fonts and scripts. So what is meant with complex?

Most Latin fonts are not complex—ligatures and kerns and maybe a little bit of contextual analysis. Here the LUA variant is the clear winner. It runs upto ten times faster. For more complex Latin fonts, like EBgaramond, that resolves ligatures in a different way, the library catches up, but still the LUA handler is faster. Keep in mind that we need to juggle discretionary nodes in any case. One difference between both methods is that the LUA handler runs over all the lists (although it has to jump over fonts not being processed then), while the library gets snippets. However, tests show that the overhead involved in that is close to zero and can be neglected. Already long ago we saw that when we compared MKIV LUATEX and MKII XETEX, the LUA based font handler is not that slow at all. This makes sense because the problem doesn't change, and maybe more importantly because LUA is a pretty fast language. If one or the other approach is less than two times faster the gain will probably go unnoticed in real runs. In my experience a few bad choices in macro or style writing is more harmful than a bit slower font machinery. Kick in some additional node processing and it might make comparison of a run even harder. By the way, one reason why font handling has been sped up over the years is because our workflows sometimes have a high load, and, for instance, processing a set of 5 documents remotely has to be fast. Also, in an edit workflow you want the runtime to be a bit comfortable.

Contrary to Latin, a pure Arabic text (normally) has no discretionary nodes, and the library profits most of this. Some day I have to pick up the thread with Idris about the potential use of discretionary nodes in Arabic typesetting. Contrary to Arabic, Latin

text has not many replacements and positioning, and, therefore, the LUA variant gets the advantage. Some of the additional features that the LUA variant provides can, of course, be provided for the library variant by adding some pre- and postprocessing of the list, but then you quickly loose any gain a library provides. So, Arabic has less complex node lists with no branches into disretinaries, but it definitely has more replacements, positioning and contextual lookups due to the many calls to helpers in the LUA code. Here the library should win because it can (I assume) use more optimized datastructures.

In Kai's prototype there are some cheats for right-to-left rendering and special scripts like Devanagari. As these tweaks mostly involve discretionary nodes; there is no real need for them. When we don't hyphenate no time is wasted anyway. I didn't test Devanagari, but there is some preprocessing needed in the LUA variant (provided by Kai and Ivo) that I might rewrite from scratch once I understand what happens there. But still, I expect the library to perform somewhat better there but I didn't test it. Eventually I might add support for some more scripts that demand special treatments, but so far there has not been any request for it.

So what is the processing speed of non-Latin scripts? An experiment with Arabic using the frequently used Arabtype font showed that the library performs faster, but when we use a mixed Latin and Arabic document the differences become less significant. On pure Latin documents the LUA variant will probably win. On pure Arabic the library might be on top. On average there is little difference in processing speed between the LUA and library engines when processing mixed documents. The main question is, does one want to loose functionality provided by the LUA variant? Of course one can depend on functionality provided by the library but not by the LUA variant. In the end the user decides.

How did we measure? The baseline measurement is the so called `none` mode: nothing is done there. It's fast but still takes a bit of time as it is triggered by a general mode identifying pass. That pass determines what font processing modes are needed for a list. `Base` mode only makes sense for Latin and has some limitations. It's fast and, basically, its run time can be neglected. That's why, for instance, PDFT<sub>E</sub>X is faster than the other engines, but it doesn't do UNICODE well. `Node` mode is the fancy name for the LUA font handler. So, in order of increasing run time we have: `none`, `base` and `node`. If we compare `node` mode with `plug` mode (in our case using the hb library), we can subtract `none` mode. This gives a cleaner (more distinctive) comparison but not a real honest one because the identifying pass always happens.

We also tested with and without hyphenation, but in practice that makes no sense. Only verbatim is typeset that way, and normally we typeset that in `none` mode anyway. On the other hand mixing fonts does happen. All the tests start with forced garbage collection in order to get rid of that variance. We also pack into horizontal boxes so that the par builder (with all kind of associated callbacks) doesn't kick in, although the `node` mode should compensate that.

Keep in mind that the tests are somewhat dumb. There is no overhead in handling structure, building pages, adding color or whatever. I never process raw text. As a reference it's no problem to let CONTEXt process hundreds of pages per second. In practice a moderate complex document like the metafun manual does some 20 pages per second. In other words, only a fraction of the time is spent on fonts. The timings for LUATEX are as follows:

### luatex latin

<b>modern</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.48	0.04	-0.75	0.39	0.05
context node	1.23	0.79	0.00	1.00	1.00
context none	0.44	0.00	-0.79	0.36	0.00
harfbuzz native	5.06	4.62	3.83	4.12	5.86
harfbuzz uniscribe	5.24	4.80	4.02	4.27	6.10
<b>pagella</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.50	0.03	-0.77	0.39	0.04
context node	1.27	0.80	0.00	1.00	1.00
context none	0.47	0.00	-0.80	0.37	0.00
harfbuzz native	4.96	4.49	3.69	3.89	5.58
harfbuzz uniscribe	5.49	5.02	4.22	4.31	6.24
<b>dejavu</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.46	0.04	-1.21	0.28	0.03
context node	1.68	1.25	0.00	1.00	1.00
context none	0.43	0.00	-1.25	0.25	0.00
harfbuzz native	4.50	4.07	2.82	2.68	3.26
harfbuzz uniscribe	4.79	4.37	3.12	2.86	3.49
<b>cambria</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.44	0.02	-1.67	0.21	0.01
context node	2.11	1.69	0.00	1.00	1.00
context none	0.43	0.00	-1.69	0.20	0.00
harfbuzz native	4.59	4.16	2.47	2.17	2.47
harfbuzz uniscribe	5.03	4.60	2.91	2.38	2.73
<b>ebgaramond</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.50	0.06	-1.86	0.21	0.03
context node	2.36	1.92	0.00	1.00	1.00
context none	0.43	0.00	-1.92	0.18	0.00
harfbuzz native	4.96	4.52	2.60	2.10	2.35
harfbuzz uniscribe	5.17	4.74	2.81	2.19	2.46

<b>lucidaot</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.48	0.01	-0.45	0.52	0.02
context node	0.93	0.45	0.00	1.00	1.00
context none	0.47	0.00	-0.45	0.51	0.00
harfbuzz native	4.28	3.81	3.35	4.62	8.42
harfbuzz uniscribe	4.68	4.21	3.76	5.06	9.32

### luatex arabic

<b>arabtype</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.42	0.00	-14.75	0.03	0.00
context node	15.17	14.76	0.00	1.00	1.00
context none	0.41	0.00	-14.76	0.03	0.00
harfbuzz native	7.14	6.73	-8.02	0.47	0.46
harfbuzz uniscribe	7.68	7.27	-7.49	0.51	0.49

<b>husayni</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.45	-0.01	-25.63	0.02	-0.00
context node	26.08	25.62	0.00	1.00	1.00
context none	0.46	0.00	-25.62	0.02	0.00
harfbuzz native	10.50	10.04	-15.58	0.40	0.39
harfbuzz uniscribe	18.96	18.50	-7.12	0.73	0.72

### luatex mixed

<b>arabtype</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.68	-0.01	-7.18	0.09	-0.00
context node	7.85	7.17	0.00	1.00	1.00
context none	0.69	0.00	-7.17	0.09	0.00
harfbuzz native	5.82	5.13	-2.03	0.74	0.72
harfbuzz uniscribe	6.21	5.53	-1.64	0.79	0.77

<b>husayni</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.72	0.05	-11.20	0.06	0.00
context node	11.92	11.25	0.00	1.00	1.00
context none	0.67	0.00	-11.25	0.06	0.00
harfbuzz native	6.93	6.25	-4.99	0.58	0.56
harfbuzz uniscribe	9.85	9.18	-2.07	0.83	0.82

The timings for LUAJITTEX are, of course, overall better. This is because the virtual machine is faster, but at the cost of some limitations. We seldom run into these limitations, but fonts with large tables can't be cached unless we rewrite some code and sacrifice clean solutions. Instead, we perform a runtime conversion which is not that noticeable

when it's just a few fonts. The numbers below are not influenced by this as the test stays away from these rare cases.

### luajittex latin

<b>modern</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.42	0.03	-0.36	0.54	0.09
context node	0.77	0.39	0.00	1.00	1.00
context none	0.38	0.00	-0.39	0.50	0.00
harfbuzz native	3.07	2.69	2.30	3.98	6.90
harfbuzz uniscribe	3.05	2.67	2.28	3.94	6.84
<b>pagella</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.44	0.02	-0.37	0.54	0.05
context node	0.80	0.39	0.00	1.00	1.00
context none	0.42	0.00	-0.39	0.52	0.00
harfbuzz native	3.02	2.61	2.22	3.77	6.74
harfbuzz uniscribe	3.01	2.59	2.20	3.74	6.69
<b>dejavu</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.40	0.04	-0.59	0.41	0.06
context node	0.98	0.62	0.00	1.00	1.00
context none	0.36	0.00	-0.62	0.37	0.00
harfbuzz native	3.02	2.66	2.04	3.07	4.28
harfbuzz uniscribe	2.97	2.60	1.98	3.01	4.19
<b>cambria</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.38	0.02	-0.79	0.33	0.02
context node	1.17	0.80	0.00	1.00	1.00
context none	0.37	0.00	-0.80	0.31	0.00
harfbuzz native	2.91	2.54	1.74	2.48	3.16
harfbuzz uniscribe	2.86	2.50	1.69	2.45	3.11
<b>ebgaramond</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.43	0.05	-0.89	0.33	0.05
context node	1.32	0.94	0.00	1.00	1.00
context none	0.38	0.00	-0.94	0.29	0.00
harfbuzz native	3.00	2.62	1.68	2.27	2.78
harfbuzz uniscribe	2.98	2.60	1.66	2.25	2.77
<b>lucidaot</b>	<i>t</i>	<i>t</i> - <i>t</i> <sub>none</sub>	<i>t</i> - <i>t</i> <sub>node</sub>	<i>t/t</i> <sub>node</sub>	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.41	-0.01	-0.21	0.66	-0.04
context node	0.63	0.20	0.00	1.00	1.00

context none	0.42	0.00	-0.20	0.67	0.00
harfbuzz native	2.61	2.18	1.98	4.16	10.71
harfbuzz uniscribe	2.59	2.17	1.97	4.14	10.65

### luajittex arabic

<b>arabtype</b>	<i>t</i>	$t - t_{\text{none}}$	$t - t_{\text{node}}$	$t/t_{\text{node}}$	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.32	-0.00	-6.85	0.04	-0.00
context node	7.17	6.84	0.00	1.00	1.00
context none	0.32	0.00	-6.84	0.04	0.00
harfbuzz native	4.63	4.31	-2.54	0.65	0.63
harfbuzz uniscribe	4.67	4.35	-2.50	0.65	0.64

<b>husayni</b>	<i>t</i>	$t - t_{\text{none}}$	$t - t_{\text{node}}$	$t/t_{\text{node}}$	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.35	-0.00	-11.90	0.03	-0.00
context node	12.25	11.90	0.00	1.00	1.00
context none	0.35	0.00	-11.90	0.03	0.00
harfbuzz native	15.28	14.93	3.03	1.25	1.25
harfbuzz uniscribe	15.25	14.90	3.00	1.25	1.25

### luajittex mixed

<b>arabtype</b>	<i>t</i>	$t - t_{\text{none}}$	$t - t_{\text{node}}$	$t/t_{\text{node}}$	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.57	-0.03	-3.47	0.14	-0.01
context node	4.04	3.44	0.00	1.00	1.00
context none	0.60	0.00	-3.44	0.15	0.00
harfbuzz native	3.69	3.09	-0.35	0.91	0.90
harfbuzz uniscribe	3.69	3.08	-0.35	0.91	0.90

<b>husayni</b>	<i>t</i>	$t - t_{\text{none}}$	$t - t_{\text{node}}$	$t/t_{\text{node}}$	$\frac{t-t_{\text{none}}}{t_{\text{node}}-t_{\text{none}}}$
context base	0.62	0.04	-5.33	0.10	0.01
context node	5.94	5.37	0.00	1.00	1.00
context none	0.57	0.00	-5.37	0.10	0.00
harfbuzz native	7.19	6.62	1.25	1.21	1.23
harfbuzz uniscribe	7.11	6.53	1.17	1.20	1.22

A few side notes. Since a library is an abstraction, one has to live with what one gets. In my case that was a crash in UTF-32 mode. I could get around it, but one advantage of using LUA is that it's hard to crash—if only because as a scripting language it manages its memory well without user interference. My policy with libraries is just to wait till things get fixed and not bother with the why and how of the internals.

Although CONTEXt will officially support the [plug](#) model, it will not be actively used by me, or in documentation, so for support users are on their own. I didn't test the

`plug` mode in real documents. Most documents that I process are Latin (or a mix), and redefining feature sets or adapting styles for testing makes no sense. So, can one just switch engines without looking at the way a font is defined? The answer is—not really, because (even without the user knowing about it) virtual fonts might be used, additional features kicked in and other mechanisms can make assumptions about how fonts are dealt with too.

The useability of `plug` mode probably depends on the workflow one has. We use CONTEX $\mathrm{T}$  in a few very specific workflows where, interestingly, we only use a small subset of its functionality. Most of which is driven by users, and tweaking fonts is popular and has resulted in all kind of mechanisms. So, for us it's unlikely that we will use it. If you process (in bursts) many documents in succession, each demanding a few runs, you don't want to sacrifice speed.

Of course timing can (and likely will) be different for plain T $\mathrm{E}\mathrm{X}$  and L $\mathrm{A}\mathrm{T}\mathrm{E}\mathrm{X}$  usage. It depends on how mechanisms are hooked into the callbacks, what extra work is done or not done compared to CONTEX $\mathrm{T}$ . This means that my timings for CONTEX $\mathrm{T}$  for sure will differ from those of other packages. Timings are a snapshot anyway. And as said, font processing is just one of the many things that goes on. If you are not using CONTEX $\mathrm{T}$  you probably will use Kai's version because it is adapted to his use case and well tested.

A fundamental difference between the two approaches is that—whereas the LUA variant operates on node lists only, the `plug` variant generates strings that get passed to a library where, in the CONTEX $\mathrm{T}$  variant of hb support, we use UTF-32 strings. Interesting, a couple of years ago I considered using a similar method for LUA but eventually decided against it, first of all for performance reasons, but mostly because one still has to use some linked list model. I might pick up that idea as a variant, but because all this T $\mathrm{E}\mathrm{X}$  related development doesn't really pay off and costs a lot of free time it will probably never happen.

I finish with a few words on how to use the plug model. Because the library initializes a default set of features,<sup>3</sup> all you need to do is load the plugin mechanism:

```
\usemodule[fonts-plugins]
```

Next you define features that use this extension:

```
\definefontfeature
  [hb-native]
  [mode=plug,
   features=harfbuzz,
   shaper=native]
```

---

<sup>3</sup> Somehow passing features to the library fails for Arabic. So when you don't get the desired result, just try with the defaults.

After this you can use this feature set when you define fonts. Here is a complete example:

```
\usemodule[fonts-plugins]

\starttext

\definefontfeature
  [hb-library]
  [mode=plug,
   features=harfbuzz,
   shaper=native]

\definedfont[Serif*hb-library]

\input ward \par

\definefontfeature
  [hb-binary]
  [mode=plug,
   features=harfbuzz,
   method=binary,
   shaper=uniscribe]

\definedfont[Serif*hb-binary]

\input ward \par

\stoptext
```

The second variant uses the `hb-shape` binary which is, of course, pretty slow, but does the job and is okay for testing.

There are a few trackers available too:

```
\enabletrackers[fonts.plugins.hb.colors]
\enabletrackers[fonts.plugins.hb.details]
```

The first one colors replaced glyphs while the second gives lot of information about what is going on. If you want to know what gets passed to the library you can use the `text` plugin:

```
\definefontfeature[test] [mode=plug,features=text]
\start
  \definedfont[Serif*test]
  \input ward \par
\stop
```

This produces something:

```
otf plugin > text > start run 3
otf plugin > text > 001 : [-] The [+]-> U+00054 U+00068 U+00065
otf plugin > text > 002 : [+] Earth, [+]-> U+00045 U+00061 U+00072 ...
otf plugin > text > 003 : [+] as [+]-> U+00061 U+00073
otf plugin > text > 004 : [+] a [+]-> U+00061
otf plugin > text > 005 : [+] habi- [-]-> U+00068 U+00061 U+00062 ...
otf plugin > text > 006 : [-] tat [+]-> U+00074 U+00061 U+00074
otf plugin > text > 007 : [+] habitat [+]-> U+00068 U+00061 U+00062 ...
otf plugin > text > 008 : [+] for [+]-> U+00066 U+0006F U+00072
otf plugin > text > 009 : [+] an- [-]-> U+00061 U+0006E U+0002D
```

You can see how hyphenation of **habi-tat** results in two snippets and a whole word. The font engine can decide to turn this word into a disc node with a pre, post and replace text. Of course the machinery will try to retain as many hyphenation points as possible. Among the tricky parts of this are lookups across and inside discretionary nodes resulting in (optional) replacements and kerning. You can imagine that there is some trade off between performance and quality here. The results are normally acceptable, especially because **TEX** is so clever in breaking paragraphs into lines.

Using this mechanism (there might be variants in the future) permits the user to cook up special solutions. After all, that is what LUATEX is about—the traditional core engine with the ability to plug in your own code using LUA. This is just an example of it.

I'm not sure yet when the plugin mechanism will be in the CONTEXt distribution, but it might happen once the **ffi** library is supported in LUATEX. At the end of this document the basics of the test setup are shown, just in case you wonder what the numbers apply to.

Just to put things in perspective, the current (February 2017) METAFUN manual has 424 pages. It takes LUATEX 18.3 seconds and LUAJITTEX 14.4 seconds on my Dell 7600 laptop with 3840QM mobile i7 processor. Of this 6.1 (4.5) seconds is used for processing 2170 METAPOST graphics. Loading the 15 fonts used takes 0.25 (0.3) seconds, which includes also loading the outline of some. Font handling is part of the, so called, hlist processing and takes around 1 (0.5) second, and attribute backend processing takes 0.7 (0.3) seconds. One problem in these timings is that font processing often goes too fast for timing, especially when we have lots of small snippets. For example, short runs like titles and such take no time at all, and verbatim needs no font processing. The difference in runtime between LUATEX and LUAJITTEX is significant so we can safely assume that we spend some more time on fonts than reported. Even if we add a few seconds, in this rather complete document, the time spent on fonts is still not that impressive. A five fold increase in processing (we use mostly Pagella and Dejavu) is a significant addition to the total run time, especially if you need a few runs to get cross referencing etc. right.

The test files are the familiar ones present in the distribution. The **tufte** example is a good torture test for discretionary processing. We preload the files so that we don't have the overhead of **\input**.

```
\edef\tufte{\cldloadfile{tufte.tex}}
\edef\khatt{\cldloadfile{khatt-ar.tex}}
```

We use six buffers for the tests. The Latin test uses three fonts and also has a paragraph with mixed font usage. Loading the fonts happens once before the test, and the local (re)definition takes no time. Also, we compensate for general overhead by subtracting the `none` timings.

```
\startbuffer[latin-definitions]
\definefont[TestA][Serif*test]
\definefont[TestB][SerifItalic*test]
\definefont[TestC][SerifBold*test]
\stopbuffer

\startbuffer[latin-text]
\TestA \tufte \par
\TestB \tufte \par
\TestC \tufte \par
\dorecurse{10} {%
    \TestA Fluffy Test Font A
    \TestB Fluffy Test Font B
    \TestC Fluffy Test Font C
}\par
\stopbuffer
```

The Arabic tests are a bit simpler. Of course we do need to make sure that we go from right to left.

```
\startbuffer[arabic-definitions]
\definedfont[Arabic*test at 14pt]
\setupinterlinespace[line=18pt]
\setupalign[r2l]
\stopbuffer

\startbuffer[arabic-text]
\dorecurse{10} {
    \khatt\space
    \khatt\space
    \khatt\blank
}
\stopbuffer
```

The mixed case use a Latin and an Arabic font and also processes a mixed script paragraph.

```
\startbuffer[mixed-definitions]
\definefont[TestL][Serif*test]
```

```

\definefont[TestA] [Arabic*test at 14pt]
\setupinterlinespace[line=18pt]
\setupalign[r2l]
\stopbuffer

\startbuffer[mixed-text]
\dorecurse {2} {
  {\TestA\khatt\space\khatt\space\khatt}
  {\TestL\lefttoright\tufte}
  \blank
  \dorecurse{10}{%
    {\TestA      }
    {\TestL\lefttoright A snippet text that makes no sense.}
  }
}
\stopbuffer

```

The related font features are defined as follows:

```

\definefontfeature
  [test-none]
  [mode=none]

\definefontfeature
  [test-base]
  [mode=base,
   liga=yes,
   kern=yes]

\definefontfeature
  [test-node]
  [mode=node,
   script=auto,
   autoscript=position,
   autolanguage=position,
   ccmp=yes,liga=yes,clig=yes,
   kern=yes,mark=yes,mkrmk=yes,
   curs=yes]

\definefontfeature
  [test-text]
  [mode=plug,
   features=text]

\definefontfeature
  [test-native]

```

```

[mode=plug,
 features=harfbuzz,
 shaper=native]

\definefontfeature
 [arabic-node]
 [arabic]

\definefontfeature
 [arabic-native]
 [mode=plug,
 features=harfbuzz,
 script=arab,language=dflt,
 shaper=native]

```

The timings are collected in LUA tables and typeset afterwards, so there is no interference there either.

*The timings are as usual a snapshot and just indications. The relative times can differ over time depending on how binaries are compiled, libraries are improved and LUA code evolves. In node mode we can have experimental trickery that is not yet optimized. Also, especially with complex fonts like Husayni, not all shapers give the same result, although node mode and Uniscribe should be the same in most cases. A future (public) version of Husayni will play more safe and use less complex sequences of features.*

## 3 Variable fonts

### Introduction

History shows the tendency to recycle ideas. Often quite some effort is made by historians to figure out what really happened, not just long ago, when nothing was written down and we have to do with stories or pictures at most, but also in recent times. Descriptions can be conflicting, puzzling, incomplete, partially lost, biased, . . .

Just as language was invented (or evolved) several times, so were scripts. The same might be true for rendering scripts on a medium. Semaphores came and went within decades and how many people know now that they existed and that encryption was involved? Are the old printing presses truly the old ones, or are older examples simply gone? One of the nice aspects of the internet is that one can now more easily discover similar solutions for the same problem, but with a different (and independent) origin.

So, how about this “new big thing” in font technology: variable fonts. In this case, history shows that it’s not that new. For most TEX users the names METAFONT and METAPOST will ring bells. They have a very well documented history so there is not much left to speculation. There are articles, books, pictures, examples, sources, and more around for decades. So, the ability to change the appearance of a glyph in a font depending on some parameters is not new. What probably *is* new is that creating variable fonts is done in the natural environment where fonts are designed: an interactive program. The METAFONT toolkit demands quite some insight in programming shapes in such a way that one can change look and feel depending on parameters. There are not that many meta fonts made and one reason is that making them requires a certain mind- and skill set. On the other hand, faster computers, interactive programs, evolving web technologies, where real-time rendering and therefore more or less real-time tweaking of fonts is a realistic option, all play a role in acceptance.

But do interactive font design programs make this easier? You still need to be able to translate ideas into usable beautiful fonts. Taking the common shapes of glyphs, defining extremes and letting a program calculate some interpolations will not always bring good results. It’s like morphing a picture of your baby’s face into yours of old age (or that of your grandparent): not all intermediate results will look great. It’s good to notice that variable fonts are a revival of existing techniques and ideas used in, for instance, multiple master fonts. The details might matter even more as they can now be exaggerated when some transformation is applied.

There is currently (March 2017) not much information about these fonts so what I say next may be partially wrong or at least different from what is intended. The perspective will be one from a TEX user and coder. Whatever you think of them, these fonts will be out there and for sure there will be nice examples circulating soon. And so, when I ran into a few experimental fonts, with POSTSCRIPT and TRUETYPE outlines, I decided to have a look at what is inside. After all, because it’s visual, it’s also fun to play with. Let’s

stress that at the moment of this writing I only have a few simple fonts available, fonts that are designed for testing and not usage. Some recommended tables were missing and no complex OPENTYPE features are used in these fonts.

## The specification

I'm not that good at reading specifications, first of all because I quickly fall asleep with such documents, but most of all because I prefer reading other stuff (I do have lots of books waiting to be read). I'm also someone who has to play with something in order to understand it: trial and error is my modus operandi. Eventually it's my intended usage that drives the interface and that is when everything comes together.

Exploring this technology comes down to: locate a font, get the OPENTYPE 1.8 specification from the MICROSOFT website, and try to figure out what is in the font. When I had a rough idea the next step was to get to the shapes and see if I could manipulate them. Of course it helped that in CONTEX $\mathrm{T}$  we already can load fonts and play with shapes (using METAPOST). I didn't have to install and learn other programs. Once I could render them, in this case by creating a virtual font with inline PDF literals, a next step was to apply variation. Then came the first experiments with a possible user interface. Seeing more variation then drove the exploration of additional properties needed for typesetting, like features.

The main extension to the data packaged in a font file concerns the (to be discussed) axis along which variable fonts operate and deltas to be applied to coordinates. The **gdef** table has been extended and contains information that is used in **gpos** features. There are new **hvar**, **vvar** and **mvar** tables that influence the horizontal, vertical and general font dimensions. The **gvar** table is used for TRUETYPE variants, while the **cff2** table replaces the **cff** table for OPENTYPE POSTSCRIPT outlines. The **avar** and **stat** tables contain some meta-information about the axes of variations.

It must be said that because this is new technology the information in the standard is not always easy to understand. The fact that we have two rendering techniques, POSTSCRIPT **cff** and TRUETYPE **ttf**, also means that we have different information and perspectives. But this situation is not much different from OPENTYPE standards a few years ago: it takes time but in the end I will get there. And, after all, users also complain about the lack of documentation for CONTEX $\mathrm{T}$ , so who am I to complain? In fact, it will be those CONTEX $\mathrm{T}$  users who will provide feedback and make the implementation better in the end.

## Loading

Before we discuss some details, it will be useful to summarize what the font loader does when a user requests a font at a certain size and with specific features enabled. When a font is used the first time, its binary format is converted into a form that makes it suitable for use within CONTEX $\mathrm{T}$  and therefore LUATE $\mathrm{X}$ . This conversion involves collecting

properties of the font as a whole (official names, general dimensions like x-height and em-width, etc.), of glyphs (dimensions, UNICODE properties, optional math properties), and all kinds of information that relates to (contextual) replacements of glyphs (small caps, oldstyle, scripts like Arabic) and positioning (kerning, anchoring marks, etc.). In the CONTEXt font loader this conversion is done in LUA.

The result is stored in a condensed format in a cache and the next time the font is needed it loads in an instant. In the cached version the dimensions are untouched, so a font at different sizes has just one copy in the cache. Often a font is needed at several sizes and for each size we create a copy with scaled glyph dimensions. The feature-related dimensions (kerning, anchoring, etc.) are shared and scaled when needed. This happens when sequences of characters in the node list get converted into sequences of glyphs. We could do the same with glyph dimensions but one reason for having a scaled copy is that this copy can also contain virtual glyphs and these have to be scaled beforehand. In practice there are several layers of caching in order to keep the memory footprint within reasonable bounds.<sup>4</sup>

When the font is actually used, interaction between characters is resolved using the feature-related information. When for instance two characters need to be kerned, a lookup results in the injection of a kern, scaled from general dimensions to the current size of the font.

When the outlines of glyphs are needed in METAFUN the font is also converted from its binary form to something in LUA, but this time we filter the shapes. For a `cff` this comes down to interpreting the `charstrings` and reducing the complexity to `moveto`, `lineto` and `curveto` operators. In the process subroutines are inlined. The result is something that METAPOST is happy with but that also can be turned into a piece of a PDF.

We now come to what a variable font actually is: a basic design which is transformed along one or more axes. A simple example is wider shapes:



We can also go taller and retain the width:




---

<sup>4</sup> In retrospect one can wonder if that makes sense; just look at how much memory a browser uses when it has been open for some time. In the beginning of LUATEX users wondered about caching fonts, but again, just look at what amounts browsers cache: it gets pretty close to the average amount of writes that a SSD can handle per day within its guarantee.

Here we have a linear scaling but glyphs are not normally done that way. There are font collections out there with lots of intermediate variants (say from light to heavy) and it's more profitable to sell each variant independently. However, there is often some logic behind it, probably supported by programs that designers use, so why not build that logic into the font and have one file that represents many intermediate forms. In fact, once we have multiple axes, even when the designer has clear ideas of the intended usage, nothing will prevent users from tinkering with the axis properties in ways that will fulfil their demands but hurt the designers eyes. We will not discuss that dilemma here.

When a variable font follows the route described above, we face a problem. When you load a TRUETYPE font it will just work. The glyphs are packaged in the same format as static fonts. However, a variable font has axes and on each axis a value can be set. Each axis has a minimum, maximum and default. It can be that the default instance also assumes some transformations are applied. The standard recommends adding tables to describe these things but the fonts that I played with each lacked such tables. So that leaves some guesswork. But still, just loading a TRUETYPE font gives some sort of outcome, although the dimensions (widths) might be weird due to lack of a (default) axis being applied.

An OPENTYPE font with POSTSCRIPT outlines is different: the internal `cff` format has been upgraded to `cff2` which on the one hand is less complicated but on the other hand has a few new operators — which results in programs that have not been adapted complaining or simply quitting on them.

One could argue that a font is just a resource and that one only has to pass it along but that's not what works well in practice. Take LUATEX. We can of course load the font and apply axis values so that we can process the document as we normally do. But at some point we have to create a PDF. We can simply embed the TRUETYPE files but no axis values are applied. This is because, even if we add the relevant information, there is no way in current PDF formats to deal with it. For that, we should be able to pass all relevant axis-related information as well as specify what values to use along these axes. And for TRUETYPE fonts this information is not part of the shape description so then we in fact need to filter and pass more. An OPENTYPE POSTSCRIPT font is much cleaner because there we have the information needed to transform the shape mostly in the glyph description. There we only need to carry some extra information on how to apply these so-called blend values. The region/axis model used there only demands passing a relatively simple table (stripped down to what we need). But, as said above, `cff2` is not backward-compatible so a viewer will (currently) simply not show anything.

Recalling how we load fonts, how does that translate with variable changes? If we have two characters with glyphs that get transformed and that have a kern between them, the kern may or may not transform. So, when we choose values on an axis, then not only glyph properties change but also relations. We no longer can share positional information and scale afterwards because each instance can have different values to start with. We could carry all that information around and apply it at runtime but

because we're typesetting documents with a static design it's more convenient to just apply it once and create an instance. We can use the same caching as mentioned before but each chosen instance (provided by the font or made up by user specifications) is kept in the cache. As a consequence, using a variable font has no overhead, apart from initial caching.

So, having dealt with that, how do we proceed? Processing a font is not different from what we already had. However, I would not be surprised if users are not always satisfied with, for instance, kerning, because in such fonts a lot of care has to be given to this by the designer. Of course I can imagine that programs used to create fonts deal with this, but even then, there is a visual aspect to it too. The good news is that in CONTEXT we can manipulate features so in theory one can create a so-called font goodie file for a specific instance.

## Shapes

For OPENTYPE POSTSCRIPT shapes we always have to do a dummy rendering in order to get the right bounding box information. For TRUETYPE this information is already present but not when we use a variable instance, so I had to do a bit of coding for that. Here we face a problem. For TEX we need the width, height and depth of a glyph. Consider the following case:



The shape has a bounding box that fits the shape. However, its left corner is not at the origin. So, when we calculate a tight bounding box, we cannot use it for actually positioning the glyph. We do use it (for horizontal scripts) to get the height and depth but for the width we depend on an explicit value. In OPENTYPE POSTSCRIPT we have the width available and how the shape is positioned relative to the origin doesn't much matter. In a TRUETYPE shape a bounding box is part of the specification, as is the width, but for a variable font one has to use so-called phantom points to recalculate the width and the test fonts I had were not suitable for investigating this.

At any rate, once I could generate documents with typeset text using variable fonts it became time to start thinking about a user interface. A variable font can have predefined instances but of course a user also wants to mess with axis values. Take one of the test fonts: Adobe Variable Font Prototype. It has several instances:

extralight	It looks like this!	weight=0.0 contrast=0.0
light	It looks like this!	weight=150.0 contrast=0.0
regular	It looks like this!	weight=394.0 contrast=0.0
semibold	<b>It looks like this!</b>	weight=600.0 contrast=0.0
bold	<b>It looks like this!</b>	weight=824.0 contrast=0.0

black high contrast	<b>It looks like this!</b>	weight=1000.0 contrast=100.0
black medium contrast	<b>It looks like this!</b>	weight=1000.0 contrast=50.0
black	<b>It looks like this!</b>	weight=1000.0 contrast=0.0

Such an instance is accessed with:

```
\definefont
[MyLightFont]
[name:adobevariablefontprototypelight*default]
```

The Avenir Next variable demo font (currently) provides:

regular	It looks like this!	weight=400.0 width=100.0
medium	It looks like this!	weight=500.0 width=100.0
bold	<b>It looks like this!</b>	weight=700.0 width=100.0
heavy	<b>It looks like this!</b>	weight=900.0 width=100.0
condensed	It looks like this!	weight=400.0 width=75.0
medium condensed	It looks like this!	weight=500.0 width=75.0
bold condensed	<b>It looks like this!</b>	weight=700.0 width=75.0
heavy condensed	<b>It looks like this!</b>	weight=900.0 width=75.0

Before we continue I will show a few examples of variable shapes. Here we use some METAFUN magic. Just take these definitions for granted.

```
\startMPcode
draw outlinetext.b
  ("\"definedfont [name:adobevariablefontprototypeextralight]foo@bar")
  (withcolor "gray")
  (withcolor red withpen pencircle scaled 1/10)
  xsized .45TextWidth ;
\stopMPcode

\startMPcode
draw outlinetext.b
  ("\"definedfont [name:adobevariablefontprototypelight]foo@bar")
  (withcolor "gray")
  (withcolor red withpen pencircle scaled 1/10)
  xsized .45TextWidth ;
\stopMPcode

\startMPcode
draw outlinetext.b
  ("\"definedfont [name:adobevariablefontprototypebold]foo@bar")
  (withcolor "gray")
  (withcolor red withpen pencircle scaled 1/10)
  xsized .45TextWidth ;
\stopMPcode
```

```

\startMPcode
draw outlinetext.b
  ("\"definefontfeature [whatever] [axis={weight:350}]%
    \definedfont [name:adobevariablefontprototype*whatever]foo@bar")
  (withcolor "gray")
  (withcolor red withpen pencircle scaled 1/10)
  xsized .45TextWidth ;
\stopMPcode

```

The results are shown in figure 3.1. What we see here is that as long as we fill the shape everything will look as expected but using an outline only won't. The crucial (control) points are moved to different locations and as a result they can end up inside the shape. Giving up outlines is the price we evidently need to pay. Of course this is not unique for variable fonts although in practice static fonts behave better. To some extent we're back to where we were with METAFONT and (for instance) Computer Modern: because these originate in bitmaps (and probably use similar design logic) we also can have overlap and bits and pieces pasted together and no one will notice that. The first outline variants of Computer Modern also had such artifacts while in the static Latin Modern successors, outlines were cleaned up.

The fact that we need to preprocess an instance but only know how to do that when we have gotten the information about axis values from the font means that the font handler has to be adapted to keep caching correct. Another definition is:

```

\definefontfeature
[lightdefault]
[default]
[axis={weight:230,contrast:50}]

\definefont
[MyLightFont]
[name:adobevariablefontprototype*lightdefault]

```

Here the complication is that where normally features are dealt with after loading, the axis feature is part of the preparation (and caching). If you want the virtual font solution you can do this:

```

\definefontfeature
[inlinelightdefault]
[default]
[axis={weight:230,contrast:50},
 variableshapes=yes]

\definefont
[MyLightFont]
[name:adobevariablefontprototype*inlinelightdefault]

```

a

b

d

**Figure 3.1** Four variants

When playing with these fonts it was hard to see if loading was done right. For instance not all values make sense. It is beyond the scope of this article, but axes like weight, width, contrast and italic values get applied differently to so-called regions (subspaces). So say that we have an  $x$  coordinate with value 50. This value can be adapted in, for instance, four subspaces (regions), so we actually get:

$$x' = x + s_1 \times x_1 + s_2 \times x_2 + s_3 \times x_3 + s_4 \times x_4$$

The (here) four scale factors  $s_n$  are determined by the axis value. Each axis has some rules about how to map the values 230 for weight and 50 for contrast to such a factor. And each region has its own translation from axis values to these factors. The deltas  $x_1, \dots, x_4$  are provided by the font. For a POSTSCRIPT-based font we find sequences like:

```
1 <setvstore>
120 [10 -30 40 -60] 1 <blend> ... <operator>
100 120 [10 -30 40 -60] [30 -10 -30 20] 2 <blend> .. <operator>
```

A store refers to a region specification. From there the factors are calculated using the chosen values on the axis. The deltas are part of the glyphs specification. Officially there can be multiple region specifications, but how likely it is that they will be used in real fonts is an open question.

For TRUETYPE fonts the deltas are not in the glyph specification but in a dedicated `gvar` table.

```
apply x deltas [10 -30 40 -60] to x 120
apply y deltas [30 -10 -30 20] to y 100
```

Here the deltas come from tables outside the glyph specification and their application is triggered by a combination of axis values and regions.

The following two examples use Avenir Next Variable and demonstrate that kerning is adapted to the variant.

```
\definefontfeature
  [default:shaped]
  [default]
  [axis={width:10}]

\definefont
  [SomeFont]
  [file:avenirnextvariable*default:shaped]

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Hermann Zapf

```
\definefontfeature  
  [default:shaped]  
  [default]  
  [axis={width:100}]
```

```
\definefont  
  [SomeFont]  
  [file:avenirnextvariable*default:shaped]
```

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Hermann Zapf

## Embedding

Once we're done typesetting and a PDF file has to be created there are three possible routes:

- We can embed the shapes as PDF images (inline literal) using virtual font technology. We cannot use so-called xforms here because we want to support color selectively in text.
- We can wait till the PDF format supports such fonts, which might happen but even then we might be stuck for years with viewers getting there. Also documents need to get printed, and when printer support might arrive is another unknown.
- We can embed a regular font with shapes that match the chosen values on the axis. This solution is way more efficient than the first.

Once I could interpret the right information in the font, the first route was the way to go. A side effect of having a converter for both outline types meant that it was trivial to create a virtual font at runtime. This option will stay in CONTeXt as pseudo-feature **variablesshapes**.

When trying to support variable fonts I tried to limit the impact on the backend code. Also, processing features and such was not touched. The inclusion of the right shapes

is done via a callback that requests the blob to be injected in the `cff` or `glyf` table. When implementing this I actually found out that the LUATEX backend also does some juggling of charstrings, to serve the purpose of inlining subroutines. In retrospect I could have learned a few tricks faster by looking at that code but I never realized that it was there. Looking at the code again, it strikes me that the whole inclusion could be done with LUA code and some day I will give that a try.

## Conclusion

When I first heard about variable fonts I was confident that when they showed up they could be supported. Of course a specimen was needed to prove this. A first implementation demonstrates that indeed it's no big deal to let CONTEXT with LUATEX handle such fonts. Of course we need to fill in some gaps which can be done once we have complete fonts. And then of course users will demand more control. In the meantime the helper script that deals with identifying fonts by name has been extended and the relevant code has been added to the distribution. At some point the CONTEXT Garden will provide the LUATEX binary that has the callback.

I end with a warning. On the one hand this technology looks promising but on the other hand one can easily get lost. Probably most such fonts operate over a well-defined domain of values but even then one should be aware of complex interactions with features like positioning or replacements. Not all combinations can be tested. It's probably best to stick to fonts that have all the relevant tables and don't depend on properties of a specific rendering technology.

Although support is now present in the core of CONTEXT the official release will happen at the CONTEXT meeting in 2017. By then I hope to have tested more fonts. Maybe the interface has also been extended by then because after all, T<sub>E</sub>X is about control.



## 4 Emoji again

Because at the CONTEX 2016 meeting color fonts<sup>5</sup> were on the agenda, some time was spent on emoji (these colorful small picture glyphs). When possible I bring kids to the BachoT<sub>E</sub>X conference so for the 2017 BachoTUG I decided to do something with emoji that, after all, are mostly used by those younger than I am. So, I had to take a look at the current state. Here are some observations.

The UNICODE standard defines a whole lot of emoji and if mankind manages to survive for a while one can assume that a lot more will be added. After all, icons as well as variants keep evolving. There are several ways to organize these symbols in groups but I will not give grouping a try. Just visit [emojipedia.org](http://emojipedia.org) and you get served well. For this story I only mention that:

- There are quite some shapes and nearly all of them are in color. The yellow ones, smilies and such, are quite prominently present but there are many more.
- A special subset is fulled by persons: man, woman, girl, boy and recently a baby.
- The grown ups can be combined in loving couples (either or not kissing) and then can form families, but only upto 2 young kids or gender neutral babies.
- All persons can be flagged with one of five skin tones so that not all persons (or heads) look bright yellow.
- Interesting is that girls and boys are still fond of magenta (pinkish) and cyan (blueish) cloths and ornaments. Also haircuts are rather specific to the gender.

For rendering color emojis we have a few color related OPENTYPE font properties available: bitmaps, SVG and stacked glyphs. Now, if you think of the combinations that can be made with skin tones, you realize that fonts can become pretty large if each combination results in a glyph. In the first half of 2017 MICROSOFT released an update for its emoji font and the company took the challenge to provide not only mixed skin tone couples, but also supported skin tones for the kids, including a baby.

This recent addition already adds over 25.000 additional glyphs<sup>6</sup> so imagine what will happen in the future. But, instead of making a picture for each variant, a different solution has been chosen. For coloring this seguiemj font uses the (very flexible) stacking technology: a color shape is an overlay of colored symbols. The colors are organized in pallets and it's no big deal to add additional pallets if needed. Instead of adding pre-composed shapes (as is needed with bitmaps and SVG) snippets are used to build

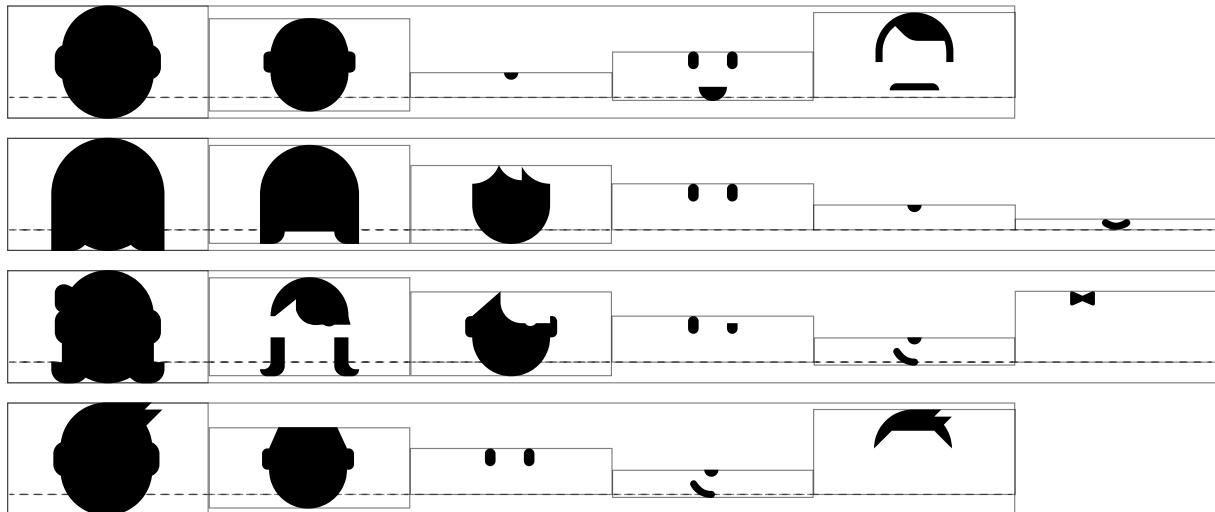
---

<sup>5</sup> For that occasion the cowfont, a practical joke concerning Dutch 'koeieletters', were turned into a color font and presented at the meeting.

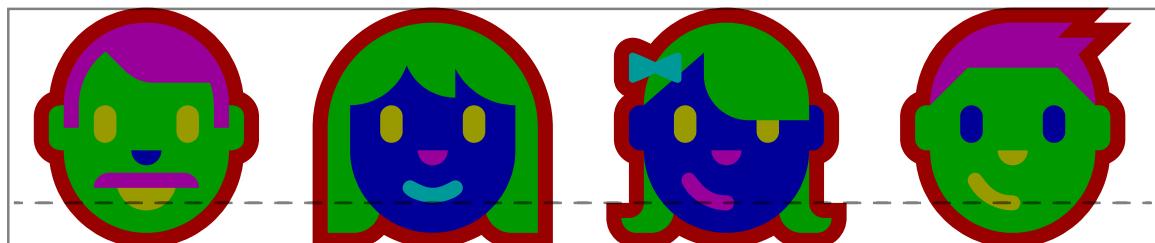
<sup>6</sup> That is the amount I counted when I added all combinations runtime but the emojipedia mentions twice that amount. Currently in CONTEX we resolve such combinations when requested.

alternative glyphs and these can be combined into new shapes by substitution and positioning (for that kerns, mark anchoring and distance compensation is used).

So, a family can be constructed of composed shapes (man, woman, etc) that each are composed of snippets (skull, hair, mouth, eyes). So, effectively a family of four is a bunch of maybe 25 small glyphs overlayed and colored. In figure 4.1 we see how a shape is constructed out of separate glyphs. Figure 4.2 shows how they can be overlaid with colors (we use a dedicated color set).



**Figure 4.1** Emoji snippets.

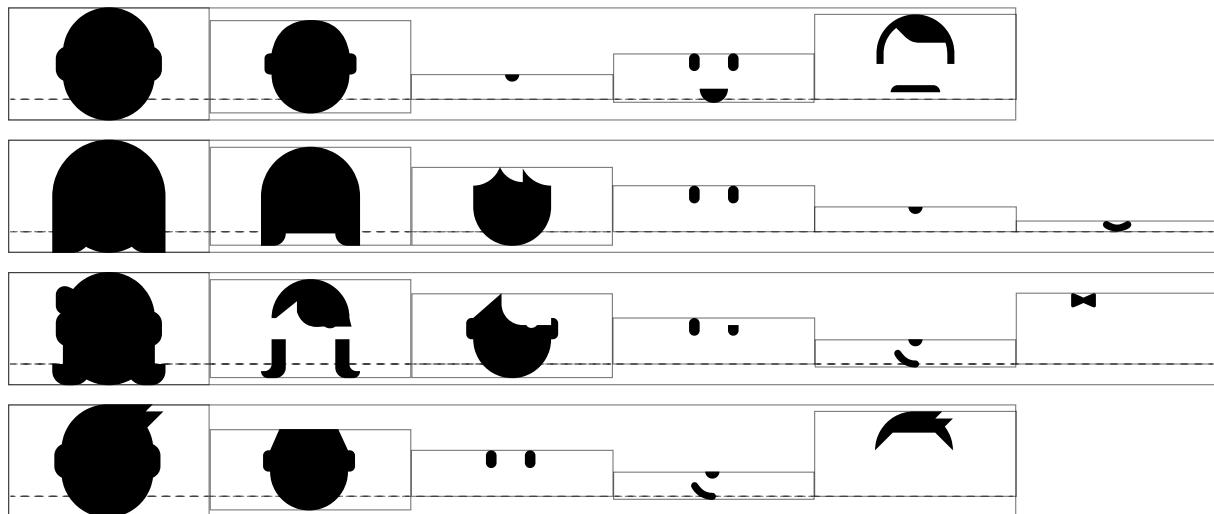


**Figure 4.2** Emoji snippets overlaid.

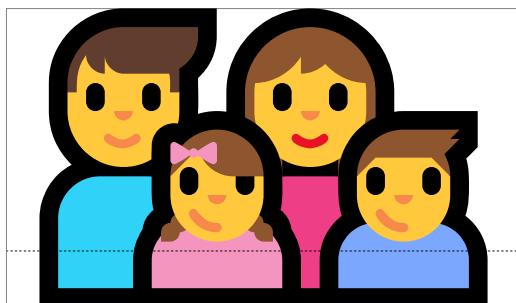
When a font supports it, a sequence of emoji can be turned into a more compact representation. In figure 4.3 we see how skin tones are applied in such combinations. Figure 4.4 shows the small snippets.

When we have to choose a font we need to take the following criteria into account:

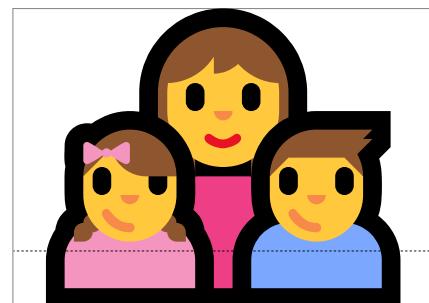
- What is the quality of the shapes? For sure, outlines are best if you want to scale too.
- How efficient is a shape constructed. In that respect a bitmap or SVG image is just one entity.
- How well can (semi) arbitrary combinations of emoji be provided. Here the glyph approach wins.



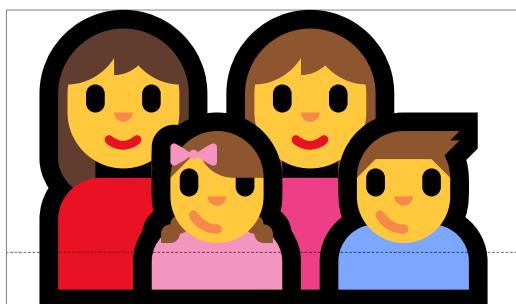
**Figure 4.4** Emoji glyphs.



family man woman girl boy



family woman girl boy



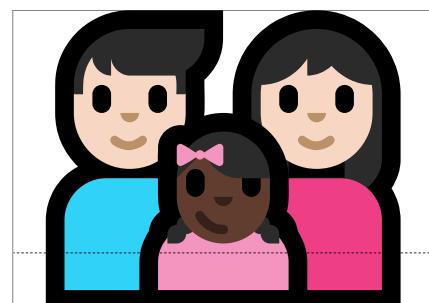
family woman woman girl boy



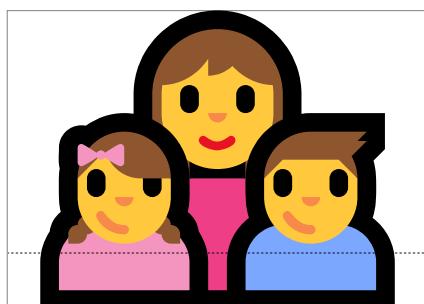
family man girl boy



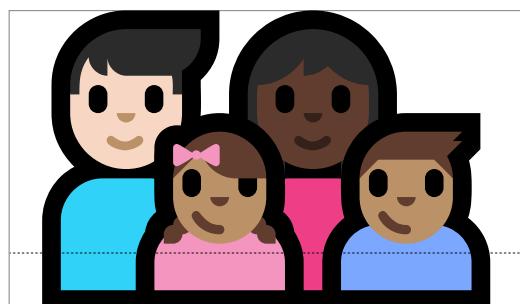
family man dark skin tone woman girl baby



family man light skin tone woman light skin tone girl dark skin tone



family woman girl boy



family man light skin tone woman dark skin tone girl medium skin tone boy medium skin tone

**Figure 4.3** Emoji families and such with skin tones.

- Are all skin colors for all human relates shapes supported? Actually it opens the possibility for racist fonts.
- Are all reasonable combinations of persons supported? It looks like (depending on time and version) kissing men or women can be missing, maybe because of social political reasons.
- Are black and white shapes provided alongside color shapes.

Maybe an SVG or bitmap image can have a lot of detail compared to a stacked glyph but, when we're just using pictographic representations, the later is the best choice.

When I was playing a bit with the skin tone variants and other combinations that should result in some composed shape, I used the UNICODE test files but I got the impression that there are some errors in the test suite, for instance with respect to modifiers. Maybe the fonts are just doing the wrong thing or maybe some implement these sequences a bit inconsistent. This will probably improve over time but the question is if we should intercept issues. I'm not in favour of this because it adds more and more fuzzy code that not only wastes cycles (energy) but is also a conceptual horror. So, when testing, imperfection has to be accepted for now. This is no big deal as until now no one ever asked for emoji support in CONTEXt.

When no combined shape is provided, the original sequence shows up. A side effect can be that zero-width-joiners and modifiers become visible. This depends on the fonts. Users probably don't care that much about it. Now how do we suppose that users enter these emoji (sequences) in a document source? One can imagine a pop up in the editor but T<sub>E</sub>Xies are often using commands for special cases.

We already showed some combined shapes. The reader might appreciate the outcome but getting there from the input takes a bit of work. For instance a two person man light skin tone woman medium skin tone girl medium-light skin tone baby medium-light skin tone involves this:

```
font      92: seguiemj.ttf @ 12.0pt
features [basic: ccmp=yes, dist=yes, mark=yes, mkmk=yes,
          script=dflt, tlig=yes, trep=yes] [extra: analyze=yes,
          autolanguage=position, autoscript=position, checkmarks=yes,
          colr=yes, curs=yes, devanagari=yes, dummies=yes,
          extensions=yes, extrafeatures=yes, extraprivates=yes,
          kern=yes, liga=yes, mathkerns=yes, mathrules=yes,
          mode=node, spacekern=yes, visualspace=yes]
step 1   ☺️肤色人种脸: [+TLT] U+1F468: ☺️ U+1F3FB: 🤵
          U+200D: | U+1F469: 🤵 U+1F3FD: 🤵 U+200D: | U+1F467: 🤶
          U+1F3FC: 🤵 U+200D: | U+1F476: 🤶 U+1F3FC: 🤵
```

```
feature 'ccmp', type 'gsub_ligature', lookup 's_s_0',
replacing U+1F468 upto U+1F3FB by ligature U+F01C5
case 2
```

```
feature 'ccmp', type 'gsub_ligature', lookup 's_s_0',
replacing U+1F469 upto U+1F3FD by ligature U+F01D2
case 2
```

```
feature 'ccmp', type 'gsub_ligature', lookup 's_s_0',
replacing U+1F467 upto U+1F3FC by ligature U+F01BC
case 2
```

```
feature 'ccmp', type 'gsub_ligature', lookup 's_s_0',
replacing U+1F476 upto U+1F3FC by ligature U+F020E
case 2
```

step 2 [+TLT] U+F01C5: U+200D: ↑ U+F01D2: U+200D: ↑ U+F01BC: U+200D: ↑ U+F020E:

```
feature 'ccmp', type 'gsub_contextchain', chain lookup
's_s_2', replacing single U+F01C5 by U+F147F
```

step 3 [+TLT] U+F147F: U+200D: ↑ U+F01D2: U+200D: ↑ U+F01BC: U+200D: ↑ U+F020E:

```
feature 'ccmp', type 'gsub_contextchain', chain lookup
's_s_3', index 1, replacing character U+200D upto
U+F01D2 by ligature U+F14A7 case 4
```

step 4 [+TLT] U+F147F: U+F14A7: U+200D: ↑ U+F01BC: U+200D: ↑ U+F020E:

```
feature 'ccmp', type 'gsub_contextchain', chain lookup
's_s_5', index 1, replacing character U+200D upto
U+F01BC by ligature U+F1474 case 4
```

step 5 [+TLT] U+F147F: U+F14A7: U+F1474: U+200D: ↑ U+F020E:

```
feature 'ccmp', type 'gsub_contextchain', chain lookup
's_s_6', index 1, replacing character U+200D upto
U+F020E by ligature U+F14C2 case 4
```

step 6 [+TLT] U+F147F: U+F14A7: U+F1474: U+F14C2:

```
feature 'ccmp', type 'gsub_contextchain', chain lookup
's_s_7', replacing single U+F1474 by U+F1467
```

step 7 [+TLT] U+F147F: U+F14A7: U+F1467: U+F14C2:

feature 'dist', type 'gpos\_single', lookup 'p\_s\_0',  
shifting single U+F147F by single xy (1.5pt,0pt) and  
wh (0pt,0pt)

- step 8  [+TLT] U+F147F: U+F14A7: U+F1467: U+F14C2: 
- feature 'dist', type 'gpos\_single', lookup 'p\_s\_1',  
shifting single U+F14A7 by single xy (0pt,0pt) and wh  
(1.5pt,0pt)
- step 9  [+TLT] U+F147F: U+F14A7: [kern] U+F1467: U+F14C2: 
- feature 'dist', type 'gpos\_contextchain', chain lookup  
'p\_s\_2', shifting single U+F147F by single (0pt,0pt)  
and correction (1.5pt,0pt)
- step 10  [+TLT] [kern] U+F147F: U+F14A7: [kern] U+F1467: U+F14C2: 
- feature 'dist', type 'gpos\_contextchain', chain lookup  
'p\_s\_3', shifting single U+F14A7 by single  
(5.71289pt,0pt) and correction (0pt,0pt)
- feature 'dist', type 'gpos\_contextchain', chain lookup  
'p\_s\_3', shifting single U+F14A7 by single (0pt,0pt)  
and correction (9.92578pt,0pt)
- step 11  [+TLT] [kern] U+F147F: [kern] U+F14A7: [kern] U+F1467: U+F14C2: 
- feature 'dist', type 'gpos\_contextchain', chain lookup  
'p\_s\_5', shifting single U+F147F by single (0pt,0pt)  
and correction (-5.71289pt,0pt)
- step 12  [+TLT] [kern] U+F147F: [kern] [kern] U+F14A7: [kern] U+F1467: U+F14C2: 
- feature 'mark', type 'gpos\_mark2base', lookup 'p\_s\_27',  
bound 1, anchoring mark U+F1467 to basechar U+F14A7  
=> (7.59375pt,0pt)
- step 13  [+TLT] [kern] U+F147F: [kern] [kern] U+F14A7: [kern] U+F1467: U+F14C2: 
- feature 'mark', type 'gpos\_mark2base', lookup 'p\_s\_28',  
bound 2, anchoring mark U+F14C2 to basechar U+F14A7  
=> (0.01172pt,0pt)

**result**  [+TLT] [kern] U+F147F: [kern] [kern] U+F14A7:  
[kern] U+F1467: U+F14C2:

A black and white example is the following **family woman girl**:

**font** 95: seguiemj.ttf @ 12.0pt

**features** [basic: ccmp=yes, dist=yes, mark=yes, mkmk=yes, script=dflt, tlig=yes, trep=yes] [extra: analyze=yes, autolanguage=position, autoscript=position, checkmarks=yes, curs=yes, devanagari=yes, dummies=yes, extensions=yes, extrafeatures=yes, extraprivates=yes, kern=yes, liga=yes, mathkerns=yes, mathrules=yes, mode=node, spacekern=yes, visualspace=yes]

**step 1**  [+TLT] U+1F469: U+200D: † U+1F467:

feature 'ccmp', type 'gsub\_contextchain', chain lookup 's\_s\_4', replacing single U+1F469 by U+F149D

**step 2**  [+TLT] U+F149D: U+200D: † U+1F467:

feature 'ccmp', type 'gsub\_contextchain', chain lookup 's\_s\_5', index 1, replacing character U+200D upto U+1F467 by ligature U+F146B case 4

**step 3**  [+TLT] U+F149D: U+F146B:

feature 'dist', type 'gpos\_single', lookup 'p\_s\_1', shifting single U+F149D by single xy (0pt,0pt) and wh (1.5pt,0pt)

**step 4**  [+TLT] U+F149D: [kern] U+F146B:

feature 'dist', type 'gpos\_contextchain', chain lookup 'p\_s\_4', shifting single U+F149D by single (0pt,0pt) and correction (1.5pt,0pt)

feature 'dist', type 'gpos\_contextchain', chain lookup 'p\_s\_4', shifting single U+F149D by single (0pt,0pt) and correction (5.71289pt,0pt)

**step 5**  [+TLT] [kern] U+F149D: [kern] U+F146B:

feature 'mark', type 'gpos\_mark2base', lookup 'p\_s\_28', bound 1, anchoring mark U+F146B to basechar U+F149D => (0.01172pt,0pt)

**result**  [+TLT] [kern] U+F149D: [kern] U+F146B:

I will not show all emoji, just the subset that contains the word **woman** in the description. As you can see the persons in the sequences are separated by a zero-width-joiner. There are some curious ones, for instance a **woman wearing turban** which in terms of UNICODE input is a female combine with a turban wearing man becomes a beardless woman wearing a turban. Woman vampires and zombies are not supported so these are male properties.

👩	👩 ♀	blondhaired woman
👩 ❤️ 👨	👩 ❤️ 👨	couple with heart woman man
👩 ❤️ 👩	👩 ❤️ 👩	couple with heart woman woman
👨 👩 👦	👨 👩 👦	family man woman boy
👨 👩 👦 👦	👨 👩 👦 👦	family man woman boy boy
👨 👩 👧	👨 👩 👧	family man woman girl
👨 👩 👧 👧	👨 👩 👧 👧	family man woman girl boy
👨 👩 👧 👧 👧	👨 👩 👧 👧 👧	family man woman girl girl
👨 👦	👨 👦	family woman boy
👩 👦 👦	👩 👦 👦	family woman boy boy
👩 👧	👩 👧	family woman girl
👩 👧 👦	👩 👧 👦	family woman girl boy
👩 👧 👧	👩 👧 👧	family woman girl girl
👩 👧 👧 👦	👩 👧 👧 👦	family woman woman boy
👩 👧 👧 👦 👦	👩 👧 👧 👦 👦	family woman woman boy boy
👩 👧 👧 👧	👩 👧 👧 👧	family woman woman girl
👩 👧 👧 👧 👦	👩 👧 👧 👧 👦	family woman woman girl boy
👩 👧 👧 👧 👧	👩 👧 👧 👧 👧	family woman woman girl girl
👩 ❤️ 💋 👨	👩 ❤️ 💋 👨	kiss woman man
👩 ❤️ 💋 👩	👩 ❤️ 💋 👩	kiss woman woman
👫	👫	man and woman holding hands
👵	👵	old woman
🤰	🤰	pregnant woman
👩	👩	woman
👩 🎨	👩 🎨	woman artist
👩 🚶	👩 🚶	woman astronaut
👩 ⚑	👩 ⚑	woman bald
👩 🚴	👩 🚴	woman biking
👢	👢	woman boot
👩 ⚽	👩 ⚽	woman bouncing ball
👩 🎹	👩 🎹	woman bowing
👩 🤵	👩 🤵	woman cartwheeling
👩 🏆	👩 🏆	woman climbing
👚	👚	woman clothes
👩 🚧	👩 🚧	woman construction worker
👩 🔎	👩 🔎	woman cook
👩 🍷	👩 🍷	woman curly haired
👩 🎵	👩 🎵	woman dancing
👩 🕵️	👩 🕵️	woman detective

👩‍ǩ	👩‍ǩ	woman elf
👩‍Ǌ	👩‍Ǌ	woman facepalming
👩‍⼯	👩‍⼯	woman factory worker
👩‍Ǫ	👩‍Ǫ	woman fairy
👩‍农	👩‍农	woman farmer
👩‍🚒	👩‍🚒	woman firefighter
👩‍👟	👩‍👟	woman flat shoe
👩‍🙍	👩‍🙍	woman frowning
👩‍🧞	👩‍🧞	woman genie
👩‍手势	👩‍手势	woman gesturing no
👩‍手势ok	👩‍手势ok	woman gesturing ok
👩‍💇	👩‍💇	woman getting haircut
👩‍💆	👩‍💆	woman getting massage
👩‍⛳	👩‍⛳	woman golfing
👩‍💂	👩‍💂	woman guard
👩‍👒	👩‍👒	woman hat
👩‍⚕️	👩‍⚕️	woman health worker
👩‍🧘	👩‍🧘	woman in lotus position
👩‍🧖	👩‍🧖	woman in steamy room
👩‍⚖️	👩‍⚖️	woman judge
👩‍🤹	👩‍🤹	woman juggling
👩‍🏋️	👩‍🏋️	woman lifting weights
👩‍🧙	👩‍🧙	woman mage
👩‍🔧	👩‍🔧	woman mechanic
👩‍🚵	👩‍🚵	woman mountain biking
👩‍💼	👩‍💼	woman office worker
👩‍✈️	👩‍✈️	woman pilot
👩‍🤾	👩‍🤾	woman playing handball
👩‍🤽	👩‍🤽	woman playing water polo
👩‍👮	👩‍👮	woman police officer
👩‍🙎	👩‍🙎	woman pouting
👩‍🙋	👩‍🙋	woman raising hand
👩‍🔴	👩‍🔴	woman red haired
👩‍🚣	👩‍🚣	woman rowing boat
👩‍🏃	👩‍🏃	woman running
👩‍👡	👩‍👡	woman sandal
👩‍🔬	👩‍🔬	woman scientist
👩‍🤷	👩‍🤷	woman shrugging
👩‍🎤	👩‍🎤	woman singer
👩‍🎓	👩‍🎓	woman student
👩‍🦸	👩‍🦸	woman superhero
👩‍🧙	👩‍🧙	woman supervillain
👩‍🏄	👩‍🏄	woman surfing
👩‍🏊	👩‍🏊	woman swimming
👩‍🏫	👩‍🏫	woman teacher

		woman technologist
		woman tipping hand
		woman vampire
		woman walking
		woman wearing turban
		woman white haired
		woman with headscarf
		woman zombie

So what if you don't like these colors? Because we're dealing with TeX you can assume that if there is some way around the fixed color sets, then it will be provided. So, when you use CONTEXT, here is away to overload them:

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

\definefontcolorpalette
[emoji-s]
[black,emoji-gray]

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

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

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

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

\definefontfeature[seguiemj-s] [ccmp=yes,dist=yes,colr=emoji-s]
\definefontfeature[seguiemj-r] [ccmp=yes,dist=yes,colr=emoji-r]
\definefontfeature[seguiemj-g] [ccmp=yes,dist=yes,colr=emoji-g]
\definefontfeature[seguiemj-b] [ccmp=yes,dist=yes,colr=emoji-b]
\definefontfeature[seguiemj-y] [ccmp=yes,dist=yes,colr=emoji-y]

\definefont[MyEmojiS] [seguiemj*seguiemj-s]
```

```
\definefont[MyEmojiR][seguiemj*seguiemj-r]
\definefont[MyEmojiG][seguiemj*seguiemj-g]
\definefont[MyEmojiB][seguiemj*seguiemj-b]
\definefont[MyEmojiY][seguiemj*seguiemj-y]
```

In figure 4.5 we see how this is applied. You can provide as many colors as needed but when you don't provide enough the last one is used. This way we get the overlayed transparent colors in the examples. By using transparency we don't obscure shapes.

The emojipedia mentions "Asked about the design, MICROSOFT told emojipedia that one of the reasons for the thick stroke was to allow each emoji to be easily read on any background color." The first glyph in the stack seems to do the trick, so just make sure that it doesn't become white. And, before I read that remark, while preparing a presentation with a colored background, I had already noticed that using a background was no problem. This font definitely sets the standard.

How do we know what colors are used? The next table shows the first color palette of seguiemj. There are quite some colors so defining your own definitely involved some studying.

1	1	2	3	4	5	6	7	8	9	10	11	12
	13	14	15	16	17	18	19	20	21	22	23	24
	25	26	27	28	29	30	31	32	33	34	35	36
	37	38	39	40	41	42	43	44	45	46	47	48
	49	50	51	52	53	54	55	56	57	58	59	60
	61	62	63	64	65	66	67	68	69	70	71	72
	73	74	75	76	77	78	79	80	81	82	83	84
	85	86	87	88	89	90	91	92	93	94	95	96
	97	98	99	100	101	102	103	104	105	106	107	108
	109	110	111	112	113	114	115	116	117	118	119	120
	121	122	123	124	125	126	127	128	129	130	131	132
	133	134	135	136	137	138	139	140	141	142	143	144
	145	146	147	148	149	150	151	152	153	154	155	156
	157	158	159	160	161	162	163	164	165	166	167	168
	169	170	171	172	173	174	175					

Normally special symbols are accessed in CONTEXt with the `symbol` command where symbols are organized in symbol sets. This is a rather old mechanism and dates from the time that fonts were limited in coverage and symbols were collected in special fonts. The emoji are accessed by their own command: `\emoji`. The font used has the font synonym `emoji` so you need to set that one first:

```
\definefontsynonym[emoji][seguiemj*seguiemj-cl]
```

Here is an example:

```
\emoji{woman light skin tone}\quad
```



**Figure 4.5** Overloading colors by plugging in a sequence of alternate colors.

```
\emoji{woman scientist}\quad
{\bf bigger} \emoji{man health worker}}
```

or typeset:   **bigger** 

The emoji symbol scales with the normal running font. When you ask for a family with skin toned members the lookup can result in another match (or no match) because one never knows to what extent a font supports it.

\expandedemoji	the sequence constructed from the given string
\resolvedemoji	a protected sequence constructed from the given string
\checkedemoji	an typeset sequence with unresolved modifiers and joiners removed
\emoji	a typeset resolved sequence using the <code>emoji</code> font synonym
\robustemoji	a typeset checked sequence using the <code>emoji</code> font synonym

In case you wonder how some of the details above were typeset, there is a module `fonts-emoji` that provides some helpers for introspection.

\ShowEmoji	show all the emoji in the current font
\ShowEmojiSnippets	show the snippets of a given emoji
\ShowEmojiSnippetsOverlay	show the overlayed snippets of a given emoji
\ShowEmojiGlyphs	show the snippets of a typeset emoji
\ShowEmojiPalettes	show the color pallets in the current font

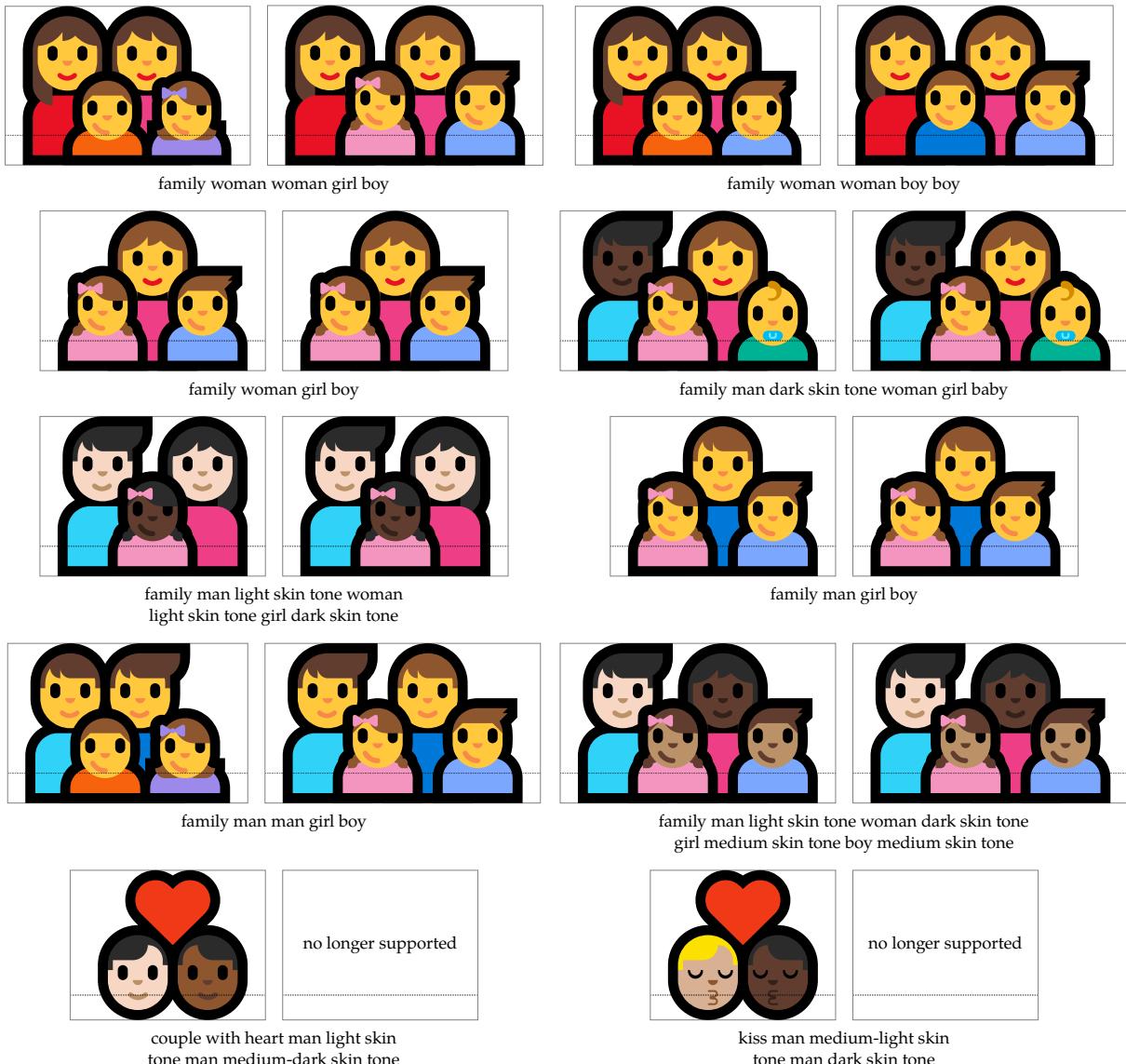
Examples of usage are:

```
\ShowEmojiSnippets[family man woman girl boy]
\ShowEmojiGlyphs [family man woman baby girl]
\ShowEmoji [^man]
\ShowEmoji
\ShowEmojiPalettes
\ShowEmojiPalettes[1]
```

A good source of information about emoji is the mentioned [emojipedia.org](http://emojipedia.org) website. There you find not only details about all these symbols but also has some history. It compares updates in fonts too. It mentions for instance that in the creative update of Windows 10, some persons grew beards in the seguiemj font and others lost an eye. Now, if you look at the snippets shown before, you can wonder if that eye is really gone. Maybe the color is wrong or the order of stacking is not right. I decided not to waste time looking into that.

Another quote: "Support for color emoji presentation on MS WINDOWS is limited. Many applications on MS WINDOWS display emojis with a black and white text presentation instead of their color version." Well, we can do better with T<sub>E</sub>X, but as usual not that many people really cares about that. But it's fun anyway.

We end with a warning. When you use 'ligatures' like this, you really need to check the outcome. For instance, when MICROSOFT updated the font end 2017, same gender couples got different hair style for the individuals so that one can still distinguish them. However, kissing couples and couples in love (indicated by a heart) seem to be removed. Who knows how and when politics creep into fonts: is public mixed couple kissing permitted, do we support families with any mix of gender, is associating pink with girls okay or not, how do we distinguish male and female anyway? In figure 4.6 we see the same combination twice, the early 2017 rendering versus the late 2017 rendering. Can you notice the differences?



**Figure 4.6** Incompatible updates.



# 5 Performance

## 5.1 Introduction

This chapter is about performance. Although it concerns LUATEX this text is only meant for CONTEXT users. This is not because they ever complain about performance, on the contrary, I never received a complain from them. No, it's because it gives them some ammunition against the occasionally occurring nagging about the speed of LUATEX (somewhere on the web or at some meeting). My experience is that in most such cases those complaining have no clue what they're talking about, so effectively we could just ignore them, but let's, for the sake of our users, waste some words on the issue.

## 5.2 What performance

So what exactly does performance refer to? If you use CONTEXT there are probably only two things that matter:

- How long does one run take.
- How many runs do I need.

Processing speed is reported at the end of a run in terms of seconds spent on the run, but also in pages per second. The runtime is made up out of three components:

- start-up time
- processing pages
- finishing the document

The startup time is rather constant. Let's take my 2013 Dell Precision with i7-3840QM as reference. A simple

```
\starttext  
\stoptext
```

document reports 0.4 seconds but as we wrap the run in an `mtxrun` management run we have an additional 0.3 overhead (auxiliary file handling, PDF viewer management, etc). This includes loading the Latin Modern font. With LUAJITTEX these times are below 0.3 and 0.2 seconds. It might look like much overhead but in an edit-preview runs it feels snappy. One can try this:

```
\stoptext
```

which bring down the time to about 0.2 seconds for both engines but as it doesn't do anything useful that is is no practice.

Finishing a document is not that demanding because most gets flushed as we go. The more (large) fonts we use, the longer it takes to finish a document but on the average

that time is not worth noticing. The main runtime contribution comes from processing the pages.

Okay, this is not always true. For instance, if we process a 400 page book from 2500 small XML files with multiple graphics per page, there is a little overhead in loading the files and constructing the XML tree as well as in inserting the graphics but in such cases one expects a few seconds more runtime. The METAFUN manual has some 450 pages with over 2500 runtime generated METAPOST graphics. It has color, uses quite some fonts, has lots of font switches (verbatim too) but still one run takes only 18 seconds in stock LUATEX and less than 15 seconds with LUAJITTEX. Keep these numbers in mind if a non-CONTEXT users barks against the performance tree that his few page mediocre document takes 10 seconds to compile: the content, styling, quality of macros and whatever one can come up with all plays a role. Personally I find any rate between 10 and 30 pages per second acceptable, and if I get the lower rate then I normally know pretty well that the job is demanding in all kind of aspects.

Over time the CONTEXT-LUATEX combination, in spite of the fact that more functionality has been added, has not become slower. In fact, some subsystems have been sped up. For instance font handling is very sensitive for adding functionality. However, each version so far performed a bit better. Whenever some neat new trickery was added, at the same time improvements were made thanks to more insight in the matter. In practice we're not talking of changes in speed by large factors but more by small percentages. I'm pretty sure that most CONTEXT users never noticed. Recently a 15–30% speed up (in font handling) was realized (for more complex fonts) but only when you use such complex fonts and pages full of text you will see a positive impact on the whole run.

There is one important factor I didn't mention yet: the efficiency of the console. You can best check that by making a format (`context --make en`). When that is done by piping the messages to a file, it takes 3.2 seconds on my laptop and about the same when done from the editor (SCITE), maybe because the LUATEX run and the log pane run on a different thread. When I use the standard console it takes 3.8 seconds in Windows 10 Creative update (in older versions it took 4.3 and slightly less when using a console wrapper). The powershell takes 3.2 seconds which is the same as piping to a file. Interesting is that in Bash on Windows it takes 2.8 seconds and 2.6 seconds when piped to a file. Normal runs are somewhat slower, but it looks like the 64 bit Linux binary is somewhat faster than the 64 bit mingw version.<sup>7</sup> Anyway, it demonstrates that when someone yells a number you need to ask what the conditions where.

At a CONTEXT meeting there has been a presentation about possible speed-up of a run for instance by using a separate syntax checker to prevent a useless run. However, the use case concerned a document that took a minute on the machine used, while the same document took a few seconds on mine. At the same meeting we also did a comparison

---

<sup>7</sup> Long ago we found that LUATEX is very sensitive to for instance the CPU cache so maybe there are some differences due to optimization flags and/or the fact that bash runs in one thread and all file IO in the main windows instance. Who knows.

of speed for a LATEX run using PDFTEX and the same document migrated to CONTEXT MKIV using LUATEX (Harald Königs XML torture and compatibility test). Contrary to what one might expect, the CONTEXT run was significantly faster; the resulting document was a few gigabytes in size.

## 5.3 Bottlenecks

I will discuss a few potential bottlenecks next. A complex integrated system like CONTEXT has lots of components and some can be quite demanding. However, when something is not used, it has no (or hardly any) impact on performance. Even when we spend a lot of time in LUA that is not the reason for a slow-down. Sometimes using LUA results in a speedup, sometimes it doesn't matter. Complex mechanisms like natural tables for instance will not suddenly become less complex. So, let's focus on the "aspects" that come up in those complaints: fonts and LUA. Because I only use CONTEXT and occasionally test with the plain TEX version that we provide, I will not explore the potential impact of using truckloads of packages, styles and such, which I'm sure of plays a role, but one neglected in the discussion.

### Fonts

According to the principles of LUATEX we process (OPENTYPE) fonts using LUA. That way we have complete control over any aspect of font handling, and can, as to be expected in TEX systems, provide users what they need, now and in the future. In fact, if we didn't had that freedom in CONTEXT I'd probably already quit using TEX a decade ago and found myself some other (programming) niche.

After a font is loaded, part of the data gets passed to the TEX engine so that it can do its work. For instance, in order to be able to typeset a paragraph, TEX needs to know the dimensions of glyphs. Once a font has been loaded (that is, the binary blob) the next time it's fetched from a cache. Initial loading (and preparation) takes some time, depending on the complexity or size of the font. Loading from cache is close to instantaneous. After loading the dimensions are passed to TEX but all data remains accessible for any desired usage. The OPENTYPE feature processor for instance uses that data and CONTEXT for sure needs that data (fast accessible) for different purposes too.

When a font is used in so called base mode, we let TEX do the ligaturing and kerning. This is possible with simple fonts and features. If you have a critical workflow you might enable base mode, which can be done per font instance. Processing in node mode takes some time but how much depends on the font and script. Normally there is no difference between CONTEXT and generic usage. In CONTEXT we also have dynamic features, and the impact on performance depends on usage. In addition to base and node we also have plug mode but that is only used for testing and therefore not advertised.

Every \hbox and every paragraph goes through the font handler. Because we support mixed modes, some analysis takes place, and because we do more in CONTEXT, the

generic analyzer is more light weight, which again can mean that a generic run is not slower than a similar CONTEXT one.

Interesting is that added functionality for variable and/or color fonts had no impact on performance. Runtime added user features can have some impact but when defined well it can be neglected. I bet that when you add additional node list handling yourself, its impact on performance is larger. But in the end what counts is that the job gets done and the more you demand the higher the price you pay.

## LUA

The second possible bottleneck when using LUATEX can be in using LUA code. However, using that as argument for slow runs is laughable. For instance CONTEXT MKIV can easily spend half its time in LUA and that is not making it any slower than MKII using PDFTEX doing equally complex things. For instance the embedded METAPOST library makes MKIV way faster than MKII, and the built-in XML processing capabilities in MKIV can easily beat MKII XML handling, apart from the fact that it can do more, like filtering by path and expression. In fact, files that take, say, half a minute in MKIV, could as well have taken 15 minutes or more in MKII (and imagine multiple runs then).

So, for CONTEXT using LUA to achieve its objectives is mandate. The combination of TEX, METAPOST and LUA is pretty powerful! Each of these components is really fast. If TEX is your bottleneck, review your macros! When LUA seems to be the bad, go over your code and make it better. Much of the LUA code I see flying around doesn't look that efficient, which is okay because the interpreter is really fast, but don't blame LUA beforehand, blame your coding (style) first. When METAPOST is the bottleneck, well, sometimes not much can be done about it, but when you know that language well enough you can often make it perform better.

For the record: every additional mechanism that kicks in, like character spacing (the ugly one), case treatments, special word and line trickery, marginal stuff, graphics, line numbering, underlining, referencing, and a few dozen more will add a bit to the processing time. In that case, in CONTEXT, the font related runtime gets pretty well obscured by other things happening, just that you know.

## 5.4 Some timing

Next I will show some timings related to fonts. For this I use stock LUATEX (second column) as well as LUAJITTEX (last column) which of course performs much better. The timings are given in 3 decimals but often (within a set of runs) and as the system load is normally consistent in a set of test runs the last two decimals only matter in relative comparison. So, for comparing runs over time round to the first decimal. Let's start with loading a bodyfont. This happens once per document and normally one has only one bodyfont active. Loading involves definitions as well as setting up math so a couple

of fonts are actually loaded, even if they're not used later on. A setup normally involves a serif, sans, mono, and math setup (in CONTEXT).<sup>8</sup>

---

### **bodyfont**

---

modern	0.023	0.019
pagella	0.127	0.079
termes	0.128	0.087
cambria	0.180	0.123
dejavu	0.140	0.092
ebgaramond	0.142	0.093
lucidaot	0.146	0.120

---

There is a bit difference between the font sets but a safe average is 150 milli seconds and this is rather constant over runs.

An actual font switch can result in loading a font but this is a one time overhead. Loading four variants (regular, bold, italic and bold italic) roughly takes the following time:

---

### **bodyfont switch and 4 style changes (first time)**

---

modern	0.028	0.028
pagella	0.035	0.031
termes	0.036	0.069
cambria	0.052	0.047
dejavu	0.091	0.069
ebgaramond	0.022	0.016
lucidaot	0.017	0.031

---

Using them again later on takes no time:

---

### **bodyfont switch and 4 style changes (follow up)**

---

modern	0.000	0.000
pagella	0.001	0.000
termes	0.000	0.001
cambria	0.000	0.000
dejavu	0.001	0.000
ebgaramond	0.000	0.000
lucidaot	0.000	0.000

---

Before we start timing the font handler, first a few baseline benchmarks are shown. When no font is applied and nothing else is done with the node list we get:

---

<sup>8</sup> The timing for Latin Modern is so low because that font is loaded already.

---

**100 hboxes with 4 texts and no font handling**

---

baseline 0.142 2.343

---

A simple monospaced, no features applied, run takes a bit more:

---

**100 hboxes with 4 texts and no features**

---

baseline 0.275 0.220

---

Now we show a one font typesetting run. As the two benchmarks before, we just typeset a text in a `\hbox`, so no par builder interference happens. We use the `sapolsky` sample text and typeset it 100 times 4 (either of not with font switches).

---

**100 hboxes with 4 texts using one font**

---

modern	0.933	0.591
pagella	1.027	0.660
termes	1.032	0.604
cambria	1.483	0.862
dejavu	1.009	0.581
ebgaramond	3.240	1.774
lucidaot	0.699	0.444

---

Much more runtime is needed when we typeset with four font switches. The garamond is most demanding. Actually we're not doing 4 fonts there because it has no bold, so the numbers are a bit lower than expected for this example. One reason for it being demanding is that it has lots of (contextual) lookups. The only comment I can make about that is that it also depends on the strategies of the font designer. Combining lookups saves space and time so complexity of a font is not always a good predictor for performance hits.

If we typeset paragraphs we get this:

---

**100 times 4 texts on pages**

---

modern	1.377	0.904
pagella	1.523	0.961
termes	1.453	0.898
cambria	1.901	1.138
dejavu	1.437	0.917
ebgaramond	3.714	2.133
lucidaot	1.117	0.767

---

We're talking of some 275 pages here.

---

**100 times 4 texts on pages using 4 styles**

---

modern	2.074	1.307
pagella	2.155	1.338
termes	2.153	1.373
cambria	3.349	2.012
dejavu	2.408	1.453
ebgaramond	4.368	2.512
lucidaot	1.682	1.056

---

There is of course overhead in handling paragraphs and pages:

---

**100 paragraphs with 4 texts and no features**

---

baseline	0.825	0.559
----------	-------	-------

---

Before I discuss these numbers in more details two more benchmarks are shown. The next table concerns a paragraph with only a few (**bold**) words.

---

**100 texts on pages with [1,2,4] bold font switches**

---

modern	0.409	0.263
pagella	0.445	0.281
termes	0.432	0.300
cambria	0.606	0.368
dejavu	0.465	0.295
ebgaramond	0.922	0.530
lucidaot	0.345	0.220

---

The following table has paragraphs with a few mono spaced words typeset using `\type`.

---

**100 texts on pages with [1,2,4] word verbatim switches**

---

modern	0.380	0.255
pagella	0.396	0.266
termes	0.384	0.278
cambria	0.535	0.355
dejavu	0.366	0.247
ebgaramond	0.939	0.533
lucidaot	0.322	0.216

---

When a node list (hbox or paragraph) is processed, each glyph is looked at. One important property of LUATeX (compared to PDFTeX) is that it hyphenates the whole text, not only the most feasible spots. For the `sapolsky` snippet this results in 200 potential breakpoints, registered in an equal number of discretionary nodes. The snippet has 688 characters grouped into 125 words and because it's an English quote we're not hampered with composed characters or complex script handling. And, when we mention 100 runs then we actually mean 400 ones when font switching and bodyfonts are compared

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.

Robert M. Sapolsky

In order to get substitutions and positioning right we need not only to consult streams of glyphs but also combinations with preceding pre or replace, or trailing post and replace texts. When a font has a bit more complex substitutions, as ebgaramond has, multiple (sometimes hundreds of) passes over the list are made. This is why the more complex a font is, the more runtime is involved.

Another factor, one you could easily deduce from the benchmarks, is intermediate font switches. Even a few such switches (in the last benchmarks) already result in a runtime penalty. The four switch benchmarks show an impressive increase of runtime, but it's good to know that such a situation seldom happens. It's also important not to confuse for instance a verbatim snippet with a bold one. The bold one is indeed leading to a pass over the list, but verbatim is normally skipped because it uses a font that needs no processing. That verbatim or bold have the same penalty is mainly due to the fact that verbatim itself is costly: the text is picked up using a different catcode regime and travels through  $\text{\TeX}$  and LUA before it finally gets typeset. This relates to special treatments of spacing and syntax highlighting and such.

Also keep in mind that the page examples are quite unreal. We use a layout with no margins, just text from edge to edge.

So what is a realistic example? That is hard to say. Unfortunately no one ever asked us to typeset novels. They are rather brain dead products for a machinery so they process fast. On the mentioned laptop 350 word pages in Dejavu fonts can be processed at a rate of 75 pages per second with LUATEX and over 100 pages per second with LUAJITTEX. On a more modern laptop or professional server performance is of course better. And for automated flows batch mode is your friend. The rate is not much worse for a document in a language with a bit more complex character handling, take accents or ligatures. Of course PDFTEX is faster on such a dumb document but kick in some more functionality and the advantage quickly disappears. So, if someone complains that LUATEX needs 10 or more seconds for a simple few page document . . . you can bet that when the fonts are seen as reason, that the setup is pretty bad. Personally I'd not waste time on such a complaint.

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. 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 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. 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 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. 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 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. 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 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. 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 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.

**Figure 5.1**

**Figure 5.2**



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 ... stratification of society and the invention of classes. Thus, it allowed for the invention of p I think that the punch line of the primate-human difference is that when humans invented poverty, t came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. 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 ... stratification of society and the invention of classes. Thus, it allowed for the invention of p I think that the punch line of the primate-human difference is that when humans invented poverty, t came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. 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 ... stratification of society and the invention of classes. Thus, it allowed for the invention of p I think that the punch line of the primate-human difference is that when humans invented poverty, t came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world. 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 ... stratification of society and the invention of classes. Thus, it allowed for the invention of p I think that the punch line of the primate-human difference is that when humans invented poverty, t came up with a way of subjugating the low-ranking like nothing ever seen before in the primate world.

**Figure 5.5**

## 5.5 Valid questions

Here are some reasonable questions that you can ask when someone complains to you about the slowness of LUATEX:

### What engines do you compare?

If you come from PDFTEX you come from an 8 bit world: input and font handling are based on bytes and hyphenation is integrated into the par builder. If you use UTF-8 in PDFTEX, the input is decoded by TeX macros which carries a speed penalty. Because in the wide engines macro names can also be UTF sequences, construction of macro names is less efficient too.

When you try to use wide fonts, again there is a penalty. Now, if you use XETEX or LUATEX your input is UTF-8 which becomes something 32 bit internally. Fonts are wide so more resources are needed, apart from these fonts being larger and in need of more processing due to feature handling. Where XETEX uses a library, LUATEX uses its own handler. Does that have a consequence for performance? Yes and no. First of all it depends on how much time is spent on fonts at all, but even then the difference is not that large. Sometimes XETEX wins, sometimes LUATEX. One thing is clear: LUATEX is more flexible as we can roll out our own solutions and therefore do more advanced font magic. For CONTEX it doesn't matter as we use LUATEX exclusively and rely on the flexible font handler, also for future extensions. If really needed you can kick in a library based handler but it's (currently) not distributed as we loose other functionality

which in turn would result in complaints about that fact (apart from conflicting with the strive for independence).

There is no doubt that **PDFTEX** is faster but for **CONTEX**T it's an obsolete engine. The hard coded solutions engine **X<sub>H</sub>TEX** is also not feasible for **CONTEX**T either. So, in practice **CONTEX**T users have no choice: **LUATEX** is used, but users of other macro packages can use the alternatives if they are not satisfied with performance. The fact that **CONTEX**T users don't complain about speed is a clear signal that this is no issue. And, if you want more speed you can use **LUAJITTEX**.<sup>9</sup> In the last section the different engines will be compared in more detail.

Just that you know, when we do the four switches example in plain **T<sub>E</sub>X** on my laptop I get a rate of 40 pages per second, and for one font 180 pages per second. There is of course a bit more going on in **CONTEX**T in page building and so, but the difference between plain and **CONTEX**T is not that large.

### What macro package is used?

If the answer is that when plain **T<sub>E</sub>X** is used, a follow up question is: what variant? The **CONTEX**T distribution ships with **luatex-plain** and that is our benchmark. If there really is a bottleneck it is worth exploring. But keep in mind that in order to be plain, not that much can be done. The **LUATEX** part is just an example of an implementation. We already discussed **CONTEX**T, and for **IATEX** I don't want to speculate where performance hits might come from. When we're talking fonts, **CONTEX**T can actually a bit slower than the generic (or **IATEX**) variant because we can kick in more functionality. Also, when you compare macro packages, keep in mind that when node list processing code is added in that package the impact depends on interaction with other functionality and depends on the efficiency of the code. You can't compare mechanisms or draw general conclusions when you don't know what else is done!

### What do you load?

Most **CONTEX**T modules are small and load fast. Of course there can be exceptions when we rely on third party code; for instance loading **tikz** takes a bit of time. It makes no sense to look for ways to speed that system up because it is maintained elsewhere. There can probably be gained a bit but again, no user complained so far.

If **CONTEX**T is not used, one probably also uses a large **T<sub>E</sub>X** installations. File lookup in **CONTEX**T is done differently and can be faster. Even loading can be more efficient in **CONTEX**T, but it's hard to generalize that conclusion. If one complains about loading fonts being an issue, just try to measure how much time is spent on loading other code.

---

<sup>9</sup> In plug mode we can actually test a library and experiments have shown that performance on the average is much worse but it can be a bit better for complex scripts, although a gain gets unnoticed in normal documents. So, one can decide to use a library but at the cost of much other functionality that **CONTEX**T offers, so we don't support it.

## Did you patch macros?

Not everyone is a  $\text{\TeX}$ pert. So, coming up with macros that are expanded many times and/or have inefficient user interfacing can have some impact. If someone complains about one subsystem being slow, then honestly demands to complain about other subsystems as well. You get what you ask for.

## How efficient is the code that you use?

Writing super efficient code only makes sense when it's used frequently. In  $\text{CONTEX}$ T most code is reasonable efficient. It can be that in one document fonts are responsible for most runtime, but in another document table construction can be more demanding while yet another document puts some stress on interactive features. When  $\text{hz}$  or protrusion is enabled then you run substantially slower anyway so when you are willing to sacrifice 10% or more runtime don't complain about other components. The same is true for enabling  $\text{SYNCTEX}$ : if you are willing to add more than 10% runtime for that, don't whiner about the same amount for font handling.<sup>10</sup>

## How efficient is the styling that you use?

Probably the most easily overseen optimization is in switching fonts and color. Although in  $\text{CONTEX}$ T font switching is fast, I have no clue about it in other macro packages. But in a style you can decide to use inefficient (massive) font switches. The effects can easily be tested by commenting bit and pieces. For instance sometimes you need to do a full bodyfont switch when changing a style, like assigning  $\text{\small\bf}$  to the **style** key in  $\text{\setuphead}$ , but often using e.g.  $\text{\tf{d}}$  is much more efficient and works quite as well. Just try it.

## Are fonts really the bottleneck?

We already mentioned that one can look in the wrong direction. Maybe once someone is convinced that fonts are the culprit, it gets hard to look at the real issue. If a similar job in different macro packages has a significant different runtime one can wonder what happens indeed.

It is good to keep in mind that the amount of text is often not as large as you think. It's easy to do a test with hundreds of paragraphs of text but in practice we have whitespace, section titles, half empty pages, floats, itemize and similar constructs, etc. Often we don't mix many fonts in the running text either. So, in the end a real document is the best test.

## If you use LUA, is that code any good?

You can gain from the faster virtual machine of  $\text{LUAJIT}\text{\TeX}$ . Don't expect wonders from the jitting as that only pays off for long runs with the same code used over and over

---

<sup>10</sup> In  $\text{CONTEX}$ T we use a  $\text{SYNCTEX}$  alternative that is somewhat faster but it remains a fact that enabling more and more functionality will make the penalty of for instance font processing relatively small.

again. If the gain is high you can even wonder how well written your LUA code is anyway.

### What if they don't believe you?

So, say that someone finds LUATEX slow, what can be done about it? Just advice him or her to stick to tool used previously. Then, if arguments come that one also wants to use UTF-8, OPENTYPE fonts, a bit of METAPOST, and is looking forward to using LUA runtime, the only answer is: take it or leave it. You pay a price for progress, but if you do your job well, the price is not that large. Tell them to spend time on learning and maybe adapting and bark against their own tree before barking against those who took that step a decade ago. Most CONTEXT users took that step and someone still using LUATEX after a decade can't be that stupid. It's always best to first wonder what one actually asks from LUATEX, and if the benefit of having LUA on board has an advantage. If not, one can just use another engine.

Also think of this. When a job is slow, for me it's no problem to identify where the problem is. The question then is: can something be done about it? Well, I happily keep the answer for myself. After all, some people always need room to complain, maybe if only to hide their ignorance or incompetence. Who knows.

## 5.6 Comparing engines

The next comparison is to be taken with a grain of salt and concerns the state of affairs mid 2017. First of all, you cannot really compare MKII with MKIV: the later has more functionality (or a more advanced implementation of functionality). And as mentioned you can also not really compare PDFTEX and the wide engines. Anyway, here are some (useless) tests. First a bunch of loads. Keep in mind that different engines also deal differently with reading files. For instance MKIV uses LUATEX callbacks to normalize the input and has its own readers. There is a bit more overhead in starting up a LUATEX run and some functionality is enabled that is not present in MKII. The format is also larger, if only because we preload a lot of useful font, character and script related data.

```
\starttext
  \dorecurse {#1} {
    \input knuth
    \par
  }
\stoptext
```

When looking at the numbers one should realize that the times include startup and job management by the runner scripts. We also run in batchmode to avoid logging to influence runtime. The average is calculated from 5 runs.

engine	50	500	2500
pdftex	0.43	0.77	2.33

<b>xetex</b>	0.85	2.66	10.79
<b>luatex</b>	0.94	2.50	9.44
<b>luajittex</b>	0.68	1.69	6.34

The second example does a few switches in a paragraph:

```
\starttext
  \dorecurse {#1} {
    \tf \input knuth
    \bf \input knuth
    \it \input knuth
    \bs \input knuth
    \par
  }
\stoptext
```

<b>engine</b>	<b>50</b>	<b>500</b>	<b>2500</b>
<b>pdftex</b>	0.58	2.10	8.97
<b>xetex</b>	1.47	8.66	42.50
<b>luatex</b>	1.59	8.26	38.11
<b>luajittex</b>	1.12	5.57	25.48

The third example does a few more, resulting in multiple subranges per style:

```
\starttext
  \dorecurse {#1} {
    \tf \input knuth \it knuth
    \bf \input knuth \bs knuth
    \it \input knuth \tf knuth
    \bs \input knuth \bf knuth
    \par
  }
\stoptext
```

<b>engine</b>	<b>50</b>	<b>500</b>	<b>2500</b>
<b>pdftex</b>	0.59	2.20	9.52
<b>xetex</b>	1.49	8.88	43.85
<b>luatex</b>	1.64	8.91	41.26
<b>luajittex</b>	1.15	5.91	27.15

The last example adds some color. Enabling more functionality can have an impact on performance. In fact, as MKIV uses a lot of LUA and is also more advanced than MKII, one can expect a performance hit but in practice the opposite happens, which can also be due to some fundamental differences deep down at the macro level.

```
\setupcolors[state=start] % default in MkIV

\starttext
  \dorecurse {#1} {
    {\red \tf \input knuth \green \it knuth}
    {\red \bf \input knuth \green \bs knuth}
    {\red \it \input knuth \green \tf knuth}
    {\red \bs \input knuth \green \bf knuth}
    \par
  }
\stoptext
```

<b>engine</b>	<b>50</b>	<b>500</b>	<b>2500</b>
<b>pdftex</b>	0.61	2.36	10.33
<b>xetex</b>	1.53	9.25	45.59
<b>luatex</b>	1.65	8.91	41.32
<b>luajittex</b>	1.15	5.93	27.34

In these measurements the accuracy is a few decimals but a pattern is visible. As expected PDFTEX wins on simple documents but starts loosing when things get more complex. For these tests I used 64 bit binaries. A 32 bit XETEX with MKII performs the same as LUAJITTEX with MKIV, but a 64 bit XETEX is actually quite a bit slower. In that case the mingw cross compiled LUATEX version does pretty well. A 64 bit PDFTEX is also slower (it looks) than a 32 bit version. So in the end, there are more factors that play a role. Choosing between LUATEX and LUAJITTEX depends on how well the memory limited LUAJITTEX variant can handle your documents and fonts.

Because in most of our recent styles we use OPENTYPE fonts and (structural) features as well as recent METAFUN extensions only present in MKIV we cannot compare engines using such documents. The mentioned performance of LUATEX (or LUAJITTEX) and MkIV on the METAFUN manual illustrate that in most cases this combination is a clear winner.

```
\starttext
  \dorecurse {#1} {
    \null \page
  }
\stoptext
```

This gives:

<b>engine</b>	<b>50</b>	<b>500</b>	<b>2500</b>
<b>pdftex</b>	0.46	1.05	3.72
<b>xetex</b>	0.73	1.80	6.56
<b>luatex</b>	0.84	1.44	4.07
<b>luajittex</b>	0.61	1.10	3.33

That leaves the zero run:

```
\starttext
  \dorecurse {#1} {
    % nothing
  }
\stoptext
```

This gives the following numbers. In longer runs the difference in overhead is negligible.

engine	50	500	2500
<b>pdftex</b>	0.36	0.36	0.36
<b>xetex</b>	0.57	0.57	0.59
<b>luatex</b>	0.74	0.74	0.74
<b>luajittex</b>	0.53	0.53	0.54

It will be clear that when we use different fonts the numbers will also be different. And if you use a lot of runtime METAPOST graphics (for instance for backgrounds), the MKIV runs end up at the top. And when we process XML it will be clear that going back to MKII is no longer a realistic option. It must be noted that I occasionally manage to improve performance but we've now reached a state where there is not that much to gain. Some functionality is hard to compare. For instance in CONTEXT we don't use much of the PDF backend features because we implement them all in LUA. In fact, even in MKII already a done in T<sub>E</sub>X, so in the end the speed difference there is not large and often in favour of MKIV.

For the record I mention that shipping out the about 1250 pages has some overhead too: about 2 seconds. Here LUAJIT<sub>T</sub><sub>E</sub>X is 20% more efficient which is an indication of quite some LUA involvement. Loading the input files has an overhead of about half a second. Starting up LUATEX takes more time than PDFTEX and XETEX, but that disadvantage disappears with more pages. So, in the end there are quite some factors that blur the measurements. In practice what matters is convenience: does the runtime feel reasonable and in most cases it does.

If I would replace my laptop with a reasonable comparable alternative that one would be some 35% faster (single threads on processors don't gain much per year). I guess that this is about the same increase in performance that CONTEXT MKIV got in that period. I don't expect such a gain in the coming years so at some point we're stuck with what we have.

## 5.7 Summary

So, how "slow" is LUATEX really compared to the other engines? If we go back in time to when the first wide engines showed up, OMEGA was considered to be slow, although I

never tested that myself. Then, when  $\text{\LaTeX}$  showed up, there was not much talk about speed, just about the fact that we could use OPENTYPE fonts and native UTF input. If you look at the numbers, for sure you can say that it was much slower than  $\text{\PDF\TeX}$ . So how come that some people complain about  $\text{\LUA\TeX}$  being so slow, especially when we take into account that it's not that much slower than  $\text{\LaTeX}$ , and that  $\text{\LUAJIT\TeX}$  is often faster than  $\text{\LaTeX}$ . Also, computers have become faster. With the wide engines you get more functionality and that comes at a price. This was accepted for  $\text{\LaTeX}$  and is also acceptable for  $\text{\LUA\TeX}$ . But the price is not that high if you take into account that hardware performs better: you just need to compare  $\text{\LUA\TeX}$  (and  $\text{\LaTeX}$ ) runtime with  $\text{\PDF\TeX}$  runtime 15 years ago.

As a comparison, look at games and video. Resolution became much higher as did color depth. Higher frame rates were in demand. Therefore the hardware had to become faster and it did, and as a result the user experience kept up. No user will say that a modern game is slower than an old one, because the old one does 500 frames per second compared to some 50 for the new game on the modern hardware. In a similar fashion, the demands for typesetting became higher: UNICODE, OPENTYPE, graphics, XML, advanced PDF, more complex (niche) typesetting, etc. This happened more or less in parallel with computers becoming more powerful. So, as with games, the user experience didn't degrade with demands. Comparing  $\text{\LUA\TeX}$  with  $\text{\PDF\TeX}$  is like comparing a low res, low frame rate, low color game with a modern one. You need to have up to date hardware and even then, the writer of such programs need to make sure it runs efficient, simply because hardware no longer scales like it did decades ago. You need to look at the larger picture.

# 6 Editing

## 6.1 Introduction

Some users like the synctex feature that is built in the  $\text{\TeX}$  engines. Personally I never use it because it doesn't work well with the kind of documents I maintain. If you have one document source, and don't shuffle around (reuse) text too much it probably works out okay but that is not our practice. Here I will describe how you can enable a more  $\text{CONTEXT}$  specific synctex support so that aware PDF viewers can bring you back to the source.

## 6.2 The premise

Most of the time we provide our customers with an authoring workflow consisting of:

- the typesetting engine  $\text{CONTEXT}$
- the styles to generate the desired PDF files
- the text editor  $\text{SCITE}$
- the  $\text{SUMATRAPDF}$  viewer

For the MATHML we advice the  $\text{MATHTYPE}$  editor and we provide them with a customized MATHML translator for the copy & paste actions. When  $\text{ASCIIMATH}$  is used to code math no special tools are needed.

What people operate this workflow? Sometimes it's an author, but most of the time they are editors with a background in copy-editing. We call them XML editors, because they are maintaining the large (sets of) XML documents and edit directly in the XML sources.

Maybe you'll ask yourself "Can they do that? Can they edit directly in the XML resource?" The answer is yes, because after they have hit the processing key they are rewarded with a publishable PDF document in a demanding layout.

The XML sources have a dual purpose. They form the basis for:

- all folio products that are generated in XML to PDF workflow(s)
- the digital web product(s)

The XML editors do their proofing chapter-wise. Sometimes a chapter is one big XML file (10.000 lines is no exception when the chapter contains hundreds of bloated MATHML snippets). In other projects they have to deal with chapters that are made up of hundreds (100 upto 500) of smaller XML files.

## 6.3 The problem

Let's keep it simple: there's a typo. Here's what an XML editor will do:

- start SciTE
- open a file
- correct the typo
- generate the PDF
- proof the PDF and see if his alteration has some undesired side effects like text flow of image floating

So far so good. When the editor dealing with one big XML file there's no problem. Hopefully the filename will indicate the specific chapter. He or she opens the file and searches for the typo. And then correction happens. But what if there are hundreds of small XML files. How does the editor know in which file the typo can be found?

First, let's give a few statistics based on two projects that are in a revision stage.

project	chapters	# of files	average # of lines
A	16	16	11000
B	132	16000 <sup>11</sup>	100

The XML resource passes three stages: a raw, a semi final and a final version. The raw XML version originates from a web authoring tool that is used by the author. Then the PDF is proofread and the XML editor goes to work.

workflow	# edit locations and adaptations	# runs <sup>12</sup>
raw to semifinal	75	105
semifinal to final	35	55

Keep in mind that altering text may cause text to flow and images to float in a way that an XML editor will have to finetune and needs multiple runs for one correction.

Just to give an idea of the work involved. A typical semi final needs some 50 runs where each run takes 20 seconds (assuming 3 runs to get all cross referencing right). The numbers of explicit pagebreaks is about 5, and (related to formulas) explicit linebreaks around 8. It takes some 2 hours to get everything right, which includes checking in detail, fixing some things and if needed moving content a bit around.

Now we broaden the earlier question into: how can we make the work of an XML editor as easy and efficient as possible?

## 6.4 Enhancing efficiency

Since it is easier to proof content for folio and web via PDF documents we generate proof PDF files in which the complete content is shown. The proof can be a massive

---

<sup>11</sup> 132 chapters consisting of  $\pm$ 120 files.

<sup>12</sup> Maybe you can now see why we put quite some effort in keeping CONTEXT working at a comfortable speed.

document. A normal 40 page chapter can explode to 140 pages visualizing all the content that is coded in the XML file(s).

The content in the proof is shown in an effective way and a functional order. Let's give a few examples of how we enhance the XML editors effectiveness:

- By default the proof PDF file is interactive which serves testing the tocs and the register.
- The web hyperlinks are active so their destination can be tested.
- The questions and their answers are displayed in eachothers proximity. This sounds logical but in folio they are two separate products (theory and answer books).
- Medium specific content (web or folio) is typographically highlighted. For example by colored backgrounds.
- When spelling mode is on the XML editor can easily pick out the colored misspelled words.
- Images can be active areas although this is of no interest to XML editors. Clicking the image results in opening the image file in its corresponding application for maintenance.
- For practical reasons the filenames and paths of the XML files are displayed. The filenames are active links and clicking them results in opening the destination XML file in Scite.

Okay. The last option is a nice feature. However, the destination file is opened at the top of the file and you still have to find the typo or whatever incorrect issue you are looking for.

So a further enhancement in efficiency would be to jump to the typo's corresponding line in the XML source. This is where SYNCTEX comes into view. This feature, present in the TeX engines, provides a way to go from PDF to source by using a secondary file with positions. Unfortunately that mechanism is hardly useable for CONTEXT because it assumes a page and file handling model different from what we use. However, as CONTEXT uses LUALaTeX, it can also provide its own alternative.

## 6.5 What we want

The SYNCTEX method roughly works as follows. Internally TeX constricts linked lists of glyphs, kerns, glue, boxes, rules etc. These elements are called nodes. Some nodes carry information about the file and line where they were created. In the backend this information gets somehow translated in a (sort of) verbose tree that describes the makeup in terms of boxes, glue and kerns. From that information the SYNCTEX parser library, hooked into a PDF viewer, can go back from a position on the screen to a line in a file. One would expect this to be a relatively simple rectangle based model, but as far as I can see it's way more complex than that. There are some comments that CONTEXT is not supported well because it has a layered page model, which indicates that there are some assumptions about how macro packages are supposed to work. Also the used heuristics not only involve some specific spot (location) but also involve the corners and

edges. It is therefore not so much a (simple) generic system but a mechanism geared for a macro package like L<sup>A</sup>T<sub>E</sub>X.

Because we have a couple of users who need to edit complex sets of documents, coded in T<sub>E</sub>X or XML, I decided to come up with a variant that doesn't use the SYNCTEX machinery but manipulates the few SYNCTEX fields directly<sup>13</sup> and eventually outputs a straightforward file for the editor. Of course we need to follow some rules so that the editor can deal with it. It took a bit of trial and error to get the right information in the support file needed by the viewer but we got there.

The prerequisites of a decent CONTEX "click on preview and goto editor" are the following:

- It only makes sense to click on text in the text flow. Headers and footers are often generated from structure, and special typographic elements can originate in macros hooked into commands instead of in the source.
- Users should not be able to reach environments (styles) and other files loaded from the (normally read-only) T<sub>E</sub>X tree, like modules. We don't want accidental changes in such files.
- We not only have T<sub>E</sub>X files but also XML files and these can normally flush in rather arbitrary ways. Although the concept of lines is sort of lost in such a file, there is still a relation between lines and the snippets that make out the content of an XML node.
- In the case of XML files the overhead related to preserving line numbers should be minimal and have no impact on loading and memory when these features are not used.
- The overhead in terms of an auxiliary file size and complexity as well as producing that file should be minimal. It should be easy to turn on and off these features. (I'd never turn them on by default.)

It is unavoidable that we get more run time but I assume that for the average user that is no big deal. It pays off when you have a workflow when a book (or even a chapter in a book) is generated from hundreds of small XML files. There is no overhead when SYNCTEX is not used.

In CONTEX we don't use the built-in SYNCTEX features, that is: we let filename and line numbers be set but often these are overloaded explicitly. The output file is not compressed and constructed by CONTEX. There is no benefit in compression and the files are probably smaller than default SYNCTEX anyway.

---

<sup>13</sup> This is something that in my opinion should have been possible right from the start but it's too late now to change the system and it would not be used beyond CONTEX anyway.

## 6.6 Commands

Although you can enable this mechanism with directives it makes sense to do it using the following command.

```
\setupsynctex[state=start]
```

The advantage of using an explicit command instead of some command line option is that in an editor it's easier to disable this trickery. Commenting that line will speed up processing when needed. This command can also be given in an environment (style). On the command line you can say

```
context --synctex somefile.tex
```

A third method is to put this at the top of your file:

```
% synctex=yes
```

Often an XML files is very structured and although probably the main body of text is flushed as a stream, specific elements can be flushed out of order. In educational documents flushing for instance answers to exercises can happen out of order. In that case we still need to make sure that we go to the right spot in the file. It will never be 100% perfect but it's better than nothing. The above command will also enable XML support.

If you don't want a file to be accessed, you can block it:

```
\blocksynctexfile[foo.tex]
```

Of course you need to configure the viewer to respond to the request for editing. In Sumatra combined with SciTE the magic command is:

```
c:\data\system\scite\wscite\scite.exe "%f" "-goto:%l"
```

Such a command is independent of the macro package so you can just consult the manual or help info that comes with a viewer, given that it supports this linking back to the source at all.

If you enable tracing (see next section) you can what has become clickable. Instead of words you can also work with ranges, which not only gives less runtime but also much smaller .synctex files. Use

```
\setupsynctex[state=start,method=min]
```

to get words clickable and

```
\setupsynctex[state=start,method=max]
```

if you want somewhat more efficient ranges. The overhead for `min` is about 10 percent while `max` slows down around 5 percent.

## 6.7 Tracing

In case you want to see what gets synced you can enable a tracker:

```
\enabletrackers[system.synctex.visualize]  
\enabletrackers[system.synctex.visualize=real]
```

The following tracker outputs some status information about XML flushing. Such trackers only make sense for developers.

```
\enabletrackers[system.synctex.xml]
```

## 6.8 Warning

Don't turn on this feature when you don't need it. This is one of those mechanism that hits performance badly.

Depending on needs the functionality can be improved and/or extended. Of course you can always use the traditional SYNCTEX method but don't expect it to behave as described here.

## 7 Tricky fences

Occasionally one of my colleagues notices some suboptimal rendering and asks me to have a look at it. Now, one can argue about “what is right” and indeed there is not always a best answer to it. Such questions can even be a nuisance; let’s think of the following scenario. You have a project where  $\text{\TeX}$  is practically the only solution. Let it be an XML rendering project, which means that there are some boundary conditions. Speaking in 2017 we find that in most cases a project starts out with the assumption that everything is possible.

Often such a project starts with a folio in mind and therefore by decent tagging to match the educational and esthetic design. When rendering is mostly automatic and concerns too many (variants) to check all rendering, some safeguards are used (an example will be given below). Then different authors, editors and designers come into play and their expectations, also about what is best, often conflict. Add to that rendering for the web, and devices and additional limitations show up: features get dropped and even more cases need to be compensated (the quality rules for paper are often much higher). But, all that defeats the earlier attempts to do well because suddenly it has to match the lesser format. This in turn makes investing in improving rendering very inefficient (read: a bottomless pit because it never gets paid and there is no way to gain back the investment). Quite often it is spacing that triggers discussions and questions what rendering is best. And inconsistency dominates these questions.

So, in case you wonder why I bother with subtle aspects of rendering as discussed below, the answer is that it is not so much professional demand but users (like my colleagues or those on the mailing lists) that make me look into it and often something that looks trivial takes days to sort out (even for someone who knows his way around the macro language, fonts and the inner working of the engine). And one can be sure that more cases will pop up.

All this being said, let’s move on to a recent example. In  $\text{CONTEX}$  we support MATHML although in practice we’re forced to a mix of that standard and ASCIIMATH. When we’re lucky, we even get a mix with good old  $\text{\TeX}$ -encoded math. One problem with an automated flow and processing (other than raw  $\text{\TeX}$ ) is that one can get anything and therefore we need to play safe. This means for instance that you can get input like this:

$f(x) + f(1/x)$

or in more structured  $\text{\TeX}$  speak:

$\$f(x) + f(\frac{1}{x})\$$

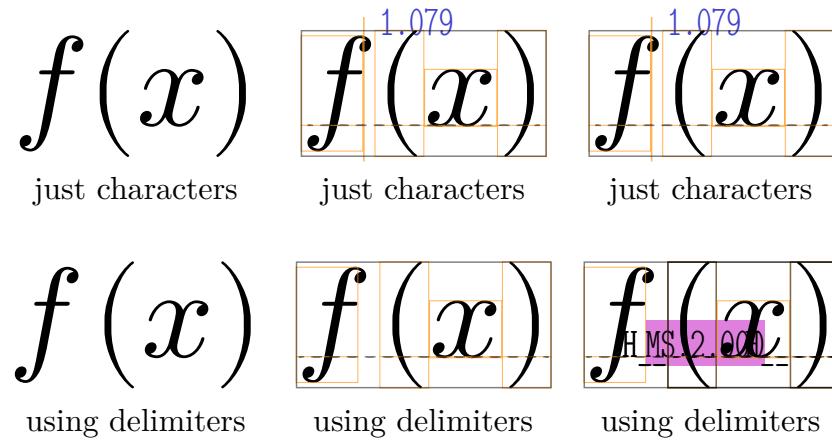
Using  $\text{\TeX}$  Gyre Pagella, this renders as:  $f(x) + f(\frac{1}{x})$ , and when seeing this a  $\text{\TeX}$  user will revert to:

```
$f(x) + f\left(\frac{1}{x}\right)$
```

which gives:  $f(x) + f\left(\frac{1}{x}\right)$ . So, in order to be robust we can always use the `\left` and `\right` commands, can't we?

```
$f(x) + f\left(x\right)$
```

which gives  $f(x) + f(x)$ , but let's blow up this result a bit showing some additional tracing from left to right, now in Latin Modern:



When we visualize the glyphs and kerns we see that there's a space instead of a kern when we use delimiters. This is because the delimited sequence is processed as a subformula and injected as a so-called inner object and as such gets spaced according to the ordinal (for the *f*) and inner ("fenced" with delimiters *x*) spacing rules. Such a difference normally will go unnoticed but as we mentioned authors, editors and designers being involved, there's a good chance that at some point one will magnify a PDF preview and suddenly notice that the difference between the *f* and *(* is a bit on the large side for simple unstacked cases, something that in print is likely to go unnoticed. So, even when we don't know how to solve this, we do need to have an answer ready.

When I was confronted by this example of rendering I started wondering if there was a way out. It makes no sense to hard code a negative space before a fenced subformula because sometimes you don't want that, especially not when there's nothing before it. So, after some messing around I decided to have a look at the engine instead. I wondered if we could just give the non-scaled fence case the same treatment as the character sequence.

Unfortunately here we run into the somewhat complex way the rendering takes place. Keep in mind that it is quite natural from the perspective of TeX because normally a user will explicitly use `\left` and `\right` as needed, while in our case the fact that we automate and therefore want a generic solution interferes (as usual in such cases).

Once read in the sequence *f(x)* can be represented as a list:

```
list = {  
{
```

```

id = "noad", subtype = "ord", nucleus = {
  {
    id = "mathchar", fam = 0, char = "U+00066",
  },
},
{
  id = "noad", subtype = "open", nucleus = {
    {
      id = "mathchar", fam = 0, char = "U+00028",
    },
  },
},
{
  id = "noad", subtype = "ord", nucleus = {
    {
      id = "mathchar", fam = 0, char = "U+00078",
    },
  },
},
{
  id = "noad", subtype = "close", nucleus = {
    {
      id = "mathchar", fam = 0, char = "U+00029",
    },
  },
},
}

```

The sequence `f \left( x \right)` is also a list but now it is a tree (we leave out some unset keys):

```

list = {
  {
    id = "noad", subtype = "ord", nucleus = {
      {
        id = "mathchar", fam = 0, char = "U+00066",
      },
    },
  },
  {
    id = "noad", subtype = "inner", nucleus = {
      {
        id = "submlist", head = {
          {
            id = "fence", subtype = "left", delim = {

```

```

{
  id = "delim", small_fam = 0, small_char = "U+00028",
},
},
},
{
  id = "noad", subtype = "ord", nucleus = {
  {
    id = "mathchar", fam = 0, char = "U+00078",
  },
  },
},
{
  id = "fence", subtype = "right", delim = {
  {
    id = "delim", small_fam = 0, small_char = "U+00029",
  },
  },
  },
  },
  },
  },
  },
  },
}

```

So, the formula `f(x)` is just four characters and stays that way, but with some inter-character spacing applied according to the rules of TeX math. The sequence `f \left( x \right)` however becomes two components: the `f` is an ordinal noad,<sup>14</sup> and `\left( x \right)` becomes an inner noad with a list as a nucleus, which gets processed independently. The way the code is written this is what (roughly) happens:

- A formula starts; normally this is triggered by one or two dollar signs.
- The `f` becomes an ordinal noad and TeX goes on.
- A fence is seen with a left delimiter and an inner noad is injected.
- That noad has a sub-math list that takes the left delimiter up to a matching right one.
- When all is scanned a routine is called that turns a list of math noads into a list of nodes.
- So, we start at the beginning, the ordinal `f`.

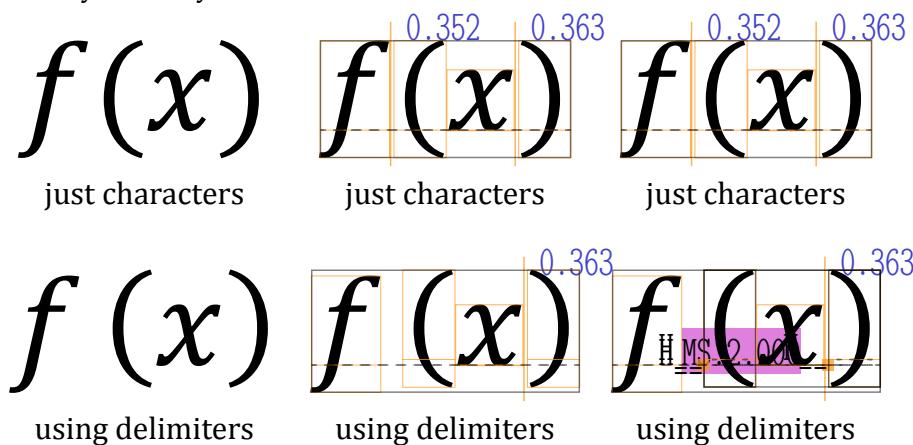
---

<sup>14</sup> Noads are the mathematical building blocks. Eventually they become nodes, the building blocks of paragraphs and boxed material.

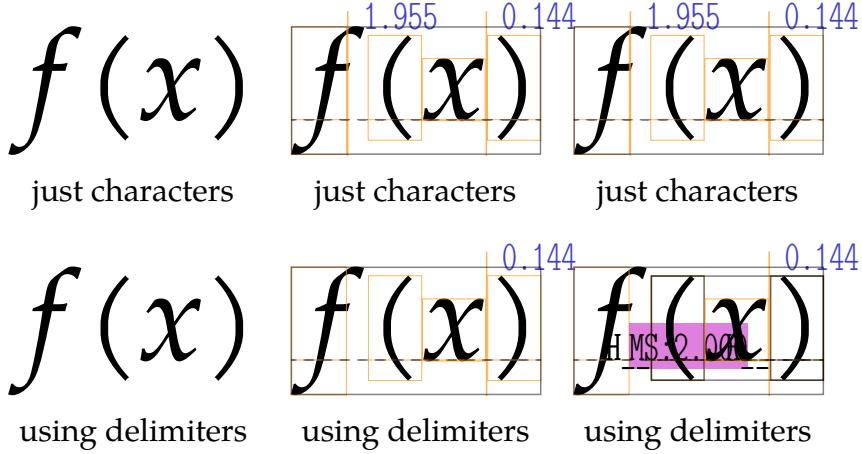
- Before moving on a check happens if this character needs to be kerned with another (but here we have an ordinal-inner combination).
- Then we encounter the subformula (including fences) which triggers a nested call to the math typesetter.
- The result eventually gets packaged into a hlist and we're back one level up (here after the ordinal  $f$ ).
- Processing a list happens in two passes and, to cut it short, it's the second pass that deals with choosing fences and spacing.
- Each time when a (sub)list is processed a second pass over that list happens.
- So, now TeX will inject the right spaces between pairs of noads.
- In our case that is between an ordinal and an inner noad, which is quite different from a sequence of ordinals.

It's these fences that demand a two-pass approach because we need to know the height and depth of the subformula. Anyway, do you see the complication? In our inner formula the fences are not scaled, but this is not communicated back in the sense that the inner noad can become an ordinal one, as in the simple  $f()$  pair. The information is not only lost, it is not even considered useful and the only way to somehow bubble it up in the processing so that it can be used in the spacing requires an extension. And even then we have a problem: the kerning that we see between  $f()$  is also lost. It must be noted that this kerning is optional and triggered by setting `\mathitalicsmode=1`. One reason for this is that fonts approach italic correction differently, and cheat with the combination of natural width and italic correction.

Now, because such a workaround is definitely conflicting with the inner workings of TeX, our experimenting demands another variable be created: `\mathdelimitersmode`. It might be a prelude to more manipulations but for now we stick to this one case. How messy it really is can be demonstrated when we render our example with Cambria.



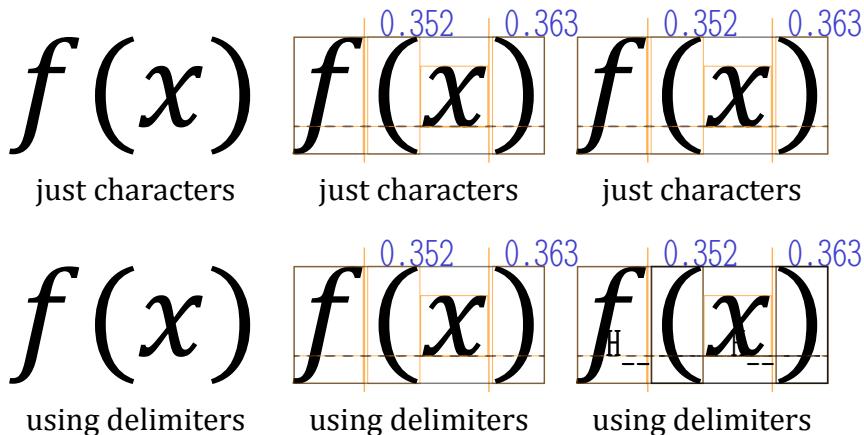
If you look closely you will notice that the parenthesis are moved up a bit. Also notice the more accurate bounding boxes. Just to be sure we also show Pagella:



When we really want the unscaled variant to be somewhat compatible with the fenced one we now need to take into account:

- the optional axis-and-height/depth related shift of the fence (bit 1)
- the optional kern between characters (bit 2)
- the optional space between math objects (bit 4)

Each option can be set (which is handy for testing) but here we will set them all, so, when `\mathdelimitersmode=7`, we want cambria to come out as follows:



When this mode is set the following happens:

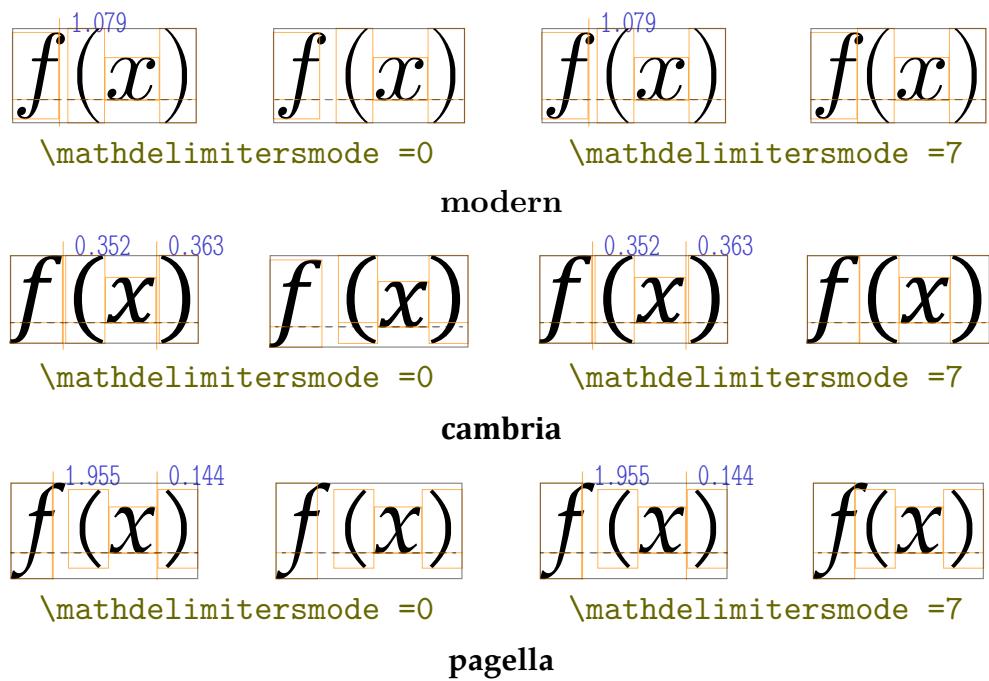
- We keep track of the scaling and when we use the normal size this is registered in the noad (we had space in the data structure for that).
- This information is picked up by the caller of the routine that does the subformula and stored in the (parent) inner noad (again, we had space for that).
- Kerns between a character (ordinal) and subformula (inner) are kept, which can be bad for other cases but probably less than what we try to solve here.
- When the fences are unscaled the inner property temporarily becomes an ordinal one when we apply the inter-noad spacing.

Hopefully this is good enough but anything more fancy would demand drastic changes in one of the most sensitive mechanisms of TeX. It might not always work out right, so for now I consider it an experiment, which means that it can be kept around, rejected or improved.

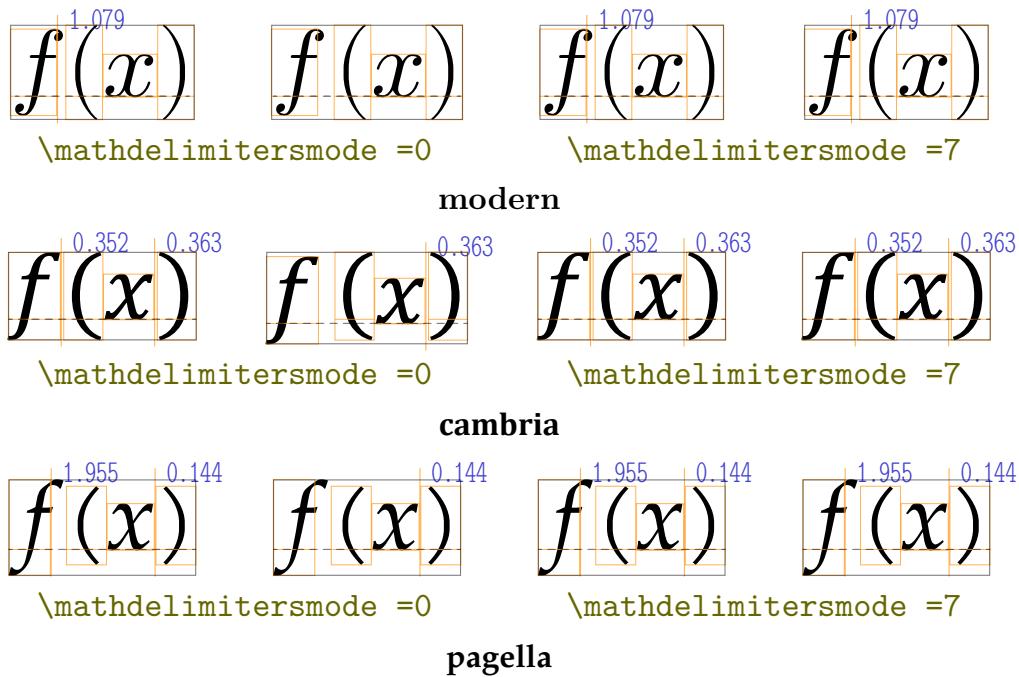
In case one wonders if such an extension is truly needed, one should also take into account that automated typesetting (also of math) is probably one of the areas where TeX can shine for a while. And while we can deal with much by using LUA, this is one of the cases where the interwoven and integrated parsing, converting and rendering of the math machinery makes it hard. It also fits into a further opening up of the inner working by modes.

Another objection to such a solution can be that we should not alter the engine too much. However, fences already are an exception and treated specially (tests and jumps in the program) so adding this fits reasonably well into that part of the design.

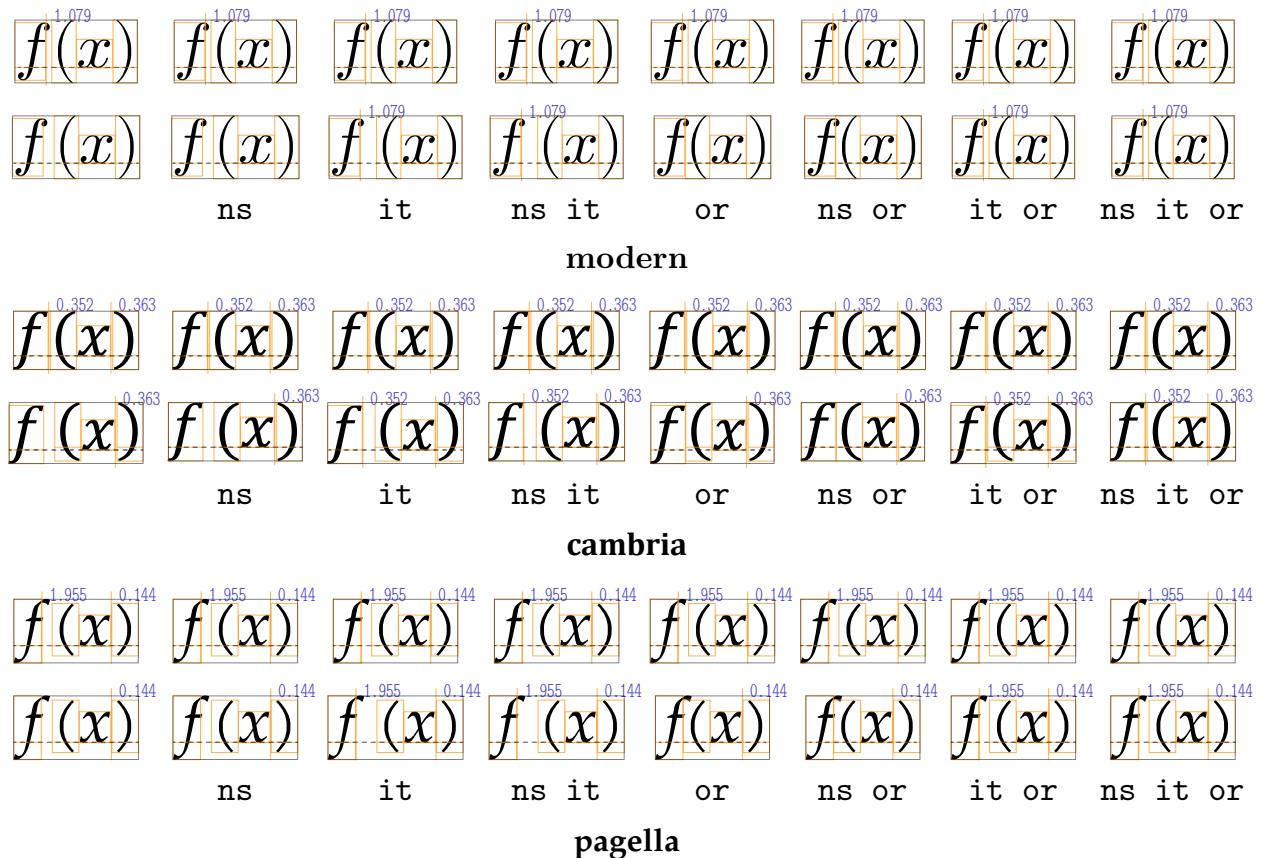
In the following examples we demonstrate the results for Latin Modern, Cambria and Pagella when `\mathdelimitersmode` is set to zero or one. First we show the case where `\mathitalicsmode` is disabled:



When we enable `\mathitalicsmode` we get:



So is this all worth the effort? I don't know, but at least I got the picture and hopefully now you have too. It might also lead to some more modes in future versions of L<sup>A</sup>T<sub>E</sub>X.



In C<sup>O</sup>N<sub>T</sub>E<sub>X</sub>T, a regular document can specify `\setupmathfences [method=auto]`, but in MATHML or ASCIIMATH this feature is enabled by default (so that we can test it).

We end with a summary of all the modes (assuming italics mode is enabled) in the table below.



# 8 The state of PDF

## 8.1 Introduction

Below I will spend some words on the state of PDF in CONTeXt mid 2018. These are just some reflections, not an in-depth discussion of the state of affairs. I sometimes feel the need to wrap up.

## 8.2 Media

For over two decades CONTeXt has supported fancy PDF features like movies and sound. In fact, as happens more, the flexibility of TeX made it possible to support such features right after they became available, often even before other applications supported them.

The first approach to support such media clips was relatively easy. In PDF one has the text flow, resulting from the typesetting process, either or not enhanced with images that are referred to from the flow. In that respect images are an integral part of PDF. On a separate layer there can be annotations. There are many kinds and they are originally a sort of extension mechanism that permits plugins to add features to a document. Examples of this are hyperlinks and the already mentioned media clips. Video was supported by the quicktime movie plugin. As far as I know in the meantime that plugin has been dropped as official part of Acrobat but one can still plug it in.

Later an extra mechanism was introduced, tagged renditions. It separates the views from the media and was more complex. When I first played with it, quite some media were possible, and I made a demo that could handle mov, mp3, smi and swf files. But last time I checked none of these really worked, apart from the swf file. One gets pop-ups for missing viewers and a look at the reader preferences makes one pessimistic about future support anyway. But one should be able to set up a list of useable players with this mechanism (although only an Adobe one seems to be okay so we're back to where we started).

At some point support for u3d was added. Interesting is that there is quite some infrastructure described in the PDF standard. Also something called rich media was introduced and that should replace the former video and audio annotations (definitely in PDF version 2) and probably some day the renditions will no longer be supported either. Open source PDF viewers just stuck to supporting text and static images.

Now, do these rich media work well? Hardly. The standard leaves it to the viewer and provides ways to define viewers (although it's unclear to me how that works out in practice.) Basically in PDF version 2 there is no native support for simple straightforward video. One has to construct a complex set of related annotations.

One can give arguments (like security risks) for not supporting all these fancy features but then why make rich media part of the specification at all? Browsers beat PDF viewers

in showing media and as browsers can operate in kiosk mode I suppose that it's not that hard to delegate showing whatever you want in an embedded window in the PDF viewer. Or why not simply support videolan out of the box. All we need is the ability to view movies and control them (play, pause, stop, rewind, etc). Where HTML evolved towards easier media support, PDF evolved to more obscurity.

So, how bad is it really? There are PDF files around that have video! Indeed, but the way they're supposed to do this is as follows: currently one actually has to embed a shockwave video player (a user interface around something built-in) and let that player show for instance an mp4 movie. However, support for shockwave (flash) will be dropped in 2020 and that renders documents that use it obsolete. This even makes one wonder about JAVASCRIPT and widgets like form fields, also a rather moving and somewhat unstable target. (I must have a document being a calculator somewhere made in the previous century, in the early days of PDF.)

I think that the plugin model failed already rather early in the PDF history if only because it made no sense to develop them when in a next version of Acrobat the functionality was copied in the core. In a similar fashion JAVASCRIPT support seems to have stalled.

Unfortunately the open source viewers never caughted on with media, forms and JAVASCRIPT and therefore there has been no momentum created to keep things supported. It all makes efforts spent on supporting this kind of PDF features a waste of time. It also makes one careful in using them: it only works on the short term.

Get me right, I'm not talking of complex media like 3d or animations but of straightforward video support. I understand that the rich media framework tries to cover complex cases but it's simple cases that carry the format. On the other hand, one can wonder why the PDF format makes it possible to specify behaviour that in practice depends on JAVASCRIPT and therefore could as well have been delegated to JAVASCRIPT as well. It would probably have been much cleaner.<sup>15</sup>

The PDF version 2 specification mentions **3D**, **Video** and **Audio** as primary content types so maybe future viewers will support video out of the box. Who knows. We try to keep up in CONTEXT because it's often not that complex to support PDF features but with hardly any possibility to test them, they have a low priority. And with Acrobat moving to the cloud and thereby creating a more or less lifelong dependency on remote resources it doesn't become much interesting to explore those routes either.

## 8.3 Accessibility

A popular PDF related topic is accessibility. One aspect of that is tagged PDF. This substandard is in my opinion not something that deserves a price for beauty. I know that there are CONTEXT users who need to be compliant but I always wonder what a

---

<sup>15</sup> It looks like muPDF in 2018 got some support related to widgets aka fields but alas not for layers which would be quite useful.

publisher really does with such a file. It's a bit like requiring XML as source but at the same time sacrificing really rich encoded sources for tweaks that suite the current limitations of for instance browsers, tool-chains and competence. We've seen it happen.

Support for tagged PDF has been available in CONTeXt already for a while but as far as I know only Acrobat professional can do something with it. The reason for tagging is that a document is then useable for (for instance) visually impaired users, but aren't they better served with a proper complete and very structured source in some format that tools suitable for it can use? How many publishers distribute PDF files while they can still make money on prints? How many are really interested in distributing enriched content that then can be reused somehow? And how many are willing to invest in tools instead of waiting for it to happen for free? It's a bit cheap trick to just expect authors (and their in the case of TeX free tools) to suit a publishers needs. Anyway, just as with advanced interactive documents or forms, I wonder if it will catch on. At least no publisher ever asked us and by the time they might do the competition of web based dissemination could have driven PDF to the background. But, in CONTeXt we will keep supporting such features anyway, if only because it's quite doable. But . . . it's user demand that drives development, not the market, which means that the motivation for implementing such features depends on user input as well as challenging aspects that make it somewhat fun to spend time on them.

## 8.4 Quality assurance

Another aspect popping up occasionally is validation. I'm not entirely sure what drives that but delegating a problem can be one reason. Often we see publishers and printers use old versions of PDF related tools. Also, some workflows are kind of ancient anyway and are more driven by POSTSCRIPT history than PDF possibilities. I sometimes get the impression that it takes at least a decade for these things to catch on, and by that time it doesn't matter any more that TeX and friends were at the front: their users are harassed by what the market demands by then.

Support for several standards related to validation is already part of CONTeXt for quite a while. For instance the bump from PDF 1.7 to 2.0 was hardly worth noticing, simply because there are not that many fundamental changes. Adapting LUATEX was trivial (and actually not really needed), and macro packages can provide what is needed without much problems. So, yes, we can support it without much hassle. Personally I never ran into a case where validation was really needed. The danger of validation is that it can give a false impression of quality. And as with everything quality control created a market. As with other features it is users who drive the availability of support for this. After all, they are the ones testing it and figuring out the often fuzzy specifications. These are things that one can always look at in retrospect (like: it has to be done this or that way) while in practice in order to be an early adopter one has to gamble a bit and see where it fails or succeeds. Fortunately it's relatively easy to adapt macro packages and CONTeXt users are willing to update so it's not really an issue.

Putting a stamp of approval on a PDF cannot hide the inconsistencies between for instance vector graphics produced by a third party. They also don't expose inconsistent use of color and fonts. The page streams produced by LUATEX are simple and clean enough to not give problems with validation. The problem lays more with resources coming from elsewhere. When you're phoned by a printing house about an issue with RGB images in a file where there is no sign of RGB being used but where a validator reports an issue, you're lucky when an experienced printer dating back decades then replies that he already had that impression and will contact the origin. There is no easy way out of this but educating users (authors) is an option. However, they are often dependent on the publishers and departments that deal with these and those tend to come with directives that the authors cannot really argue with (or about).

## 8.5 Interactivity

This is an area where T<sub>E</sub>X (and therefore also C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T) always had an edge. There is a lot possible and in principle all that PDF provides can be supported. But the more fancy one goes, the more one depends on Acrobat. Interactivity in PDF evolved stepwise and is mostly market driven. As a result it is (or was) not always consistent. This is partly due to the fact that we have a chicken-egg issue: you need typesetting machinery, viewer as well as a standard.

The regular hyperlinks, page or named driven are normally supported by viewers. Some redefined named destinations (like going to a next page, or going back in a chain of followed links) not always. Launching applications, as it also relates to security, can be qualified as an unreliable mechanism. More advanced linking, for instance using JAVASCRIPT is hardly supported. In that respect PDF viewers lag way behind HTML browsers. I understand that there can be security risks involved. It's interesting to see that in Acrobat one can mess with internals of files which makes the API large and complex, but if we stick to the useful core, the amount of interfacing needed is quite small. Lack of support in open source viewers (we're talking of about two decades now) made me loose interest in these features but they are and will be supported in C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T. We'll see if and when viewers catch up.

Comments and attachments are also part of interactivity and of course we supported them right from the start. Some free viewers also support them by now. Personally I never use comments but they can be handy for popping up information or embedding snippets or (structured) sources (like MATHML or bibliographic data). In C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T we can even support PDF inclusion with (a reasonable) subset of these so called annotations. As the PDF standard no longer evolves much we can expect all these features to become stable.

## 8.6 Summary

We have always supported the fancy PDF features and we will continue doing so in C<sub>O</sub>N<sub>T</sub>E<sub>X</sub>T. However, many of them depends on what viewers support, and after decades

of PDF that is still kind of disappointing, which is not that motivating. We'll see what happens.



## 9 From LUA 5.2 to 5.3

When we started with LUATEX we used LUA 5.1 and moved to 5.2 when that became available. We didn't run into issues then because there were no fundamental changes that could not be dealt with. However, when LUA 5.3 was announced in 2015 we were not sure if we should make the move. The main reason was that we'd chosen LUA because of its clean design which meant that we had only one number type: double. In 5.3 on the other hand, deep down a number can be either an integer or a floating point quantity.

Internally  $\text{\TeX}$  is mostly (up to) 32-bit integers and when we go from LUA to  $\text{\TeX}$  we round numbers. Nonetheless one can expect some benefits in using integers. Performance-wise we didn't expect much, and memory consumption would be the same too. So, the main question then was: can we get the same output and not run into trouble due to possible differences in serializing numbers; after all  $\text{\TeX}$  is about stability. The serialization aspect is for instance important when we compare quantities and/or use numbers in hashes.

Apart from this change in number model, which comes with a few extra helpers, another extension in 5.3 was that bit-wise operations are now part of the language. The lpeg library is still not part of stock LUA. There is some minimal UTF8 support, but less than we provide in LUATEX already. So, looking at these changes, we were not in a hurry to update. Also, it made sense to wait till this important number-related change was stable.

But, a few years later, we still had it on our agenda to test, and after the CONTEXT 2017 meeting we decided to give it a try; here are some observations. A quick test was just dropping in the new LUA code and seeing if we could make a CONTEXT format. Indeed that was no big deal but a test run failed because at some point a (for instance) 1 became a 1.0. It turned out that serializing has some side effects. And with some ad hoc prints for tracing (in the LUATEX source) I could figure out what went on. How numbers are seen can (to some extent) be deduced from the `string.format` function, which is in LUA a combination of parsing, splitting and concatenation combined with piping to the C `sprintf` function.<sup>16</sup>

```
local a = 2 * (1/2) print(string.format("%s", a),math.type(x))
local b = 2 * (1/2) print(string.format("%d", b),math.type(x))
local c = 2         print(string.format("%d", c),math.type(x))
local d = -2        print(string.format("%d", d),math.type(x))
local e = 2 * (1/2) print(string.format("%i", e),math.type(x))
```

---

<sup>16</sup> Actually, at some point I decided to write my own formatter on top of `format` and I ended up with splitting as well. It's only now that I realize why this is working out so well (in terms of performance): simple format (single items) are passed more or less directly to `sprintf` and as LUA itself is fast, due to some caching, the overhead is small compared to the built-in splitter method. And the CONTEXT formatter has many more options and is extensible.

```

local f = 2.1          print(string.format("%.0f",f),math.type(x))
local g = 2.0          print(string.format("%.0f",g),math.type(x))
local h = 2.1          print(string.format("%G", h),math.type(x))
local i = 2.0          print(string.format("%G", i),math.type(x))
local j = 2             print(string.format("%.0f",j),math.type(x))
local k = -2            print(string.format("%.0f",k),math.type(x))

```

This gives the following results:

<b>a</b>	$2 * (1/2)$	s	1.0	float
<b>b</b>	$2 * (1/2)$	d	1	float
<b>c</b>	2	d	2	integer
<b>d</b>	-2	d	2	integer
<b>e</b>	$2 * (1/2)$	i	1	float
<b>f</b>	2.1	.0f	2	float
<b>g</b>	2.0	.0f	2	float
<b>h</b>	2.1	G	2.1	float
<b>i</b>	2.0	G	2	float
<b>j</b>	2	.0f	2	integer
<b>k</b>	-2	.0f	2	integer

This demonstrates that we have to be careful when we need these numbers represented as strings. In CONTEXT the number of places where we had to check for that was not that large; in fact, only some hashing related to font sizes had to be done using explicit rounding.

Another surprising side effect is the following. Instead of:

```
local n = 2^6
```

we now need to use:

```
local n = 0x40
```

or just:

```
local n = 64
```

because we don't want this to be serialized to  $64.0$  which is due to the fact that a power results in a float. One can wonder if this makes sense when we apply it to an integer.

At any rate, once we could process a file, two documents were chosen for a performance test. Some experiments with loops and casts had demonstrated that we could expect a small performance hit and indeed, this was the case. Processing the LUATEX manual takes 10.7 seconds with 5.2 on my 5-year-old laptop and 11.6 seconds with 5.3. If we consider that CONTEXT spends 50% of its time in LUA, then we see a 20% performance penalty. Processing the METAFUN manual (which has lots of METAPOST images) went from less than 20 seconds (LUAJITTEX does it in 16 seconds) up to more than 27 seconds.

So there we lose more than 50% on the LUA end. When we observed these kinds of differences, Luigi and I immediately got into debugging mode, partly out of curiosity, but also because consistent performance is important to us.

Because these numbers made no sense, we traced different sub-mechanisms and eventually it became clear that the reason for the speed penalty was that the core `string.format` function was behaving quite badly in the `mingw` cross-compiled binary, as seen by this test:

```
local t = os.clock()
for i=1,1000*1000 do
    -- local a = string.format("%.3f",1.23)
    -- local b = string.format("%i",123)
    local c = string.format("%s",123)
end
print(os.clock()-t)
```

	<b>lua 5.3</b>	<b>lua 5.2</b>	<b>texlua 5.3</b>	<b>texlua 5.2</b>
<b>a</b>	0.43	0.54	3.71 (0.47)	0.53
<b>b</b>	0.18	0.24	3.78 (0.17)	0.22
<b>c</b>	0.26	0.68	3.67 (0.29)	0.66

The 5.2 binaries perform the same but the 5.3 Lua binary greatly outperforms LUATEX, and so we had to figure out why. After all, all this integer optimization could bring some gain! It took us a while to figure this out. The numbers in parentheses are the results after fixing this.

Because font internals are specified in integers one would expect a gain in running:

```
mtxrun --script font --reload force
```

and indeed that is the case. On my machine a scan results in 2561 registered fonts from 4906 read files and with 5.2 that takes 9.1 seconds while 5.3 needs a bit less: 8.6 seconds (with the bad format performance) and even less once that was fixed. For a test:

```
\setupbodyfont[modern]      \tf \bf \it \bs
\setupbodyfont[pagella]     \tf \bf \it \bs
\setupbodyfont[dejavu]      \tf \bf \it \bs
\setupbodyfont[termes]      \tf \bf \it \bs
\setupbodyfont[cambria]     \tf \bf \it \bs
\starttext \stoptext
```

This code needs 30% more runtime so the question is: how often do we call `string.format` there? A first run (when we wipe the font cache) needs some 715,000 calls while successive runs need 115,000 calls so that slow down definitely comes from the bad handling of `string.format`. When we drop in a LUA update or whatever other dependency we don't want this kind of impact. In fact, when one uses external libraries that are or can be compiled under the TEX Live infrastructure and the impact would be such, it's

bad advertising, especially when one considers the occasional complaint about LUATEX being slower than other engines.

The good news is that eventually Luigi was able to nail down this issue and we got a binary that performed well. It looks like LUA 5.3.4 (cross)compiles badly with GCC 5.3.0 and 6.3.0.

So in the end caching the fonts takes:

	caching	running
<b>5.2 stock</b>	8.3	1.2
<b>5.3 bugged</b>	12.6	2.1
<b>5.3 fixed</b>	6.3	1.0

So indeed it looks like 5.3 is able to speed up LUATEX a bit, given that one integrates it in the right way! Using a recent compiler is needed too, although one can wonder when a bad case will show up again. One can also wonder why such a slow down can mostly go unnoticed, because for sure LUATEX is not the only compiled program.

The next examples are some edge cases that show you need to be aware that (1) an integer has its limits, (2) that hexadecimal numbers are integers and (3) that LUA and LUAJIT can be different in details.

	print(0xFFFFFFFFFFFFFFFFF)	print(0x7FFFFFFFFFFFFFFF)
<b>lua 52</b>	1.844674407371e+019	9.2233720368548e+018
<b>luajit</b>	1.844674407371e+19	9.2233720368548e+18
<b>lua 53</b>	-1	9223372036854775807

So, to summarize the process. A quick test was relatively easy: move 5.3 into the code base, adapt a little bit of internals (there were some LUATEX interfacing bits where explicit rounding was needed), run tests and eventually fix some issues related to the Makefile (compatibility) and C obscurities (the slow `sprintf`). Adapting CONTEXT was also not much work, and the test suite uncovered some nasty side effects. For instance, the valid 5.2 solution:

```
local s = string.format("02X",u/1024)
local s = string.char      (u/1024)
```

now has to become (both 5.2 and 5.3):

```
local s = string.format("02X",math.floor(u/1024))
local s = string.char      (math.floor(u/1024))
```

or (both 5.2 and (emulated or real) 5.3):

```
local s = string.format("02X",bit32.rshift(u,10))
local s = string.char      (bit32.rshift(u,10))
```

or (only 5.3):

```
local s = string.format("02X",u >> 10)
local s = string.char      (u >> 10)
```

or (only 5.3):

```
local s = string.format("02X",u//1024)
local s = string.char      (u//1024)
```

A conditional section like:

```
if LUAVERSION >= 5.3 then
    local s = string.format("02X",u >> 10)
    local s = string.char      (u >> 10)
else
    local s = string.format("02X",bit32.rshift(u,10))
    local s = string.char      (bit32.rshift(u,10))
end
```

will fail because (of course) the 5.2 parser doesn't like that. In CONTEXt we have some experimental solutions for that but that is beyond this summary.

In the process a few UTF helpers were added to the string library so that we have a common set for LUAJIT and LUA (the `utf8` library that was added to 5.3 is not that important for LUATEX). For now we keep the `bit32` library on board. Of course we'll not mention all the details here.

When we consider a gain in speed of 5-10% with 5.3 that also means that the gain of LUAJITTEX compared to 5.2 becomes less. For instance in font processing both engines now perform closer to the same.

As I write this, we've just entered 2018 and after a few months of testing LUATEX with LUA 5.3 we're confident that we can move the code to the experimental branch. This means that we will use this version in the CONTEXt distribution and likely will ship this version as 1.10 in 2019, where it becomes the default. The 2018 version of TeX Live will have 1.07 with LUA 5.2 while intermediate versions of the LUA 5.3 binary will end up on the CONTEXt garden, probably with number 1.08 and 1.09 (who knows what else we will add or change in the meantime).



# 10 Executing T<sub>E</sub>X

Much of the LUA code in CON<sub>T</sub>EXT originates from experiments. When it survives in the source code it is probably used, waiting to be used or kept for educational purposes. The functionality that we describe here has already been present for a while in CON<sub>T</sub>EXT, but improved a little starting with LUAT<sub>E</sub>X 1.08 due to an extra helper. The code shown here is generic and not used in CON<sub>T</sub>EXT as such.

Say that we have this code:

```
for i=1,10000 do
    tex.sprint("1")
    tex.sprint("2")
    for i=1,3 do
        tex.sprint("3")
        tex.sprint("4")
        tex.sprint("5")
    end
    tex.sprint("\space")
end
```

When we call `\directlua` with this snippet we get some 30 pages of 12345345345. The printed text is saved till the end of the LUA call, so basically we pipe some 170.000 characters to T<sub>E</sub>X that get interpreted as one paragraph.

Now imagine this:

```
\setbox0\hbox{xxxxxxxxxx} \number\wd0
```

which gives 4461336. If we check the box in LUA, with:

```
tex.sprint(tex.box[0].width)
tex.sprint("\enspace")
tex.sprint("\setbox0\hbox{!}")
tex.sprint(tex.box[0].width)
```

the result is 4461336 4461336, which is not what you would expect at first sight. However, if you consider that we just pipe to a T<sub>E</sub>X buffer that gets parsed after the LUA call, it will be clear that the reported width is the width that we started with. It will work all right if we say:

```
tex.sprint(tex.box[0].width)
tex.sprint("\enspace")
tex.sprint("\setbox0\hbox{!}")
tex.sprint("\directlua{tex.sprint(tex.box[0].width)}")
```

because now we get: 4461336 443625. It's not that complex to write some support code that makes this more convenient. This can work out quite well but there is a drawback. If we use this code:

```
print(status.input_ptr)
tex.sprint(tex.box[0].width)
tex.sprint("\\enspace")
tex.sprint("\\setbox0\\hbox{!}")
tex.sprint("\\directlua{print(status.input_ptr) \
tex.sprint(tex.box[0].width)}")
```

Here we get 6 and 7 reported. You can imagine that when a lot of nested `\directlua` calls happen, we can get an overflow of the input level or (depending on what we do) the input stack size. Ideally we want to do a LUA call, temporarily go to T<sub>E</sub>X, return to LUA, etc. without needing to worry about nesting and possible crashes due to LUA itself running into problems. One charming solution is to use so-called coroutines: independent LUA threads that one can switch between — you jump out from the current routine to another and from there back to the current one. However, when we use `\directlua` for that, we still have this nesting issue and what is worse, we keep nesting function calls too. This can be compared to:

```
\def\whatever{\ifdone\whatever\fi}
```

where at some point `\ifdone` is false so we quit. But we keep nesting when the condition is met, so eventually we can end up with some nesting related overflow. The following:

```
\def\whatever{\ifdone\expandafter\whatever\fi}
```

is less likely to overflow because there we have tail recursion which basically boils down to not nesting but continuing. Do we have something similar in LUATEX for LUA? Yes, we do. We can register a function, for instance:

```
lua.get_functions_table()[1] = function() print("Hi there!") end
```

and call that one with:

```
\luafunction 1
```

This is a bit faster than calling a function like:

```
\directlua{HiThere()}
```

which can also be achieved by

```
\directlua{print("Hi there!")}
```

which sometimes can be more convenient. Anyway, a function call is what we can use for our purpose as it doesn't involve interpretation and effectively behaves like a tail call. The following snippet shows what we have in mind:

```

tex.routine(function()
    tex.sprint(tex.box[0].width)
    tex.sprint("\\enspace")
    tex.sprint("\\setbox0\\hbox{!}")
    tex.yield()
    tex.sprint(tex.box[0].width)
end)

```

We start a routine, jump out to  $\text{\TeX}$  in the middle, come back when we're done and continue. This gives us: 4461336 218508, which is what we expect.

4461336 218508

This mechanism permits efficient (nested) loops like:

```

tex.routine(function()
    for i=1,10000 do
        tex.sprint("1")
        tex.yield()
        tex.sprint("2")
        tex.routine(function()
            for i=1,3 do
                tex.sprint("3")
                tex.yield()
                tex.sprint("4")
                tex.yield()
                tex.sprint("5")
            end
        end)
        tex.sprint("\\space")
        tex.yield()
    end
end)

```

We do create coroutines, go back and forwards between LUA and  $\text{\TeX}$ , but avoid memory being filled up with printed content. If we flush paragraphs (instead of e.g. the space) then the main difference is that instead of a small delay due to the loop unfolding in a large set of prints and accumulated content, we now get a steady flushing and processing.

However, we can still have an overflow of input buffers because we still nest them: the limitation at the  $\text{\TeX}$  end has moved to a limitation at the LUA end. How come? Here is the code that we use:

```

local stepper = nil
local stack   = { }
local fid     = 0xFFFFFFF

```

```

local goback = "\\\luafunction" .. fid .. "\\relax"

function tex.resume()
    if coroutine.status(stepper) == "dead" then
        stepper = table.remove(stack)
    end
    if stepper then
        coroutine.resume(stepper)
    end
end

lua.get_functions_table()[fid] = tex.resume

function tex.yield()
    tex.sprint(goback)
    coroutine.yield()
    texio.closeinput()
end

function tex.routine(f)
    table.insert(stack,stepper)
    stepper = coroutine.create(f)
    tex.sprint(goback)
end

```

The `routine` creates a coroutine, and `yield` gives control to T<sub>E</sub>X. The `resume` is done at the T<sub>E</sub>X end when we're finished there. In practice this works fine and when you permit enough nesting and levels in T<sub>E</sub>X then you will not easily overflow.

When I picked up this side project and wondered how to get around it, it suddenly struck me that if we could just quit the current input level then nesting would not be a problem. Adding a simple helper to the engine made that possible (of course figuring it out took a while):

```

local stepper = nil
local stack   = { }
local fid     = 0xFFFFF
local goback = "\\\luafunction" .. fid .. "\\relax"

function tex.resume()
    if coroutine.status(stepper) == "dead" then
        stepper = table.remove(stack)
    end
    if stepper then
        coroutine.resume(stepper)
    end

```

```

end

lua.get_functions_table()[fid] = tex.resume

if texio.closeinput then
    function tex.yield()
        tex.sprint(goback)
        coroutine.yield()
        texio.closeinput()
    end
else
    function tex.yield()
        tex.sprint(goback)
        coroutine.yield()
    end
end

function tex.routine(f)
    table.insert(stack,stepper)
    stepper = coroutine.create(f)
    tex.sprint(goback)
end

```

The trick is in `texio.closeinput`, a recent helper and one that should be used with care. We assume that the user knows what she or he is doing. On an old laptop with a i7-3840 processor running WINDOWS 10 the following snippet takes less than 0.35 seconds with LUATEX and 0.26 seconds with LUAJITTEX.

```

tex.routine(function()
    for i=1,10000 do
        tex.sprint("\setbox0\hpack{x}")
        tex.yield()
        tex.sprint(tex.box[0].width)
        tex.routine(function()
            for i=1,3 do
                tex.sprint("\setbox0\hpack{xx}")
                tex.yield()
                tex.sprint(tex.box[0].width)
            end
        end)
    end
end)

```

Say that we run the bad snippet:

```

for i=1,10000 do
    tex.sprint("\setbox0\hpack{x}")

```

```

tex.sprint(tex.box[0].width)
for i=1,3 do
    tex.sprint("\\setbox0\\hpack{xx}")
    tex.sprint(tex.box[0].width)
end
end

```

This time we need 0.12 seconds in both engines. So what if we run this:

```

\dorecurse{10000}{%
    \setbox0\hpack{x}
    \number\wd0
    \dorecurse{3}{%
        \setbox0\hpack{xx}
        \number\wd0
    }%
}

```

Pure TeX needs 0.30 seconds for both engines but there we lose 0.13 seconds on the loop code. In the LUA example where we yield, the loop code takes hardly any time. As we need only 0.05 seconds more it demonstrates that when we use the power of LUA the performance hit of the switch is quite small: we yield 40.000 times! In general, such differences are far exceeded by the overhead: the time needed to typeset the content (which `\hpack` doesn't do), breaking paragraphs into lines, constructing pages and other overhead involved in the run. In CONTEXt we use a slightly different variant which has 0.30 seconds more overhead, but that is probably true for all LUA usage in CONTEXt, but again, it disappears in other runtime.

Here is another example:

```

\def\TestWord#1%
{\directlua{
    tex.routine(function()
        tex.sprint("\\setbox0\\hbox{\\tttf #1}")
        tex.yield()
        tex.sprint(math.round(100 * tex.box[0].width/tex.hsize))
        tex.sprint(" percent of the hsize: ")
        tex.sprint("\\box0")
    end)
}}

```

The width of next word is `\TestWord {inline!}`!

The width of next word is 9 percent of the hsize: `inline!`

Now, in order to stay realistic, this macro can also be defined as:

```
\def\TestWord#1%
```

```

{\setbox0\hbox{\tttf #1}%
 \directlua{
   tex.sprint(math.round(100 * tex.box[0].width/tex.hsize))
 } %
 percent of the hsize: \box0\relax}

```

We get the same result: “The width of next word is 9 percent of the hsize: `inline!`”.

We have been using a LUA- $\text{\TeX}$  mix for over a decade now in CONTeXt, and have never really needed this mixed model. There are a few places where we could (have) benefitted from it and we might use it in a few places, but so far we have done fine without it. In fact, in most cases typesetting can be done fine at the  $\text{\TeX}$  end. It’s all a matter of imagination.

## 103 Executing T<sub>E</sub>X

# 11 Modern Latin

## 11.1 Introduction

In CONTeXt, already in MKII, we have a feature tagged ‘effects’ that can be used to render a font in outline or bolder versions. It uses some low level PDF directives to accomplish this and it works quite well. When a user on the CONTeXt list asked if we could also provide it as a font feature in the repertoire of additional features in CONTeXt, I was a bit reluctant to provide that because it operates at another level than the glyph stream. Also, such a feature can be abused and result in a bad looking document. However, by adding a few simple options to the LUATEX engine such a feature could actually be achieved rather easy: it was trivial to implement given that we can influence font handling at the LUA end. In retrospect extended and pseudo slanted fonts could be done this way too but there we have some historic ballast. Also, the backend now handles such transformations very efficient because they are combined with font scaling. Anyway, by adding this feature in spite of possible objections, I could do some more advanced experiments.

In the following pages I will demonstrate how we support effects as a feature in CONTeXt. Instead of simply applying some magic PDF text operators in the backend a more integrated approach is used. The difference with the normal effect mechanism is that where the one described here is bound to a font instance while the normal mechanism operates on the glyph stream.

## 11.2 The basics

Let’s start with a basic boldening example. First we demonstrate a regular Latin Modern sample (using `ward.tex`):

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

This font looks rather thin (light). Next we define an effect or `0.2` and typeset the same sample:

```
\definefontfeature
  [effect-1]
  [effect=.2]
```

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

This simple call gives reasonable default results. But you can have more control than this. The previous examples use the following properties:

```
id      : 107  factor  : 0  wdelta : 1
effect  : both  hfactor : 0  hdelta  : 1
width   : 0.2  vfactor : 0  ddelta  : 1
```

```
\definefontfeature
  [effect-2]
  [effect={width=.3}]
```

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

This time we use:

```
id      : 108  factor  : 0  wdelta : 1
effect  : both  hfactor : 0  hdelta  : 1
width   : 0.3  vfactor : 0  ddelta  : 1
```

```
\definefontfeature
  [effect-3]
  [effect={width=.3,delta=0.4}]
```

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

We have now tweaked one more property and show the fontkerns in order to see what happens with them:

```
id      : 109  factor  : 0  wdelta : 0.4
effect  : both  hfactor : 0  hdelta  : 0.4
width   : 0.3  vfactor : 0  ddelta  : 0.4
```

```
\definefontfeature
  [effect-4]
  [effect={width=.3,delta=0.4,factor=0.3}]
```

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

An additional parameter **factor** will influence the way (for instance) kerns get affected:

```

id      : 111  factor  : 0.3  wdelta : 0.4
effect  : both  hfactor : 0.3  hdelta  : 0.4
width   : 0.3   vfactor : 0.3  ddelta  : 0.4

```

## 11.3 Outlines

There are four effects. Normally a font is rendered with effect `inner`. The `outer` effect just draws the outlines while `both` gives a rather fat result. The `hidden` effect hides the text.

```
\definefontfeature
[effect-5]
[effect={width=0.2,delta=0.4,factor=0.3,effect=inner}]
```

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

An inner effect is rather useless unless you want to use the other properties of this mechanism.

```
\definefontfeature
[effect-6]
[effect={width=.2,delta=0.4,factor=0.3,effect=outer}]
```

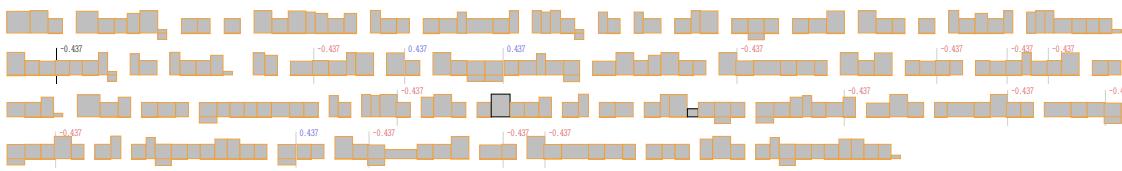
The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

```
\definefontfeature
[effect-7]
[effect={width=.2,delta=0.4,factor=0.3,effect=both}]
```

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

```
\definefontfeature
[effect-8]
[effect={width=.2,delta=0.4,factor=0.3,effect=hidden},
boundingbox=yes] % to show something
```

We also show the boundingboxes of the glyphs here so that you can see what you're missing. Actually this text is still there and you can select it in the viewer.



## 11.4 The logic

In order to support this I had to make some choices. The calculations involved are best explained in terms of CONTEXT font machinery.

$$\Delta_{\text{wd}} = \text{effect}_{\text{wdelta}} \times \text{parameter}_{\text{hfactor}} \times \text{effect}_{\text{width}} \times 100$$

$$\Delta_{ht} = \text{effect}_{h\delta} \times \text{parameter}_{vf} \times \text{effect}_w \times 100$$

$$\Delta_{dp} = \text{effect}_{ddelta} \times \text{parameter}_{vfactor} \times \text{effect}_{width} \times 100$$

The factors in the parameter namespace are adapted according to:

$$\Delta_{\text{factor}} = \text{effect}_{\text{factor}} \times \text{parameters}_{\text{factor}}$$

$$\Delta_{\text{hfactor}} = \text{effect}_{\text{hfactor}} \times \text{parameters}_{\text{hfactor}}$$

$$\Delta_{\text{vfactor}} = \text{effect}_{\text{vfactor}} \times \text{parameters}_{\text{vfactor}}$$

The horizontal and vertical scaling factors default to the normal factor that defaults to zero so by default we have no additional scaling of for instance kerns. The width (wd), height (ht) and depth (dp) of a glyph are adapted in relation to the line width. A glyph is shifted in its bounding box by half the width correction. The delta defaults to one.

## 11.5 About features

This kind of boldening has limitations especially because some fonts use positioning features that closely relate to the visual font properties. Let's give some examples. The most common positioning is kerning. Take for instance these shapes:



The first one is that we start with. The circle and square have a line width of one unit and a distance (kern) of five units. The second pair has a line width of two units and the same distance while the third pair has a distance of seven units. So, in the last case we have just increased the kern with a value relative to the increase of line width.



In this example we have done the same but we started with a distance of zero. You can consider this a kind of anchoring. This happens in for instance cursive scripts where

entry and exit points are used to connect shapes. In a latin script you can think of a poor-mans attachment of a cedilla or ogonek. But what to do with for instance an accent on top of a character? In that case we could do the same as with kerning. However, when we mix styles we would like to have a consistent height so maybe there scaling is not a good idea. This is why we can set the factors and deltas explicitly for vertical and horizontal movements. However, this will only work well when a font is consistent in how it applies these movements. In this case, if could recognize cursive anchoring (the last pair in the example) we could compensate for it.



So, an interesting extension to the positioning part of the font handler could be to influence all the scaling factors: anchors, cursives, single and pair wise positioning in both directions (so eight independent factors). Technically this is no big deal so I might give it a go when I have a need for it.

## 11.6 Some (extreme) examples

The last decade buying a font has become a bit of a nightmare simply because you have to choose the weights that you need. It's the business model to not stick to four shapes in a few weights but offer a whole range and each of course costs money.

Latin Modern is based on Computer Modern and is meant for high resolution rendering. The design of the font is such that you can create instances but in practice that isn't done. One property that let the font stand out is its bold which runs rather wide. However, how about cooking up a variant? For this we will use a series of definitions:

```
\definefontfeature [effect-2-0-0]
  [effect={width=0.2,delta=0}]
\definefontfeature [effect-2-3-0]
  [effect={width=0.2,delta=0.3}]
\definefontfeature [effect-2-6-0]
  [effect={width=0.2,delta=0.6}]
\definefontfeature [effect-4-0-0]
  [effect={width=0.4,delta=0}]
\definefontfeature [effect-4-3-0]
  [effect={width=0.4,delta=0.3}]
\definefontfeature [effect-4-6-0]
  [effect={width=0.4,delta=0.6}]
\definefontfeature [effect-8-0-0]
  [effect={width=0.8,delta=0}]
\definefontfeature [effect-8-3-0]
  [effect={width=0.8,delta=0.3}]
\definefontfeature [effect-8-6-0]
  [effect={width=0.8,delta=0.6}]
```

```
\definefontfeature[effect-8-6-2]
  [effect={width=0.8,delta=0.6,factor=0.2}]
\definefontfeature[effect-8-6-4]
  [effect={width=0.8,delta=0.6,factor=0.4}]
```

And a helper macro:

```
\starttexdefinition ShowOneSample #1#2#3#4
%\testpage[5]
%\startsubsubsubject[title=\type{#1}]
\start
  \definedfont[#2*#3 @ 10pt]
  \setupinterlinespace
  \startlinecorrection
    \showglyphs \showfontkerns
    \scale[sx=#4,sy=#4]{effective n\"ots}
  \stoplinecorrection
  \blank[samepage]
  \dontcomplain
  \showfontkerns
  \margintext{\tt\txx\maincolor{#1}}
  \samplefile{ward}
  \par
\stop
%\stopsubsubsubject
\stoptexdefinition
```

We show some extremes, using the font used in this document. so don't complain about beauty here.

## Serif



**no effect** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.



**width=0.2 delta=0** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.



**width=0.2** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

# effective nöts

**width=0.2** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

# effective nöts

**width=0.4** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

# effective nöts

**width=0.4** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

# effective nöts

**width=0.4** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

# effective nöts

**width=0.8** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

# effective nöts

**width=0.8** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

# effective nöts

width=0.8

delta=0.6

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

# effective nöts

width=0.8

delta=0.6

factor=0.2

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

# effective nöts

width=0.8

delta=0.6

factor=0.4

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

SerifBold

# effective nöts

no effect

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

# effective nöts

width=0.2

delta=0

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

# effective nöts

width=0.2

delta=0.3

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

# effective nöts

width=0.2

delta=0.6

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

# effective nöts

width=0.4  
delta=0

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

# effective nöts

width=0.4  
delta=0.3

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

# effective nöts

width=0.4  
delta=0.6

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

# effective nöts

width=0.8  
delta=0

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

# effective nöts

width=0.8  
delta=0.3

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

# effective nöts

width=0.8  
delta=0.6

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

# effective nöts

width=0.8  
delta=0.6  
factor=0.2

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the

**effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.**

**effective nöts**

width=0.8  
delta=0.6  
factor=0.4

**The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.**

SerifItalic

*effective nöts*

**no effect** *The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

*effective nöts*

**width=0.2  
delta=0** *The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

*effective nöts*

**width=0.2  
delta=0.3** *The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

*effective nöts*

**width=0.2  
delta=0.6** *The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

*effective nöts*

width=0.4 The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

**effective nöts**

width=0.4 The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

**effective nöts**

width=0.4 The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

**effective nöts**

width=0.8 The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

**effective nöts**

width=0.8 The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

**effective nöts**

width=0.8 The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

**effective nöts**

width=0.8  
delta=0.6  
factor=0.2

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



effective nöts

width=0.8  
delta=0.6  
factor=0.4

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

SerifBoldItalic



effective nöts

no effect

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



effective nöts

width=0.2  
delta=0

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



effective nöts

width=0.2  
delta=0.3

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



effective nöts

width=0.2  
delta=0.6

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



effective nöts

width=0.4

delta=0

*The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

**effective nöts**

width=0.4

delta=0.3

*The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

**effective nöts**

width=0.4

delta=0.6

*The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

**effective nöts**

width=0.8

delta=0

*The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

**effective nöts**

width=0.8

delta=0.3

*The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

**effective nöts**

width=0.8

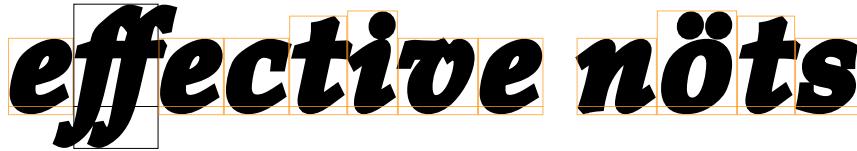
delta=0.6

*The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

**effective nöts**

width=0.8  
delta=0.6  
factor=0.2

*The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*



width=0.8  
delta=0.6  
factor=0.4

*The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.*

Sans



no effect

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



width=0.2  
delta=0

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



width=0.2  
delta=0.3

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



width=0.2  
delta=0.6

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



width=0.4  
delta=0

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

# effective nöts

width=0.4  
delta=0.3

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

# effective nöts

width=0.4  
delta=0.6

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

# effective nöts

width=0.8  
delta=0

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

# effective nöts

width=0.8  
delta=0.3

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

# effective nöts

width=0.8  
delta=0.6

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

# effective nöts

width=0.8  
delta=0.6  
factor=0.2

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

# effective nöts

width=0.8  
delta=0.6  
factor=0.4

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

Mono

# effective nöts

**no effect** The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

**width=0.2** The Earth, as a habitat for animal life, is in old age and has a fatal illness. **delta=0** Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

**width=0.2** The Earth, as a habitat for animal life, is in old age and has a fatal illness. **delta=0.3** Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

**width=0.2** The Earth, as a habitat for animal life, is in old age and has a fatal illness. **delta=0.6** Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

**width=0.4** The Earth, as a habitat for animal life, is in old age and has a fatal illness. **delta=0** Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

**width=0.4** The Earth, as a habitat for animal life, is in old age and has a fatal illness. **delta=0.3** Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

**width=0.4** The Earth, as a habitat for animal life, is in old age and has a fatal illness. **delta=0.6** Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

width=0.8  
delta=0  
The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

width=0.8  
delta=0.3  
The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

width=0.8  
delta=0.6  
The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

width=0.8  
delta=0.6  
factor=0.2  
The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

**effective nöts**

width=0.8  
delta=0.6  
factor=0.4  
The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day-and we humans are the cigarettes.

## 11.7 Pitfall

The quality of the result depends on how the font is made. For instance, ligatures can be whole shapes, replaced glyphs and/or repositioned glyphs, or whatever the designer thinks reasonable. In figure 11.1 this is demonstrated. We use the following feature sets:

```
\definefontfeature
  [demo-1]
  [default]
  [hlig=yes]
```

```
\definefontfeature
  [demo-2]
  [demo-1]
  [effect=0.5]
```

fist effe fist effe

texgyre pagella regular

fist effe fist effe

cambria

fist effe fist effe

ebgaramond 12 regular

**Figure 11.1** The effects on ligatures.

Normally the artifacts (as in the fi ligature in ebgaramond as of 2018) will go unnoticed at small sized. Also, when the user has a low res display, printer or when the publishers is one of those who print a scanned PDF the reader might not notice it at all. Most readers don't even know what to look at.

## 11.8 A modern Modern

So how can we make an effective set of Latin Modern that fits in todays look and feel. Of course this is a very subjective experiment but we've seen experiments with these fonts before (like these cm super collections). Here is an example of a typescript definition:

```
\starttypescriptcollection[modernlatin]

\definefontfeature[lm-rm-regular] [effect={width=0.15,delta=1.00}]
\definefontfeature[lm-rm-bold]      [effect={width=0.30,delta=1.00}]
\definefontfeature[lm-ss-regular]  [effect={width=0.10,delta=1.00}]
\definefontfeature[lm-ss-bold]     [effect={width=0.20,delta=1.00}]
\definefontfeature[lm-tt-regular]  [effect={width=0.15,delta=1.00}]
\definefontfeature[lm-tt-bold]    [effect={width=0.30,delta=1.00}]
\definefontfeature[lm-mm-regular] [effect={width=0.15,delta=1.00}]
\definefontfeature[lm-mm-bold]   [effect={width=0.30,delta=1.00}]

\starttypescript [serif] [modern-latin]
  \definefontsynonym
    [Serif] [file:lmroman10-regular]
    [features={default,lm-rm-regular}]
  \definefontsynonym
    [SerifItalic] [file:lmroman10-italic]
```

```

    [features={default,lm-rm-regular}]
\definefontsynonym
    [SerifSlanted] [file:lmromanslant10-regular]
    [features={default,lm-rm-regular}]
\definefontsynonym
    [SerifBold] [file:lmroman10-regular]
    [features={default,lm-rm-bold}]
\definefontsynonym
    [SerifBoldItalic] [file:lmroman10-italic]
    [features={default,lm-rm-bold}]
\definefontsynonym
    [SerifBoldSlanted] [file:lmromanslant10-regular]
    [features={default,lm-rm-bold}]
\stoptypescript

\starttypescript [sans] [modern-latin]
\definefontsynonym
    [Sans] [file:lmsans10-regular]
    [features={default,lm-ss-regular}]
\definefontsynonym
    [SansItalic] [file:lmsans10-oblique]
    [features={default,lm-ss-regular}]
\definefontsynonym
    [SansSlanted] [file:lmsans10-oblique]
    [features={default,lm-ss-regular}]
\definefontsynonym
    [SansBold] [file:lmsans10-regular]
    [features={default,lm-ss-bold}]
\definefontsynonym
    [SansBoldItalic] [file:lmsans10-oblique]
    [features={default,lm-ss-bold}]
\definefontsynonym
    [SansBoldSlanted] [file:lmsans10-oblique]
    [features={default,lm-ss-bold}]
\stoptypescript

\starttypescript [mono] [modern-latin]
\definefontsynonym
    [Mono] [file:lmmono10-regular]
    [features={default,lm-tt-regular}]
\definefontsynonym
    [MonoItalic] [file:lmmono10-italic]
    [features={default,lm-tt-regular}]
\definefontsynonym
    [MonoSlanted] [file:lmmonoslant10-regular]

```

```

        [features={default,lm-tt-regular}]
\definefontsynonym
    [MonoBold] [file:lmmono10-regular]
    [features={default,lm-tt-bold}]
\definefontsynonym
    [MonoBoldItalic] [file:lmmono10-italic]
    [features={default,lm-tt-bold}]
\definefontsynonym
    [MonoBoldSlanted] [file:lmmonoslant10-regular]
    [features={default,lm-tt-bold}]
\stotypescript

\starttypescript [math] [modern-latin]
    \loadfontgoodies[lm]
\definefontsynonym
    [MathRoman] [file:latinmodern-math-regular.otf]
    [features={math\mathsizesuffix,lm-mm-regular,mathextra},
     goodies=lm]
\definefontsynonym
    [MathRomanBold] [file:latinmodern-math-regular.otf]
    [features={math\mathsizesuffix,lm-mm-bold,mathextra},
     goodies=lm]
\stotypescript

\starttypescript [modern-latin]
    \definetypeface [\typescriptone]
        [rm] [serif] [modern-latin] [default]
    \definetypeface [\typescriptone]
        [ss] [sans] [modern-latin] [default]
    \definetypeface [\typescriptone]
        [tt] [mono] [modern-latin] [default]
    \definetypeface [\typescriptone]
        [mm] [math] [modern-latin] [default]
    \quittypescriptscanning
\stotypescript

\stotypescriptcollection

```

We show some more samples now for which we use `zapf.tex`.

```
\switchtobodyfont[modern-latin,rm,10pt]
```

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks,

and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

*Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.*

*Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.*

*Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.*

*Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.*

*Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.*

\switchtobodyfont [modern-latin,ss,10pt]

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely-praised program, called up on the screen, will make everything automatic from now on.

\switchtobodyfont [modern-latin,tt,10pt]

Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely--praised program, called up on the screen, will make everything automatic from now on.

**Coming 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 PC or software. There 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. Many people  
are just fascinated by their PC's tricks, and think that a  
widely--praised program, called up on the screen, will make  
everything automatic from now on.**

*Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely--praised program, called up on the screen, will make everything automatic from now on.*

***Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely--praised program, called up on the screen, will make everything automatic from now on.***

*Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely--praised program, called up on the screen, will make everything automatic from now on.*

***Coming 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 PC or software. There 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. Many people are just fascinated by their PC's tricks, and think that a widely--praised program, called up on the screen, will make everything automatic from now on.***

## 11.9 Finetuning

In practice we only need to compensate the width but can leave the height and depth untouched. In the following examples we see the normal bold next to the regular as well as the boldened version. For this we will use a couple of definitions:

```

\definefontfeature[lm-bald] [effect={width=0.25, effect=both}]
\definefontfeature[pg-bald] [effect={width=0.25, effect=both}]
\definefontfeature[dj-bald] [effect={width=0.35, effect=both}]

\definefontfeature
  [lm-bold]
  [effect={width=0.25, hdelta=0, ddelta=0, effect=both},
   extend=1.10]

\definefontfeature
  [pg-bold]
  [effect={width=0.25, hdelta=0, ddelta=0, effect=both},
   extend=1.00]

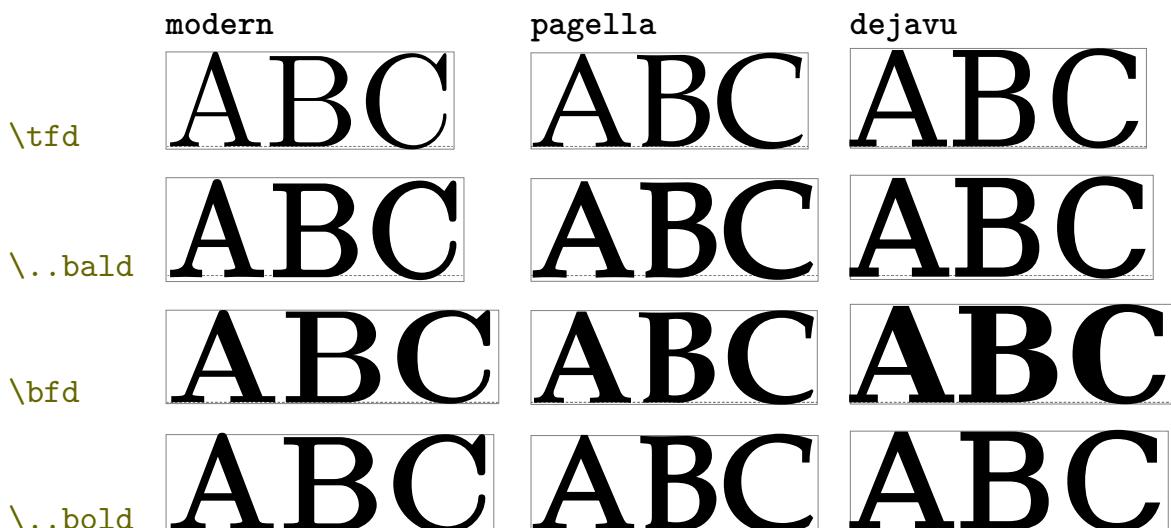
\definefontfeature
  [dj-bold]
  [effect={width=0.35, hdelta=0, ddelta=0, effect=both},
   extend=1.05]

\definefont[lmbald] [Serif*default, lm-bald sa d]
\definefont[pgbald] [Serif*default, pg-bald sa d]
\definefont[djbald] [Serif*default, dj-bald sa d]

\definefont[lmbold] [Serif*default, lm-bold sa d]
\definefont[pgbold] [Serif*default, pg-bold sa d]
\definefont[djbold] [Serif*default, dj-bold sa d]

```

We can combine the extend and effect features to get a bold running as wide as a normal bold. We limit the height and depth so that we can use regular and bold in the same sentence. It's all a matter of taste, but some control is there.



Let's take another go at Pagella. We define a few features, colors and fonts first:

```

\definefontfeature
  [pg-fake-1]
  [effect={width=0.25, effect=both}]

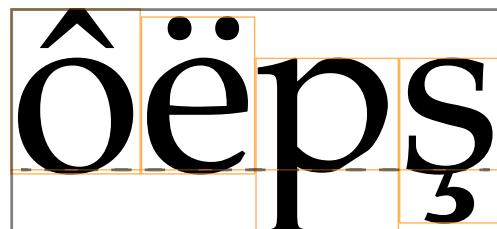
\definefontfeature
  [pg-fake-2]
  [effect={width=0.25, hdelta=0, ddelta=0, effect=both}]

\definefont [pgregular]  [Serif*default]
\definefont [pgbold]     [SerifBold*default]
\definefont [pgfakebolda] [Serif*default, pg-fake-1]
\definefont [pgfakeboldb] [Serif*default, pg-fake-2]

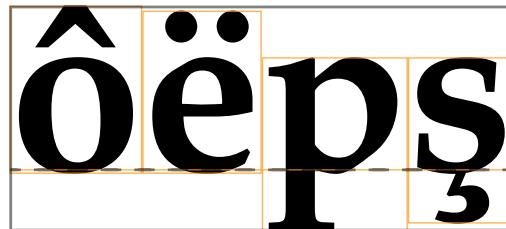
\definecolor [color-pgregular]  [t=.5, a=1, r=.6]
\definecolor [color-pgbold]     [t=.5, a=1, g=.6]
\definecolor [color-pgfakebolda] [t=.5, a=1, b=.6]
\definecolor [color-pgfakeboldb] [t=.5, a=1, r=.6, g=.6]

```

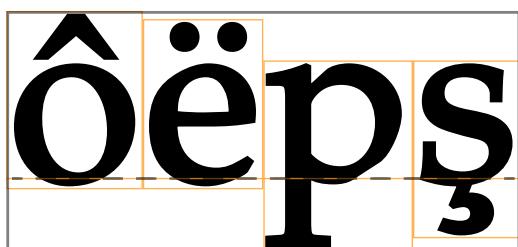
When we apply these we get the results of figure 11.2 while we show the same overlaid in figure 11.3. As you can see, the difference in real bold and fake bold is subtle: the inner shape of the ‘o’ differs. Also note that the position of the accents doesn’t change in the vertical direction but moves along with the width.



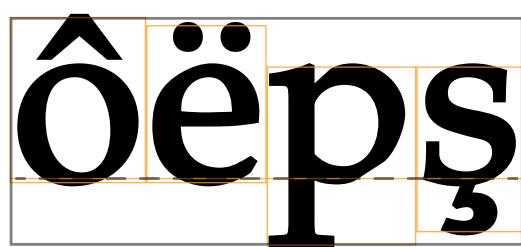
regular (red)



bold (green)



fakebolda (blue)



fakeboldb (yellow)

**Figure 11.2** Four pagella style variants compared.

## 11.10 The code

The amount of code involved is not that large and is a nice illustration of what L<sup>A</sup>T<sub>E</sub>X provides (I have omitted a few lines of tracing and error reporting). The only thing added to the font scaler elsewhere is that we pass the `mode` and `width` parameters to T<sub>E</sub>X so that they get used in the backend to inject the few operators needed.

ôëps ôëps

bold over regular

ôëps ôëps

fakebolda over regular

ôëps ôëps

fakeboldb over regular

ôëps ôëps

fakeboldb over bold

ôëps ôëps

fakeboldb over fakebolda

ôëps ôëps

all four overlaid

Figure 11.3 Four pagella style variants overlaid.

```
local effects = {  
    inner  = 0,  
    outer  = 1,  
    both   = 2,  
    hidden = 3,  
}  
  
local function initialize(tfmdata,value)  
    local spec  
    if type(value) == "number" then  
        spec = { width = value }  
    else  
        spec = utilities.parsers.settings_to_hash(value)  
    end  
    local effect = spec.effect or "both"  
    local width  = tonumber(spec.width) or 0  
    local mode   = effects[effect]  
    if mode then  
        local factor  = tonumber(spec.factor) or 0  
        local hfactor = tonumber(spec.vfactor) or factor  
        local vfactor = tonumber(spec.hfactor) or factor
```

```

local delta    = tonumber(spec.delta)    or 1
local wdelta   = tonumber(spec.wdelta)   or delta
local hdelta   = tonumber(spec.hdelta)   or delta
local ddelta   = tonumber(spec.ddelta)   or hdelta
tfmdata.parameters.mode    = mode
tfmdata.parameters.width   = width * 1000
tfmdata.properties.effect = {
    effect  = effect, width   = width,
    wdelta   = wdelta, factor  = factor,
    hdelta   = hdelta, hfactor = hfactor,
    ddelta   = ddelta, vfactor = vfactor,
}
end
end

local function manipulate(tfmdata)
    local effect = tfmdata.properties.effect
    if effect then
        local characters = tfmdata.characters
        local parameters = tfmdata.parameters
        local multiplier = effect.width * 100
        local wdelta = effect.wdelta * parameters.hfactor * multiplier
        local hdelta = effect.hdelta * parameters.vfactor * multiplier
        local ddelta = effect.ddelta * parameters.vfactor * multiplier
        local hshift = wdelta / 2
        local factor  = (1 + effect.factor) * parameters.factor
        local hfactor = (1 + effect.hfactor) * parameters.hfactor
        local vfactor = (1 + effect.vfactor) * parameters.vfactor
        for unicode, char in next, characters do
            local oldwidth = char.width
            local oldheight = char.height
            local olddepth = char.depth
            if oldwidth and oldwidth > 0 then
                char.width = oldwidth + wdelta
                char.commands = {
                    { "right", hshift },
                    { "char", unicode },
                }
            end
            if oldheight and oldheight > 0 then
                char.height = oldheight + hdelta
            end
            if olddepth and olddepth > 0 then
                char.depth = olddepth + ddelta
            end
        end
    end
end

```

```

    end
    parameters.factor  = factor
    parameters.hfactor = hfactor
    parameters.vfactor = vfactor
  end
end

local specification = {
  name      = "effect",
  description = "apply effects to glyphs",
  initializers = {
    base = initialize,
    node = initialize,
  },
  manipulators = {
    base = manipulate,
    node = manipulate,
  },
}
}

fonts.handlers.otf.features.register(specification)
fonts.handlers.afm.features.register(specification)

```

The real code is slightly more complex because we want to stack virtual features properly but the principle is the same.

## 11.11 Arabic

It is tempting to test effects with arabic but we need to keep in mind that for that we should add some more support in the CONTEXt font handler. Let's define some features.

```

\definefontfeature
  [bolden-arabic-1]
  [effect={width=0.4}]

\definefontfeature
  [bolden-arabic-2]
  [effect={width=0.4,effect=outer}]

\definefontfeature
  [bolden-arabic-3]
  [effect={width=0.5,wdelta=0.5,delta=.2,hdelta=.2,factor=.1}]

```

With MICROSOFT Arabtype the `khatt-ar.tex` looks as follows:

```

\setupalign
[righttoleft]

\setupinterlinespace
[1.5]

\start
\definedfont[arabictest*arabic,bolden-arabic-1 @ 30pt]
\samplefile{khatt-ar}\par
\definedfont[arabictest*arabic,bolden-arabic-2 @ 30pt]
\samplefile{khatt-ar}\par
\definedfont[arabictest*arabic,bolden-arabic-3 @ 30pt]
\samplefile{khatt-ar}\par
\stop

```

قالَ عَلَيْهِ بْنُ أَبِي طَالِبٍ لِّكَاتِيهِ عُبَيْدِ اللَّهِ بْنِ أَبِي رَافِعٍ: أَلْقُ دَوَاتِكَ، وَأَطِلْ جِلْفَةَ قَلْمِيكَ، وَ فَرَّخْ بَيْنَ السُّطُورِ، وَ قَرْمَطْ بَيْنَ الْحُرُوفِ؛ فَإِنَّ ذَلِكَ أَجْدَرُ بِصَبَاحةِ الْخَطِّ.

قالَ عَلَيْهِ بْنُ أَبِي طَالِبٍ لِّكَاتِيهِ عُبَيْدِ اللَّهِ بْنِ أَبِي رَافِعٍ: أَلْقُ دَوَاتِكَ، وَأَطِلْ جِلْفَةَ قَلْمِيكَ، وَ فَرَّخْ بَيْنَ السُّطُورِ، وَ قَرْمَطْ بَيْنَ الْحُرُوفِ؛ فَإِنَّ ذَلِكَ أَجْدَرُ بِصَبَاحةِ الْخَطِّ.

قالَ عَلَيْهِ بْنُ أَبِي طَالِبٍ لِّكَاتِيهِ عُبَيْدِ اللَّهِ بْنِ أَبِي رَافِعٍ: أَلْقُ دَوَاتِكَ، وَأَطِلْ جِلْفَةَ قَلْمِيكَ، وَ فَرَّخْ بَيْنَ السُّطُورِ، وَ قَرْمَطْ بَيْنَ الْحُرُوفِ؛ فَإِنَّ ذَلِكَ أَجْدَرُ بِصَبَاحةِ الْخَطِّ.

And with Idris' Husayni we get:

```

\setupalign
[righttoleft]

\setupinterlinespace
[1.5]

\start
\definedfont[arabictest*arabic,bolden-arabic-1 @ 30pt]

```

```

\samplefile{khatt-ar}\par
\definedfont[arabictest*arabic,bolden-arabic-2 @ 30pt]
\samplefile{khatt-ar}\par
\definedfont[arabictest*arabic,bolden-arabic-3 @ 30pt]
\samplefile{khatt-ar}\par
\stop

```

قَالَ عَلِيُّ بْنُ أَبِي طَالِبٍ لِكَاتِبِهِ عُبَيْدِ اللَّهِ بْنِ أَبِي رَافِعٍ: أَلْقِ  
دَوَاتَكَ، وَأَطِلْ جِلْفَةَ قَلْمِيكَ، وَفَرِّجْ بَيْنَ السُّطُورِ، وَقَرْمَطْ بَيْنَ  
الْحُرُوفِ؛ فَإِنَّ ذَلِكَ أَجْدَرُ بِصَبَاحَةِ الْخَطِّ.

قَالَ عَلِيُّ بْنُ أَبِي طَالِبٍ لِكَاتِبِهِ عُبَيْدِ اللَّهِ بْنِ أَبِي رَافِعٍ: أَلْقِ  
دَوَاتَكَ، وَأَطِلْ جِلْفَةَ قَلْمِيكَ، وَفَرِّجْ بَيْنَ السُّطُورِ، وَقَرْمَطْ بَيْنَ  
الْحُرُوفِ؛ فَإِنَّ ذَلِكَ أَجْدَرُ بِصَبَاحَةِ الْخَطِّ.

قَالَ عَلِيُّ بْنُ أَبِي طَالِبٍ لِكَاتِبِهِ عُبَيْدِ اللَّهِ بْنِ أَبِي رَافِعٍ: أَلْقِ  
دَوَاتَكَ، وَأَطِلْ جِلْفَةَ قَلْمِيكَ، وَفَرِّجْ بَيْنَ السُّطُورِ، وَقَرْمَطْ بَيْنَ  
الْحُرُوفِ؛ فَإِنَّ ذَلِكَ أَجْدَرُ بِصَبَاحَةِ الْخَطِّ.

Actually, quite okay are the following. We don't over do bold here and to get a distinction we make the original thinner.

```

\definefontfeature[effect-ar-thin] [effect={width=0.01, effect=inner}]
\definefontfeature[effect-ar-thick] [effect={width=0.20, extend=1.05}]

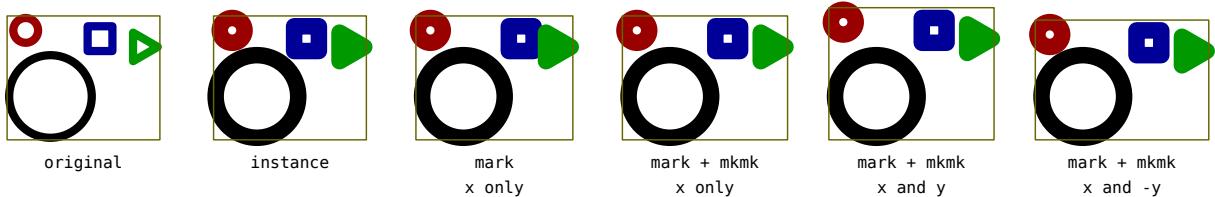
```

قَالَ عَلِيُّ بْنُ أَبِي طَالِبٍ لِكَاتِبِهِ عُبَيْدِ اللَّهِ بْنِ أَبِي رَافِعٍ: أَلْقِ  
دَوَاتَكَ، وَأَطِلْ جِلْفَةَ قَلْمِيكَ، وَفَرِّجْ بَيْنَ السُّطُورِ، وَقَرْمَطْ بَيْنَ  
الْحُرُوفِ؛ فَإِنَّ ذَلِكَ أَجْدَرُ بِصَبَاحَةِ الْخَطِّ.

فَتَالَ عَلِيُّ بْنُ أَبِي طَالِبٍ لَكَاتِبٍ عُبَيْدِ اللَّهِ بْنِ أَبِي رَافِعٍ: أَلْقَ دَوَاتَكَ، وَأَطِلْ جُلْفَةَ قَلْمِيكَ، وَفَرْجٌ بَيْنَ السُّطُورِ، وَقَرْمَطٌ بَيْنَ الْحُرُوفِ؛ فَإِنَّ ذَلِكَ أَجْدَرُ بِصَبَاحَةِ الْخَطِّ.

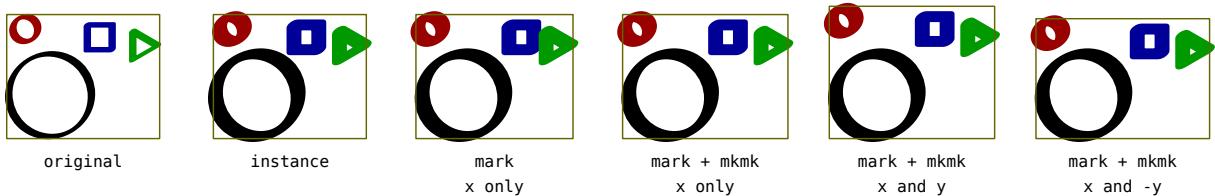
The results are acceptable at small sizes but at larger sizes you will start to see kerning, anchoring and cursive artifacts. The outline examples show that the amount of overlap differs per font and the more overlap we have the better boldening will work.

In arabic (and sometimes latin) fonts the marks (or accents in latin) are attached to base shapes and normally one will use the `mark` to anchor a mark to a base character or specific component of a ligature. The `mkmk` feature is then used to anchor marks to other marks. Consider the following example.



We start with `original`: a base shape with three marks: the red circle and blue square anchor to the base and the green triangle anchors to the blue square. When we bolden, the shapes will start touching. In the case of latin scripts, it's normal to keep the accents on the same height so this is why the third picture only shifts in the horizontal direction. The fourth picture demonstrates that we need to compensate the two bound marks. One can decide to move the lot up as in the fifth picture but that is no option here.

Matters can be even more complex when a non circular pen is introduced. In that case a transformation from one font to another using the transformed OPENTYPE positioning logic (values) is even more tricky and unless one knows the properties (and usage) of a mark it makes no sense at all. Actually the sixths variant is probably nicer here but there we actually move the marks down!



For effects this means that when it gets applied to such a font, only small values work out well.

## 11.12 Math

Math is dubious as there is all kind of positioning involved. Future versions might deal with this, although bolder math (math itself has bold, so actually we're talking of bold with some heavy) is needed for titling. If we keep that in mind we can actually just bolden math and probably most will come out reasonable well. One of the potential troublemakers is the radical (root) sign that can be bound to a rule. Bumping the rules is no big deal and patching the relevant radical properties neither, so indeed we can do:

```
\switchtobodyfont [modernlatin,17.3pt]
$ 
  \mr \darkblue \getbuffer[mathblob] \quad
  \mb \darkgreen \getbuffer[mathblob]
$
```

$$2 \times \sqrt{\frac{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}} \quad 2 \times \sqrt{\frac{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}}$$

Where the `mathblob` buffer is:

```
2\times\sqrt{\frac{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}}
```

Here you also see a fraction rule that has been bumped. In display mode we get:

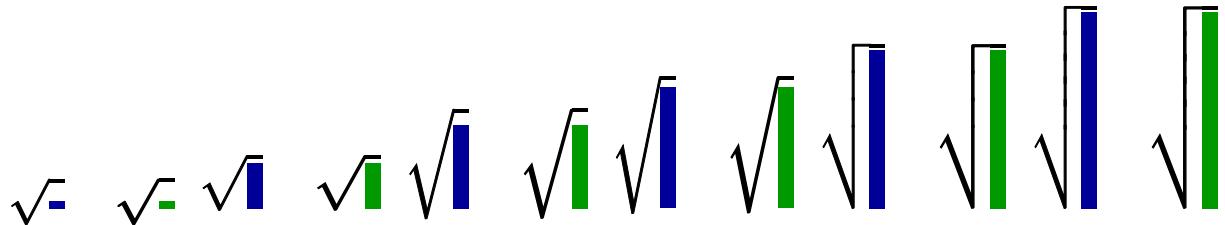
```
\switchtobodyfont [modernlatin,17.3pt]
\startformula
  \mr \darkblue \getbuffer[mathblob] \quad
  \mb \darkgreen \getbuffer[mathblob]
\stopformula
```

$$2 \times \sqrt{\frac{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}} \quad 2 \times \sqrt{\frac{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}}$$

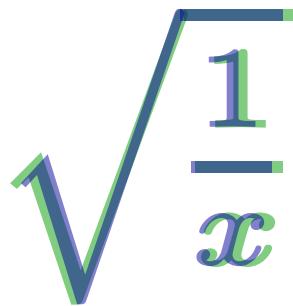
Extensibles behave well too:

```
\switchtobodyfont [modernlatin,17.3pt]
\dostepwiserecurse {1} {30} {5} {
$ 
  \mr \sqrt{\blackrule[width=2mm,height=#1mm,color=darkblue]}
  \quad
```

```
\mb \sqrt{\blackrule[width=2mm,height=#1mm,color=darkgreen]}
$}
}
```



In figure 11.4 we overlay regular and bold. The result doesn't look that bad after all, does it? It took however a bit of experimenting and a fix in LUATEX: pickup the value from the font instead of the currently used (but frozen) math parameter.



**Figure 11.4** Modern Latin regular over bold.

In case you wonder how currently normal Latin Modern bold looks, here we go:

```
\switchtobodyfont[latinmodern,17.3pt]
\startformula
  \mr \darkblue \getbuffer[mathblob] \quad
  \mb \darkgreen \getbuffer[mathblob]
\stopformula
```

$$2 \times \sqrt{\frac{\sqrt{2}}{\sqrt{2}}} \quad 2 \times \sqrt{\frac{\sqrt{2}}{\sqrt{2}}}$$

Two expressions showing the square root of the fraction  $\sqrt{2}/\sqrt{2}$ . The first expression uses a standard font for the numbers and a bold font for the square root symbol. The second expression uses a bold font for both the numbers and the square root symbol.

$$2 \times \sqrt{\frac{\sqrt{2}}{\sqrt{2}}} \quad 2 \times \sqrt{\frac{\sqrt{2}}{\sqrt{2}}}$$

dejavu: 2 2 2 **2 2 2**

pagella: 2 2 2 **2 2 2**

termes: 2 2 2 **2 2 2**

$$2 \times \sqrt{\frac{\sqrt{2}}{\sqrt{2}}} \quad 2 \times \sqrt{\frac{\sqrt{2}}{\sqrt{2}}}$$

bonum: 2 2 2 **2 2 2**

schola: 2 2 2 **2 2 2**

cambria: 2 2 2 **2 2 2**

I must admit that I cheat a bit. In order to get a better looking pseudo math we need to extend the shapes horizontally as well as squeeze them a bit vertically. So, the real effect definitions more look like this:

```
\definefontfeature
[boldened-30]
[effect={width=0.3,extend=1.15,squeeze=0.985,%
delta=1,hdelta=0.225,delta=0.225,vshift=0.225}]
```

and because we can calculate the funny values sort of automatically, this gets simplified to:

```
\definefontfeature
[boldened-30]
[effect={width=0.30,auto=yes}]
```

We leave it to your imagination to figure out what happens behind the screens. Just think of some virtual font magic combined with the engine supported `extend` and `squeeze` function. And because we already support bold math in CONTeXt, you will get it when you are doing bold titling.

```
\def\MathSample
{\overbrace{2 +
\sqrt{\frac{\sqrt{\frac{\sqrt{2}}{\sqrt{2}}}}{\sqrt{2}}}}
{\sqrt{\frac{\sqrt{\underbrace{2}}}{\overline{2}}}}}}
```

```
\definehead
[mysubject]
[subject]
```

```
\setuphead
[mysubject]
[style=\tfc,
```

```

color=darkblue,
before=\blank,
after=\blank]

\mysubject{Regular \quad \$\mathit{MathSample}\quad \mathit{mb}\mathit{MathSample\$} }

\setuphead
[mysubject]
[style=\bfc,
 color=darkred]

\mysubject{Bold \quad \$\mathit{MathSample}\quad \mathit{mb}\mathit{MathSample\$} }

```

The image shows four mathematical expressions arranged in a 2x2 grid. The top row is labeled "Regular" and the bottom row is labeled "Bold". The left column contains expressions in dark blue, and the right column contains expressions in dark red. All expressions are identical:  $2 + \sqrt{\frac{\sqrt{2}}{\sqrt{2}}}$ . The text "Regular" is in blue, and the text "Bold" is in red. The numbers and symbols are rendered in a serif font, while the text labels are in a sans-serif font.

Of course one can argue about the right values for boldening and compensation if dimensions so don't expect the current predefined related features to be frozen yet.

For sure this mechanism will create more fonts than normal but fortunately it can use the low level optimizations for sharing instances so in the end the overhead is not that large. This chapter uses 36 different fonts, creates 270 font instances (different scaling and properties) of which 220 are shared in the backend. The load time is 5 seconds in LUATEX and 1.2 seconds in LUAJITTEX on a somewhat old laptop with a i7-3840QM processor running 64 bit MS WINDOWS. Of course we load a lot of bodyfonts at different sizes so in a normal run the extra loading is limited to just a couple of extra instances for math (normally 3, one for each math size).

## 11.13 Conclusion

So what can we conclude? When we started with LUATEX, right from the start CONTEXT supported true UNICODE math by using virtual UNICODE math fonts. One of the objectives of the TEXGyre project is to come up with a robust complete set of math

fonts, text fonts with a bunch of useful symbols, and finally a subset bold math font for titling. Now we have real OPENTYPE math fonts, although they are still somewhat experimental. Because we're impatient, we now provide bold math by using effects but the future will learn to what extent the real bold math fonts will differ and be more pleasant to look at. After all, what we describe here is just an experiment that got a bit out of hands.

## 12 More (new) expansion trickery

Contrary to what one might expect when looking at macro definitions,  $\text{\TeX}$  is pretty efficient. Occasionally I wonder if some extra built in functionality could help me write better code but when you program with a bit care there is often not much to gain in terms of tokens and performance.<sup>17</sup> Also, some possible extensions probably only would be applied a few times which makes them low priority. When you look at the extensions brought by  $\varepsilon\text{-}\text{\TeX}$  the number is not that large, and LUATEX only added a few that deal with the language, for instance `\expanded` which is like an `\edef` without the defining a macro and acts on a token list wrapped in (normally) curly braces. Just as reference we mention some of the expansion related helpers.

command	argument	comment
<code>\expandafter</code>	<code>token</code>	The token after the next token gets expanded (one level only). In tricky $\text{\TeX}$ code you can often see multiple such commands in sequence which makes a nice puzzle.
<code>\noexpand</code>	<code>token</code>	The token after this command is not expanded in the context of expansion.
<code>\expanded</code>	<code>{tokens}</code>	The given token list is expanded. This command showed up early in LUATEX development and was taken from $\varepsilon\text{-}\text{\TeX}$ follow-ups. I have mails from 2011 mentioning its presence in PDFTEX 1.50 (which was targeted in 2008) but somehow it never ended up in a production version at that time (and we're still not at that version). In CONTEXt we already had a command with that name so there we use <code>\normalexpanded</code> . Users normally can just use the CONTEXt variant of <code>\expanded</code> .
<code>\unexpanded</code>	<code>{tokens}</code>	The given token list is hidden from expansion. Again, in CONTEXt we already had a command serving as prefix for definitions so instead we use <code>\normalunexpanded</code> . In the core of CONTEXt this new $\varepsilon\text{-}\text{\TeX}$ command is hardly used.
<code>\detokenize</code>	<code>{tokens}</code>	The given tokenlist becomes (basically) verbatim $\text{\TeX}$ code. We had something like that in CONTEXt but have no nameclash. It is used in a few places. It's also an $\varepsilon\text{-}\text{\TeX}$ command.
<code>\scantokens</code>	<code>{tokens}</code>	This primitive interprets its argument as a pseudo file. We don't really use it.

---

<sup>17</sup> The long trip to the yearly BachoTeX meeting is always a good opportunity to ponder  $\text{\TeX}$  and its features. The new functionality discussed here is a side effect of the most recent trip.

\scantextokens {tokens}	This LUATEX primitive does the same but has no end-of-file side effects. This one is also not really used in CONTEXT.
\protected \.def	The definition following this prefix, introduced in $\varepsilon$ -TEX, is unexpandable in the context of expansion. We already used such a command in CONTEXT but with a completely different meaning so use \normalprotected as prefix or \unexpanded which is an alias.

Here I will present two other extensions in LUATEX that can come in handy, and they are there simply because their effect can hardly be realized otherwise (never say never in TEX). One has to do with immediately applying a definition, the other with user defined conditions. The first one relates directly to expansion, the second one concerns conditions and relates more to parsing branches which on purpose avoids expansion.

For the first one I use some silly examples. I must admit that although I can envision useful application, I really need to go over the large amount of CONTEXT source code to really find a place where it is making things better. Take the following definitions:

```
\newcount\NumberOfCalls

\def\TestMe{\advance\NumberOfCalls1 }

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

\meaning\Tested
```

The result is a macro \Tested that not only has the unexpanded incrementing code in its body but also hasn't done any advancing:

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

Of course when you're typesetting something, this kind of expansion normally is not needed. Instead of the above definition we can define \TestMe in a way that expands the assignment immediately. You need of course to be aware of preventing look ahead interference by using a space or \relax (often an expression works better as it doesn't leave an \relax).

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

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

\meaning\Tested
```

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

macro:->bar:3

Here is a somewhat silly example of an expanded comparison of two 'strings':

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

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

\meaning\Tested
```

I don't remember many cases where I needed such an expanded comparison. We have a variant in CONTeXt that uses LUA but that one is not really used in the core. Anyway, the above code gives:

macro:->(nop/yes)

You can do the same assignments as in preambles of \halign and after \accent which means that assignments to box registers are blocked (boxing involves grouping and delayed assignments and so). The error you will get when you use a non-assignment command refers to a prefix, because internally such commands are called prefixed commands. Leading spaces and \relax are ignored.

In addition to this one-time immediate assignment a pseudo token list variant is provided, so the above could be rewritten to:

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

```
\next}
```

While `\expanded` first builds a token lists that then gets used, the `\immediateas-signed` primitive just walls over the list delimited by curly braces.

A next extension concerns conditions. If you have done a bit of extensive TeX programming you know that nested conditions need to be properly constructed in for instance macro bodies. This is because (for good reason) TeX goes into a fast scanning mode when there is a match and it has to skip the `\else` upto `\fi` branch. In order to do that properly a nested `\if` in there needs to have a matching `\fi`.

In practice this is no real problem and careful coding will never give a problem here: you can either hide nested code in a macro or somehow jump over nested conditions if really needed. Actually you only need to care when you pickup a token inside the branch because likely you don't want to pick up for instance a `\fi` but something that comes after it. Say that we have a sane conditional setup like this:

```
\newif\iffloor \foofalse
\newif\ifbar \bartrue

\ifloor
  \ifbar \else \fi
\else
  \ifbar \else \fi
\fi
```

Here the `\iffloor` and `\ifbar` need to be equivalent to `\iftrue` or `\iffalse` in order to succeed well and that is what for instance `\footrue` and `\foofalse` will do: change the meaning of `\iffloor`.

But imagine that you want something more complex. You want for instance to let `\ifbar` do some calculations. In that case you want it to behave a bit like what a so called `vardef` in METAPOST does: the end result is what matters. Now, because TeX macros often are a complex mix of expandable and non-expandable this is not that trivial. One solution is a dedicated definer, say `\cdef` for defining a macro with conditional properties. I actually implemented such a definer a few years ago but left it so long in a folder with ideas that I only found it back after I had come up with another solution. It was probably proof that it was not that good an idea.

The solution implemented in LUATEX is just a special case of a test: `\ifcondition`. When looking at the next example, keep in mind that from the perspective of TeX's scanner it only needs to know if something is a token that does some test and has a matching `\fi`. For that purpose you can consider `\ifcondition` to be `\iftrue`. When TeX actually wants to do a test, which is the case in the true branch, then it will simply ignore this `\ifcondition` primitive and expands what comes after it (which is TeX's natural behaviour). Effectively `\ifcondition` has no meaning except from when it has to be skipped, in which case it's a token flagged as `\if` kind of command.

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

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

Wrapped in a macro you can actually make this fully expandable when you use the previously mentioned immediate assignment. Here is another example:

```
\unexpanded\def\onoddpag
{\ifodd\count0 }

\ifcondition\onoddpag odd \else even \fi page
```

The previously defined comparison macro can now be rewritten as:

```
\def\equaltokens#1#2%
{\immediateassignment\edef\tempa{#1}%
 \immediateassignment\edef\tempb{#2}%
 \ifx\tempa\tempb

\def\expandeddoifelse#1#2#3#4%
{\ifcondition\equaltokens{#1}{#2}%
 \immediateassignment\def\next{#3}%
 \else
 \immediateassignment\def\next{#4}%
 \fi
\next}
```

When used this way it will of course also work without the `\ifcondition` but when used nested it can be like this. This last example also demonstrates that this feature probably only makes sense in more complicated cases where more work is done in the `\onoddpag` or `\equaltokens` macro. And again, I am not sure if for instance in CONTEXt I have a real use for it because there are only a few cases where nesting like

this could benefit. I did some tests with a low level macro where it made the code look nicer. It was actually a bit faster but most core macros are not called that often. Although the overhead of this feature can be neglected, performance should not be the reason for using it: in CONTEXT for instance one can often only measure such possible speed-ups on macros that are called tens or hundreds of thousands of times and that seldom happens in a real run and even then a change from say 0.827 seconds to 0.815 seconds for 10K calls of a complex case is just noise as the opposite can also happen.

Although not strictly necessary these extensions might make some code look better so that is why they officially will be available in the 1.09 release of LUATEX in fall 2018. It might eventually inspire me to go over some code and see where I can improve the look and feel.

The last few years I have implemented some more ideas as local experiments, for instance `\futurelet` variant or a simple (one level) `\expand`, but in the end rejected them because there is no real benefit in them (no better looking code, no gain in performance, hard to document, possible side effects, etc.), so it is very unlikely that we will have more extensions like this. After all, we could do more than 40 years without them. Although . . . who knows what we will provide in LUATEX version 2.

## 145 More (new) expansion trickery

# 13 Amputating code

## 13.1 Introduction

Because CONTEXt is already rather old in terms of software life and because it evolves over time, code can get replaced by better code. Reasons for this can be:

- a better understanding of the way T<sub>E</sub>X and METAPOST work
- demand for more advanced options
- a brainwave resulting in a better solution
- new functionality provided in T<sub>E</sub>X engine used
- the necessity to speed up a core process

Replacing code that in itself does a good job but is no longer the best to be used comes with sentiments. It can be rather satisfying to cook up a (conceptually as well as code-wise) good solution and therefore removing code from a file can result in a somewhat bad feeling and even a feeling of losing something. Hence the title of this chapter.

Here I will discuss one of the more complex subsystems: the one dealing with typeset text in METAPOST graphics. I will stick to the principles and not present (much) code as that can be found in archives. This is not a tutorial, but more a sort of wrap-up for myself. It anyhow show the thinking behind this mechanism. I'll also introduce a new LUATEX feature here: subruns.

## 13.2 The problem

METAPOST is meant for drawing graphics and adding text to them is not really part of the concept. Its a bit like how T<sub>E</sub>X sees images: the dimensions matter, the content doesn't. This means that in METAPOST a blob of text is an abstraction. The native way to create a typeset text picture is:

```
picture p ; p := btex some text etex ;
```

In traditional METAPOST this will create a temporary T<sub>E</sub>X file with the words `some text` wrapped in a box that when typeset is just shipped out. The result is a DVI file that with an auxiliary program will be transformed into a METAPOST picture. That picture itself is made from multiple pictures, because each sequences of characters becomes a picture and kerns become shifts.

There is also a primitive `infont` that takes a text and just converts it into a low level text object but no typesetting is done there: so no ligatures and no kerns are found there. In CONTEXt this operator is redefined to do the right thing.

In both cases, what ends up in the POSTSCRIPT file is references to fonts and characters and the original idea is that DVIPS understands what fonts to embed. Details are communicated via specials (comments) that DVIPS is supposed to intercept and understand. This all happens in an 8 bit (font) universe.

When we moved on to PDF, a converter from METAPost's rather predictable and simple POSTSCRIPT code to PDF was written in `TEX`. The graphic operators became PDF operators and the text was retypeset using the font information and snippets of strings and injected at the right spot. The only complication was that a non circular pen actually produced two path of which one has to be transformed.

At that moment it already had become clear that a more tight integration in CONTEXT would happen and not only would that demand a more sophisticated handling of text, but it would also require more features not present in METAPost, like dealing with CMYK colors, special color spaces, transparency, images, shading, and more. All this was implemented. In the next sections we will only discuss texts.

### 13.3 Using the traditional method

The `btx` approach was not that flexible because what happens is that `btx` triggers the parser to just grabbing everything upto the `etx` and pass that to an external program. It's special scanner mode and because because of that using macros for typesetting texts is a pain. So, instead of using this method in CONTEXT we used `textext`. Before a run the METAPost file was scanned and for each `textext` the argument was copied to a file. The `btx` calls were scanned to and replaced by `textext` calls.

For each processed snippet the dimensions were stored in order to be loaded at the start of the METAPost run. In fact, each text was just a rectangle with certain dimensions. The PDF converter would use the real snippet (by typesetting it).

Of course there had to be some housekeeping in order to make sure that the right snippets were used, because the order of definition (as picture) can be different from them being used. This mechanism evolved into reasonable robust text handling but of course was limited by the fact that the file was scanned for snippets. So, the string had to be string and not assembled one. This disadvantage was compensated by the fact that we could communicate relevant bits of the environment and apply all the usual context trickery in texts in a way that was consistent with the rest of the document.

A later implementation could communicate the text via specials which is more flexible. Although we talk of this method in the past sense it is still used in MKII.

### 13.4 Using the library

When the MPLIB library showed up in LUATEX, the same approach was used but soon we moved on to a different approach. We already used specials to communicate extensions to the backend, using special colors and fake objects as signals. But at that time paths got pre- and postscripts fields and those could be used to really carry information with objects because unlike specials, they were bound to that object. So, all extensions using specials as well as texts were rewritten to use these scripts.

The `texttext` macro changed its behaviour a bit too. Remember that a text effectively was just a rectangle with some transformation applied. However this time the postscript field carried the text and the prescript field some specifics, like the fact that we are dealing with text. Using the script made it possible to carry some more information around, like special color demands.

```
draw texttext("foo") ;
```

Among the prescripts are `tx_index=trial` and `tx_state=trial` (multiple prescripts are prepended) and the postscript is `foo`. In a second run the prescript is `tx_index=trial` and `tx_state=final`. After the first run we analyze all objects, collect the texts (those with a `tx_` variables set) and typeset them. As part of the second run we pass the dimensions of each indexed text snippet. Internally before the first run we ‘reset’ states, then after the first run we ‘analyze’, and after the second run we ‘process’ as part of the conversion of output to PDF.

## 13.5 Using runscript

When the `runscript` feature was introduced in the library we no longer needed to pass the dimensions via subscripted variables. Instead we could just run a LUA snippets and ask for the dimensions of a text with some index. This is conceptually not much different but it saves us creating METAPost code that stored the dimensions, at the cost of potentially a bit more runtime due to the `runscript` calls. But the code definitely looks a bit cleaner this way. Of course we had to keep the dimensions at the LUA end but we already did that because we stored the preprocessed snippets for final usage.

## 13.6 Using a sub T<sub>E</sub>X run

We now come the current (post LUATEX 1.08) solution. For reasons I will mention later a two pass approach is not optimal, but we can live with that, especially because CONTEX with METAFUN (which is what we’re talking about here) is quit efficient. More important is that it’s kind of ugly to do all the not that special work twice. In addition to text we also have outlines, graphics and more mechanisms that needed two passes and all these became one pass features.

A T<sub>E</sub>X run is special in many ways. At some point after starting up T<sub>E</sub>X enters the main loop and begins reading text and expanding macros. Normally you start with a file but soon a macro is seen, and a next level of input is entered, because as part of the expansion more text can be met, files can be opened, other macros be expanded. When a macro expands a token register, another level is entered and the same happens when a LUA call is triggered. Such a call can print back something to T<sub>E</sub>X and that has to be scanned as if it came from a file.

When token lists (and macros) get expanded, some commands result in direct actions, others result in expansion only and processing later as one of more tokens can end up in

the input stack. The internals of the engine operate in miraculous ways. All commands trigger a function call, but some have their own while others share one with a switch statement (in C speak) because they belong to a category of similar actions. Some are expanded directly, some get delayed.

Does it sound complicated? Well, it is. It's even more so when you consider that  $\text{\TeX}$  uses nesting, which means pushing and popping local assignments, knows modes, like horizontal, vertical and math mode, keeps track of interrupts and at the same time triggers typesetting, par building, page construction and flushing to the output file.

It is for this reason plus the fact that users can and will do a lot to influence that behaviour that there is just one main loop and in many aspects global state. There are some exceptions, for instance when the output routine is called, which creates a sort of closure: it interrupts the process and for that reason gets grouping enforced so that it doesn't influence the main run. But even then the main loop does the job.

Starting with version 1.10 LUATEX provides a way to do a local run. There are two ways provided: expanding a token register and calling a LUA function. It took a bit of experimenting to reach an implementation that works out reasonable and many variants were tried. In the appendix we give an example of usage.

The current variant is reasonably robust and does the job but care is needed. First of all, as soon as you start piping something to  $\text{\TeX}$  that gets typeset you'd better in a valid mode. If not, then for instance glyphs can end up in a vertical list and LUATEX will abort. In case you wonder why we don't intercept this: we can't because we don't know the user's intentions. We cannot enforce a mode for instance as this can have side effects, think of expanding `\everypar` or injecting an indentation box. Also, as soon as you start juggling nodes there is no way that  $\text{\TeX}$  can foresee what needs to be copied to discarded. Normally it works out okay but because in LUATEX you can cheat in numerous ways with LUA, you can get into trouble.

So, what has this to do with METAPOST? Well, first of all we could now use a one pass approach. The `texttext` macro calls LUA, which then let  $\text{\TeX}$  do some typesetting, and then gives back the dimensions to METAPOST. The 'analyze' phase is now integrated in the run. For a regular text this works quite well because we just box some text and that's it. However, in the next section we will see where things get complicated.

Let's summarize the one pass approach: the `texttext` macro creates rectangle with the right dimensions and for doing passes the string to LUA using `runcscript`. We store the argument of `texttext` in a variable, then call `runtoks`, which expands the given token list, where we typeset a box with the stored text (that we fetch with a LUA call), and the `runcscript` passes back the three dimensions as fake RGB color to METAPOST which applies a `scantokens` to the result. So, in principle there is no real conceptual difference except that we now analyze in-place instead of between runs. I will not show the code here because in CONTEXt we use a wrapper around `runcscript` so low level examples won't run well.

## 13.7 Some aspects

An important aspect of the text handling is that the whole text can be transformed. Normally this is only some scaling but rotation is also quite valid. In the first approach, the original METAPOST one, we have pictures constructed of snippets and pictures transform well as long as the backend is not too confused, something that can happen when for instance very small or large font scales are used. There were some limitations with respect to the number of fonts and efficient inclusion when for instance randomization was used (I remember cases with thousands of font instances). The PDF backend could handle most cases well, by just using one size and scaling at the PDF level. All the `texttext` approaches use rectangles as stubs which is very efficient and permits all transforms.

How about color? Think of this situation:

```
\startMPcode
  draw texttext("some \color[red]{text}")
    withcolor green ;
\stopMPcode
```

And what about the document color? We suffice by saying that this is all well supported. Of course using transparency, spot colors etc. also needs extensions. These are however not directly related to texts although we need to take it into account when dealing with the inclusion.

```
\startMPcode
  draw texttext("some \color[red]{text}")
    withcolor "blue"
    withtransparency (1,0.5) ;
\stopMPcode
```

What if you have a graphic with many small snippets of which many have the same content? These are by default shared, but if needed you can disable it. This makes sense if you have a case like this:

```
\useMPlibrary[dum]

\startMPcode
  draw texttext("\externalfigure[unknown]") notcached ;
  draw texttext("\externalfigure[unknown]") notcached ;
\stopMPcode
```

Normally each unknown image gets a nice placeholder with some random properties. So, do we want these two to have the same or not? At least you can control it.

When I said that things can get complicated with the one pass approach the previous code snippet is a good example. The dummy figure is generated by METAPOST. So, as

we have one pass, and jump temporarily back to  $\text{\TeX}$ , we have two problems: we reenter the MPLIB instance again in the middle of a run, and we might pipe back something to and/or from  $\text{\TeX}$  nested.

The first problem could be solved by starting a new MPLIB session. This normally is not a problem as both runs are independent of each other. In CONTEXT we can have META-POST runs in many places and some produce some more or less stand alone graphic in the text while other calls produce PDF code in the backend that is used in a different way (for instance in a font). In the first case the result gets nicely wrapped in a box, while in the second case it might directly end up in the page stream. And, as  $\text{\TeX}$  has no knowledge of what is needed, it's here that we can get the complications that can lead to aborting a run when you are careless. But in any case, if you abort, then you can be sure you're doing the wrong thing. So, the second problem can only be solved by careful programming.

When I ran the test suite on the new code, some older modules had to be fixed. They were doing the right thing from the perspective of intermediate runs and therefore independent box handling, putting a text in a box and collecting dimensions, but interwoven they demanded a bit more defensive programming. For instance, the multi-pass approach always made copies snippets while the one pass approach does that only when needed. And that confused some old code in a module, which incidentally is never used today because we have better functionality built-in (the METAFUN `followtext` mechanism).

The two pass approach has special code for cases where a text is not used. Imagine this:

```
picture p ; p := texttext("foo") ;  
  
draw boundingbox p;
```

Here the ‘analyze’ stage will never see the text because we don’t flush  $p$ . However because `texttext` is called it can also make sure we still know the dimensions. In the next case we do use the text but in two different ways. These subtle aspects are dealt with properly and could be made a bit simpler in the single pass approach.

```
picture p ; p := texttext("foo") ;  
  
draw p rotated 90 withcolor red ;  
draw p withcolor green ;
```

## 13.8 One or two runs

So are we better off now? One problem with two passes is that if you use the equation solver you need to make sure that you don’t run into the redundant equation issue. So, you need to manage your variables well. In fact you need to do that anyway because you can call out to METAPOST many times in a run so old variables can interfere anyway. So yes, we’re better off here.

Are we worse off now? The two runs with in between the text processing is very robust. There is no interference of nested runs and no interference of nested local  $\text{\TeX}$  calls. So, maybe we're also bit worse off. You need to anyhow keep this in mind when you write your own low level  $\text{\TeX}$ -METAPOST interaction trickery, but fortunately now many users do that. And if you did write your own plugins, you now need to make them single pass.

The new code is conceptually cleaner but also still not trivial because due to the mentioned complications. It's definitely less code but somehow amputating the old code does hurt a bit. Maybe I should keep it around as reference of how text handling evolved over a few decades.

## 13.9 Appendix

Because the single pass approach made me finally look into a (although somewhat limited) local  $\text{\TeX}$  run, I will show a simple example. For the sake of generality I will use `\directlua`. Say that you need the dimensions of a box while in LUA:

```
\directlua {
    tex.sprint("result 1: <")

    tex.sprint("\setbox0\hbox{one}")
    tex.sprint("\number\wd0")

    tex.sprint("\setbox0\hbox{\directlua{tex.print('first')}}")
    tex.sprint(",")
    tex.sprint("\number\wd0")

    tex.sprint(">")
}

result 1: <1263102,1375500>
```

This looks ok, but only because all printed text is collected and pushed into a new input level once the LUA call is done. So take this then:

```
\directlua {
    tex.sprint("result 2: <")

    tex.sprint("\setbox0\hbox{one}")
    tex.sprint(tex.getbox(0).width)

    tex.sprint("\setbox0\hbox{\directlua{tex.print('first')}}")
    tex.sprint(",")
    tex.sprint(tex.getbox(0).width)

    tex.sprint(">")
```

```
}
```

```
result 2: <1375500,1375500>
```

This time we get the widths of the box known at the moment that we are in LUA, but we haven't typeset the content yet, so we get the wrong dimensions. This however will work okay:

```
\toks0{\setbox0\hbox{one}}
\toks2{\setbox0\hbox{first}}
\directlua {
    tex.forcehmode(true)

    tex.sprint("<")

    tex.runtoks(0)
    tex.sprint(tex.getbox(0).width)

    tex.runtoks(2)
    tex.sprint(",")
    tex.sprint(tex.getbox(0).width)

    tex.sprint(">")
}
```

```
<1263102,1375500>
```

as does this:

```
\toks0{\setbox0\hbox{\directlua{tex.sprint(MyGlobalText)}}}
\directlua {
    tex.forcehmode(true)

    tex.sprint("result 3: <")

    MyGlobalText = "one"
    tex.runtoks(0)
    tex.sprint(tex.getbox(0).width)

    MyGlobalText = "first"
    tex.runtoks(0)
    tex.sprint(",")
    tex.sprint(tex.getbox(0).width)

    tex.sprint(">")
}
```

result 3: <1263102,1375500>

Here is a variant that uses functions:

```
\directlua {
    tex.forcehmode(true)

    tex.sprint("result 4: <")

    tex.runtoks(function()
        tex.sprint("\\setbox0\\hbox{one}")
    end)
    tex.sprint(tex.getbox(0).width)

    tex.runtoks(function()
        tex.sprint("\\setbox0\\hbox{\\directlua{tex.print('first')}}}")
    end)
    tex.sprint(",")
    tex.sprint(tex.getbox(0).width)

    tex.sprint(">")
}
```

result 4: <1263102,1375500>

The `forcemode` is needed when you do this in vertical mode. Otherwise the run aborts. Of course you can also force horizontal mode before the call. I'm sure that users will be surprised by side effects when they really use this feature but that is to be expected: you really need to be aware of the subtle interference of input levels and mix of input media (files, token lists, macros or LUA) as well as the fact that TeX often looks one token ahead, and often, when forced to typeset something, also can trigger builders. You're warned.

## 155 Amputating code

## 14 Getting there, version 1.10

When we decided to turn experiments with a LUA extensions to PDF<sub>T</sub>E<sub>X</sub> into developing LUAT<sub>E</sub>X as alternative engine we had, in addition to opening up some of T<sub>E</sub>X's internals, some extensions in mind. Around version 1.00 most was already achieved and with version 1.10 we're pretty close to where we want to be. The question is, when are we ready? In order to answer that I will look at four aspects:

- objectives
- functionality
- performance
- stability

The main *objective* was to open up T<sub>E</sub>X in a way that permit extensions without the need to patch the engine. Although it might suit us, we don't want to change too much the internals, first of all because T<sub>E</sub>X is T<sub>E</sub>X, the documented program with a large legacy.<sup>18</sup> Discussions about how to extend T<sub>E</sub>X are not easy and seldom lead to an agreement so better is to provide a way to do what you like without bothering other users and/or interfering with macro packages. I think that this objective is met quite well now. Other objectives, like embedding basic graphic capabilities using METAPOST have already been met long ago. There is more control over the backend and modern fonts can be dealt with.

The *functionality* in terms of primitives has been extended but within reasonable bounds: we only added things that make coding a bit more natural but we realize that this is very subjective. So, here again we can say that we met our goals. A lot can be achieved via LUA code and users and developers need to get accustomed to that if they want to move on with LUAT<sub>E</sub>X. We will not introduce features that get added to or are part of other engines.

We wanted to keeping *performance* acceptable. The core T<sub>E</sub>X engine is already pretty fast and it's often the implementation of macros (in macro packages) that creates a performance hit. Going UTF has a price as do modern fonts. At the time of this writing processing the 270 page LUAT<sub>E</sub>X manual takes about 12 seconds (one run), which boils down to over 27 pages per second.

	runtime	overhead
LUAT <sub>E</sub> X	12.0	+0.6
LUAJITT <sub>E</sub> X	9.7	+0.5

Is this fast or slow? One can do tests with specific improvements (using new primitives) but in practice it's very hard to improve performance significantly. This is because a test with millions of calls that show a .05 second improvement disappears when one

---

<sup>18</sup> This is reflected in the keywords that exposed mechanisms use: they reflect internal variable names and constants and as a consequence there is inconsistency there.

only has a few thousand calls. Many small improvements can add up, but less than one thinks, especially when macros are already quite optimal. Also this runtime includes time normally used for running additional programs (e.g. for getting bibliographies right).

It must be said that performance is not completely under our control. For instance, we have patched the LUAJIT hash function because it favours URL's and therefore favours hashing the middle of the string which is bad for our use as we are more interested in the (often unique) start of strings. We also compress the format which speeds up loading but not on the native windows 64 bit binary. At the time this writing the extra overhead is 2 seconds due to some suboptimal gzip handling; the cross compiled 64 bit mingw binaries that I use don't suffer from this. When I was testing the 32 bit binaries on the machine of a colleague, I was surprised to measure the following differences on a complex document with hundreds of XML files, many images and a lot of manipulations.

### 1.08 with LUA 5.2    1.09 with LUA 5.3

<b>LUATEX</b>	21.5	15.2
<b>LUAJITTEX</b>	10.7	10.3

Now, these are just rough numbers but they demonstrate that the gap between LUATEX and LUAJITTEX is becoming less which is good because at this moment it looks like LUAJIT will not catch up with LUA 5.3 so at some point we might drop it. It will be interesting to see what LUA 5.4 will bring as it offers an alternative garbage collector. And imagine that the regular LUA virtual machine gets more optimized.

You also have to take into account that having a browser open in the background of a TEX run has way more impact than a few tenths of a second in LUATEX performance. The same is true for memory usage: why bother about LUATEX taking tens of megabytes for fonts while a few tabs in a browser can bump memory consumption to gigabytes of memory usage. Also, using a large TEX tree (say the whole of TEXLIVE) can have a bit of a performance hit! Or what about inefficient callbacks, using inefficient LUA code of badly designed solutions? What we could gain here we loose there, so I think we can safely say that the current implementation of LUATEX is as good as you can (and will) get. Why should we introduce obscure optimizations where on workstations TEX is just one of the many processes? Why should we bother too much to speed up on servers that have start-up or job management overhead or are connected to relatively slow remote file system? Why squeeze out a few more milliseconds when badly written macros or styles can have an way more impact on performance? So, for now we're satisfied with performance. Just for the record, the ratio between CONTEXT MKII running other engines and LUATEX with MKIV for the next snippet of code:

```
\dorecurse{250}{\input tufte\par}
```

is 2.8 seconds for XETEX, 1.5 seconds for LUATEX, 1.2 seconds for LUAJITTEX, and 0.9 seconds for PDFTEX. Of course this is not really a practical test but it demonstrates the

baseline performance on just text. The 64 bit version of PDFT<sub>E</sub>X is actually quite a bit slower on my machine. Anyway, LUAT<sub>E</sub>X (1.09) with MKIV is doing okey here.

That brings us to *stability*. In order to achieve that we will not introduce many more extensions. That way users get accustomed to what is there (read: there is no need to search for what else is possible). Also, it makes that existing functionality can become bug free because no new features can interfere. So, at some point we have to decide that this is it. If we can do what we want now, there are no strong arguments for more. in that perspective version 1.10 can be considered very close to what we want to achieve.

Of course development will continue. For instance, the PDF inclusion code will be replaced by more lightweight and independent code. Names of functions and symbolic constants might be normalized (as mentioned, currently they are often related to or derived from internals). More documentation will be added. We will quite probably keep up with LUA versions. Also the FFI interface will become more stable. And for sure bugs will be fixed. We might add a few more options to control behaviour of for instance of math rendering. Some tricky internals (like alignments) might get better attribute support if possible. But currently we think that most fundamental issues have been dealt with.

