Still

going on

Hans Hagen

Content

Introduction

This document is a follow up on ‘mk’, ‘hybrid’ and ‘about’. The first one is written when LuaTEX evolved to version 0.50, the second one stops around version 0.70 while the last one goes on after that. The title of this document refers to the fact that we’re still working towards version 1.00. In the meantime we have done a lot of testing and the engine has become quite stable. The LuajitTEX variant has become part of the standard distribution and we’re working on a library support framework. At the same time we keep experimenting and here we will report on some issues that we run into as well as discuss the way ConTEXt MkIV keeps adapting itself.

Hans Hagen
Hasselt NL
2013–2016

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

1 Math new style: are we better off?

1.1 Introduction

In this article I will summarize the state of upgrading math support in ConTEXt per mid 2013 in the perspective of demand, usability, font development and LuaTEX. There will be some examples, but don't consider this a manual: there are enough articles in the `mkiv`, `hybrid` and `about` series about specific topics; after all, we started with this many years ago. Where possible I will draw some conclusions with respect to the engine. Some comments might sound like criticism, but you should keep in mind that I wouldn't spend so much time on TEX if I would not like it that much. It's just that the environment wherein TEX is and can be used is not always as perfect as one likes it to be, i.e. bad habits and decisions once made can be pretty persistent and haunt us forever. I'm not referring to TEX the language and program here, but more to its use in scientific publishing: in an early stage standards were set and habits were nurtured which meant that to some extent the coding resembles the early days of computing and the look and feel got frozen in time, in spite of developments in coding and evolving typographic needs. I think that the community has missed some opportunities to influence and improve matters which means that we're stuck with suboptimal situations and, although they are an improvement, Unicode math and OpenType math have their flaws.

This is not a manual. Some aspects will be explained with examples, others are just mentioned. I've written down enough details in the documents that describe the history of LuaTEX and MkIV and dedicated manuals and repeating myself makes not much sense. Even if you think that I talk nonsense, some of the examples might set you thinking. This article was written for the tug 2013 conference in Japan. Many thanks to Barbara Beeton for proofreading and providing feedback.

1.2 Some basic questions

Is there still a need for a program like TEX? Those who typeset math will argue that there is. After all, one of the reasons why TEX showed up is typesetting math. In this perspective we should ask ourselves a few questions:

- Is TEX still the most adequate tool?
- Does it make sense to invest in better machinery?

- Have we learned from the past and improved matters?
- What drives development and choices to be made?

The first question is not that easy to answer, unless you see proof in the fact that TEX is still popular for typesetting a wide range of complex content (with critical editions being among the most complex). Indeed the program still attracts new users and developers. But we need to be realistic. First of all, there is some bias involved: if you have used a tool for many years, it becomes the one and only and best tool. But that doesn't necessarily make it the best tool for everyone.

In this internet world finding a few thousand fellow users gives the impression that there is a wide audience but there can be of course thousandfold more users of other systems that don't fall into your scope. This is fine: I always wonder why there is not more diversity; for instance, we have only a few operating systems to choose from, and in communities around computer languages there is a tendency to evangelize (sometimes quite extreme). We should also take into account that a small audience can have a large impact so size doesn't matter much.

As TEX is still popular among mathematicians, we can assume that it hasn't lost its charm yet and often it is their only option. We have a somewhat curious situation that scientific publishers still want to receive TEX documents —a demand that is not much different from organizations demanding MS Word documents— but at the same time don't care too much about TEX at all. Their involvement in user groups has started degrading long ago, compared to their profits; they don't invest in development; they are mostly profit driven, i.e. those who submit their articles don't even own their sources any more, etc.

On the other hand, we have users who make their own books (self-publishing) and who go, certainly in coding and style, beyond what publishers do: they want to use all kinds of fonts (and mixtures), color, nicely integrated graphics, more interesting layouts, experiment with alternative presentations. But especially for documents that contain math that also brings a price: you have to spend more time on thinking about presenting the content and coding of the source. This all means that if we look at the user side, alternative input is an option, especially if they want to publish on different media. I know that there are ConTEXt users who make documents (or articles) with ConTEXt, using whatever coding suits best, and do some conversion when it has to be submitted to a journal. Personally I think that the lack of interest of (commercial) publishers, and their rather minimal role in development, no longer qualifies them to come up with requirements for the input, if only because in the end all gets redone anyway (in Far Far Away).

It means that, as long as TEX is feasible, we are relatively free to move on and experiment with alternative input. Therefore the other two questions become relevant. The TEX engines are adapted to new font technology and a couple of math fonts are being developed (funded by the user groups). Although the TEX community didn't take the lead in math font technology we are catching up. At the same time we're investing much time in new tools, but given the fact that much math is produced for publishers it doesn't get much exposure. Scientific publishing is quite traditional and like other publishing lags behind and eventually will disappear in its current form. It could happen that one morning we find out that all that 'publishers want it this or that way' gets replaced by ways of publishing where authors do all themselves. A publisher (or his supplier) can keep using a 20-year old TEX ecosystem without problems and no one will notice, but users can go on and come up with more modern designs and output formats and in that perspective the availability of modern engines and fonts is good. I've said it before: for ConTEXt user demand drives development.

In the next sections I will focus on different aspects of math and how we went from MkII to MkIV. I will also discuss some (pending) issues. For each aspect I will try to answer the third question: did matters improve and if not, and how do we cope with it (in ConTEXt).

1.3 The math script

All math starts with symbols and/or characters that have some symbolic meaning and in TEX speak this can be entered in a rather natural way:

```
$ y = 2x + b $
```

In order to let TEX know it's math (the equivalent of) two dollar signs are used as triggers. The output of this input is: $y = 2x + b$. But not all is that simple, for instance if we want to square the x, we need to use a superscript signal:

```
$ y = x^2 + ax + b $
```

The `^` symbol results in a smaller `2` raised after the `x` as in $y = x^2 + ax + b$. Ok, this `^` and its cousin `_` are well known conventions so we stick to this kind of input.

A next level of complexity introduces special commands, for instance a command that will wrap its argument in a square root symbol: $y = \sqrt{x^2 + ax + b}$.

```
$ y = \sqrt { x^2 + ax + b } $
```

It is no big deal to avoid the backslash and use this kind of coding:

```
\asciimath { y = sqrt ( x^2 + ax + b ) }
```

In fact, we have been supporting scientific calculator input for over a decade in projects where relatively simple math had to be typeset. In one of our longest-running math related projects the input went from TEX, to content MathML to OpenMath and via presentation MathML ended up as a combination of some kind of encoding that web browsers can deal with. This brings us to reality: it's web technology that drives (and will drive math) coding. Unfortunately content driven coding (like content MathML) does not seem to be the winner here, even if it renders easier and is more robust.

Later I will discuss fences, like parentheses. Take this dummy formula:

```
$ (x + 1) / a = (x - 1) / b $
```

In a sequential (inline) rendering this will come out okay. A more display mode friendly variant can be:

```
$ \frac{x + 1}{a} = \frac{x - 1}{b} $
```

which in pure TEX would have been:

```
$ {x + 1} \over {a} = {x - 1} \over {b} $
```

The main difference between these two ways of coding is that in the second (plain) variant the parser doesn't know in advance what it is dealing with. There are a few cases in TEX where this kind of parsing is needed and it complicates not only the parser but also is not too handy at the macro level. This is why the `\frac` macro is often used instead. In LuaTEX we didn't dare to get rid of `\over` and friends, even if we're sure they are not used that often by users.

In inline or in more complex display math, the use of fences is quite normal.

```
$ ( \frac{x + 1}{a}  + 1 )^2  = \frac{x - 1}{b} $
```

Here we have a problem. The parentheses don't come out well.

$(\frac{x+1}{a} + 1)^2 = \frac{x-1}{b}$

We have to do this:

```
$ \left( \frac{x + 1}{a}  + 1 \right)^2  = \frac{x - 1}{b} $
```

in order to get:

$\left(\frac{x+1}{a} + 1\right)^2 = \frac{x-1}{b}$

Doing that `\left`-`\right` trick automatically is hard, although in MathML, where we have to interpret operators anyway it is somewhat easier. The biggest issue here is that these two directives need to be paired. In ε-TEX a `\middle` primitive was added to provide a way to have bars adapt their height to the surroundings. Interesting is that where at the character level a `(` has a math property `open` and `)` has `close`. The bar, as we will see later, can also act as separator but this property does not exist. Because properties (classes in TEX speak) determine spacing we have a problem here. So far we didn't extend the repertoire of properties in LuaTEX to suit our needs (although in ConTEXt we do have more properties).

If you are a TEX user typesetting math, you can without doubt come up with more cases of source coding that have the potential of introducing complexities. But you will also have noticed that in most cases TEX does a pretty good job on rendering math out of the box. And macro packages can provide additional constructs that help to hide the details of fine tuning (because there is a lot that *can* be fine tuned).

In TEX there are a couple of special cases that we can reconsider in the perspective of (for instance) faster machines. Normally a macro cannot have a `\par` in one of its arguments. By defining them as `\long` this limitation goes away. This default limitation was handy in times when a run was relatively slow and grabbing a whole document source as argument due to a missing brace had a price. Nowadays this is no real issue which is why in LuaTEX we can disable `\long` which indeed we do in ConTEXt. On the agenda is to also permit `\par` in a math formula, as currently TEX complains loudly. Permitting a bit more spacy formula definitions (by using empty lines) would be a good thing.

Another catch is that in traditional TEX math characters cannot be used outside math. That restriction has been lifted. Of course users need to be aware of the fact that a mix of math and text symbols can be visually incompatible.

In the examples we used `^` and `_` and in math mode these have special meanings. Traditionally in text mode they trigger an error message. In ConTEXt MkIV we have made these characters regular characters but in math mode they still behave as expected.[1] In a similar fashion the `&` is an ampersand and when you enable `\asciimode` the dollar and percent signs also become regular.[2] In LuaTEX we have introduced primitives for all characters (or more precisely: catcodes) that TEX uses for special purposes like opening and closing math mode, scripts, table alignment, etc.

In projects that involve xml we use MathML. In TEX many characters can be inserted using commands that are tuned for some purpose. The same character can be associated with several commands. In MathML entities and Unicode characters are used instead of commands. Interesting is that whenever we get math coded that way, there is a good chance that the coding is inconsistent. Of course there are ways in MathML to make sure that a character gets interpreted in the right way. For instance, the `mfenced` element drives the process of (matching) parenthesis, brackets, etc. and a renderer can use this property to make sure these symbols stretch vertically when needed. However, using `mo` in an `mrow` for a fence is also an option, but that demands some more (fuzzy) analysis. I will not go into details here, but some of the more obscure options and flags in ConTEXt relate to overcoming issues with such cases.

I have no experience with how MS Word handles math input, apart from seeing some demos. But I know that there is some input parsing involved that is a mixture between TEX and analysis. Just as word processing has driven math font technology it might be that at some point users expect more clever processing of input. To a large extent TEX users already expect that. Where till now TEX could inspire the way word processers do math, word processors can inspire TEXies way of inputting text.

So, we have MathML, which, in spite of being structured, is still providing users a lot of freedom. Then there are word processors, where mouse clicks and interpretation does the job. And of course we have TEX, with its familiar backslashes. Let us consider math, when seen in print, as a script to express the math language. And indeed, in OpenType, math is one of the official scripts although one where a rather specific kind of machinery is needed in order to get output.

---

[1] In an intermediate version `\nonknuthmode` and `\donknuthmode` controlled this.

[2] Double percent signs act as comments then which is comparable to comments in some programming languages.

I could show more complex math formulas but no matter what notation is used, coding will always be somewhat cumbersome and handywork. Math formula coding and typesetting remains a craft in itself and TEX notation will keep its place for a while. So, with that aspect settled we can continue to discuss rendering.

1.4 Alphabets

I have written about math alphabets before so let's keep it simple here. I think we can safely say that most math support mechanisms in macro packages are inspired by plain TEX. In traditional TEX we have fonts with a limited number of glyphs and an eight-bit engine, so in order to get the thousands of possible characters mapped onto glyphs the right one has to be picked from some font. In addition to characters that you find in Unicode, there are also variants, additional sizes and bits and pieces that are used in constructing large characters, so in practice a math font is quite large. But it is unlikely that we will ever run into a situation where fonts pose limits.

The easiest way is of course a direct mapping: an 'a' entered in math mode becomes an 'a' simply because the current font at that time has an italic shape in the slot referenced by the character. If we want a bold shape instead, we can switch to another font and still input an 'a'. The 16 families available are normally enough for the alphabets that we need. Because symbols can be collected in any font, they are normally accessed by name, like `\oplus` or $\oplus$.

In Unicode math the math italic 'a' has slot `U+1D44E` and directly entering this character in a Unicode aware TEX engine also has to give that 'a'. In fact, it is the only official way to get that character and the fact that we can enter the traditional ascii characters and get an italic shape is a side effect of the macro package, for instance the way it defines math fonts and families.[3]

Before we move on, let's stress a limitation in Unicode with respect to math alphabets. It has always been a principle of Unicode committees to never duplicate entries. So, thanks to the availability of some characters in traditional (font) encodings, we ended up with some symbols that are used for

---

[3] Our experience is that even when for instance MathML permits coding of math in xml, copy editors have no problem with abusing regular italic font switches to simulate math. This can result is a weird mix of math rendering.

math in the older regions of Unicode. As a consequence some alphabets have gaps. The only real reason I can come up with for accepting these gaps is that old documents using these symbols would be not compatible with gapfull Unicode math but I could argue that a document that uses those old codepoints uses commands (and needs some special fonts) to get the other symbols anyway, so it's unlikely to be a real math document. On the other hand, once we start using Unicode math we could benefit from gapless alphabets simply because otherwise each application would have to deal with the exceptions. One can come up with arguments like "just use this or that library" but that assumes persistence, and also forces everyone to use the same approach. In fact, if we hide behind a library we could as well have hidden the vectors (alphabets) as well. But as they are exposed, the gaps stand out as an anomaly.[4] Let's illustrate this with an example. Say that we load the TeXGyre Pagella math font and call up a few characters:

```
\definefont[mathdemo][file:texgyrepagellamath*mathematics]
\mathdemo \char"0211C \char"1D507 \char"1D515
```

The Unicode fraktur math alphabet is continuous but the 'MATHEMATICAL FRAKTUR CAPITAL R' is missing as we already have the `BLACK-LETTER CAPITAL R` instead. So, this is why we only see two characters show up. It means that in the input we cannot have a `U+1D515`.

ℜ𝔇

Of course we can cheat and fill in the gap:

```
\definefontfeature
  [mymathematics]
  [mathematics]
  [mathgaps=yes]
```

This feature will help us cheat:

```
\definefont[mathdemo][file:texgyrepagellamath*mymathematics]
\mathdemo \char"0211C \char"1D507 \char"1D515
```

This time we can use the character. I wonder what would happen if the TeX community would simply state that slot `U+1D515` is valid. I bet that math

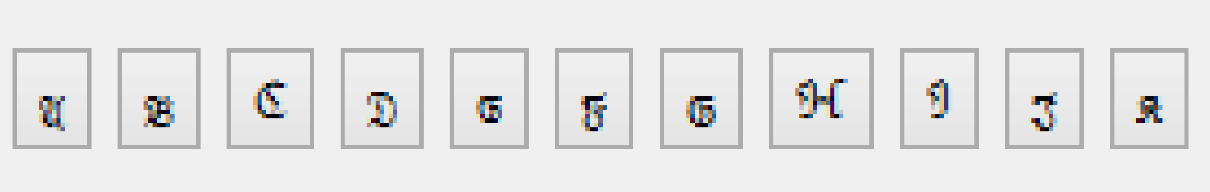
---

[4] One good reason for not having the gaps is that when users cut and paste there is no way to know if `U+210E` is used as Planck constant or variable of some sort, i.e. the not existing `0x1D455`. There is no official way to tag it as something math, and even then, as it has no code point it so has lost it's meaning, contrary to a copied *i*.

related applications would support it, as they also support more obscure properties of TEX input encoding.

ℜ𝔇ℜ

If you still wonder why I bother about this, here is a practical example. The SciTE editor that I use is rather flexible and permits me to implement advanced lexers for ConTEXt (and especially hybrid usage). It also permits to hook in Lua code and that way the editor can (within bounds) be extended. As an example I've added some button bars that permit entering math alphabets. Of course the appearance depends on the font used but operating systems tend to consult multiple fonts when the core font of the editor doesn't provide a glyph.



Here I show a small portion of the stripe with buttons that inject the shown characters. What happens in the rendering is that first the used font is consulted and that one has a couple of 'BLACK LETTER CAPITAL's so they get used. The others are 'MATHEMATICAL FRAKTUR CAPITAL's and since the font is not a math font the renderer takes them from (in this case) Cambria Math, which is why they look so different, especially in proportion. Of course we could start out with Cambria but it has no monospace (which I want for editing) and is a less complete text font, so we have a chicken–egg problem here. It is one reason why as part of the math font project we extend the Dejavu Sans Mono with proper (consistent) math symbols. Anyhow, it illustrates why gaps are kind of evil from the application point of view.

gap	**char**	**meant**	**unicode**	**used**
`U+1D455`	ℎ	MATHEMATICAL ITALIC SMALL H	`U+0210E`	PLANCK CONSTAN
`U+1D49D`	ℬ	MATHEMATICAL SCRIPT CAPITAL B	`U+0212C`	SCRIPT CAPITAL B
`U+1D4A0`	ℰ	MATHEMATICAL SCRIPT CAPITAL E	`U+02130`	SCRIPT CAPITAL E
`U+1D4A1`	ℱ	MATHEMATICAL SCRIPT CAPITAL F	`U+02131`	SCRIPT CAPITAL F
`U+1D4A3`	ℋ	MATHEMATICAL SCRIPT CAPITAL H	`U+0210B`	SCRIPT CAPITAL H
`U+1D4A4`	ℐ	MATHEMATICAL SCRIPT CAPITAL I	`U+02110`	SCRIPT CAPITAL I
`U+1D4A7`	ℒ	MATHEMATICAL SCRIPT CAPITAL L	`U+02112`	SCRIPT CAPITAL L
`U+1D4A8`	ℳ	MATHEMATICAL SCRIPT CAPITAL M	`U+02133`	SCRIPT CAPITAL M
`U+1D4AD`	ℛ	MATHEMATICAL SCRIPT CAPITAL R	`U+0211B`	SCRIPT CAPITAL R

U+1D4BA	ℯ	MATHEMATICAL SCRIPT SMALL E	U+0212F	SCRIPT SMALL E
U+1D4BC	ℊ	MATHEMATICAL SCRIPT SMALL G	U+0210A	SCRIPT SMALL G
U+1D4C4	ℴ	MATHEMATICAL SCRIPT SMALL O	U+02134	SCRIPT SMALL O
U+1D506	ℭ	MATHEMATICAL FRAKTUR CAPITAL C	U+0212D	BLACK-LETTER CA
U+1D50B	ℌ	MATHEMATICAL FRAKTUR CAPITAL H	U+0210C	BLACK-LETTER CA
U+1D50C	ℑ	MATHEMATICAL FRAKTUR CAPITAL I	U+02111	BLACK-LETTER CA
U+1D515	ℜ	MATHEMATICAL FRAKTUR CAPITAL R	U+0211C	BLACK-LETTER CA
U+1D51D	ℨ	MATHEMATICAL FRAKTUR CAPITAL Z	U+02128	BLACK-LETTER CA
U+1D53A	ℂ	MATHEMATICAL DOUBLE-STRUCK CAPITAL C	U+02102	DOUBLE-STRUCK
U+1D53F	ℍ	MATHEMATICAL DOUBLE-STRUCK CAPITAL H	U+0210D	DOUBLE-STRUCK
U+1D545	ℕ	MATHEMATICAL DOUBLE-STRUCK CAPITAL N	U+02115	DOUBLE-STRUCK
U+1D547	ℙ	MATHEMATICAL DOUBLE-STRUCK CAPITAL P	U+02119	DOUBLE-STRUCK
U+1D548	ℚ	MATHEMATICAL DOUBLE-STRUCK CAPITAL Q	U+0211A	DOUBLE-STRUCK
U+1D549	ℝ	MATHEMATICAL DOUBLE-STRUCK CAPITAL R	U+0211D	DOUBLE-STRUCK
U+1D551	ℤ	MATHEMATICAL DOUBLE-STRUCK CAPITAL Z	U+02124	DOUBLE-STRUCK

Barbara Beeton told me that, although it took some convincing arguments in the discussions about math in Unicode, we have at least one hole less than to be expected: slot `U+1D4C1` has not been seen as already covered by `U+02113`. So is there really this distinction between a `MATHEMATICAL SCRIPT SMALL L` and `SCRIPT SMALL L` (usually `\ell` in macro packages? Indeed there is, although at the time of this writing interestingly Latin Modern fonts lacked the mathematical one (which in ConTeXt math mode normally results in an upright drop–in). Such details become important when math is edited by someone not familiar with the distinction between a variable (or whatever) represented by a script shape and the length operator. There seems not to be agreement by font designers about the shapes being upright or italic, so some confusion will remain, although this does not matter as long as within the font they differ.

font	**U+1D4C1**	**U+02113**
latin modern		ℓ
stix/xits	𝓁	ℓ
bonum	𝓁	ℓ
termes	𝓁	ℓ
pagella	𝓁	ℓ
lucida	𝓁	ℓ

As math uses greek and because greek was already present in Unicode when math was recognized as script and got its entries, you can imagine that there are some issues there too, but let us move on to using alphabets.

In addition to a one–to–one mapping from a font slot onto a glyph, you can assign properties to characters that map them onto a slot in some family (which itself relates to a font). This means that in a traditional approach you can choose among two methods:

- You define several fonts (or instances of the same font) where the positions of regular characters point to the relevant shape. So, when an italic family is active the related font maps character `U+61` as well as `U+1D44E` to the same italic shape ‘$x1D44E$’. A switch from italic to bold italic is then a switch in family and in that family the `U+61` as well as `U+1D482` become bold italic ‘$x1D482$’.
- You define just one font. The alphabet (uppercase, lowercase and sometimes digits and a few symbols) gets codes that point to the right shape. When we switch from italic to bold italic, these codes get reassigned.

The first method has some additional overhead in defining fonts (you can use copies but need to make sure that the regular ascii slots are overloaded) but the switch from italic to bold italic is fast, while in the second variant there is less overhead in fonts but reassigning the codes with a style switch has some overhead (although in practice this overhead is can be neglected because not that many alphabet switches take place). In fact, many TeX users will probably stick to traditional approaches where verbose names are used and these can directly point to the right shape.

In ConTeXt, when we started with MkIV, we immediately decided to follow another approach. We only have one family and we assume Unicode math input. Ok, we do have a few more families, but these relate to a full bold math switch and right–to–left math. We cannot expect users to enter Unicode math, if only because support in editors is not that advanced, so we need to support the ascii input method as well.

We have one family and don’t redefine character codes, but set properties instead. We don’t switch fonts, but properties. These properties (often a combination) translates into the remapping of a specific character in the input onto a Unicode math code point that then directly maps onto a shape. This approach is quite clean and efficient at the TeX end but carries quite a lot of overhead at the Lua end. So far users never complained about it, maybe because ConTeXt math support is rather optimized. Also, dealing with characters is only part of math typesetting and we have subsystems that use far more processing power.

Because math characters are organized in classes, we need to set them up. Because for several reasons we collect character properties in a database we also define these character properties in Lua. This means that the `math-*` files are relatively small. So we have much less code at the TeX end, but quite a lot at the Lua end. This assumes a well managed Lua subsystem because as soon as users start plugging in their code, we have to make sure that the core system still functions well. The amount of code involved in virtual math fonts is also relatively large but most of that is becoming sort of obsolete.

Relatively new in ConTeXt is the possibility in some mathematical constructs to configure the math style (text, script, etc.) and in some cases math classes can be influenced. Control over styles is somewhat more convenient in LuaTeX, because we can consult the current style in some cases. I expect more of this kind of control in ConTeXt, although most users probably never need it. These kinds of features are meant for users like Aditya Mahajan, who likes to explore such features and also takes advantage of the freedom to experiment with the look and feel of math.

The font code that relates to math is not the easiest to understand but this is because it has to deal with bold as well as bidirectional math in efficient ways. Because in ConTeXt we have additional sizes (`x`, `xx`, `a`, `b`, `c`, `d`, ...) we also have some delayed additional defining going on. This all might sound slower to set up but in the end we win some back by the fact that we have fewer fonts to load. The price that a ConTeXt user pays in terms of runtime is more influenced by the by now large sequence of math list manipulators than by loading a font.

An unfortunate shortcoming of Unicode math is that some alphabets have gaps. This is because characters can only end up once in the standard. Given the number of weird characters showing up in recent versions, I think this condition is somewhat over the top. It forces applications that deal with Unicode math to implement exceptions over and over again. In ConTeXt we assume no gaps and compensate for that.

There are several ways that characters can become glyphs. An ‘a’ can become an italic, bold, bold italic but also end up sans serif or monospace. Because there are several artistic interpretations possible, some fonts provide a so-called alternate. In the case of for instance greek we can also distinguish upright or slanted (italic). A less well known transformation is variants driven by Unicode modified directives. If we forget about bidirectional math and full bold (heavy) math we can (currently) identify 6 axes:

axis	**use**	**choices**

1	type	digits, lowercase & uppercase latin & greek, symbols
2	alphabet	regular, sans serif, monospace, blackboard, fraktur, script
3	style	upright, italic, bold, bolditalic
4	variant	alternative rendering provided by font
5	shape	unchanged, upright, italic
6	Unicode	alternative rendering driven by Unicode modifier

Apart from the last one, this is not new, but it is somewhat easier to support this consistently. It's one of the areas where Unicode shines, although the gaps in vectors are a bad thing. One thing that I decided early in the MkIV math development is that all should fit into the same model: it makes no sense to cripple a whole system because of a few exceptions.

Users expect their digits to be rendered upright and letters to be rendered with italic shapes, but use regular ascii input. This means that we need to relocate the letters to the relevant alphabet in Unicode. In ConTEXt this happens as part of several analysis steps that more or less are the same as the axis mentioned. In addition there is collapsing, remapping, italic correction, boldening, checking, intercepting of special input, and more going on. Currently there are (depending on what gets enabled) some 10 to 15 manipulation passes over the list and there will be more.

So how does the situation compare to the old one? I think we can safely say that we're better off now and that LuaTEX behaves quite okay. There is not much that can be improved, apart from more complete fonts (especially bold). A nice bonus of LuaTEX is that math characters can be used in text mode as well (given that the current font provides them).

It will be clear that by following this route we moved far away from the MkII approach and the dependency on Lua has become rather large in this case. The benefit is that we have rather clean code with hardly any exceptions. It came at the price of lots of experiments and (re)coding but I think it pays off for users.

1.5 Bold

Bold is sort of special. There are bold symbols and some bold alphabets and that *is* basically what bold math is: just a different rendering. In a proper OpenType math fonts these bold characters are covered.

Section titles or captions are often typeset bolder and when they contain math all of it needs to be bolder too. So, a regular italic shape becomes a bold italic shape but a bold shape becomes heavy. This means that we need a full blown bold font for that purpose. And although some are on the agenda of the font team, often we need to fake it. This is seldom an issue as (at least in the documents that I deal with) section titles are not that loaded with math.

A proper implementation of such a mechanism involves two aspects: first there needs to be a complete bold math font with heavy bold included, and second the macro package must switch to bold math in a bold context. When no real bold font is available, some automatic mapping can take place, but that might give interpretation issues if bold is used in a formula. For the average highschool math that we render this is not an issue. Currently there are no full bold math fonts that have enough coverage. (The Xits font, derived from Stix, has a bold companion that does provide for instance bold radicals but lacks many bolder alphabets and symbols.)

```
\startimath
       \sqrt{x^2\over 4x}  \qquad
  {\bf \sqrt{x^2\over 4x}} \qquad
  {\mb \sqrt{x^2\over 4x}} \qquad
       \sqrt{x^2 + 4x}     \qquad
  {\bf \sqrt{x^2 + 4x}}    \qquad
  {\mb \sqrt{x^2 + 4x}}
\stopimath
```

This gives:

$$\sqrt{\frac{x^2}{4x}} \qquad \sqrt{\frac{\mathbf{x^2}}{\mathbf{4x}}} \qquad \sqrt{\frac{\boldsymbol{x}^{\mathbf{2}}}{\mathbf{4}\boldsymbol{x}}} \qquad \sqrt{x^2+4x} \qquad \sqrt{\mathbf{x^2+4x}} \qquad \sqrt{\boldsymbol{x}^{\mathbf{2}}+\mathbf{4}\boldsymbol{x}}$$

Here it is always a bit of a guess if bold extensibles are (already) supported so it’s dangerous to go wild with full bold/heavy combinations unless you check carefully what results you get. Another aspect you need to be aware of is that there is an extensive fallback mechanism present. When possible a proper alphabet will be used, but when one is not present there is a fallback on another. This ensures that we get at least something.

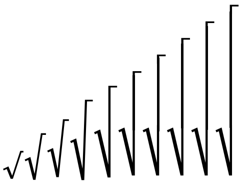
There is not much that an engine can do about it, apart from providing enough families to implement it. In a Type1 universe indeed we need lots of families already so the traditional 16-family pool is drained soon. In LuaTEX we can have 256 families which means that additional Type1 bases family sets are no issue any longer. But as in MkIV we no longer follow that route, bold math can be set up relatively easy, given that we have a bold font. If we don’t have such a font, we have an intermediate mode where a bold font is simulated. Keep in mind that this always will need checking, at least as long as don't have complete enough bold fonts with heavy bold included.

1.6 Radicals

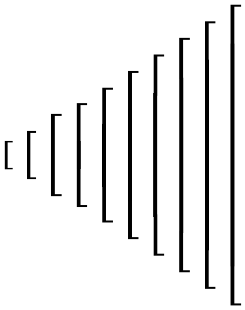
In most cases a TEX user is not that aware of what happens in order to get a nicely wrapped up root on paper. In traditional TEX this is an interplay between rather special font properties and macros. In LuaTEX it has become a bit more simple because we introduced a primitive for it. Also, in OpenType fonts, the radical is provided in a somewhat more convenient way. In an OpenType math font there are some variables that control the rendering:

`RadicalExtraAscender`
`RadicalRuleThickness`
`RadicalVerticalGap`
`RadicalDisplayStyleVerticalGap`

The engine will use these to construct the symbol. The root symbols can grow in two dimensions: the left bit grows vertically but due to the fact that there is a slope involved it happens in steps using different symbols.

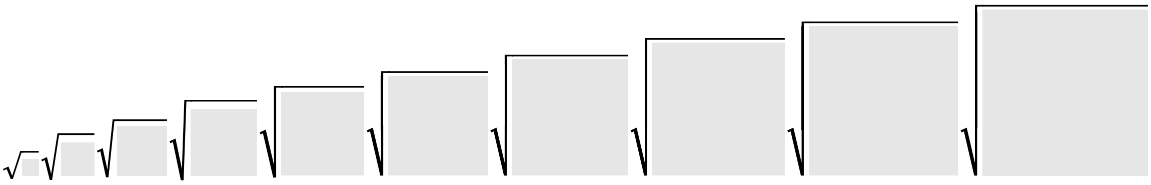


Compare this to for instance how a bracket grows:



The bracket is a so-called vertical extensible character. It grows in steps using different glyphs and when we run out of variants a last resort kicks in: a symbol gets constructed from three pieces, a top and bottom piece and in between a repeated middle segment. The root symbol is also vertically extensible but there the change to the stretched variant is visually rather distinct. This has a reason: the specification cannot deal with slopes. So, in order to stretch the last resort, as with the bracket, goes vertical and provides a middle segment.

The root can also grow horizontally; just watch this:



The font specification can handle vertical as well as horizontal extensibles but surprise: it cannot handle a combination. Maybe the reason is that there is only one such symbol: the radical. So, instead of expecting a symmetrical engine, an exception is made that is controlled by the mentioned variables. So, while we go upwards with a proper middle glyph, we go horizontal using a rule.

One can argue that the traditional TEX machinery is complex because it uses special font properties and macros, but once you start looking into the modern variant it becomes clear that although we can have a somewhat cleaner implementation, it still is a kludge. And, because rendering on paper no longer drives development it is not to be expected that this will change. The TEX community didn't come up with a better approach and there is no reason to believe that it will in the future.

One of the reasons for users to use TEX is control over the output: instead of some quick and dirty job authors can spend time on making their documents look the way they want. Even in these internet times with dynamic rendering, there is still a place for a more frozen rendering, explicitly driven by the author. But, that only makes sense when the author can influence the rendering, maybe even without bounds.

So, because in ConTEXt I really want to provide control, as one of the last components, math radicals were made configurable too. In fact, the code involved is not that complex because most was already in place. What is interesting is that when I rewrapped radicals once again I realized that instead of delegating something to the engine and font one could as well forget about it and do all in dedicated code. After all, what is a root symbol more that a variation of a framed bit of text. Here are some examples.

```
$
  y = \sqrt   { x^2 + ax + b } \quad
  y = \sqrt[2]{ x^2 + ax + b } \quad
  y = \sqrt[3]{ \frac{x^2 + ax + b }{c} }
$
```

By default this gets rendered as follows:

$$y = \sqrt{x^2 + ax + b} \quad y = \sqrt[2]{x^2 + ax + b} \quad y = \sqrt[3]{\frac{x^2+ax+b}{c}}$$

We can change the rendering alternative to one that permits some additional properties (like color):

```
\setupmathradical[sqrt][alternative=normal,color=maincolor]
```

This looks more or less the same:

$$y = \sqrt{x^2 + ax + b} \quad y = \sqrt[2]{x^2 + ax + b} \quad y = \sqrt[3]{\frac{x^2+ax+b}{c}}$$

We can go a step further and instead of a font use a symbol that adapts itself:

```
\setupmathradical
  [sqrt]
  [alternative=mp,
   color=darkgreen]
```

Now we get this:

$$y = \sqrt{x^2 + ax + b} \quad y = \sqrt[2]{x^2 + ax + b} \quad y = \sqrt[3]{\frac{x^2+ax+b}{c}}$$

Such a variant can be more subtle, as we not only can adapt the slope dynamically, but also add a nice finishing touch to the end of the horizontal line. Take this variant:

```
\startuniqueMPgraphic{math:radical:extra}
  draw
    math_radical_simple(OverlayWidth,OverlayHeight,OverlayDepth,OverlayOf
    withpen pencircle
      xscaled (2OverlayLineWidth)
      yscaled (3OverlayLineWidth/4)
      rotated 30
    dashed evenly
    withcolor OverlayLineColor ;
\stopuniqueMPgraphic
```

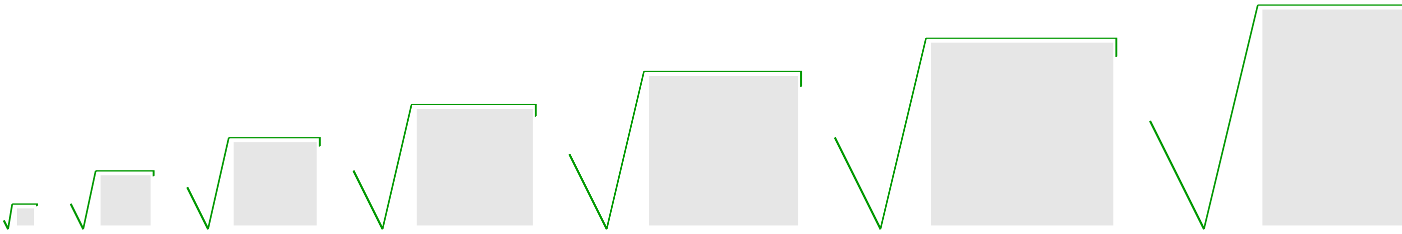
We hook this graphic into the macro:

```
\setupmathradical
  [sqrt]
  [alternative=mp,
   mp=math:radical:extra,
   color=darkred]
```

And this time we see a dashed line:

$$y = \sqrt{x^2 + ax + b} \quad y = \sqrt[2]{x^2 + ax + b} \quad y = \sqrt[3]{\frac{x^2+ax+b}{c}}$$

Of course one can argue about esthetics but let's face it: much ends up in print, also by publishers, that doesn't look pretty at all, so I tend to provide the author the freedom to make what he or she likes most. If someone is willing to spend time on typesetting (using TEX), let's at least make it a pleasant experience.



Here we see the symbol adapt. We can think of alternative symbols, for instance the first part becomes wider dependent on the height, but this can be made less prominent. Depending on user input I will provide some more variants as it's relatively easy to implement.

Before I wrap up, let's see what exactly we have in stock deep down. Traditionally TEX provides a `\surd` command which is just the root symbol. Then there is a macro `\root..\of..` that wraps the last argument in a root and typesets a degree as well (of given). In ConTEXt we now provide this:

```
$\surd x \quad \surdradical x  \quad \rootradical{3}{x} \quad \sqrt[3]{x}
```

I don't remember ever having used the `\surd` command, but this is what it renders:

$$\surd x \quad \sqrt{x} \quad \sqrt[3]{x} \quad \sqrt[3]{x}$$

Only the last command, `\sqrt` is a macro defined in one of the math modules, the others are automatically defined from the database:

```
[0x221A] = { -- there are a few more properties set
  unicodeslot = 0x221A,
  description = "SQUARE ROOT",
  adobename   = "radical",
  category    = "sm",
  mathspec    = {
    { class = "root",     name = "rootradical" },
    { class = "radical",  name = "surdradical" },
    { class = "ordinary", name = "surd"        },
  },
}
```

So we get the following definitions:

command	**meaning**	**usage**
`\surd`	`\Umathchar"0"00"00221A`	`\surd`
`\surdradical`	`\protected macro:->\Uradical "0 "221A`	`\surdradical{body}`
`\rootradical`	`\protected macro:->\Uroot "0 "221A`	`\rootradical{degree}`

So, are we better off? Given that a font sticks to how Cambria does it, we only need a minimal amount of code to implement roots. This is definitely an improvement at the engine level. However, in the font there are no fundamental differences between the traditional and more modern approach, but we've lost the opportunity to make a proper two–dimensional extensible. Eventually the user won't care as long as the macro package wraps it all up in useable macros.

1.7 Primes

Another rather disturbing issue is with primes. A prime is an accent-like symbol that as a kind of superscript is attached to a variable or function. In good old TEX tradition this is entered as follows:

```
$ f'(x) $ and $ f''(x) $
```

which produces: $f'(x)$ and $f''(x)$. The upright quote symbols are never used for anything else than primes and magically get remapped onto a prime symbol. This might look trivial, but there are several aspects to deal with, especially when using traditional fonts. In the eight-bit `lmsy10` math symbol font, which is derived from the original `cmsy10` the prime symbol looks like this:

′

The bounding box is rather tight and the reason for this becomes clear when we put it alongside another character:

$x′$

The prime is not only pretty large, it also sits on the baseline. It means that in order to make it a real prime (basically an operator pointing back to the preceding symbol), we need to raise it. Of course we can define a `\prime` command that takes care of this, and indeed that is what happens in plain TeX and derived formats. The more direct `'` input is supported by making that character an active character in math mode. Active characters behave like commands and in this case the `\prime` command.

In the OpenType latin modern fonts the prime (`U+2032`) looks like this:

xx2032

So here we have an already raised and also smaller prime symbol. And, because we also have double (`U+2033`) and triple primes (`U+2034`) a few more characters are available

xx2032 xx2033 xx2034

In the traditional approach these second and third order primes are built from the first order primes. And this introduces, in addition to the raising, another complexity: the `\prime` command has to look ahead and intercept future primes. And as there can also be a following raised symbol (or number) it needs to take a superscript trigger into account as well. So, let's look at some possible input:

`$f'(x)$`	$f'(x)$
`$f''(x)$`	$f''(x)$
`$f'''(x)$`	$f'''(x)$
`$f\prime ^2$`	$f'^2$
`$f\prime \prime ^2$`	f''^2
`$f\prime \prime \prime ^2$`	f'''^2
`$f'\prime '^2$`	f'''^2
`$f^'(x)$`	$f'(x)$
`$f'^2$`	$f'^2$
`$f{\prime }^2$`	$f'^2$

Now imagine that you have this big prime character sitting on the baseline and you need to turn `'''` into a a triple prime, but don't want `^'` to be double raised, while on the other hand `^2` should be. This is of course doable with some macro juggling but how about supporting traditional fonts in combination with OpenType, where the primes are already raised.

When we started with LuaTEX and ConTEXt MkIV, one of the first decisions I made was to go Unicode math and drop eight-bit. In order to compensate for the lack of fonts, a mechanism was provided to construct virtual Unicode math fonts, as a prelude to the lm/gyre OpenType math fonts. In the meantime we have these fonts and the virtual variants are only kept as historic reference and for further experiments.

As a starter I wrote a variant of the traditional ConTEXt `\prime` command that could recognize somehow if it was dealing with a Type1 or OpenType font. As a consequence it also had the traditional raise and look ahead mess on board. However, there was also some delegation to the Lua enhanced math support code, so the macro was not that complex. When the real OpenType math fonts showed up the macro was dropped and the virtual fonts were adapted to the raised-by-default situation, which in itself was somewhat complicated by the fact that a smaller symbol had to be used, i.e. some more information about the current set of defined math sizes has to be passed around.[5]

Anyhow, the current implementation is rather clean and supports collapsing of combinations rather well. There are four prime symbols but only three reverse prime symbols. If needed I can provide a virtual `REVERSED TRIPLE PRIME` if needed, but I guess it's not needed.

U+2032	PRIME	′	′							
U+2033	DOUBLE PRIME	″	′′	″						
U+2034	TRIPLE PRIME	‴	′′′	′″	″′	‴				
U+2057	QUADRUPLE PRIME	⁗	′′′′	′′″	′″′	″′′	″″	‴′	′‴	⁗
U+2035	REVERSED PRIME	‵	‵							
U+2036	REVERSED DOUBLE PRIME	‶	‵‵	‶						
U+2037	REVERSED TRIPLE PRIME	‷	‵‵‵	‵‶	‶‵	‷				

Of course no one will use this ligature approach but I've learned to be prepared as it wouldn't be the first time when we encounter input that is cut and paste from someplace or clicked-till-it-looks-okay.

There is one big complication and that is that where in TEX there is only one big prime that gets raised and repeated in case of multiple primes, in OpenType the primes are already raised. They are in fact not supposed to be superscripted, as they are already. In plain TEX the prime is entered using an upright single quote and that one is made active: it is in fact a macro. That macro looks ahead and intercepts following primes as well as

---

[5] The actual solution for this qualifies as a dirty trick so we are not freed from tricks yet.

subscripts. In the end, a superscript (the prime) and optional subscripts are attached to the preceding symbol. If we want to benefit from the Unicode primes as well as support collapsing, such a macro quickly becomes messy. Therefore, in MkIV the optional subscript is handled in the collapser. We cheat a bit by relocating super- and subscripts and at the same time remap the primes to virtual characters that are smashed to a smaller height, lowered to the baseline, and eventually superscripted. Indeed, it sounds somewhat complex and it is. In a next version I will also provide ways to influence the size as one might want larger of smaller primes to show up. This is one case where the traditional TeX fonts have a benefit as the primes are superscriptable characters, but we have to admit that the Unicode and OpenType approach is conceptually more correct. The only way out of this is to have a primitive operation for primes just as we have for radicals but that also has some drawbacks. Eventually I might come up with a cleaner solution for this dilemma.

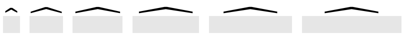
Let us summarize the situation and solution used in MkIV now:

- When (still) using the virtual Unicode math fonts, we construct a virtual glyph that has properties similar to proper OpenType math fonts.
- We collapse a sequence of primes into proper double and triple primes.
- We unraise primes so that users who (for some reason) superscript them (maybe because they still assume big ones sitting on the baseline) get the desired outcome.
- We accept mixtures of `'` and `\prime`.

We can do this because in ConTeXt MkIV we don’t care too much about exact visual compatibility as long as we can make users happy with clean mechanisms. So, this is one of the situations where the new situation is better, thanks to on the one hand the way primes are provided in fonts, and on the other hand the enhanced math machinery in MkIV.

1.8 Accents

There are a few special character types in math and accents are one of them. Personally I think that the term accent is somewhat debatable but as they are symbols drawn on top of or below something we can stick to that description for the moment. In addition to some regular fixed width variants, we have adaptive versions: `\hat` as well as `\widehat` and more.



I have no clue if wider variants are needed but such a partial coverage definitely looks weird. So, as an escape users can kick in their own code. After all, who says that a user cannot come up with a new kind of math. The following example demonstrates how this is done:

```
\startMPextensions
    vardef math_ornament_hat(expr w,h,d,o,l) text t =
        image (
            fill
                (w/2,10l) -- (w + o/2,o/2) --
                (w/2, 7l) -- (  - o/2,o/2) --
                cycle shifted (0,h-o) t ;
            setbounds
                currentpicture
            to
                unitsquare xysized(w,h) enlarged (o/2,0)
        )
    enddef ;
\stopMPextensions
```

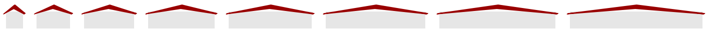
This defines a hat-like symbol. Once the sources of the math font project are published I can imagine that an ambitious user defines a whole set of proper shapes. Next we define an adaptive instance:

```
\startuniqueMPgraphic{math:ornament:hat}
    draw
        math_ornament_hat(
            OverlayWidth,
            OverlayHeight,
            OverlayDepth,
            OverlayOffset,
            OverlayLineWidth
        )
    withpen
        pencircle
            xscaled (2OverlayLineWidth)
            yscaled (3OverlayLineWidth/4)
            rotated 30
    withcolor
        OverlayLineColor ;
\stopuniqueMPgraphic
```

Last we define a symbol:

```
\definemathornament [mathhat] [mp=math:ornament:hat,color=darkred]
```

And use it as `\mathhat{...}`:



Of course this completely bypasses the accent handler and in fact even writing the normal stepwise one is not that hard to do in macros. But, there is a built–in mechanism that helps us for those cases and it can even deal with font based stretched alternatives of which there are a few: curly braces, brackets and parentheses. The reason that these can stretch is that they don't have slopes and therefore can be constructed out of pieces: in the case of a curly brace we have 4 snippets: begin, end, middle and repeated rules, and in the case of braces and brackets 3 snippets will do. But, if we really want we can use MetaPost code similar to the code shown above to get a nicer outcome.

There are in good TEX tradition four accents that can also stretch horizontally: bar, brace, parenthesis and bracket. When using fonts such an accent looks like this:

$$\overbrace{a+b+c+d} \quad \underbrace{a+b+c+d} \quad \overbrace{\underbrace{a+b+c+d}}$$

this is coded like:

```
$ \overbrace{a+b+c+d} \quad \underbrace{a+b+c+d} \quad \doublebrace{a+b+c
$
```

As with radicals, for more fancy math you can plug in MetaPost variants. Of course this kind of rendering should fit into the layout of the document but I can imagine that for schoolbooks this makes sense.

```
\useMPlibrary[mat]

\setupmathstackers
  [vfenced]
  [color=darkred,
   alternative=mp]
```

Applied in an example we get:

$$\overbracket{a+b+c+d} \quad \underbracket{a+b+c+d} \quad \overbracket{\underbracket{a+b+c+d}}$$

$$\overparen{a+b+c+d} \quad \underparen{a+b+c+d} \quad \overparen{\underparen{a+b+c+d}}$$

$$\overbrace{a+b+c+d} \quad \underbrace{a+b+c+d} \quad \overbrace{\underbrace{a+b+c+d}}$$

$$\overline{a+b+c+d} \quad \underline{a+b+c+d} \quad \overline{\underline{a+b+c+d}}$$

This kind of magic is partly possible because in LuaTEX (and therefore MkIV) we can control matters a bit better. And of course the fact that we have MetaPost embedded means that the impact of using graphics is not that large.

We used the term ‘stackers’ in the setup command so although these are officially accents, in ConTEXt we implement them as instances of a more generic mechanism: things stacked on top of each other. We will discuss these in the next section.

1.9 Stackers

In plain TEX and derived work you will find lots of arrow builders. In most cases we’re talking of a combination of one or more single or double arrow heads combined with a rule. In any case it is something that is not so much font driven but macro magic. Optionally there can be text before and/or after as well as text above and/or below them. The later is for instance the case in chemistry. This text is either math or upright properly kerned and spaced non–mathematical text so we’re talking of some mixed math and text usage. The size is normally somewhat smaller.

Arrows can also go on top or below regular math so in the end we end up with several cases:

- Something stretchable on top of or centered around the baseline, optionally with text above or below.
- Something stretchable on top of a running (piece of) text or math.
- Something stretchable below a running (piece of) text or math.
- Something stretchable on top as well as below a running (piece of) text or math.

These have in common that the symbol gets stretched. In fact the last three cases are quite similar to accents but in traditional TEX and its fonts arrows and alike never made it to accents. One reason is probably that because a macro language was available and because fonts were limited, it was rather easy to use rules to extend an arrowhead.

In ConTEXt this kind of vertically stacked stretchable material is implemented as stackers. In the chapter `mathstackers` of `about.pdf` you can read more about the details so here I stick to a short summary to illustrate what we're dealing with. Say that you want an arrow that stretches over a given width.

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

In traditional TEX with traditional fonts the definition of this arrow looks as follows:

```
\def\leftarrowfill {$
  \mathsurround=0pt
  \mathord{\mathchar"2190}
  \mkern-7mu
  \cleaders
   \hbox {$
     \mkern-2mu
     \mathchoice
      {\setbox0\hbox{$\displaystyle     -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\textstyle        -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\scriptstyle      -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\scriptscriptstyle-$}\ht0=0pt\dp0=0pt\box0}
     \mkern-2mu
    $}
   \hfill
  \mkern-7mu
  \mathchoice
   {\setbox0\hbox{$\displaystyle     -$}\ht0=0pt\dp0=0pt\box0}
   {\setbox0\hbox{$\textstyle        -$}\ht0=0pt\dp0=0pt\box0}
   {\setbox0\hbox{$\scriptstyle      -$}\ht0=0pt\dp0=0pt\box0}
   {\setbox0\hbox{$\scriptscriptstyle-$}\ht0=0pt\dp0=0pt\box0}
$}
```

When using Type1 fonts we don't use a `\mathchar` but more something like this:

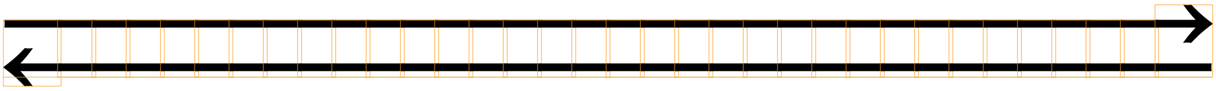
```
\leftarrow = \mathchardef\leftarrow="3220
```

What we see in this macro is a left arrow head at the start and as minus sign at the end. In between the `\cleaders` will take care of filling up the available hsize with more minus signs. The overlap is needed in order to avoid gaps due to rounding in the renderer and also obscures the rounded caps of the used minus sign.

The minus sign is used because it magically connects well to the arrow head. This is of course a property of the design but even then you can consider it a dirty trick. We don't specify a width here as this macro adapts itself to the current width due to the leader. But if we do know the width an easier approach becomes possible. Take this combination of a left and right arrow on top of each other:

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

The `\mathstylehbox` macro is a ConTEXt helper. When we take a closer look at the result (scaled up a bit) we see again snippets being used:[6].



But this time the engine itself deals with the filling. Unfortunately for the accent approach to work we need to specify the width. Given how these arrows are used, this is no problem: because we often put text on top and/or below, we need to do some packaging and therefore know the dimensions, but a generic alternative would be nice. This is why for LuaTEX we have on the low priority agenda:

```
\leaders"2190\hfill
```

or a similar primitive. This way we can let the engine do some work and keep macros simple. Normally `\leaders` delegate part of repeating to the backend but in the case of math it has to be part of constructing the formula because the extensible constructor has to be used.

If you've looked into the LuaTEX manual you might have noticed that there is a new primitive that permits this:

```
\mathstylehbox{\Uoverdelimiter\fam"21C4{\hskip4cm}}
```

However, it is hardly useable for our purpose for several reasons. First of all, when the argument is narrower than the smallest possible delimiter both get left aligned, so the delimiter sticks out (this can be considered a bug). But also, the placement is influenced by a couple of parameters that we then need to force to zero values, which might interfere. Another property of this mechanism is that the style is influenced and so we need to mess more with that. These are enough reasons to ignore this extension

---

[6] We cheat a bit here: as we use Xits in this document, and that font doesn't yet provide this magic we switch temporarily to the Pagella font

for a while. Maybe at some point, when really needed, I will write a proper wrapper for this primitive.

When we started with MkIV we stuck with the leaders approach for a while if only because there was no real need to redefine the old macros. But after a while one starts wondering if this is still the way to go, especially when reimplementing the chemistry macros didn’t lead to nicer looking code. Part of the problem was that putting two arrows on top of each other where each one goes into another direction gave issues due to the fact that we don’t have the right snippets to do it nicely. A way out was to create virtual characters for combinations of begin and end snippets as well as middle pieces, construct a proper virtual extensible and use the LuaTEX extensible constructor. Although we still have a character that gets built out of snippets, at least the begin and end snippet indicate that we have to do with one codepoint, contrary to two independent stacked arrows.

This was also the moment that I realized that it was somewhat weird that OpenType math fonts didn’t have that kind of support. After discussing this with Bogusław Jackowski of the math font project we decided that it made sense to add proper native extensibles to the upcoming math fonts. Of course I still had to support other math fonts but at least we had a conceptually clean example font now. So, from that moment on the implementation used extensibles when possible and falls back on the fake approach when needed.

In ConTEXt all these vertically stacked items are now handled by the math stacker subsystem, including a decent set of configuration options. As said, the symbols that need to stretch currently use the accent primitives which is okay but somewhat messy because that mechanism is hard to control (after all it wants to put stuff on top or below something). For (mostly) chemistry we can put text on top or below arrows and control offsets of the text as well as the axis of the arrows. We can use color and set the style. In addition there are constructs where there is text in the middle and arrows (or other symbols that need to adapt) on top or at the bottom.

Many arrows come in sizes. For instance there are two sizes of right pointing arrows as well as stretched variants, and use as top and bottom accents.

```
$\rightarrow \quad \char "2192$      →  →
$\longrightarrow \quad \char "27F6$  ⟶  ⟶

\hbox to 2cm{$\rightarrowfill $}     ────────→
\hbox to 4cm{$\rightarrowfill $}     ────────────────→
```

`$\overrightarrow {a+b+c}$`	$\overrightarrow{a+b+c}$
`$\underrightarrow {a+b+c}$`	$\underrightarrow{a+b+c}$

The first two arrows are just characters. The boxed ones are extensibles using leaders that build the arrow from snippets (a hack till we have proper character leaders) and the last two are implemented by abusing the accent mechanism and thereby use the native extensibles of the first character.

The problem here is in names and standards. The first characters have a fixed size while the later are composed. The short ones have the extensibles and can therefore be used as accents (or when supported as character leader). However from the user’s perspective, the distinction between the two Unicode characters might be less clear, not so much when they are used as character, but when used on top of or below something. As a coincidence, while writing this section, a colleague dropped a snippet of MathML on my desk:

```
<m:math>
  <m:mrow>
    <m:mover accent='true'>
      <m:mrow>
        <m:mi>A</m:mi>
        <m:mi>S</m:mi>
      </m:mrow>
      <m:mo stretchy='true'>→</m:mo>
    </m:mover>
  </m:mrow>
</m:math>
```

However, instead of <m:mo>→</m:mo> there was used `<m:mo>&xrarr;</m:mo>` and that entity is the long arrow. As is often the case in MathML the rendering is supposed to be quite tolerant and here both should stretch over the row. When a TEX user renders his or her source and sees something wrong, the search for what character or command should be used instead starts. A MathML user probably just expects things to work. This means that in a system like ConTEXt there will always be hacks and kludges to deal with such matters. It is again one of these areas where optimally the TEX community could have influenced proper and systematic coding, but it didn’t happen. So, no matter now good we make an engine or macro package, we always need to be prepared to adapt to what users expect. Let’s face it: it’s not that trivial to explain why one should favor one or the other arrow as accent: the more it has to cover, the longer it gets and the more we think of long arrows, but adding a whole bunch of `\longrightarrow...` commands to ConTeXt makes no sense.

Nevertheless, we might eventually provide more MathML compliant commands at the TeX end. Just consider the following MathML snippets:[7]

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:mover>
            <m:mo>&xrarr;</m:mo>
            <m:ms>arrow + text</m:ms>
        </m:mover>
        <m:mi>b</m:mi>
        <m:mover>
            <m:ms>text + arrow</m:ms>
            <m:mo>&xrarr;</m:mo>
        </m:mover>
        <m:mi>c</m:mi>
    </m:mrow>
</m:math>
```

This renders as:

a xrarr arrow + text b text + arrow xrarr c

Here the same construct is being used for two purposes: put an arrow on top of content that sits on the math axis or put text on an arrow that sits on the math axis. In TeX we have different commands for these:

```
$ a \overrightarrow{b+c} d $ and $ a \mrightarrow{b+c} d $
```

or

$a\,\overrightarrow{b+c}\,d$ and $a \xrightarrow{b+c} d$

The same is the case for:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:munder>
```

---

[7] These examples are variations on what we run into in Dutch school math (age 14–16).

```
            <m:mo>&xrarr;</m:mo>
            <m:ms>arrow + text</m:ms>
        </m:munder>
        <m:mi>b</m:mi>
        <m:munder>
            <m:ms>text + arrow</m:ms>
            <m:mo>&xrarr;</m:mo>
        </m:munder>
        <m:mi>c</m:mi>
    </m:mrow>
</m:math>
```

or:

a xrarr arrow + text b text + arrow xrarr c

When no arrow (or other stretchable character) is used, we still need to put one on top of the other, but in any case we need to recognize the two cases that need the special stretch treatment. There is also a combination of over and under:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:munderover>
            <m:mo>&xrarr;</m:mo>
            <m:ms>text 1</m:ms>
            <m:ms>text 2</m:ms>
        </m:munderover>
        <m:mi>b</m:mi>
    </m:mrow>
</m:math>
```

a xrarr text 1 text 2 b

And again we need to identify the special stretchable characters from anything otherwise.

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:munderover>
            <m:ms>text 1</m:ms>
            <m:ms>text 2</m:ms>
            <m:ms>text 3</m:ms>
        </m:munderover>
        <m:mi>b</m:mi>
    </m:mrow>
</m:math>
```

or:

a text 1 text 2 text 3 b

And we even can have this:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:munderover>
            <m:ms>text 1</m:ms>
            <m:mo>&xrarr;</m:mo>
            <m:ms>text 2</m:ms>
        </m:munderover>
        <m:mi>b</m:mi>
    </m:mrow>
</m:math>
```

a text 1 xrarr text 2 b

We have been supporting MathML in ConTeXt for a long time and will continue doing it. I will probably reimplement the converter (given a good reason) using more recent subsystems. It doesn't change the fact that in order to support it, we need to have some robust analytical support macros (functions) to deal with situations as mentioned. The TeX engine is not made for that but in the meantime it has become more easy thanks to a combination of TeX, Lua and data tables. Consistent availability of extensibles (either or not virtual) helps too.

Among the conclusions we can draw is that quite a lot of development (font as well as engine) is driven by what we have had for many years. A generic multi–dimensional glyph handler could have covered all odd cases that used to be done with macros but for historic reasons we could still be stuck with several slightly different and overlapping mechanisms. Nevertheless we can help macro writers by providing for instance leaders that accept characters as well in which case in math mode extensibles can be used.

1.10 Fences

Fences are symbols that are put left and/or right of a formula. They adapt their height and depth to the content they surround, so they are vertical extensibles. Users tend to minimize their coding but this is probably not a good idea with fences as there is some magic involved. For instance, TEX always wants a matching left and right fence, even if one is a phantom. So you will normally have something like this:

```
\left\lparent x \right\rparent
```

and when you don't want one of them you use a period:

```
\left\lparent x \right.
```

The question is, can we make the users live easier by magically turning braces, brackets and parentheses etc. into growing ones. As with much in MkIV, it could be that Lua can be of help. However, look at the following cases:

```
\startformula (x) \stopformula
```

$$(x)$$

This internally becomes something like this:

```
open  noad : nucleus : mathchar : U+00028
ord   noad : nucleus : mathchar : U+00078
close noad : nucleus : mathchar : U+00029
```

We get a linked list of three so-called noads where each nucleus is a math character. In addition to a nucleus there can be super- and subscripts.

```
\startformula \mathinner { (x) } \stopformula
```

$$(x)$$

```
inner noad : nucleus : submlist :
  open  noad : nucleus : mathchar : U+00028
  ord   noad : nucleus : mathchar : U+00078
  close noad : nucleus : mathchar : U+00029
```

This is still simple, although the inner primitive results in three extra levels.

```
\startformula \left( x \right) \stopformula
```

$$(x)$$

Now it becomes more complex, although we can still quite well recognize the input. The question is: how easily can we translate the previous examples into this structure.

```
inner noad : nucleus : submlist :
  left  fence : delim    : U+00028
  ord   noad  : nucleus : mathchar U+00078
  right fence : delim    : U+00029
```

```
\startformula ||x|| \stopformula
```

$$\|x\|$$

Again, we can recognize the sequence in the input:

```
ord noad : nucleus : mathchar : U+0007C
ord noad : nucleus : mathchar : U+0007C
ord noad : nucleus : mathchar : U+00078
ord noad : nucleus : mathchar : U+0007C
ord noad : nucleus : mathchar : U+0007C
```

Here we would have to collapse the two bars into one. Now, say that we manage to do this, even if it will cost a lot of code to check all border cases, then how about this?

```
\startformula \left|| x \right|| \stopformula
```

$$\left\| x \right| |$$

```
inner noad : nucleus : submlist noad :
  left  fence : delim : U+00028
  ord   noad  : nucleus : mathchar : U+0007C
  ord   noad  : nucleus : mathchar : U+00078
  right fence : delim : U+00029
ord noad : nucleus : mathchar : U+0007C
```

This time we have to look over the sublist and compare the last fence with the character following the sublist. If you keep in mind that there can be all kind of nodes in between, like glue, and that we can have multiple nested fences, it will be clear that this is a no-go. Maybe for simple cases it could work out but for a bit more complex math one ends up in constantly fighting asymmetrical input at the Lua end and occasionally fighting the heuristics at the TEX end.

It is for this reason that we provide a mechanism that users can use to avoid the primitives `\left` and `\right`.

```
\setupmathfences
  [color=red]

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

\startformula
  a \fenced[bar]      {\frac{1}{b}} c \qquad
  a \fenced[doublebar]{\frac{1}{b}} c \qquad
  a \fenced[triplebar]{\frac{1}{b}} c \qquad
  a \fenced[bracket]  {\frac{1}{b}} c \qquad
  a \fancybracket     {\frac{1}{b}} c
\stopformula
```

So, you can either use a generic instance of fences (`\fenced`) or you can define your own commands. There can be several classes of fences and they can inherit and be cloned.

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

As a bonus ConTeXt provides a few wrappers:

```
\startformula
\Lparent     \frac{1}{a} \Rparent     \quad
\Lbracket    \frac{1}{b} \Rbracket    \quad
\Lbrace      \frac{1}{c} \Rbrace      \quad
\Langle      \frac{1}{d} \Rangle      \quad
\Lbar        \frac{1}{e} \Rbar        \quad
\Ldoublebar  \frac{1}{f} \Rdoublebar  \quad
\Ltriplebar  \frac{1}{f} \Rtriplebar  \quad
\Lbracket    \frac{1}{g} \Rparent     \quad
\Langle      \frac{1}{h} \Rnothing
\stopformula
```

which gives:

$$\left(\frac{1}{a}\right) \quad \left[\frac{1}{b}\right] \quad \left\{\frac{1}{c}\right\} \quad \left\langle\frac{1}{d}\right\rangle \quad \left|\frac{1}{e}\right| \quad \left\|\frac{1}{f}\right\| \quad \frac{1}{f} \quad \left[\frac{1}{g}\right) \quad \left\langle\frac{1}{h}\right.$$

For bars, the same applies as for primes: we collapse them into proper Unicode characters when applicable:

```
U+007C  VERTICAL LINE                  |  |
U+2016  DOUBLE VERTICAL LINE           ‖  ‖  ‖
U+2980  TRIPLE VERTICAL BAR DELIMITER  ?  ⦀  ⦀  ⦀  ?
```

The question is always: to what extent do users want to structure their input. For instance, you can define this:

```
\definemathfence [weirdrange] [left="0028,right="005D]
```

and use it as:

```
$ (a,b] = \fenced[weirdrange]{a,b}$
```

This gives $(a, b] = (a, b]$ and unless you want to apply color or use specific features there is nothing wrong with the direct way. Interesting is that the complications are seldom in regular TEX input, but MathML is a different story. There is an `mfenced` element but as users can also use the more direct route, a bit more checking is needed in order to make sure that we have matching open and close symbols. For reasons mentioned before we cannot delegate this to Lua but have to use special versions of the `\left` and `\right` commands.

One complication of making a nice mechanism for this is that we cannot use the direct characters. For instance curly braces are also used for grouping and the less and equal signs serve different purposes. So, no matter what we come up with, these cases remain special. However, in ConTEXt the following is valid:

```
\setupmathfences[color=darkgreen]
\setupmathfences[mirrored][color=darkred]

\startformula
\left { \frac{1}{a} \right } \quad
\left [ \frac{1}{b} \right ] \quad
\left ( \frac{1}{c} \right ) \quad
\left < \frac{1}{d} \right > \quad
\left ⟨ \frac{1}{d} \right ⟩ \quad
\left | \frac{1}{e} \right | \quad
\left  \frac{1}{e} \right  \quad
\left  \frac{1}{e} \right  \quad
```

```
\left [ \frac{1}{d} \right [ \quad
\left ] \frac{1}{d} \right [ \quad
\stopformula
```

In the background mapping onto the mentioned left and right commands happens so we do get color support as well. And, it doesn't look that bad in your document source either. Of course other combinations are also possible.

$$\left\{\frac{1}{a}\right\} \quad \left[\frac{1}{b}\right] \quad \left(\frac{1}{c}\right) \quad \left\langle\frac{1}{d}\right\rangle \quad \left\langle\frac{1}{d}\right\rangle \quad \left|\frac{1}{e}\right| \quad \left\langle\!\left\langle\frac{1}{e}\right\rangle\!\right\rangle \quad \left\rangle\!\right\rangle\frac{1}{e}\left\langle\!\left\langle\right.\right. \quad \left[\frac{1}{d}\right[\quad \left]\frac{1}{d}\right[$$

As there are many ways to get fences and users can come from other macro packages (or use them mixed) we support them all as well as possible.

```
\left (        \frac{1}{x} \right )        =
      (        \frac{1}{x}        )        =
\left\(        \frac{1}{x} \right\)        =
     \(        \frac{1}{x}       \)        =
\left\lparent \frac{1}{x} \right\rparent =
     \lparent \frac{1}{x}       \rparent =
     \Lparent \frac{1}{x}       \Rparent
```

$$\left(\frac{1}{x}\right) = (\tfrac{1}{x}) = \left(\frac{1}{x}\right) = (\tfrac{1}{x}) = \left(\frac{1}{x}\right) = (\tfrac{1}{x}) = \left(\frac{1}{x}\right)$$

Unfortunately Unicode math doesn't free us from some annoyances with respect to paired fences. On the one hand coding math is a symbolic, abstract matter: a left parenthesis opens something and a right one closes something. The same is true for brackets and braces. However, the bar is used for left and right fencing as well as separating pieces of a formula (e.g. in conditions). Because traditionally these left and right bars were purely vertical with no slope, or hooks, or other thingies attached, in Unicode there is only one slot for it. Where paired fences can play a role in analyzing content, bars are rather useless for that. It also means that when coding a formula one cannot rely on the bar symbol to determine a left or right property. Normally this is no problem as we can use symbolic names (that include the `\left` or `\right` directive) but for instance in rendering MathML it demands some fuzzy logic to be applied. It would have been nice to have code points for the three cases.

```
\ruledhbox{$\left|x\right|$}
\ruledhbox{$\left(x\middle|x\right)$}
\ruledhbox{$\startcheckedfences\left(x\leftorright|x\right)\stopcheckedfe
```

```
\ruledhbox{$\startcheckedfences\leftorright|x\leftorright|\stopcheckedfen
\ruledhbox{$\startcheckedfences\leftorright|x\stopcheckedfences$}
\ruledhbox{$\startcheckedfences\left(x\leftorright|\stopcheckedfences$}
```

Believe me: we run into any combination of these bars and parentheses. And we're no longer surprised to see code like this (generated from applications):

```
<math>
  <mrow>
    <mo>(</mo>
    <mi>y</mi>
    <mrow>
      <mo>|</mo>
    </mrow>
    <mi>y</mi>
    <mo>)</mo>
  </mrow>
</math>
```

Here the bar sits in its own group, so what is it? A lone left, right or middle symbol, meant to stretch with the surroundings or not?

To summarize: there is no real difference (or progress) with respect to fences in LuaTeX compared to traditional TeX. We still need matching `\left` and `\right` usage and catching mismatches automatically is hard. By adding some hooks at the TeX end we can easily check for a missing `\right` but a missing `\left` needs a two-pass approach. Maybe some day in ConTeXt we will end up with multipass math processing and then I'll look into this again.

1.11 Directions

The first time I saw right-to-left math was at a Dante and later at a TUG meeting hosted in Morocco where Azzeddine Lazrek again demonstrated right-to-left math. It was only after Khaled Hosny added some support to the Xits font that I came to supporting it in ConTeXt. Apart from some housekeeping nothing special is needed: the engine is ready for it. Of course it would be nice to extend the lm and gyre fonts as well but currently it's not on the agenda. I expect to add some more control and features in the future, if only because it is a nice visual experience. And writing code for such features is kind of fun.

As this is about as complex as it can gets, it makes a nice example of how we control math font definitions, so let's see how we can define a Xits use case. Because we have a bold (heavy) font too, we define that as well. First we define the two fonts.

```
\starttypescript [math] [xits,xitsbidi] [name]
  \loadfontgoodies [xits-math]
  \definefontsynonym
    [MathRoman]
    [file:xits-math.otf]
    [features=math\mathsizesuffix,goodies=xits-math]
  \definefontsynonym
    [MathRomanBold]
    [file:xits-mathbold.otf]
    [features=math\mathsizesuffix,goodies=xits-math]
\stoptypescript
```

Discussing font goodies is beyond this article so I stick to a simple explanation. We use so-called goodie files for setting special properties of fonts, but also for defining special treatment, for instance runtime patches. The current `xits-math` goodie file looks as follows:

```
return {
  name = "xits-math",
  version = "1.00",
  comment = "Goodies that complement xits (by Khaled Hosny).",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  mathematics = {
    italics = {
      ["xits-math"] = {
         defaultfactor = 0.025,
         disableengine = true,
         corrections   = {
           [0x1D453] = -0.0375, -- f
        },
      },
    },
    alternates = {
      cal       = { feature = 'ss01', value = 1,
        comment = "Mathematical Calligraphic Alphabet" },
      greekssup = { feature = 'ss02', value = 1,
        comment = "Mathematical Greek Sans Serif Alphabet" },
      greekssit = { feature = 'ss03', value = 1,
        comment = "Mathematical Italic Sans Serif Digits" },
      monobfnum = { feature = 'ss04', value = 1,
        comment = "Mathematical Bold Monospace Digits" },
      mathbbbf  = { feature = 'ss05', value = 1,
        comment = "Mathematical Bold Double-Struck Alphabet" },
      mathbbit  = { feature = 'ss06', value = 1,
        comment = "Mathematical Italic Double-Struck Alphabet" },
      mathbbbi  = { feature = 'ss07', value = 1,
        comment = "Mathematical Bold Italic Double-Struck Alphabet"
},
      upint     = { feature = 'ss08', value = 1,
        comment = "Upright Integrals" },
      vertnot   = { feature = 'ss09', value = 1,
        comment = "Negated Symbols With Vertical Stroke" },
    },
  }
}
```

There can be many more entries but here the most important one is the `alternates` table. It defines the additional styles available in the font. Alternaties are chosen using commands like

```
\mathalternate{cal}\cal
```

and of course shortcuts for this can be defined.

Of course there is more than math, so we define a serif collection too:

```
\starttypescript [serif] [xits] [name]
  \setups[font:fallback:serif]
  \definefontsynonym[Serif]          [xits-regular.otf]    [features=defa
  \definefontsynonym[SerifBold]      [xits-bold.otf]       [features=defa
  \definefontsynonym[SerifItalic]    [xits-italic.otf]     [features=defa
  \definefontsynonym[SerifBoldItalic][xits-bolditalic.otf] [features=defa
\stoptypescript
```

If needed you can redefine the `default` feature before this typescript is used. Once we have the fonts defined we can start building a typeface:

```
\starttypescript[xits]
  \definetypeface [xits] [rm] [serif] [xits]   [default]
  \definetypeface [xits] [ss] [sans]  [heros]  [default] [rscale=0.9]
  \definetypeface [xits] [tt] [mono]  [modern] [default] [rscale=1.05]
  \definetypeface [xits] [mm] [math]  [xits]   [default]
\stoptypescript
```

We can now switch to this typeface with:

```
\setupbodyfont[xits]
```

But, as we wanted bidirectional math, something more is needed. Instead of the two fonts we define six. We could have a more abstract reference to the Xits fonts but in cases like this we prefer file names because then at least we can be sure that we get what we ask for.

So, we use the same fonts several times but apply different features to them. This time the typeface definition explicitly turns on both directions. When we don’t do that we get only left to right support, which is of course more efficient in terms of font usage.

We can now switch to the bidirectional typeface with:

```
\setupbodyfont[xitsbidi]
```

However, in order to get bidirectional math indeed, we need to turn it on.

```
\setupmathematics[align=r2l]
```

You might have wondered what this special way of defining the features using `\mathsizesuffix` means? The value of this macro is set at font definition time, and can be one of three values: `text`, `script` and `scriptscript`. At this moment the features are defined as follows:

```
\definefontfeature
  [mathematics]
  [mode=base,
   liga=yes,
   kern=yes,
   tlig=yes,
   trep=yes,
   mathalternates=yes,
   mathitalics=yes,
 % nomathitalics=yes, % don't pass to tex
   language=dflt,
   script=math]
```

From this we clone:

```
\definefontfeature
  [mathematics-l2r]
  [mathematics]
  []

\definefontfeature
  [mathematics-r2l]
  [mathematics]
  [language=ara,
   rtlm=yes,
   locl=yes]
```

Watch how we enable two specific features, where `rtlm` is a Xits-specific one. The eventually used features are defined as follows.

```
\definefontfeature[math-text]               [mathematics]    [ssty=no]
\definefontfeature[math-script]             [mathematics]    [ssty=1,mathsiz
\definefontfeature[math-scriptscript]       [mathematics]    [ssty=2,mathsiz

\definefontfeature[math-text-l2r]           [mathematics-l2r][ssty=no]
\definefontfeature[math-script-l2r]         [mathematics-l2r][ssty=1,mathsiz
\definefontfeature[math-scriptscript-l2r][mathematics-l2r][ssty=2,mathsiz

\definefontfeature[math-text-r2l]           [mathematics-r2l][ssty=no]
\definefontfeature[math-script-r2l]         [mathematics-r2l][ssty=1,mathsiz
\definefontfeature[math-scriptscript-r2l][mathematics-r2l][ssty=2,mathsiz
```

Even if it is relatively simple to do, it makes no sense to build complex mixed mode system, so currently we have to decide before we typeset a formula:

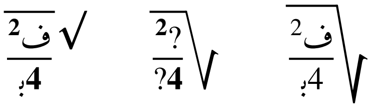
```
\setupmathematics[align=l2r]
\startformula
       \sqrt{x^2\over 4x}  \qquad
  {\bf \sqrt{x^2\over 4x}} \qquad
  {\mb \sqrt{x^2\over 4x}}
\stopformula
```

This gives a left to right formula:

$$\sqrt{\frac{x^2}{4x}} \qquad \sqrt{\frac{\mathbf{x^2}}{\mathbf{4x}}} \qquad \sqrt{\frac{\boldsymbol{x^2}}{\boldsymbol{4x}}}$$

```
\setupmathematics[align=r2l]
\startformula
       \sqrt{ف^2\over 4ب}  \qquad
  {\bf \sqrt{ف^2\over 4ب}} \qquad
  {\mb \sqrt{ف^2\over 4ب}}
\stopformula
```

And here we get an Arabic formula, where the quality of course is determined by the completeness of the font.



The bold font has a partial bold implementation so unless I implement a more complex pseudo-bold mechanism you should not expect results. Because we have no official Arabic math alphabets they are not seen by the ConTEXt MkIV analyzers that normally take care of this. It’s all a matter of demand and supply (combined with a dose of motivation). For instance while a base size might be covered, the extensibles might be missing.

About the time of writing this another variation was requested at the mailing list. For Persian math we keep the direction from left to right but the digits have to be in an Arabic font. We cannot use the bidirectional handler for this so we need to swap regular and bold digits in another way. We can use the fallback mechanism for this and a definition roughly boils down to this:

```
\definefontfallback
  [mathdigits]
  [dejavusansmono]
  [digitsarabicindic]
  [check=yes,
   force=yes,
   offset=digitsnormal]
```

This is used in:

```
\definefontsynonym
  [MathRoman]
  [file:xits-math.otf]
  [features=math\mathsizesuffix,
   goodies=xits-math,
   fallbacks=mathdigits]
```

The problem with this kind of feature is not so much in the implementation, because by now in ConTEXt we have plenty of ways to deal with such issues in a convenient way. The biggest challenge is to come up with an interface that somehow fits in the model of typescripts and with a couple of predefined typescripts we now have:

```
\usetypescriptfile[mathdigits]
\usetypescript [mathdigits] [xits-dejavu] [arabicindic]
\setupbodyfont[dejavu]
```

After that a formula like `$2 + 3 = 5$` comes out as ٢ + ٣ = ٥. In fact, if you want that in text mode, you can just use the ConTEXt MkIV font feature `anum`:

```
\definefontfeature [persian-fake-math] [arabic] [anum=yes]

\definefont[persianfakemath][dejavusans*persian-fake-math]
```

But of course you won't have proper math then. But as right-to-left math is still under construction, in due time we might end up with more advanced rendering. Currently you can exercise a little control. For instance by using the `align` parameter in combination with the `bidi` parameter. Of course support for special symbols like square roots depends on the font as well. We probably need to mirror a few more characters.

```
\m{   (  1 =   1)   }\quad
\m{   (123 = 123)   }\quad
\m{ a (  1 =   1) b }\quad
\m{ a (123 = 123) b }\quad
\m{ x = 123 y + (1 / \sqrt {x}) }
```

As in math we can assume sane usage of fences, we don't need extensive tests on pairing.

align	**bidi**					
`l2r`	`no`	$(1 = 1)$	$(123 = 123)$	$a(1 = 1)b$	$a(123 = 123)b$	$x = 123y + (1/\sqrt{x})$
`l2r`	`yes`	$(1 = 1)$	$(123 = 123)$	$a(1 = 1)b$	$a(123 = 123)b$	$x = 123y + (1/\sqrt{x})$
`r2l`	`no`	$)1 = 1($	$)321 = 321($	$b)1 = 1(a$	$b)321 = 321(a$	$)\overline{x}\surd/1(+ y321 = x$
`r2l`	`yes`	$(1 = 1)$	$(123 = 123)$	$b(1 = 1)a$	$b(123 = 123)a$	$(\overline{x}\surd/1) + y123 = x$

1.12 Structure

At some point publishers started asking for tagged pdf and as a consequence a typeset math formula suddenly becomes more than a blob of ink.

There are several arguments for tagging content. One is accessibility and another is reflow. Personally I think that both arguments are not that relevant. For instance, if you want to help a visually impaired reader, it's far better to start from a well structured original and ship that along with the typeset version. And, if you want reflow, you can better provide a (probably) simplified version in for instance html format.

We are surrounded by all kinds of visualizations, and text on paper or some medium is one. We don't make a painting accessible either. If accessibility is a demand, it should be done as best as can be, and the source is then the starting point. Of course publishers don't like that because when a source is available, it's one step closer to reuse by others. But that problem can simply be ignored as we consider publishers to be some kind of facilitating organization that deliver content from others. Alas publishers don't play that humble role so as long as they're around they can demand from their suppliers tagging of something visual.

Of course when you use TEX tagging is no real issue as you can make the input as verbose and structured as you like. But authors don't always want to be verbose, take this:

```
$ f(x) = x^2 + 3x + 7 $
```

This enters TEX as a sequence of characters: $f(x) = x^2 + 3x + 7$. These characters can have properties, for instance they can represent a relation or be an opening or closing symbol, but in most cases they are just classified as ordinary. These properties to some extent control spacing and interplay between math elements. They are not structure. If you have seen presentation MathML you have noticed that there are operators (`mo`), identifiers (`mi`) and numbers (`mn`), as well as some structural elements like fences (`mfenced`), superscripts (`msup`), subscripts (`msub`). Because it is a presentational encoding, there is no guarantee about the quality of the input as well as the rendering, but it somehow made it into a standard that is also used for tagging pdf content.

Going from mostly unstructured TEX math input to more structured output is complicated by the fact that the intermediate somewhat structured math lists eventually become regular boxes, glyphs, kerns, glue etc. In ConTEXt we carry some persistent information around so that we can still reverse engineer the output to structured input but this can be improved by more explicit tagging. We plan to add some more of that to future versions but here is an example:

```
$ \apply{f}{(x)} = x^2 + 3x + 7 $
```

You can go over the top too:

```
$ \apply{f}{(x)} = \mi{x}^\mi{2} + \mi{3}\mi{x} + \mi{7} $
```

The trick is to find an optimal mix of structure and readability. For instance, in `\sin` we already have the apply done by default, so often extra tagging is only needed in situations where there are several ways to interpret the text. Of course we're not enforcing this, but by providing some structure related features, at least we hope to make users aware of the issue. Directly inputting MathML is also an option but has never become popular.

All this is mostly a macro package issue, and ConTEXt has the basics on board. Because there is no need to adapt LuaTEX the most we will do is add a bit more consistency in building the lists (two way pointers) and carrying over properties (like attributes). We also have on the agenda a math table model that suits MathML, because some of those tables are somewhat hard to deal with.

How the export and tagging evolves depends on demand. I must admit that I implemented it as an exercise mostly because these are features I don't need myself (and no one really asked for it anyway).

1.13 Italic correction

Here we face a special situation. In regular OpenType italic correction is not part of the game, although one can cook up some positioning feature that does a similar job. In OpenType math there is italic correction, but also a more powerful sharpe-related kerning which is to be preferred. In traditional TEX the italic correction was present but since it is a font specific feature there is no way to make it work across fonts, and Type1 based math has lots of them.

At some point we have discussed throwing italic correction out of the engine, if only because it was unclear how and when to apply it. In the meantime there is some compromise reached. Because ConTEXt is always in sync with the latest LuaTEX, we oscillated between solutions and this was complicated by the fact that we had to support a mix of OpenType math fonts and virtualized Type1 legacy fonts.

The italic correction related code is still somewhat experimental, but we have several options.[8] In most cases we insert the italic correction ourselves and as the engine then sees a kern already it will not add another one. This has the advantage that we can be more consistent if only because not all fonts have these corrections and not all cases are considered by the engine.

1. A math font can have italic correction per glyph. The engine gets this passed but before it can apply them we already inject them into the mathlist where needed.
2. This is a variant of the first one, but is always applied, and not controlled by the font. This makes it possible to add additional corrections. This method is kind of obsolete as we no longer generate missing corrections at font definition time.[9]
3. This variant looks at the shape and if it is italic (or bolditalic) then correction is applied. Here the correction is related to the emwidth and controlled by a factor. We use this method by default.
4. The fourth variant is a mixture of the first (font driven) and the third (emwidth driven).

Are we better off? I honestly don't know. It is a bit of a mess and will always be, simply because the reference font (cambria) and reference implementation (msword) is not clear about it and we follow them. In that respect I consider it a macro package issue mostly. In ConTEXt at least we can offer some options.

1.14 Big

When migrating math to MkIV I couldn't resist looking into some functionality that currently uses macro magic. An example is big delimiters.

```
$ ( \big( \Big( \bigg( \Bigg( x $
```

$(\big(\Big(\bigg(\Bigg(x$

Personally I never use these, I just trust `\left` and `\right` to do the right job, but I'm no reference at all when it comes to math. The reason for looking into the bigs is that in plain TEX there are some magic numbers

---

[8] In text mode we also have an advanced mechanism for italic correction but this operates independent from math.

[9] Because the font loader is also used for the generic code, we don't want to add such features there.

involved. The macros, when translated to ConTEXt boil down to this:

```
\left<delimiter>\vbox to 0.85\bodyfontsize{}\right.
\left<delimiter>\vbox to 1.15\bodyfontsize{}\right.
\left<delimiter>\vbox to 1.45\bodyfontsize{}\right.
\left<delimiter>\vbox to 1.75\bodyfontsize{}\right.
```

Knowing that we have a chain of sizes in the font, I was tempted to go for a solution where a specific size is chosen from the linked list of next sizes. There are several strategies possible when we delegate this to Lua but we don't provide a high level interface yet. Personally I'd like to set the low level configuration options as:

```
\setconstant\bigmathdelimitermethod \plusone
\setconstant\bigmathdelimitervariant\plusthree
```

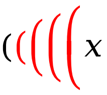
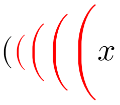
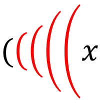
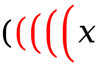
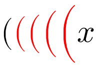
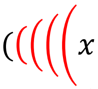
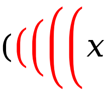
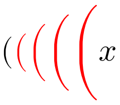
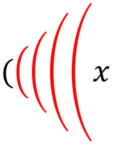
But as users might expect plain–like behaviour, ConTEXt also provides the command

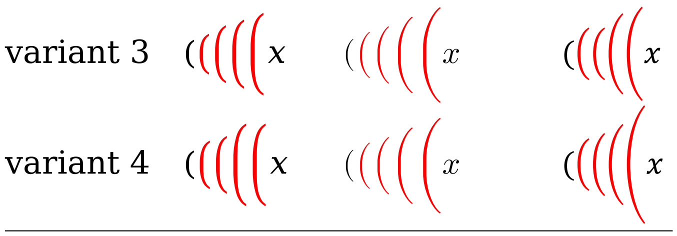
```
\plainbigdelimiters
```

which sets the method to 2. Currently that is the default. When method 1 is chosen there are four variants and the reason for keeping them all is that they are part of experiments and explorations.

1 choose size n from the available sizes
2 choose size 2n from the available sizes
3 choose the first variant that has $1.33^{n} \times (\mathrm{ht} + \mathrm{dp}) > \mathrm{size}$
4 choose the first variant that has $1.33^{n} \times \mathrm{bodyfontsize} > \mathrm{size}$

The last three variants give similar results but they are not always the same as the plain method. This is because not all fonts provide the same range.

	pagella	latin modern	cambria
plain			
variant 1			
variant 2			



So, we are somewhat unpredictable but at least we have several ways to control the situation and better solutions might show up.

1.15 Macros

I already discussed roots and the traditional `\root` command is a nice example of one that can be simplified in LuaTEX thanks to a new primitive. A macro package often has quite a lot of macros related to math that deal with tables and LuaTEX doesn't change that. But there is a category of commands that became obsolete: the ones that are used to construct characters that are not in the fonts. Keep in mind that the number of fonts as well as their size was limited at the time TEX was written, so by providing building blocks additional characters could be made. Think of for instance the negated symbols: a new symbol could be made by overlaying a slash. The same is true for arrows: by prepending or appending minus signs, arrows of arbitrary length could be constructed.

Here I will stick to another example: dots. In plain TEX we have this definition:

```
\def\vdots
  {\vbox
     {\baselineskip4pt
      \lineskiplimit0pt
      \kern6pt
      \hbox{.}%
      \hbox{.}%
      \hbox{.}}}
```

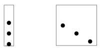
This will typeset vertical dots, while the next does them diagonally:

```
\def\ddots
  {\mathinner
     {\mkern1mu
      \raise7pt\vbox{\kern7pt\hbox{.}}%
      \mkern2mu
      \raise4pt\hbox{.}%
      \mkern2mu
      \raise1pt\hbox{.}%
      \mkern1mu}}
```

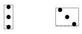
Of course these dimensions relate to the font size of plain TEX so in ConTEXt MkII we have something like this:

```
\def\vdots
  {\vbox
     {\baselineskip4\points
      \lineskiplimit\zeropoint
      \kern6\points
      \hbox{$\mathsurround\zeropoint.$}%
      \hbox{$\mathsurround\zeropoint.$}%
      \hbox{$\mathsurround\zeropoint.$}}}

\def\ddots
  {\mathinner
     {\mkern1mu
      \raise7\points\vbox{\kern 7\points\hbox{$\mathsurround\zeropoint.$}
      \mkern2mu
      \raise4\points\hbox{$\mathsurround\zeropoint.$}%
      \mkern2mu
      \raise \points\hbox{$\mathsurround\zeropoint.$}%
      \mkern1mu}}
```

These two symbols are rendered (in MkII) as follows:



I must admit that I only noticed the rather special height when I turned these macros into virtual characters for the initial virtual Unicode math that we needed in the first versions of MkIV. This is a side effect of their use in matrices. However, in MkIV we just use the characters in the font and get:



These characters look different because instead of three text periods a real symbol is used. The fact that we have more complete fonts and rely less on special font properties to achieve effects is a good thing, and in this respect it cannot be denied that LuaTEX triggered the development of more complete fonts. Of course from the user’s perspective the outcome is often the same, although … using a single character instead of three has the advantage of smaller files (neglectable), less runtime (really neglectable) and cleaner output files (undeniable) from where such characters can now be copied as one.

1.16 Unscripting

If you ever looked into plain TEX you might have noticed this following section. The symbols are more related to programming languages than to math.

```
% The following changes define internal codes as recommended
% in Appendix C of The TeXbook:
\mathcode`\^^@="2201 % \cdot
\mathcode`\^^A="3223 % \downarrow
\mathcode`\^^B="010B % \alpha
\mathcode`\^^C="010C % \beta
\mathcode`\^^D="225E % \land
\mathcode`\^^E="023A % \lnot
\mathcode`\^^F="3232 % \in
\mathcode`\^^G="0119 % \pi
\mathcode`\^^H="0115 % \lambda
\mathcode`\^^I="010D % \gamma
\mathcode`\^^J="010E % \delta
\mathcode`\^^K="3222 % \uparrow
\mathcode`\^^L="2206 % \pm
\mathcode`\^^M="2208 % \oplus
\mathcode`\^^N="0231 % \infty
\mathcode`\^^O="0140 % \partial
\mathcode`\^^P="321A % \subset
\mathcode`\^^Q="321B % \supset
\mathcode`\^^R="225C % \cap
\mathcode`\^^S="225B % \cup
\mathcode`\^^T="0238 % \forall
\mathcode`\^^U="0239 % \exists
\mathcode`\^^V="220A % \otimes
\mathcode`\^^W="3224 % \leftrightarrow
\mathcode`\^^X="3220 % \leftarrow
\mathcode`\^^Y="3221 % \rightarrow
\mathcode`\^^Z="8000 % \ne
\mathcode`\^^[="2205 % \diamond
\mathcode`\^^\="3214 % \le
\mathcode`\^^]="3215 % \ge
\mathcode`\^^^="3211 % \equiv
\mathcode`\^^_="225F % \lor
```

This means as much as: when I hit `Ctrl-Z` on my keyboard and my editor honors that by injecting character `U+1A` into the input then TeX will turn that into ≠, given that you're in math mode. I'm not sure how many keyboards and editors there are around that still do that but it illustrates that inputting in some kind of wysiwyg is not alien to TeX.[10]

One of the subprojects of the ongoing TeX user group font project is to extend the already extensive Dejavu font with all relevant math characters so that we can edit a document in a more Unicode savvy way. So, after more than three decades we might arrive where Don Knuth started: you see what you input and a similar shape will end up on paper.

Does this mean that all such input is good? Definitely not, because in Unicode we find all kinds of characters that somehow ended up there as a result of merging existing encodings. At work we're accustomed to getting input that is a mix of everything a word processor can produce and often we run into characters that users find normal but are not that handy from a TeX perspective. It's the main reason why in math mode we intercept some of them, for instance in:

```
$ y = x² + x³ + x²³ + x²ᵃ $ % not all characters are in monospace
```

These superscripts are an inconsistent bunch so they will never be real substitutes for the `^` syntax, simply because a mix like above looks bad. But fortunately it comes out well: $y = x^2 + x^3 + x^{23} + x^{2a}$. This is because ConTeXt will transform such super- and subscripts into real ones and in the process also collapse multiple scripts into a group. This is typically one of the features that already showed up early in MkIV.

Here we have a feature that doesn't relate to fonts, the math machinery or the engine, but is just a macro package goodie. It's a way to respond to the variation in input, although probably hardly any TeX math user will need it. It's one of those features that comes in handy when you use TeX as invisible backend where the input is never seen by humans.

---

[10] There are more such hidden features, for instance, in some fonts special ligatures can be implemented that no one ever uses.

1.17 Combining fonts

I already mentioned that we started out with virtual math fonts. Defining them is not that hard and boils down to defining what fonts make up the desired math font. Normally one starts out with a decent complete OpenType math font followed by mapping Type1 fonts onto specific alphabets and symbols. On top of this there are additional virtual characters constructed (including extensibles). However, this method will become kind of obsolete (read: not used) when all relevant OpenType math fonts are available.

Does this mean that we have only simple font setups? In practice yes: you can set up a math font in a few lines in a regular typescript. There are of course a few more lines needed when defining bold and/or right-to-left math but users don't need to bother about it. All is predefined. There are signals that users want to combine fonts so the already present fallback mechanism for text fonts has been made to work with math fonts as well. This permits for instance to complement the not-yet-finished OpenType Euler math fonts with Pagella. Of course you always need to keep consistency into account, but in principle you can overload for instance specific alphabets, something that can make sense when simple math is mixed with a font that has no math companion. In that case using the text italic in math mode might look better. For the at the time of this writing incomplete Euler font we can add characters like this:

```
\loadtypescriptfile[texgyre]
\loadtypescriptfile[dejavu]

\resetfontfallback  [euler]

\definefontfallback [euler] [texgyrepagella-math] [0x02100-0x02BFF]
\definefontfallback [euler] [texgyrepagella-math] [0x1D400-0x1D7FF]

\starttypescript [serif] [euler] [name]
  \setups[font:fallback:serif]
  \definefontsynonym [Serif] [euler] [features=default]
\stoptypescript

\starttypescript [math] [euler] [name]
  \definefontsynonym [MathRoman] [euler] [features=math\mathsizesuffix,fa
\stoptypescript

\starttypescript [euler]
```

```
  \definetypeface [\typescriptone] [rm] [serif] [euler]  [default]
  \definetypeface [\typescriptone] [tt] [mono]  [dejavu] [default]
[rscale=0.9]
  \definetypeface [\typescriptone] [mm] [math]  [euler]  [default]
\stoptypescript
```

If needed one can use names instead of code ranges (like `uppercasescript`) as well as map one range onto another. This last option is handy for merging a regular text font into an alphabet (in which case the Unicode's don't match).

We expect math fonts to be rather complete because after all, a font designer has a large repertoire of free alphabets to choose from. So, in practice combining math fonts will happen seldom. In text mode this is more common, especially when multiple scripts are mixed. There is a whole bunch of modules that can generate all kind of tables and overviews for testing.

1.18 Experiments

I won't describe all experiments here. An example of an experiment is a better way of dealing with punctuation, especially the cultural determined period/comma treatment. I still have the code somewhere but the heuristics are too messy to keep around.

There are also some planned experiments, like breaking and aligning display math, but they have a low priority. It's not that hard to do, but I need a good reason. The same is true for equation number placement where primitives are used that can sometimes interfere or not be used in all cases. Currently that placement in combination with alignments is implemented with quite a lot of fuzzy macro code.

One of the areas where experimenting will continue is with fonts. Early in the development of MkIV font goodies showed up. A font (or collection of fonts) can have a file (or more files) that control functionality and can have fixes. There are some in place for math fonts. It is a convenient way to use the latest greatest fonts as we have ways to circumvent issues, for instance with math parameters. The virtual math fonts are also defined as goodies.

Some mechanisms will probably be made accessible from the TEX end so that users can exercise more control. And because we're not done yet, additional features will show up for sure. There are some math related subsystems like physics and chemistry and these already demanded some extensions and might need more. Introducing math symbol (and property) dictionaries as in OpenMath is probably a next step.

I already mentioned that typesetting and rendering related technology is driven by the web. This also reflects on Unicode and OpenType. For instance, we find not only emoticons like `U+1F632` (ASTONISHED FACE) in the standard but also ‘MOUNT FUJI’, `TOKYO TOWER`, `STATUE OF LIBERTY`, `SILHOUETTE OF JAPAN`. On the other hand, in one of our older projects we still have to provide some tweak for the unary minus (as when discussing scientific calculators used in math lessons) a distinction has to be made with a regular minus sign. And there are no symbols to refer to use of media (simulation, applet, etc.) and there is as far as I know no emoticon for a student asking a question. Somehow it’s hard to defend that the Planck constant is as different from a math italic h as a ‘GRINNING FACE’ is from a ‘GRINNING FACE WITH SMILING EYES’, but the last both got a code point. I wonder with an `UNAMUSED FACE`.

Of course we can argue that this is all too visual to end up in Unicode, but the main point that I want to make is that as a TEX community (which is also related to education) we are of not that much importance and influence. Maybe it is because we always had a programmable system at hand, and folks who could make fonts, and were already extending and exploring before the web became a factor. Anyhow, in ConTEXt we solve these issues by making mechanisms extensible. For instance we can extend fonts with virtual glyphs and add features to existing fonts on the fly. Simple examples are adding some glyphs and properties to math fonts or adding color properties to whatever font. More complex examples are implementing paragraph optimizers using feature sets of fonts (most noticeably the upcoming Husayni font for advanced arabic typesetting). And, math typesetting is a speciality anyway.

Upcoming extensions to Unicode and OpenType will demonstrate that the TEX community could have been a bit more demanding and innovative, given that it had known what to demand. Interesting is that some innovation already happened by providing special fonts and macros and engines, but I guess much gets unnoticed. On the other hand, I must admit that experimenting and providing solutions independent of evolving technology also has benefits: it made (and makes) some user group meetings interesting to go to and creates interesting niches of users. Without this experimental playground I for sure would not be around.

1.19 Tracing

Tracing is available for nearly all mechanisms and math is no exception. Most tracing happens at the Lua end and can be enabled with the tracker mechanism. Users will seldom use this, but for development the situation is definitely more comfortable in MkIV. Of course it helps that the penalty of tracing and logging has become less in recent times because memory as well as runtime is hardly influenced.

We provide several styles (modules) for generating lists and tables of characters and extensibles, visualizing features and comparing fonts. Here we benefit from Lua because we can use the database embedded in ConTEXt and looping and testing is more convenient in this language. Of course the rendering is done by TEX, so this is a typical example of hybrid usage.

1.20 Conclusion

It is somewhat ironic that while ConTEXt is sometimes tagged as ‘not to be used when you need to do math typesetting’ it is this macro package that drives the development of LuaTEX with its updated math engine, which in turn influences the updated math engine in XƎTEX, that is used by other macro packages. In a similar fashion the possibility to process OpenType math fonts in LuaTEX triggered the development of such fonts as follow up on the Latin Modern and TEX Gyre projects. So, the fact that in ConTEXt we have a bit more freedom in experimenting with math (and engines) has some generic benefits as well.

I think that overall we’re better off. The implementation at the TEX end is much cleaner because we no longer have to deal with different math encodings and multiple families. Because in ConTEXt we’re less bound to traditional approaches and don’t need to be code compatible with other engines we can follow different routes than usual. After all, that was also one of the main motivations behind starting the LuaTEX project: clean (better understandable code), less mean (no more hacks at the TEX end), even if that means to be less lean (quite a lot of Lua code). Between the lines above you can read that I think that we’ve missed some opportunities but that’s a side effect of the community not being that innovative which in turn is probably driven by more or less standard expectations of publishers, as they are more served by good old stability instead of progress. Therefore, we’re probably stuck for a while, if not forever, with what we have now. And a decent ConTEXt math implementation is not going to change that. What matters is that we can (still) keep up with developments outside our sphere of influence.

I don’t claim that the current implementation of math in MkIV is flawless, but eventually we will get there.

2 Removing something (typeset)

2.1 Introduction

The primitive `\unskip` often comes in handy when you want to remove a space (or more precisely: a glue item) but sometimes you want to remove more. Consider for instance the case where a sentence is built up stepwise from data. At some point you need to insert some punctuation but as you cannot look ahead it needs to be delayed. Keeping track of accumulated content is no fun, and a quick and dirty solution is to just inject it and remove it when needed. One way to achieve this is to wrap this optional content in a box with special dimensions. Just before the next snippet is injected we can look back for that box (that can then be recognized by those special dimensions) and either remove it or unbox it back into the stream.

To be honest, one seldom needs this feature. In fact I never needed it until Alan Braslau and I were messing around with (indeed messy) bibliographic rendering and we thought it would be handy to have a helper that could remove punctuation. Think of situations like this:

```
John Foo, Mary Bar and others.
John Foo, Mary Bar, and others.
```

One can imagine this list to be constructed programmatically, in which case the comma before the `and` can be superfluous. So, the `and others` can be done like this:

```
\def\InjectOthers
  {\removeunwantedspaces
   \removepunctuation
   \space and others}

John Foo, Mary Bar, \InjectOthers.
```

Notice that we first remove spaces. This will give:

John Foo, Mary Bar and others.

where the commas after the names are coming from some not-too-clever automatism or are the side effect of lazy programming. In the sections below I will describe a bit more generic mechanism and also present a solution for non-ConTEXt users.

2.2 Marked content

The example above can be rewritten in a more general way. We define a couple macros (using ConTEXt functionality):

```
\def\InjectComma
  {\markcontent
     [punctuation]
     {\removeunwantedspaces,\space}}

\def\InjectOthers
  {\removemarkedcontent[punctuation]%
   \space and others}
```

These can be used as:

```
John Foo\InjectComma Mary Bar\InjectComma \InjectOthers.
```

Which gives us:

John Foo, Mary Bar and others.

Normally one doesn’t need this kind of magic for lists because the length of the list is known and injection can be done using the index in the list. Here is a more practical example:

```
\def\SomeTitle {Just a title}
\def\SomeAuthor{Just an author}
\def\SomeYear  {2015}
```

We paste the three snippets together:

```
\SomeTitle,\space \SomeAuthor\space (\SomeYear).
```

Just a title, Just an author (2015).

But to get even more abstract, we can do this:

```
\def\PlaceTitle
  {\SomeTitle
   \markcontent[punctuation]{.}}

\def\PlaceAuthor
  {\removemarkedcontent[punctuation]%
```

```
   \markcontent[punctuation]{,\space}%
   \SomeAuthor
   \markcontent[punctuation]{,\space}}

\def\PlaceYear
  {\removemarkedcontent[punctuation]%
   \space(\SomeYear)%
   \markcontent[punctuation]{.}}
```

Used as:

```
\PlaceTitle\PlaceAuthor\PlaceYear
```

we get the output:

Just a title, Just an author (2015).

but when we have no author,

```
\def\SomeAuthor{}

\PlaceTitle\PlaceAuthor\PlaceYear
```

Now we get:

Just a title (2015).

Even more clever is this:

```
\def\SomeAuthor{}
\def\SomeYear{}
\def\SomePeriod{\removemarkedcontent[punctuation].}

\PlaceTitle\PlaceAuthor\PlaceYear\SomePeriod
```

The output is:

Just a title.

Of course we can just test for a variable like `\SomeAuthor` being empty before we place punctuation, but there are cases where a period becomes a comma or a comma becomes a semicolon. Especially with bibliographies your worst typographical nightmares come true, so it is handy to have such a mechanism available when it's needed.

2.3 A plain solution

For users of LuaTEX who don't want to use ConTEXt I will now present an alternative implementation. Of course more clever variants are possible but the principle remains. The trick is simple enough to show here as an example of Lua coding as it doesn't need much help from the infrastructure that the macro package provides. The only pitfall is the used signal (attribute number) but you can set another one if needed. We use the `gadgets` namespace to isolate the code.

```
\directlua {
  gadgets         = gadgets or { }
  local marking   = { }
  gadgets.marking = marking

  local marksignal   = 5001
  local lastmarked   = 0
  local marked       = { }
  local local_par    = 6
  local whatsit_node = 8

  function marking.setsignal(n)
    marksignal = tonumber(n) or marksignal
  end

  function marking.mark(str)
    local currentmarked = marked[str]
    if not currentmarked then
      lastmarked    = lastmarked + 1
      currentmarked = lastmarked
      marked[str]   = currentmarked
    end
    tex.setattribute(marksignal,currentmarked)
  end

  function marking.remove(str)
    local attr = marked[str]
    if not attr then
      return
    end
    local list = tex.nest[tex.nest.ptr]
    if list then
```

```
      local head = list.head
      local tail = list.tail
      local last = tail
      if last[marksignal] == attr then
        local first = last
        while true do
          local prev = first.prev
          if not prev or prev[marksignal] ~= attr or
               (prev.id == whatsit_node and
                  prev.subtype == local_par) then
             break
          else
            first = prev
          end
        end
        if first == head then
          list.head = nil
          list.tail = nil
        else
          local prev = first.prev
          list.tail  = prev
          prev.next  = nil
        end
        node.flush_list(first)
      end
    end
  end
}
```

These functions are called from macros. We use symbolic names for the marked snippets. We could have used numbers but meaningful tags can be supported with negligible overhead. The remover starts at the end of the current list and goes backwards till no matching attribute value is seen. When a valid range is found it gets removed.

```
\def\setmarksignal#1%
  {\directlua{gadgets.marking.setsignal(\number#1)}}

\def\marksomething#1#2%
  {{\directlua{gadgets.marking.mark("#1")}{#2}}}

\def\unsomething#1%
```

```
  {\directlua{gadgets.marking.remove("#1")}}
```

The working of these macros can best be shown from a few examples:

```
before\marksomething{gone}{\em HERE}\unsomething{gone}after
before\marksomething{kept}{\em HERE}\unsomething{gone}after
\marksomething{gone}{\em HERE}\unsomething{gone}last
\marksomething{kept}{\em HERE}\unsomething{gone}last
```

This renders as:

beforeafter
before*HERE*after
last
***HERE*last**

The remover needs to look at the beginning of a paragraph marked by a local par whatsit. If we removed that, LuaTEX would crash because the list head (currently) cannot be set to nil. This is no big deal because this macro is not meant to clean up across paragraphs.

A close look at the definition of `\marksomething` will reveal an extra grouping in the definition. This is needed to make content that uses `\aftergroup` trickery work correctly. Here is another example:

```
\def\SnippetOne  {first\marksomething{punctuation}{, }}
\def\SnippetTwo  {second\marksomething{punctuation}{, }}
\def\SnippetThree{\unsomething{punctuation} and third.}
```

We can paste these snippets together and make the last one use `and` instead of a comma.

```
\SnippetOne \SnippetTwo  \SnippetThree\par
\SnippetOne \SnippetThree\par
```

We get:

first, second and third.

first and third.

Of course in practice one probably knows how many snippets there are and using a counter to keep track of the state is more efficient than first typesetting something and removing it afterwards. But still it looks like a cool feature and it can come in handy at some point, as with the title-author-year example given before.

The plain code shown here is in the distribution in the file `luatex-gadgets` and gets preloaded in the `luatex-plain` format.

3 Scanning input

3.1 Introduction

Tokens are the building blocks of the input for TEX and they drive the process of expansion which in turn results in typesetting. If you want to manipulate the input, intercepting tokens is one approach. Other solutions are preprocessing or writing macros that do something with their picked-up arguments. In ConTEXt MkIV we often forget about manipulating the input but manipulate the intermediate typesetting results instead. The advantage is that only at that moment do you know what you're truly dealing with, but a disadvantage is that parsing the so-called node lists is not always efficient and it can even be rather complex, for instance in math. It remains a fact that until LuaTEX version 0.80 ConTEXt hardly used the token interface.

In version 0.80 a new scanner interface was introduced, demonstrated by Taco Hoekwater at the ConTEXt conference 2014. Luigi Scarso and I integrated that code and I added a few more functions. Eventually the team will kick out the old token library and overhaul the input-related code in LuaTEX, because no callback is needed any more (and also because the current code still has traces of multiple Lua instances). This will happen stepwise to give users who use the old mechanism an opportunity to adapt.

Here I will show a bit of the new token scanners and explain how they can be used in ConTEXt. Some of the additional scanners written on top of the built-in ones will probably end up in the generic LuaTEX code that ships with ConTEXt.

3.2 The TEX scanner

The new token scanner library of LuaTEX provides a way to hook Lua into TEX in a rather natural way. I have to admit that I never had any real demand for such a feature but now that we have it, it is worth exploring.

The TEX scanner roughly provides the following sub-scanners that are used to implement primitives: keyword, token, token list, dimension, glue and integer. Deep down there are specific variants for scanning, for instance, font dimensions and special numbers.

A token is a unit of input, and one or more characters are turned into a token. How a character is interpreted is determined by its current catcode.

For instance a backslash is normally tagged as `escape character' which means that it starts a control sequence: a macro name or primitive. This means that once it is scanned a macro name travels as one token through the system. Take this:

```
\def\foo#1{\scratchcounter=123#1\relax}
```

Here TEX scans `\def` and turns it into a token. This particular token triggers a specific branch in the scanner. First a name is scanned with optionally an argument specification. Then the body is scanned and the macro is stored in memory. Because `\scratchcounter`, `\relax` and `#1` are turned into tokens, this body has 7 tokens.

When the macro `\foo` is referenced the body gets expanded which here means that the scanner will scan for an argument first and uses that in the replacement. So, the scanner switches between different states. Sometimes tokens are just collected and stored, in other cases they get expanded immediately into some action.

3.3 Scanning from LUA

The basic building blocks of the scanner are available at the Lua end, for instance:

```
\directlua{print(token.scan_int())} 123
```

This will print `123` to the console. Or, you can store the number and use it later:

```
\directlua{SavedNumber = token.scan_int())} 123

We saved: \directlua{tex.print(SavedNumber)}
```

The number of scanner functions is (on purpose) limited but you can use them to write additional ones as you can just grab tokens, interpret them and act accordingly.

The `scan_int` function picks up a number. This can also be a counter, a named (math) character or a numeric expression. In TEX, numbers are integers; floating-point is not supported naturally. With `scan_dimen` a dimension is grabbed, where a dimen is either a number (float) followed by a unit, a dimen register or a dimen expression (internally, all become integers).

Of course internal quantities are also okay. There are two optional arguments, the first indicating that we accept a filler as unit, while the second indicates that math units are expected. When an integer or dimension is scanned, tokens are expanded till the input is a valid number or dimension. The `scan_glue` function takes one optional argument: a boolean indicating if the units are math.

The `scan_toks` function picks up a (normally) brace-delimited sequence of tokens and (LuaTeX 0.80) returns them as a table of tokens. The function `get_token` returns one (unexpanded) token while `scan_token` returns an expanded one.

Because strings are natural to Lua we also have `scan_string`. This one converts a following brace-delimited sequence of tokens into a proper string.

The function `scan_keyword` looks for the given keyword and when found skips over it and returns `true`. Here is an example of usage:[11]

```
function ScanPair()
  local one = 0
  local two = ""
  while true do
    if token.scan_keyword("one") then
      one = token.scan_int()
    elseif token.scan_keyword("two") then
      two = token.scan_string()
    else
      break
    end
  end
  tex.print("one: ",one,"\\par")
  tex.print("two: ",two,"\\par")
end
```

This can be used as:

```
\directlua{ScanPair()}
```

You can scan for an explicit character (class) with `scan_code`. This function takes a positive number as argument and returns a character or `nil`.

1	0	`escape`
2	1	`begingroup`

---

[11] In LuaTeX 0.80 you should use `newtoken` instead of `token`.

4	2	endgroup
8	3	mathshift
16	4	alignment
32	5	endofline
64	6	parameter
128	7	superscript
256	8	subscript
512	9	ignore
1024	10	space
2048	11	letter
4096	12	other
8192	13	active
16384	14	comment
32768	15	invalid

So, if you want to grab the character you can say:

```
local c = token.scan_code(2^10 + 2^11 + 2^12)
```

In ConTeXt you can say:

```
local c = tokens.scanners.code(
  tokens.bits.space +
  tokens.bits.letter +
  tokens.bits.other
)
```

When no argument is given, the next character with catcode letter or other is returned (if found).

In ConTeXt we use the `tokens` namespace which has additional scanners available. That way we can remain compatible. I can add more scanners when needed, although it is not expected that users will use this mechanism directly.

(new)token	tokens	arguments
	scanners.boolean	
scan_code	scanners.code	(bits)
scan_dimen	scanners.dimension	(fill,math)
scan_glue	scanners.glue	(math)
scan_int	scanners.integer	
scan_keyword	scanners.keyword	
	scanners.number	

`scan_token`	`scanners.token`	
`scan_tokens`	`scanners.tokens`	
`scan_string`	`scanners.string`	
`scan_word`	`scanners.word`	
`get_token`	`getters.token`	
`set_macro`	`setters.macro`	`(catcodes,cs,str,global)`

All except `get_token` (or its alias `getters.token`) expand tokens in order to satisfy the demands.

Here are some examples of how we can use the scanners. When we would call `Foo` with regular arguments we do this:

```
\def\foo#1{%
  \directlua {
    Foo("whatever","#1",{n = 1})
  }
}
```

but when `Foo` uses the scanners it becomes:

```
\def\foo#1{%
  \directlua{Foo()} {whatever} {#1} n {1}\relax
}
```

In the first case we have a function `Foo` like this:

```
function Foo(what,str,n)
  --
  -- do something with these three parameters
  --
end
```

and in the second variant we have (using the `tokens` namespace):

```
function Foo()
  local what = tokens.scanners.string()
  local str  = tokens.scanners.string()
  local n    = tokens.scanners.keyword("n") and
               tokens.scanners.integer() or 0
  --
  -- do something with these three parameters
  --
end
```

The string scanned is kind of special as the result depends ok what is seen. Given the following definition:

```
           \def\bar  {bar}
\unexpanded\def\ubar {ubar} % \protected in plain etc
           \def\foo  {foo-\bar-\ubar}
           \def\wrap {{foo-\bar}}
           \def\uwrap{{foo-\ubar}}
```

We get:

```
{foo}         foo
{foo-\bar }   foo-bar
{foo-\ubar }  foo-\ubar
foo-\bar      foo-bar
foo-\ubar     foo-ubar
foo$bar$      foobar
\foo          foo-bar-ubar
\wrap         foo-bar
\uwrap        foo-\ubar
```

Because scanners look ahead the following happens: when an open brace is seen (or any character marked as left brace) the scanner picks up tokens and expands them unless they are protected; so, effectively, it scans as if the body of an `\edef` is scanned. However, when the next token is a control sequence it will be expanded first to see if there is a left brace, so there we get the full expansion. In practice this is convenient behaviour because the braced variant permits us to pick up meanings honouring protection. Of course this is all a side effect of how TEX scans.[12]

With the braced variant one can of course use primitives like `\detokenize` and `\unexpanded` (in ConTEXt: `\normalunexpanded`, as we already had this mechanism before it was added to the engine).

---

[12] This lookahead expansion can sometimes give unexpected side effects because often TEX pushes back a token when a condition is not met. For instance when it scans a number, scanning stops when no digits are seen but the scanner has to look at the next (expanded) token in order to come to that conclusion. In the process it will, for instance, expand conditionals. This means that intermediate catcode changes will not be effective (or applied) to already-seen tokens that were pushed back into the input. This also happens with, for instance, futurelet.

3.4 Considerations

Performance-wise there is not much difference between these methods. With some effort you can make the second approach faster than the first but in practice you will not notice much gain. So, the main motivation for using the scanner is that it provides a more TEX-ified interface. When playing with the initial version of the scanners I did some tests with performance-sensitive ConTEXt calls and the difference was measurable (positive) but deciding if and when to use the scanner approach was not easy. Sometimes embedded Lua code looks better, and sometimes TEX code. Eventually we will end up with a mix. Here are some considerations:

- In both cases there is the overhead of a Lua call.
- In the pure Lua case the whole argument is tokenized by TEX and then converted to a string that gets compiled by Lua and executed.
- When the scan happens in Lua there are extra calls to functions but scanning still happens in TEX; some token to string conversion is avoided and compilation can be more efficient.
- When data comes from external files, parsing with Lua is in most cases more efficient than parsing by TEX.
- A macro package like ConTEXt wraps functionality in macros and is controlled by key/value specifications. There is often no benefit in terms of performance when delegating to the mentioned scanners.

Another consideration is that when using macros, parameters are often passed between `{}`:

```
\def\foo#1#2#3%
  {...}
\foo {a}{123}{b}
```

and suddenly changing that to

```
\def\foo{\directlua{Foo()}}
```

and using that as:

```
\foo {a} {b} n 123
```

means that `{123}` will fail. So, eventually you will end up with something:

```
\def\myfakeprimitive{\directlua{Foo()}}
\def\foo#1#2#3{\myfakeprimitive {#1} {#2} n #3 }
```

and:

```
\foo {a} {b} {123}
```

So in the end you don't gain much here apart from the fact that the fake primitive can be made more clever and accept optional arguments. But such new features are often hidden for the user who uses more high-level wrappers.

When you code in pure TEX and want to grab a number directly you need to test for the braced case; when you use the Lua scanner method you still need to test for braces. The scanners are consistent with the way TEX works. Of course you can write helpers that do some checking for braces in Lua, so there are no real limitations, but it adds some overhead (and maybe also confusion).

One way to speed up the call is to use the `\luafunction` primitive in combinations with predefined functions and although both mechanisms can benefit from this, the scanner approach gets more out of that as this method cannot be used with regular function calls that get arguments. In (rather low level) Lua it looks like this:

```
luafunctions[1] = function()
  local a token.scan_string()
  local n token.scan_int()
  local b token.scan_string()
  -- whatever --
end
```

And in TEX:

```
\luafunction1 {a} 123 {b}
```

This can of course be wrapped as:

```
\def\myprimitive{\luafunction1 }
```

3.5 Applications

The question now pops up: where can this be used? Can you really make new primitives? The answer is yes. You can write code that exclusively stays on the Lua side but you can also do some magic and then print back something to TEX. Here we use the basic token interface, not ConTEXt:

```
\directlua {
local token = newtoken or token
function ColoredRule()
  local w, h, d, c, t
  while true do
    if token.scan_keyword("width") then
      w = token.scan_dimen()
    elseif token.scan_keyword("height") then
      h = token.scan_dimen()
    elseif token.scan_keyword("depth") then
      d = token.scan_dimen()
    elseif token.scan_keyword("color") then
      c = token.scan_string()
    elseif token.scan_keyword("type") then
      t = token.scan_string()
    else
      break
    end
  end
  if c then
    tex.sprint("\\color[",c,"]{")
  end
  if t == "vertical" then
    tex.sprint("\\vrule")
  else
    tex.sprint("\\hrule")
  end
  if w then
    tex.sprint("width ",w,"sp")
  end
  if h then
    tex.sprint("height ",h,"sp")
  end
  if d then
    tex.sprint("depth ",d,"sp")
  end
  if c then
    tex.sprint("\\relax}")
  end
end
}
```

This can be given a TEX interface like:

```
\def\myhrule{\directlua{ColoredRule()} type {horizontal} }
\def\myvrule{\directlua{ColoredRule()} type {vertical} }
```

And used as:

```
\myhrule width \hsize height 1cm color {darkred}
```

giving:

Of course ConTEXt users can use the following commands to color an otherwise-black rule (likewise):

```
\blackrule[width=\hsize,height=1cm,color=darkgreen]
```

The official ConTEXt way to define such a new command is the following. The conversion back to verbose dimensions is needed because we pass back to TEX.

```
\startluacode
local myrule = tokens.compile {
  {
    { "width",  "dimension", "todimen" },
    { "height", "dimension", "todimen" },
    { "depth",  "dimension", "todimen" },
    { "color",  "string" },
    { "type",   "string" },
  }
}

interfaces.scanners.ColoredRule = function()
  local t = myrule()
  context.blackrule {
    color  = t.color,
    width  = t.width,
    height = t.height,
    depth  = t.depth,
  }
end
\stopluacode
```

With:

```
\unprotect \let\myrule\clf_ColoredRule \protect
```

and

```
\myrule width \textwidth height 1cm color {maincolor} \relax
```

we get:

There are many ways to use the scanners and each has its charm. We will look at some alternatives from the perspective of performance. The timings are more meant as relative measures than absolute ones. After all it depends on the hardware. We assume the following shortcuts:

```
local scannumber  = tokens.scanners.number
local scankeyword = tokens.scanners.keyword
local scanword    = tokens.scanners.word
```

We will scan for four different keys and values. The number is scanned using a helper `scannumber` that scans for a number that is acceptable for Lua. Thus, `1.23` is valid, as are `0x1234` and `12.12E4`.

```
function getmatrix()
  local sx, sy = 1, 1
  local rx, ry = 0, 0
  while true do
    if scankeyword("sx") then
      sx = scannumber()
    elseif scankeyword("sy") then
      sy = scannumber()
    elseif scankeyword("rx") then
      rx = scannumber()
    elseif scankeyword("ry") then
```

```
      ry = scannumber()
    else
      break
    end
  end
  -- action --
end
```

Scanning the following specification 100000 times takes 1.00 seconds:

```
sx 1.23 sy 4.5 rx 1.23 ry 4.5
```

The ‘tight’ case takes 0.94 seconds:

```
sx1.23 sy4.5 rx1.23 ry4.5
```

We can compare this to scanning without keywords. In that case there have to be exactly four arguments. These have to be given in the right order which is no big deal as often such helpers are encapsulated in a user-friendly macro.

```
function getmatrix()
  local sx, sy = scannumber(), scannumber()
  local rx, ry = scannumber(), scannumber()
  -- action --
end
```

As expected, this is more efficient than the previous examples. It takes 0.80 seconds to scan this 100000 times:

```
1.23 4.5 1.23 4.5
```

A third alternative is the following:

```
function getmatrix()
  local sx, sy = 1, 1
  local rx, ry = 0, 0
  while true do
    local kw = scanword()
    if kw == "sx" then
      sx = scannumber()
    elseif kw == "sy" then
      sy = scannumber()
    elseif kw == "rx" then
```

```
      rx = scannumber()
    elseif kw == "ry" then
      ry = scannumber()
    else
      break
    end
  end
  -- action --
end
```

Here we scan for a keyword and assign a number to the right variable. This one call happens to be less efficient than calling `scan_keyword` 10 times $(4+3+2+1)$ for the explicit scan. This run takes 1.11 seconds for the next line. The spaces are really needed as words can be anything that has no space.[13]

```
sx 1.23 sy 4.5 rx 1.23 ry 4.5
```

Of course these numbers need to be compared to a baseline of no scanning (i.e. the overhead of a Lua call which here amounts to 0.10 seconds. This brings us to the following table.

keyword checks	0.9 sec
no keywords	0.7 sec
word checks	1.0 sec

The differences are not that impressive given the number of calls. Even in a complex document the overhead of scanning can be negligible compared to the actions involved in typesetting the document. In fact, there will always be some kind of scanning for such macros so we're talking about even less impact. So you can just use the method you like most. In practice, the extra overhead of using keywords in combination with explicit checks (the first case) is rather convenient.

If you don't want to have many tests you can do something like this:

```
local keys = {
  sx = scannumber, sy = scannumber,
  rx = scannumber, ry = scannumber,
}
```

---

[13] Hard-coding the word scan in a C helper makes little sense, as different macro packages can have different assumptions about what a word is. And we don't extend LuaTeX for specific macro packages.

```
function getmatrix()
  local values = { }
  while true do
    for key, scan in next, keys do
      if scankeyword(key) then
        values[key] = scan()
      else
        break
      end
    end
  end
  -- action --
end
```

This is still quite fast although one now has to access the values in a table. Working with specifications like this is clean anyway so in ConTEXt we have a way to abstract the previous definition.

```
local specification = tokens.compile {
  {
    { "sx", "number" }, { "sy", "number" },
    { "rx", "number" }, { "ry", "number" },
  },
}

function getmatrix()
  local values = specification()
  -- action using values.sx etc --
end
```

Although one can make complex definitions this way, the question remains if it is a better approach than passing Lua tables. The standard ConTEXt way for controlling features is:

```
\getmatrix[sx=1.2,sy=3.4]
```

So it doesn’t matter much if deep down we see:

```
\def\getmatrix[#1]%
  {\getparameters[@@matrix][sx=1,sy=1,rx=1,ry=1,#1]%
   \domatrix
     \@@matrixsx
     \@@matrixsy
```

```
      \@@matrixrx
      \@@matrixry
   \relax}
```

or:

```
\def\getmatrix[#1]%
  {\getparameters[@@matrix][sx=1,sy=1,rx=1,ry=1,#1]%
   \domatrix
     sx \@@matrixsx
     sy \@@matrixsy
     rx \@@matrixrx
     ry \@@matrixry
   \relax}
```

In the second variant (with keywords) can be a scanner like we defined before:

```
\def\domatrix#1#2#3#4%
  {\directlua{getmatrix()}}
```

but also:

```
\def\domatrix#1#2#3#4%
  {\directlua{getmatrix(#1,#2,#3,#4)}}
```

given:

```
function getmatrix(sx,sy,rx,ry)
    -- action using sx etc --
end
```

or maybe nicer:

```
\def\domatrix#1#2#3#4%
  {\directlua{domatrix{
     sx = #1,
     sy = #2,
     rx = #3,
     ry = #4
   }}}
```

assuming:

```
function getmatrix(values)
```

```
    -- action using values.sx etc --
end
```

If you go for speed the scanner variant without keywords is the most efficient one. For readability the scanner variant with keywords or the last shown example where a table is passed is better. For flexibility the table variant is best as it makes no assumptions about the scanner — the token scanner can quit on unknown keys, unless that is intercepted of course. But as mentioned before, even the advantage of the fast one should not be overestimated. When you trace usage it can be that the (in this case matrix) macro is called only a few thousand times and that doesn't really add up. Of course many different sped-up calls can make a difference but then one really needs to optimize consistently the whole code base and that can conflict with readability. The token library presents us with a nice chicken–egg problem but nevertheless is fun to play with.

3.6 Assigning meanings

The token library also provides a way to create tokens and access properties but that interface can change with upcoming versions when the old library is replaced by the new one and the input handling is cleaned up. One experimental function is worth mentioning:

```
token.set_macro("foo","the meaning of bar")
```

This will turn the given string into tokens that get assigned to `\foo`. Here are some alternative calls:

```
set_macro("foo")                       \def \foo{}
set_macro("foo","meaning")             \def \foo{meaning}
set_macro("foo","meaning","global")  \gdef \foo{meaning}
```

The conversion to tokens happens under the current catcode regime. You can enforce a different regime by passing a number of an allocated catcode table as the first argument, as with `tex.print`. As we mentioned performance before: setting at the Lua end like this:

```
token.set_macro("foo","meaning")
```

is about two times as fast as:

```
tex.sprint("\\def\\foo{meaning}")
```

or (with slightly more overhead) in ConTEXt terms:

```
context("\\def\\foo{meaning}")
```

The next variant is actually slower (even when we alias `setvalue`):

```
context.setvalue("foo","meaning")
```

but although 0.4 versus 0.8 seconds looks like a lot on a TEX run I need a million calls to see such a difference, and a million macro definitions during a run is a lot. The different assignments involved in, for instance, 3000 entries in a bibliography (with an average of 5 assignments per entry) can hardly be measured as we're talking about milliseconds. So again, it's mostly a matter of convenience when using this function, not a necessity.

3.7 Conclusion

For sure we will see usage of the new scanner code in ConTEXt, but to what extent remains to be seen. The performance gain is not impressive enough to justify many changes to the code but as the low-level interfacing can sometimes become a bit cleaner it will be used in specific places, even if we sacrifice some speed (which then probably will be compensated for by a little gain elsewhere).

The scanners will probably never be used by users directly simply because there are no such low level interfaces in ConTEXt and because manipulating input is easier in Lua. Even deep down in the internals of ConTEXt we will use wrappers and additional helpers around the scanner code. Of course there is the fun-factor and playing with these scanners is fun indeed. The macro setters have as their main benefit that using them can be nicer in the Lua source, and of course setting a macro this way is also conceptually cleaner (just like we can set registers).

Of course there are some challenges left, like determining if we are scanning input of already converted tokens (for instance in a macro body or tokenlist expansion). Once we can properly feed back tokens we can also look ahead like `\futurelet` does. But for that to happen we will first clean up the LuaTEX input scanner code and error handler.

4 Profiling lines

4.1 Introduction

Although TEX is pretty good at typesetting simple texts like novels, in practice it's often used for getting more complex stuff on paper (or screen). Math is of course the first thing that comes to mind. If for instance you look at the books typeset by Don Knuth you will see a rendering that is rather consistent in spacing. This is no surprise as the author pays a lot of attention to detail and uses inline versus display math properly. No publisher will complain about the result.

In the documents that I have to write styles for, the content is rather mixed, and in particular inline math can have display math properties. In a one-column layout this is not a real problem especially because lots of short sentences and white space is used: we're talking of secondary-school educational math where arguments for formatting something this or that way is not always rational and consistent but more based on “this is what the student expects”, “the competitor also does it that way” or just “we like this more”. For instance in a recent project, the books with answers to questions had to be typeset in a multicolumn layout and because math was involved, we end up with lines with more height and depth than normal. That can not only result in more pages but also can make the result look a bit messy.

This paragraph demonstrates how lines are handled: when a paragraph is broken into lines each line becomes a horizontal box with a height and depth determined by the size of the characters that make up the line. There is a minimal distance between baselines (`baselineskip`) and when lines touch there can optionally be a `\lineskip`. In the end we get a vertical list of boxes and glue (either of not flexible) mixed with penalties that determine optimal paragraph breaks. This paragraph shows that there is normally enough space available to do the job.

We already have some ways to control this. For instance the dimensions of math can be limited a bit and lines can be made to snap on a grid (which is what publishers often want anyway). However, another alternative is to look at the line and decide if successive lines can be moved closer, of course within the constraints of the height and and depth of the lines. There is no real way to see if some ugly clash can happen simply because when we run into boxed material there can be anything inside and the dimensions can be set on purpose. This means that we have to honour all dimensions and only can mess around with dimensions when we’re reasonably confident. In ConTEXt this messing is called profiling and that is what we will discuss next.

4.2 Line heights and depths

In this section we will use some (Dutch) examples from documents that we’ve processed. We show unprofiled versions, with two different paragraph widths, in figure 4.1. All three examples shown demonstrate that as soon as we use something more complex than a number or variable in a subscript we exceed the normal line height, and thus the line spacing becomes somewhat irregular.

The profiled rendering of the same examples are shown in figure 4.2. Here we use the minimal heights and depths plus a minimum distance of 1pt. This default method is called `strict`.

In the first and last example there are some lines where the depth of one line combined with the height of the following exceeds the standard line height. This forces TEX to insert `\lineskip` (mentioned in the demonstration paragraph above), a dimension that is normally set to a fraction of the line spacing (for instance 1pt for a 10pt body font and 12pt line spacing). When we are profiling, `\lineskip` is ignored and we use a settable distance instead. The second example (with superscripts) normally comes out fine as the math stays within limits and we make sure that smaller fractions and scripts stay within the natural limits of the line, but nested scripts can be an issue.

In figure 4.3 we see the profile of a regular text with no math. The average text stays well within the limits of height and depth. If this doesn’t happen for prose then you need to adapt the height/depth ratio to the ascender/descender ratio of the bodyfont. For regular text it makes no sense to use the profiler, it only slows down typesetting.

4.3 When lines exceed boundaries

Let’s now take a more detailed look at what happens when lines get too high or low. First we’ll zoom in on a simple example: in figure 4.4, we compare a sample text rendered using the variants of profiling currently implemented. (This is still experimental code so there might be more in the future). Seeing profiles helps to get a picture of the complications we have to deal with. In `hsize 12cm` `unprofiled`

Regelmatig kom je procenten tegen. ‘Pro centum’ is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: $45\% = \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

`hsize 10cm` `unprofiled`

Regelmatig kom je procenten tegen. ‘Pro centum’ is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: $45\% = \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

example 1

`hsize 12cm` `unprofiled`

Je gaat uit van de bekende eigenschappen van machten. Bijvoorbeeld: $g^r * g^s = g^{(r+s)}$. Neem je hierin $r =^g \log(a)$ en $s =^g \log b$, dan vind je: $g^{{}^g\log(a) + {}^g\log(b)} = g^{{}^g\log a} \times g^{{}^g\log b} = a \times b$. Hierbij gebruik je de definitieformules.

`hsize 10cm` `unprofiled`

Je gaat uit van de bekende eigenschappen van machten. Bijvoorbeeld: $g^r * g^s = g^{(r+s)}$. Neem je hierin $r =^g \log(a)$ en $s =^g \log b$, dan vind je: $g^{{}^g\log(a) + {}^g\log(b)} = g^{{}^g\log a} \times g^{{}^g\log b} = a \times b$. Hierbij gebruik je de definitieformules.

example 2

`hsize 12cm` `unprofiled`

Omdat volgens de eigenschappen van machten en exponenten geldt $\frac{1}{x^4} = x^{-4}$ is ook hier sprake van een machtsfunctie, namelijk $f(x) = \frac{6}{x^4} = 6 \times \frac{1}{x^4} = 6x^{-4}$.

`hsize 10cm` `unprofiled`

Omdat volgens de eigenschappen van machten en exponenten geldt $\frac{1}{x^4} = x^{-4}$ is ook hier sprake van een machtsfunctie, namelijk $f(x) = \frac{6}{x^4} = 6 \times \frac{1}{x^4} = 6x^{-4}$.

example 3

Figure 4.1 Unprofiled examples.

addition to the normal `vbox` variant (used in the previous examples), we show `none` which only analyzes, `strict` that uses the natural dimensions of lines and `fixed` that is supposed to cooperate with grid snapping.

Figure 4.4 we show what happens when we add some more excessive height and depth to lines. The samples are:

```
line 1 x\lower2ex\hbox{xxx}\par
line 2 x\raise2ex\hbox{xxx}\par
line 3 \par
```

`hsize 12cm`
`profiled`

Regelmatig kom je procenten tegen. ‘Pro centum’ is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: $45\% = \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

`hsize 10cm`
`profiled`

Regelmatig kom je procenten tegen. ‘Pro centum’ is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: $45\% = \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

example 1

`hsize 12cm`
`profiled`

Je gaat uit van de bekende eigenschappen van machten. Bijvoorbeeld: $g^r * g^s = g^{(r+s)}$. Neem je hierin $r = {}^g\log(a)$ en $s = {}^g\log b$, dan vind je: $g^{{}^g\log(a)+{}^g\log(b)} = g^{{}^g\log a} \times g^{{}^g\log b} = a \times b$. Hierbij gebruik je de definitieformules.

`hsize 10cm`
`profiled`

Je gaat uit van de bekende eigenschappen van machten. Bijvoorbeeld: $g^r * g^s = g^{(r+s)}$. Neem je hierin $r = {}^g\log(a)$ en $s = {}^g\log b$, dan vind je: $g^{{}^g\log(a)+{}^g\log(b)} = g^{{}^g\log a} \times g^{{}^g\log b} = a \times b$. Hierbij gebruik je de definitieformules.

example 2

`hsize 12cm`
`profiled`

Omdat volgens de eigenschappen van machten en exponenten geldt $\frac{1}{x^4} = x^{-4}$ is ook hier sprake van een machtsfunctie, namelijk $f(x) = \frac{6}{x^4} = 6 \times \frac{1}{x^4} = 6x^{-4}$.

`hsize 10cm`
`profiled`

Omdat volgens de eigenschappen van machten en exponenten geldt $\frac{1}{x^4} = x^{-4}$ is ook hier sprake van een machtsfunctie, namelijk $f(x) = \frac{6}{x^4} = 6 \times \frac{1}{x^4} = 6x^{-4}$.

example 3

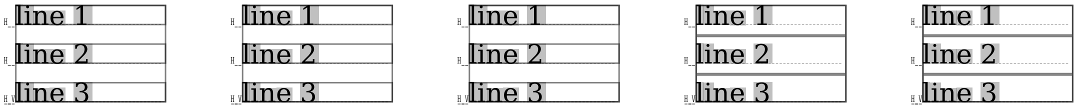
Figure 4.2 Profiled examples.

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.

Figure 4.3 Normal lines profiled (quote by Hermann Zapf)

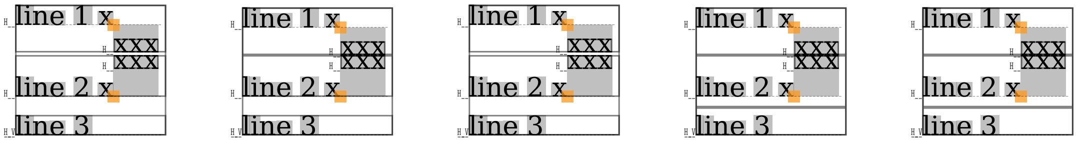
and:

```
x\lower2ex\hbox{xxx} line 1 \par
line 2 x\raise2ex\hbox{xxx}\par
```



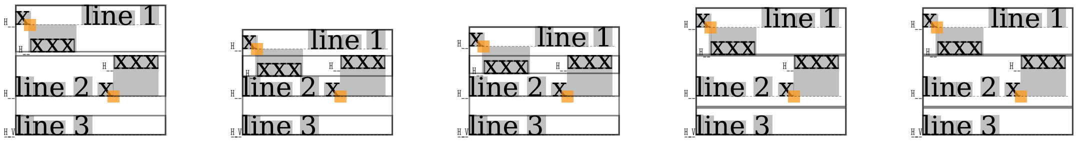
none strict/0pt strict/1pt fixed/0pt fixed/1pt

no excessive height and depth



none strict/0pt strict/1pt fixed/0pt fixed/1pt

some excessive height and depth (overlapping)



none strict/0pt strict/1pt fixed/0pt fixed/1pt

some excessive height and depth (out of touch)

Figure 4.4 Variants of profiling, using a constructed two-line text.

```
line 3 \par
```

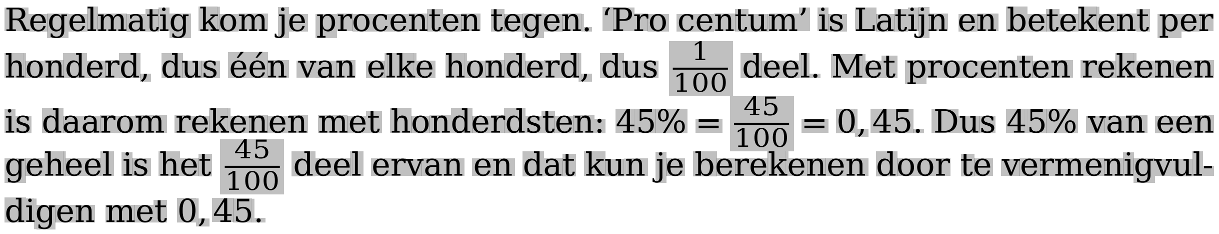
Here the `strict` variant has some effect while `fixed` only has some influence on the height and depth of lines. Later we will see that `fixed` operates in steps and the default step is large so here we never meet the criteria for closing up.[14]

A profiled box is typeset with `\profiledbox`. There is some control possible but the options are not yet set in stone so we won't use them all here. Profiling can be turned on for the whole document with `\setprofile` but I'm sure that will seldom happen, and these examples show why: one cannot beforehand say if the result looks good. Let's now apply profiling to a real text. If you play with this yourself you can show profiles in gray with a tracker:

```
\enabletrackers[profiling.show]
```

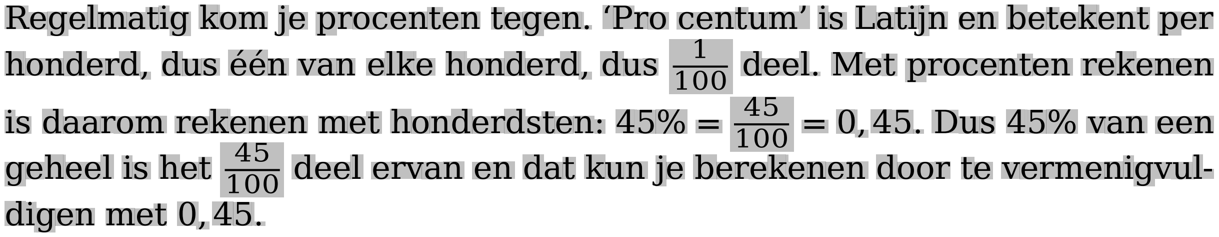
---

[14] In ConTEXt we normally use `\high` and `\low` and both ensure that we don't exceed the natural height and depth.



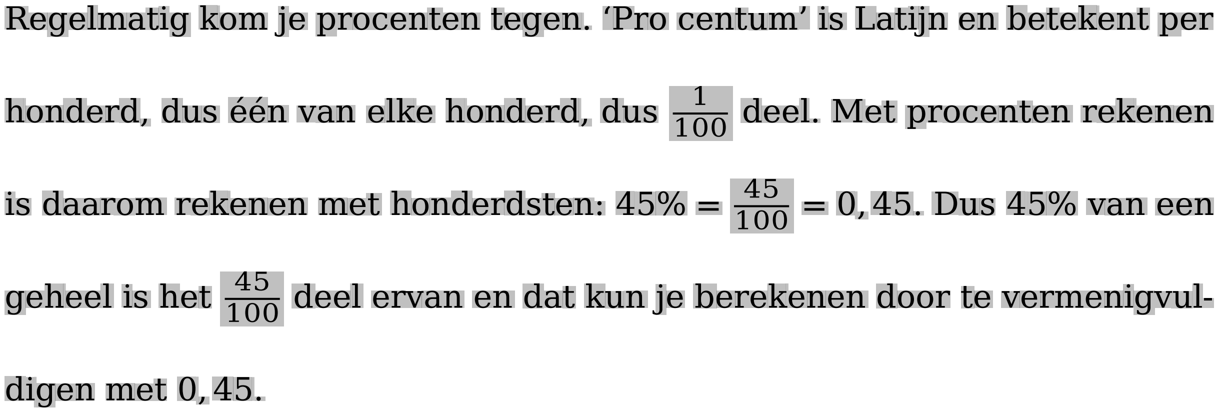
Regelmatig kom je procenten tegen. ‘Pro centum’ is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: $45\% = \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

zero distance, resulting height 83.5265pt



Regelmatig kom je procenten tegen. ‘Pro centum’ is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: $45\% = \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

distance, resulting height 85.5265pt



Regelmatig kom je procenten tegen. ‘Pro centum’ is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: $45\% = \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

distance, double height and depth, resulting height 151.302pt

Figure 4.5 Examples width different dimensions.

We show the effects of setting distances in figure 4.5. We start with a zero distance:

```
\profiledbox
  [strict]
  [distance=0pt]
  {\nl\getbuffer[example-1]}
```

Because we don’t want lines to touch we then set the minimum distance to a reasonable value (1pt).

```
\profiledbox
  [strict]
  [distance=1pt]
  {\nl\getbuffer[example-1]}
```

Finally we also double the height and depth of lines, something that normally will not be done. The defaults are the standard height and depth (the ones you get when you inject a so-called `\strut`).

```
\profiledbox
  [strict]
  [height=2\strutht,
   depth=2\strutdp,
   distance=1pt]
  {\nl\getbuffer[example-1]}
```

The problem with this kind of analysis is that deciding when and how to use this information to improve spacing is non-trivial. One of the characteristics of user demand is that it nearly always concerns rather specific situations and that suggested solutions could work only in those cases. But as soon as we have one exceptional situation, intervention is needed which in turn means that a mechanism has to be under complete user control. That itself assumes that the user still has control, which is not the case in automated workflows. In fact, as soon as one is in control over the source and rendering, there are often easier ways to deal with the few cases that need treatment. Possible interference can come from, for instance:

- whitespace between paragraphs
- section titles (using different fonts and spacing)
- descriptions and other intermezzos
- images that interrupt the flow, or end up next to text
- ornaments like margin words (we catch some)
- text backgrounds making spacing assumptions

After a few decades of using TEX and writing solutions, it has become pretty clear that fully automated typesetting is a dream, if only because the input can be pretty weird and inconsistent and demands (from those who are accustomed to tweaking manually in a desktop publishing application) can be pretty weird and inconsistent too. So, the only real solution is to use some kind of artificial intelligence that one can feed with demands and constraints and that hopefully is clever enough to deal with the inconsistencies. As this kind of computing is unlikely to happen in my lifetime, poor man explicit solutions have to do the job for now. One can add all kinds of heuristics to the profiler but this can backfire when control is needed. Alternatively one can end up with many options like we have in grid snapping.

4.4 Where to use profiling

In ConTEXt there are four places (maybe a few more eventually) where this kind of control over spacing makes sense:

- the main text flow in single column mode
- multi-column mode, especially mixed columns
- framed texts, used for all kinds of content
- explicitly (balanced) split boxes

Because framed texts are used all over, for instance in tables, it means that if we provide control over spacing using profiles, many ConTEXt mechanisms can use it. However, enabling this for all packaging has a significant overhead so it has to be used with care so that there is no performance hit when it is not used. Here is an easy example using `\framed`:

```
\framed
  [align=normal,
   profile=fixed,
   frame=off]
  {some text ...}
```

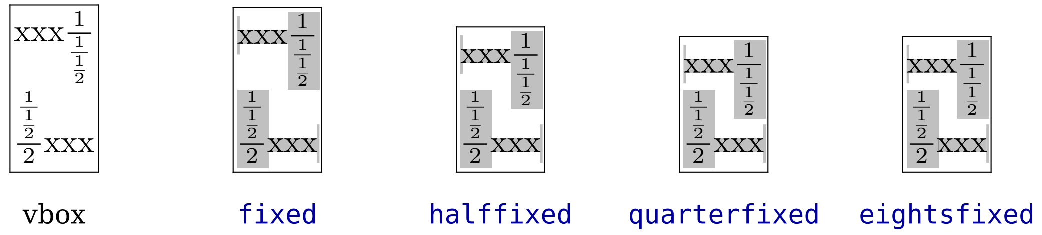
For the following examples we define this helper:

```
\starttexdefinition demo-profile-1 #1
  \framed
    [align=normal,profile=#1]
    {xxx$\frac{1}{\frac{1}{\frac{1}{2}}}$
     \par
     $\frac{\frac{1}{\frac{1}{2}}}{2}$xxx}
\stoptexdefinition
```

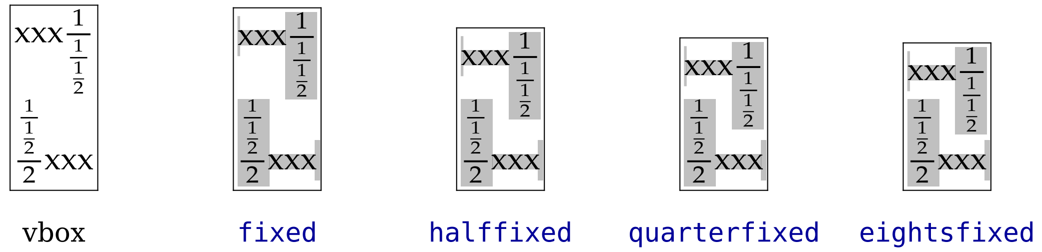
We apply this to predefined profiles. The macro is called like this:

```
\texdefinition{demo-profile-1}{fixed}
```

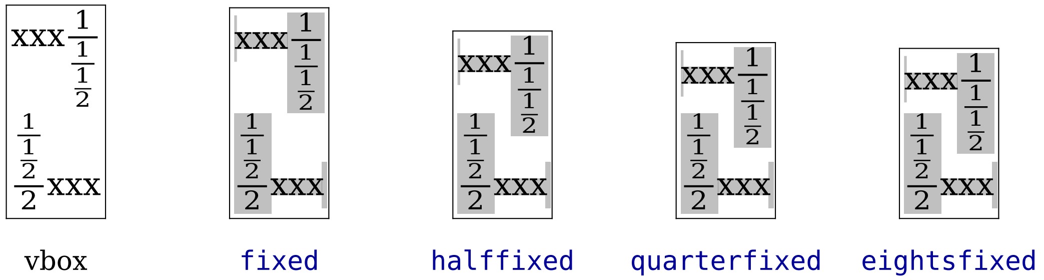
The outcome can depend on the font used: in figure 4.6 we show Latin Modern, TEX Gyre Pagella and Dejavu. Because in ConTEXt the line height depends on the bodyfont; each case is different.



Latin Modern



Pagella

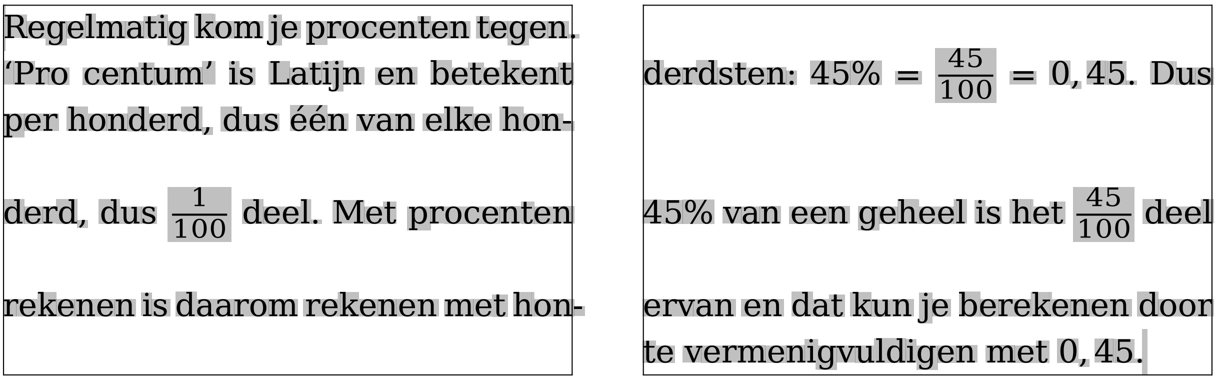


Dejavu

Figure 4.6 A few fonts compared.

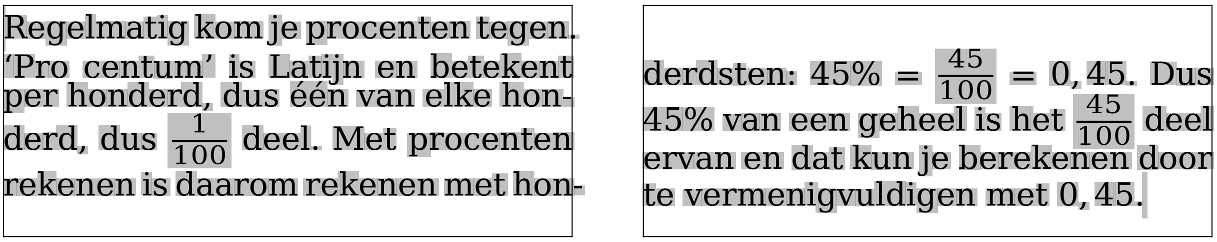
As mentioned, we need this kind of profiling in multi-column typesetting, so let us have a look at that now. Columns are processed in grid mode but this is taken into account. We can simulate this by using boxed columns; see figure 4.7. One of the biggest problems is what to do with the bottom and top of a page or column. This will probably take a bit more to get right, and likely we will end up with different strategies. We can also think of special handlers but that will come with a high speed penalty. In the `strict` variant we don’t mess with the dimension of a line too much, but the `fixed` alternative will get some more control.

Although using this feature looks promising it is also dangerous. For instance a side effect can be that interline spacing becomes inconsistent and even ugly. It really depends on the content. Also, as soon as some grid snapping is used, the gain becomes less, simply because the solution space is smaller. Then of course there is the matter of overall look and feel: most documents that need this kind of magic look bad anyway, so why bother. In this respect it is comparable to applying protrusion and expansion. There



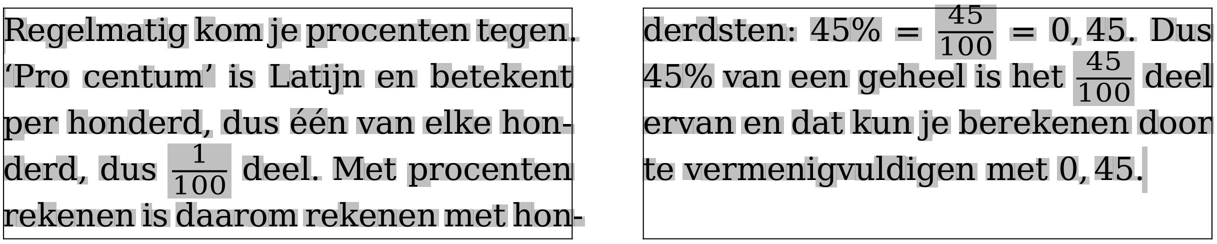
Regelmatig kom je procenten tegen. 'Pro centum' is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: 45% $= \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

none on grid



Regelmatig kom je procenten tegen. 'Pro centum' is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: 45% $= \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

strict on grid



Regelmatig kom je procenten tegen. 'Pro centum' is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: 45% $= \frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

fixed on grid

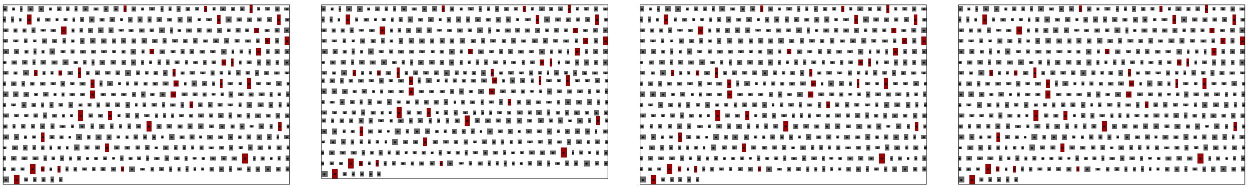
Figure 4.7 Boxed columns without profile.

are hardly any combinations of design and content where micro-typography makes sense to use: in prose perhaps, but not in mixed content. On the other hand, profiling makes more sense in mixed content than in prose.

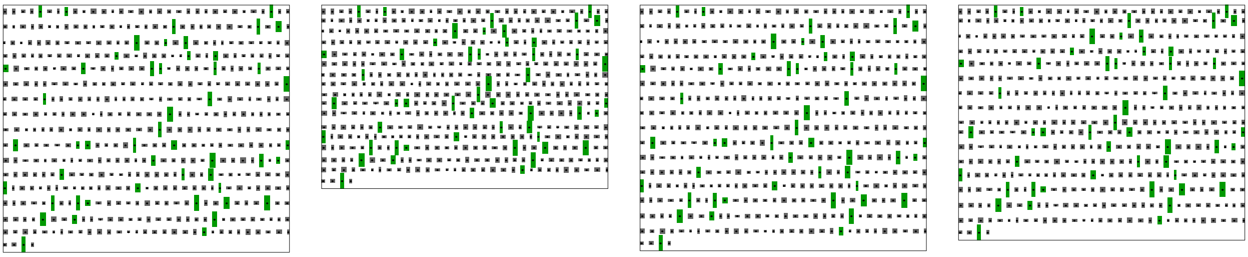
Not everything that is possible should be used. In figure 4.8 we show some fake paragraphs with profiles applied, the first series (random range 2) has a few excessive snippets, the last one (random range 5) has many. In figure 4.9 we show them in a different arrangement. Although there are differences it is hard to say if the results look better. We scaled down the results and used gray fake blurs instead of real text in order to get a better impression of the so-called (overall) grayness of a text.

4.5 Conclusion

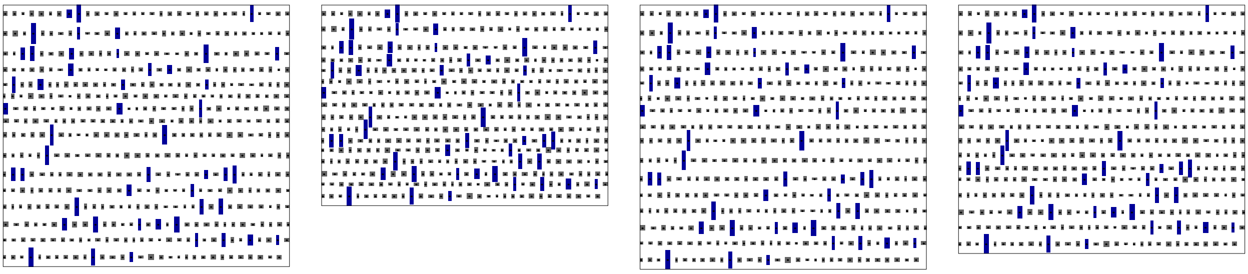
Although profiling seems interesting, in practice it does not have much value in an automated flow. Ultimately, in the project for which I inves-



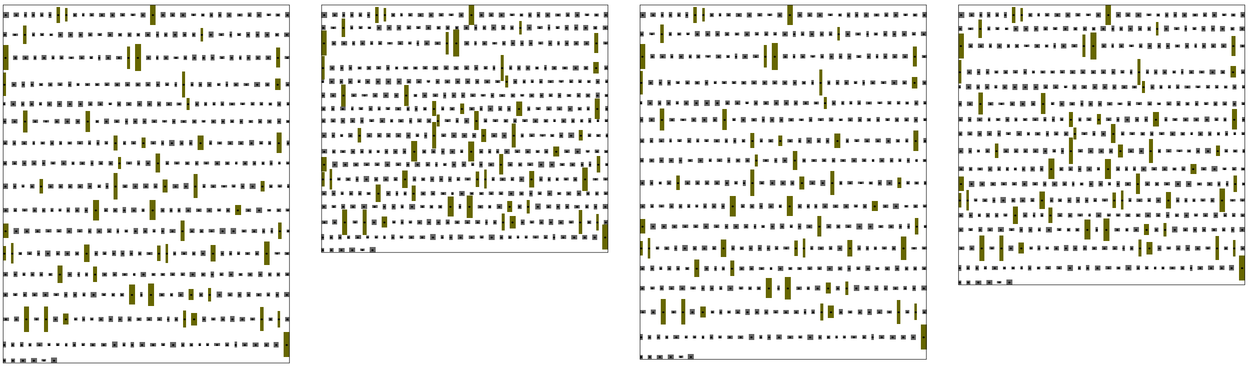
none / 2 strict / 2 fixed / 2 halffixed / 2



none / 3 strict / 3 fixed / 3 halffixed / 3



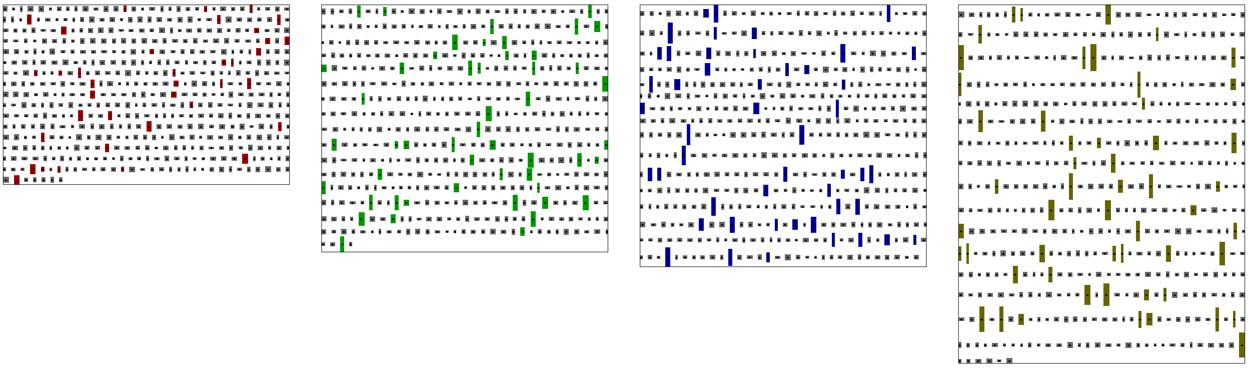
none / 4 strict / 4 fixed / 4 halffixed / 4



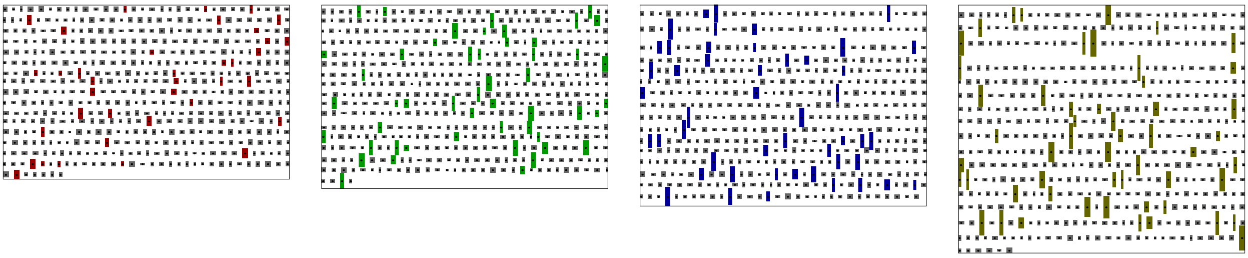
none / 5 strict / 5 fixed / 5 halffixed / 5

Figure 4.8 Some examples, each row has progressively more excessive snippets.

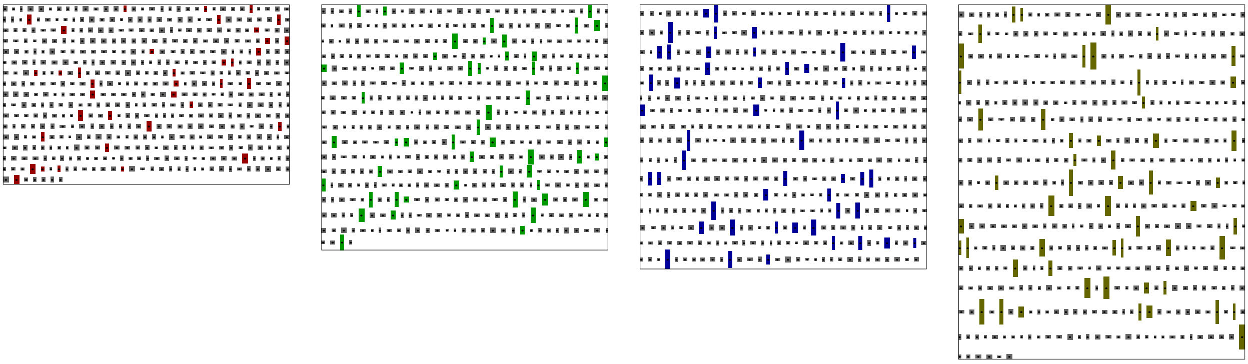
tigated this trickery, only in the final stage was some last minute optimization of the rendering done. We did that by injecting directives. Think of page breaks that make the result look more balanced. Optimizing image placement happens in an earlier stage because the text can refer to images like “in the picture on the left, we see …”. Controlling profiles is much harder. In fact, the more clever we are, the harder it gets to beat it when we want an exception. All these mechanisms: spacing, snapping, profiling,



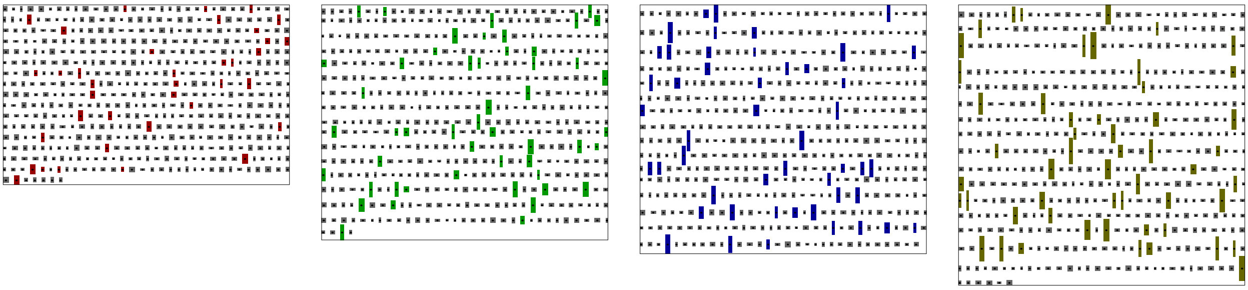
none / 2 none / 3 none / 4 none / 5



strict / 2 strict / 3 strict / 4 strict / 5



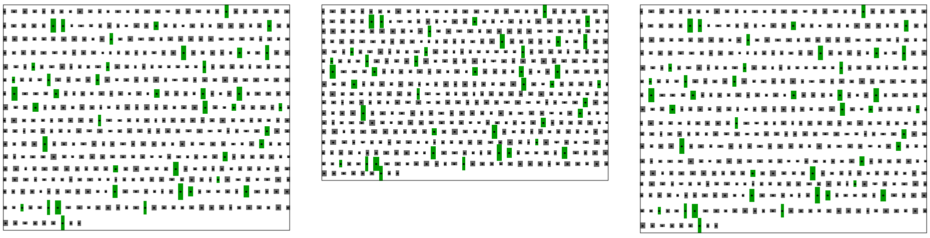
fixed / 2 fixed / 3 fixed / 4 fixed / 5



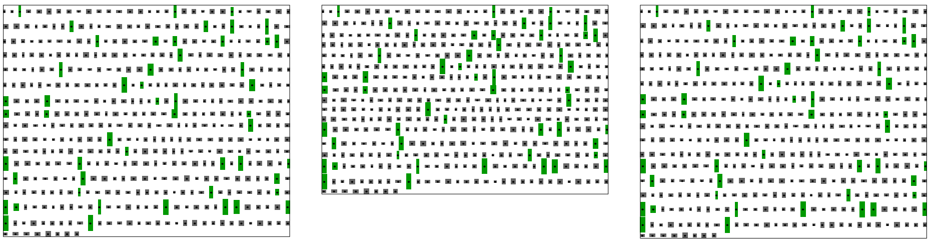
halffixed / 2 halffixed / 3 halffixed / 4 halffixed / 5

Figure 4.9 The same examples, rearranged such that each row has a different profiling variant.

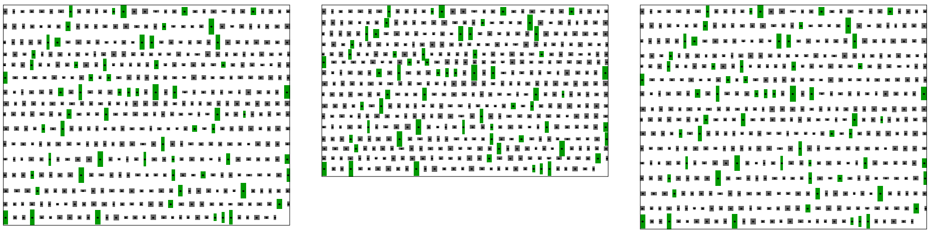
breaking pages, image placement, to mention a few, have to work together. For projects that depend on such placement, it might be better to write dedicated mechanisms than to try to fight with clever built-in features.



vbox 1 strict 1 fixed 1



vbox 2 strict 2 fixed 2



vbox 3 strict 3 fixed 3

Figure 4.10 Three similar random cases.

In practice, probably only the `fixed` alternative makes sense and as that one has a boundary condition similar to (or equal, depending on other settings) snapping on gridsteps, the end result might not be that different from doing nothing. In figure 4.10 you see that the vbox variant is not that bad. And extremely difficult content is unlikely to ever look perfect unless some manual intervention happens. Therefore, from the perspective of “fine points of text typesetting” some local (manual) control might be more interesting and relevant.

In the end, I didn’t need this profiling feature at all: because there are expectations with respect to how many pages a book should have, typesetting in columns was not needed. It didn’t save that many pages, and the result would never look that much better, simply because of the type of content. Large images were also spoiling the game. Nevertheless we will keep profiles in the core and it might even get extended. One question remains: at what point do we stop adding such features? The answer would be easier if TEX wasn’t so flexible.

5 Opentype math

5.1 Introduction

When TEX typesets mathematics it makes some assumptions about the properties of fonts and dimensions of glyphs. Due to practical limitations in the traditional eight-bit fonts, such as the number of available characters in a font and a limited number of heights and depths, some juggling takes place. For instance, TEX sometimes uses dimensions as a signal to treat some characters as special. This is not a problem as long as one knows how to make a font and in practice that was done by looking at the properties of Computer Modern to implement similar shapes. After all, there are not that many math fonts around and basically there is only one engine that can deal with them properly.

However, when Microsoft set the standard for OpenType math fonts it also steered the direction of their use in rendering mathematics. This means that the LuaTEX engine, which handles OpenType fonts, has to implement some alternative code paths. At the start, this involved a bit of gambling because there was no real specification; since then we now have a better picture. One of the more complex changes that took place is in the way italic correction is applied. A dirty way out of this dilemma would be to turn the math fonts into virtual ones that match traditional TEX properties, but this would not be a nice solution.

It must be noted that in the process of implementing support for the new fonts, Taco turned some noad types (see below) into a generic noad with a subtype. This simplified the transition. At the same time, a lot of detailed control was added in the way successive characters are spaced.

In LuaTEX pre 0.85, the italic correction was always added when a character got boxed (a frequently used preparation in the math builder). Now this is only done for the traditional fonts because, concerning italic correction, the OpenType standard states:[15]

1. When a run of slanted characters is followed by a straight character (such as an operator or a delimiter), the italics correction of the last glyph is added to its advance width.
2. When positioning limits on an N-ary operator (e.g., integral sign), the horizontal position of the upper limit is moved to the right by ½ of the

---

[15] Recently version 1.7 was published on the Microsoft website.

italics correction, while the position of the lower limit is moved to the left by the same distance.

3. When positioning superscripts and subscripts, their default horizontal positions are also different by the amount of the italics correction of the preceding glyph.

And, with respect to kerning:

4. Set the default horizontal position for the superscript as shifted relative to the position of the subscript by the italics correction of the base glyph.

I must admit that when the first implementation showed up, my natural reaction to unexpected behaviour was just to compensate it. One such solution was simply not to pass the italic correction to the engine and deal with it in Lua. In practice, that didn't work out well for all cases; one reason was that the engine saw the combination of old fonts as a new one and followed a mixed code path.[16] Another approach I tried was a mix of manipulated italic values and Lua, but finally as specifications settled I decided to leave it to the engine completely, if only because successive versions of LuaTEX behaved much better.

So, as we are closing in on the first release of LuaTEX, I decided to fix the pending issues and sat down to look at the math related code. I must admit that I had never looked in depth into that part of the machinery. In the next sections I will discuss some of the outcome of this exercise.

I will also discuss some extensions that have been on the agenda for years. They are rather generic and handy, but I must also admit that the MkIV code related to math has so many options to control rendering that I'm not sure if they will ever be used in ConTEXt. Nevertheless, these generic extensions fit will into the set of basic features of LuaTEX.

5.2 Italic correction

As stated above, the normal code path included italic correction in all the math boxes that are made. This meant that, in some places, this correction had to be removed and/or moved to another place in the chain. This is a natural side effect of the fact that TEX runs over the intermediate list of math nodes (noads) and turns them into regular nodes, mostly glyphs, kerns, glue and boxes.

---

[16] ConTEXt employed Unicode math right from the start of LuaTEX.

The complication is not so much these italic corrections themselves, because we could just continue to do the same, but the fact that these corrections are to be interpreted differently in case of integrals. There, the problem is that we have to (kind of) look back at what is done in order to determine what italic corrections are to be applied.

The original solution was to keep track of the applied correction via variables but that still made some analysis necessary. In the new implementation, more information is stored in the processed noads. This is a logical choice given that we have already added other information. It also makes it possible to fix cases that will (for sure) show up in the future.

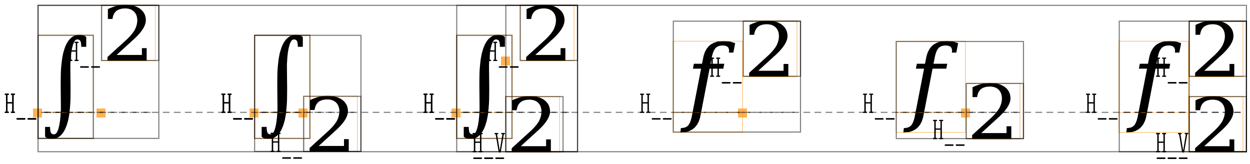


Figure 5.1 Some examples of italic correction (1).

In figure 5.1 we show two examples of inline italic correction. The superscripts are shifted to the right and the subscripts to the left. In the case of an integral sign, we need to move half the correction. This is triggered by the `\nolimits` primitive. In figure 5.2 we show the difference between just an integral character and one tagged as having limits.[17]

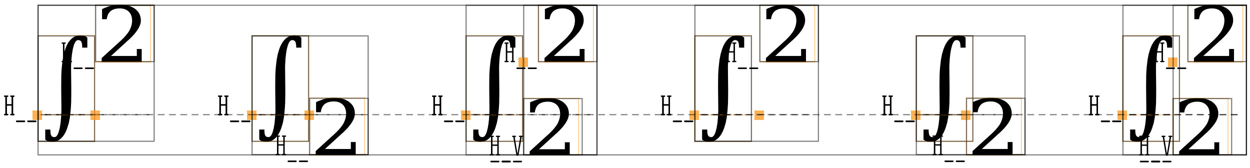


Figure 5.2 Some examples of italic correction (2).

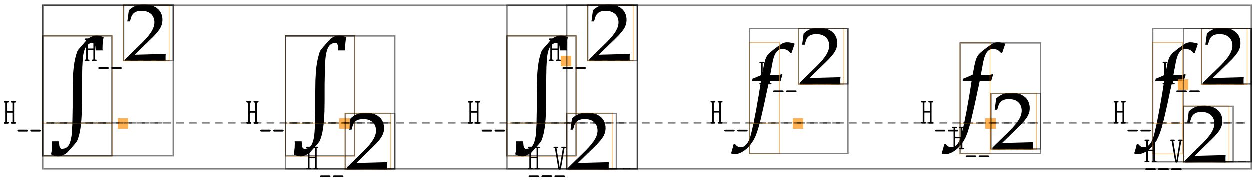
The amount of correction, if present at all, depends on the font, and in this document we use Dejavu math. Figure 5.3 shows a few variants. As you can see, the amount of correction is very font dependent.

5.3 Vertical delimiters

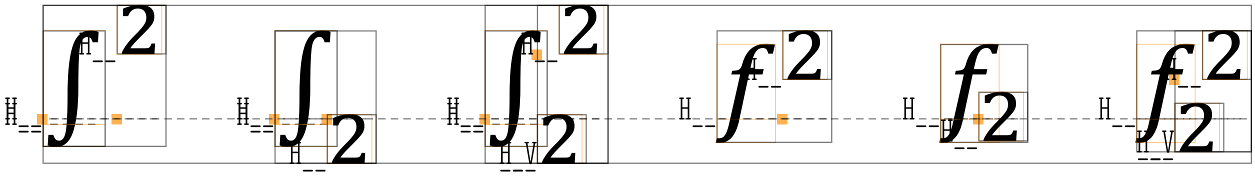
When we go into display math, there is a good chance that an integral has to be enlarged. The integral sign in Unicode has slot `0x222B`, so we can define a bigger one as follows:

---

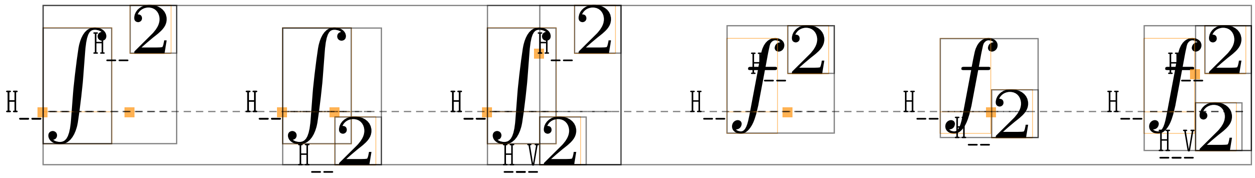
[17] We show some boxes so that you get an idea what TEX is doing. Basically TEX puts superscripts and subscripts on top of each other with some kern in between and then corrects the dimensions.



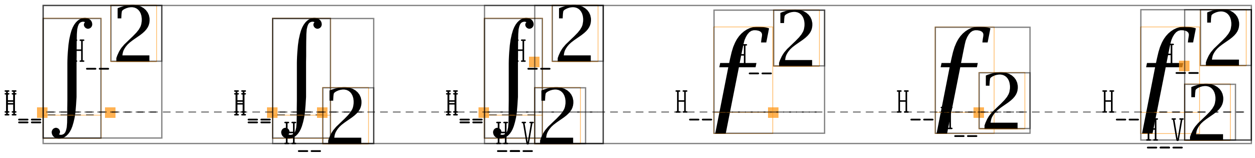
cambria



pagella



latin modern



lucida ot

Figure 5.3 Some examples of italic correction (3).

```
\def\standardint{
    \Umathchar "1 "0 "222B
}
\def\wrappedint{\mathop{
    \Umathchar "1 "0 "222B
}}
\def\biggerint{\mathop{
    \Uleft  height 3ex depth 3ex axis \Udelimiter "0 "0 "222B
    \Uright .
}}
\def\evenbiggerint{\mathop{
    \Uleft  height 6ex depth 6ex axis \Udelimiter "0 "0 "222B
    \Uright .
}}
```

The `axis` keyword will apply a shift up over the size of the current styles math axis. We use this in some examples as:

```
$
\displaystyle\standardint  ^a_b\enspace
\displaystyle\wrappedint   ^a_b\enspace
\displaystyle\biggerint    ^a_b\enspace
\displaystyle\evenbiggerint^a_b\enspace
$
```

In figure 5.4 you can see some subtle differences. The wrapped version doesn't shift the superscript and subscript. The reason is that the operator is hidden in its own wrapper and the scripts attach at an outer level. So, unless we start analyzing the innermost noad and apply that to the outer, we cannot know the shift. Such analyzing is asking for problems: where do we stop and what slight variations do we take into account? It's better to be predictable.

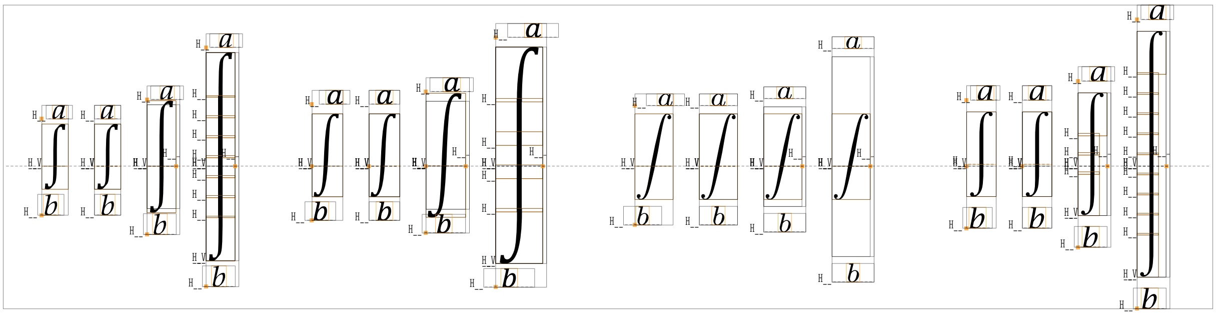


Figure 5.4 pagella, cambria, latin modern and lucida

Another observation is that Latin Modern does not provide (at least not yet) large integrals at all.

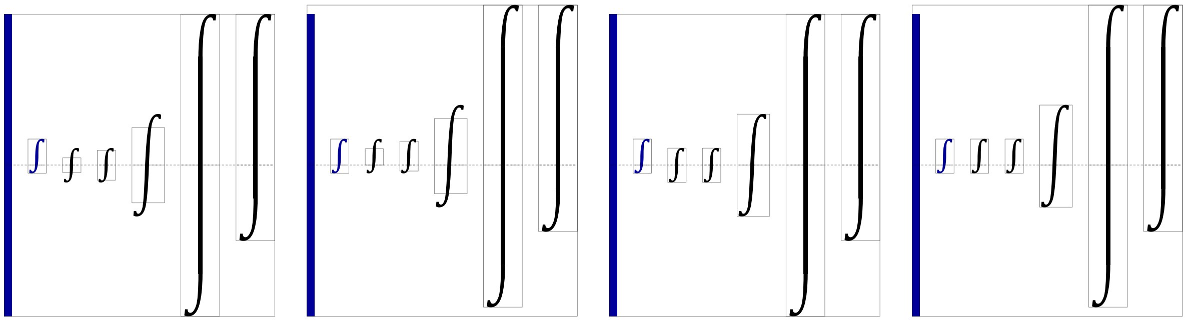
The following four cases are equivalent:

```
\Uleft   height 3ex depth 3ex axis \Udelimiter "0 "0 "222B
\Uright .

\Uleft  .
\Uright  height 3ex depth 3ex axis \Udelimiter "0 "0 "222B

\Uleft   .
\Umiddle height 3ex depth 3ex axis \Udelimiter "0 "0 "222B
\Uright  .

\Uleft   .
\Umiddle height 3ex depth 3ex axis \Udelimiter "0 "0 "222B
\Uright  .
```

However, because this all looks a bit clumsy, we now provide a new primitive:

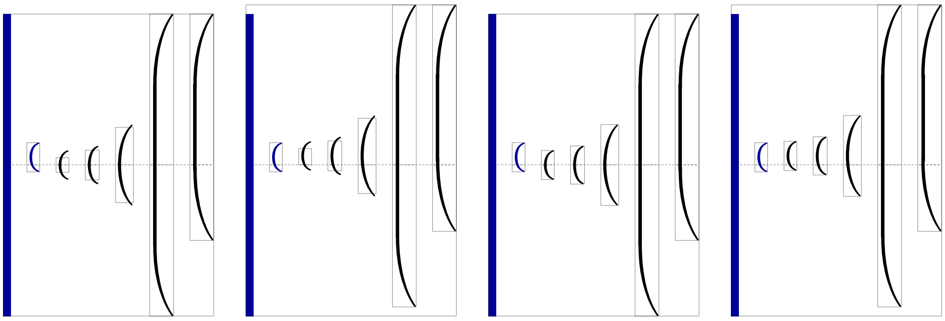
```
\Uvextensible
    height <dimension>
    depth <dimension>
    axis
    exact
    <delimiter>
```

The symbol to be constructed will have size `height` plus `depth`. When an `axis` is specified, the symbol will be shifted up, which is normally the case for such symbols. The keyword `exact` will correct the dimensions when no exact match is made, and this can be the case as long as we use the stepwise larger glyphs and before we end up using the composed shapes. When no dimensions are specified, the normal construction takes place and the only keyword that can be used then is `noaxis` which keeps the axis out of the calculations. After about a week of experimenting and exploring options, this combination made most sense, read: no fuzzy heuristics but predictable behaviour; after all, one might need different solutions for different fonts or circumstances and the applied logic (and expectations) can (and will, for sure) differ per macro package.



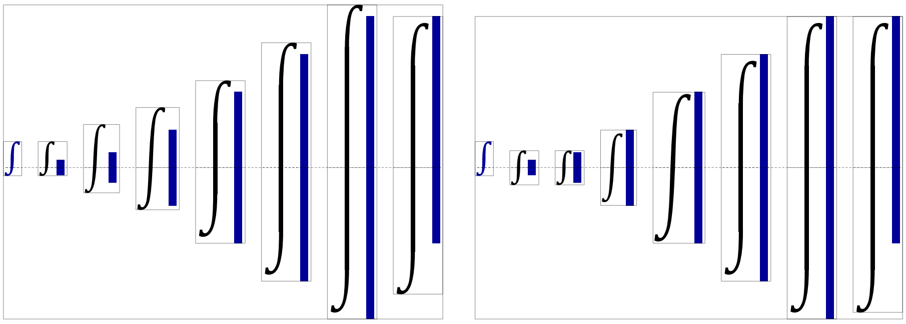
axis exact axis exact

Figure 5.5 cambria integrals with dimensions



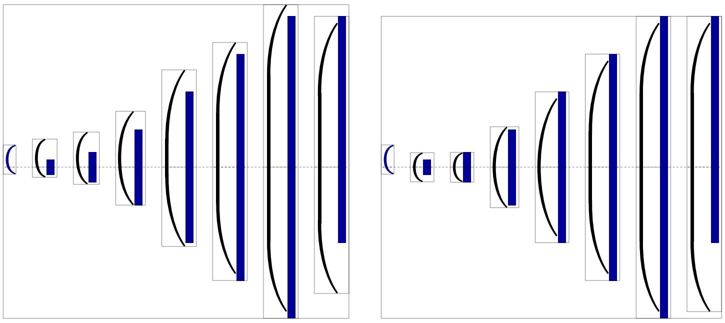
axis exact axis exact

Figure 5.6 cambria left parenthesis with dimensions



noaxis

Figure 5.7 cambria integrals adaptive



noaxis

Figure 5.8 cambria left parenthesis adaptive

5.4 Horizontal delimiters

Horizontal extenders also got some new options. Although one can achieve similar results with macros, the following might look a bit more natural. Also, some properties are lost once the delimiter is constructed, so macros can become complex when trying to determine the original dimensions involved.

We start with the new `\Uhextensible` primitive that accepts a dimension. It’s just a variant of the over and under delimiters with no content part.

```
\Uhextensible
    height <dimension>
    depth <dimension>
    left | middle | right
    <family>
    <slot>
```

So for example you can say:

```
$\Uhextensible width 30pt 0 "2194$
```

The `left`, `middle` and `right` keywords are only interpreted when the requested size can't be met due to stepwise larger glyph selection (i.e., before we start using arbitrary sizes made of snippets). Figure 5.9 shows what we get when we step from 2 up to 20 points with increments of 2 points in cambria.

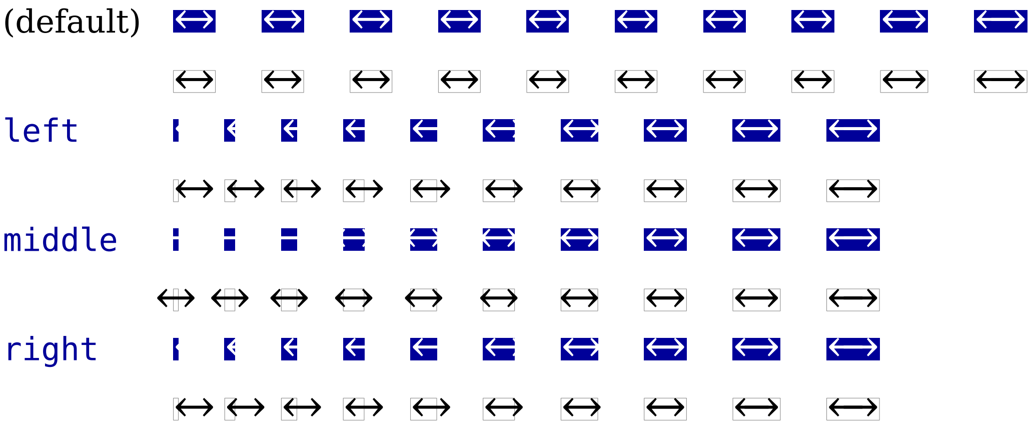


Figure 5.9 Stepwise wider `\Uhextensible` with options (cambria).

The dimensions and options can also be given to the `\Uoverdelimiter`, `\Uunderdelimiter`, `\Udelimiterover` and `\Udelimiterunder` primitives. Figure 5.10 shows what happens when the delimiter is smaller than requested. The samples look like this:

```
$\Udelimiterunder width 1pt 0 "2194 {\hbox{\strut !}}
```

When no dimension is given the keywords are ignored as it makes no sense to mess with the extensible then.

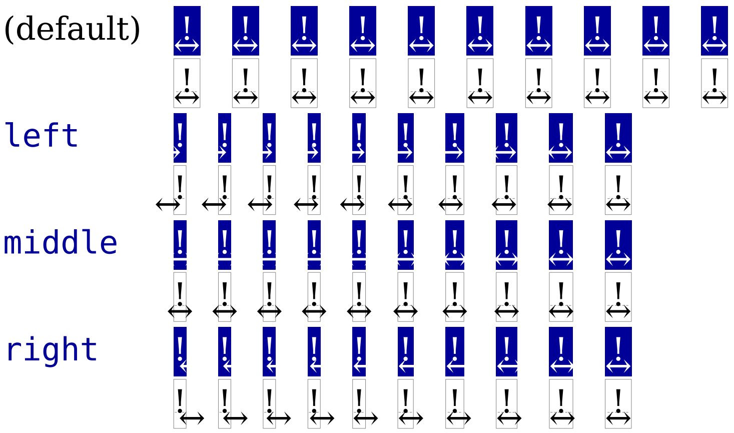


Figure 5.10 Stepwise wider `\Udelimiterunder` with options (cambria).

5.5 Accents

Already many years ago, I observed that overlaying characters (which happens when we negate an operator that has no composed glyph) didn't always give nice results and, therefore, a tracker item was created. When going over the todo list, I ran across a suggested patch by Khaled Hosny that added an overlay accent type. As the suggested solution fits in with the other extensions, a variant has been implemented.

The results really depend on the quality and completeness of the font, so here we will show xits. The placement of an `overlay` also depends on the top accent shift as specified in the font for the used glyph. Instead of a fixed criterion for trying to find the best match, an additional `fraction` (numerator) parameter can be specified. A value of 800 means that the target width is 800/1000.

The `\Umathaccent` command now has the following syntax:

```
\Umathaccent
    [top|bottom|overlay]
    [fixed]
    [fraction <number>]
    <delimiter>
    {content}
```

When we have an overlay, the fraction concerns the height; otherwise it concerns the width of the nucleus. In both cases, it is only applied when searching for stepwise larger glyphs, as extensibles are not influenced. An example of a specification is:

```
\Umathaccent
    overlay "0 "0 "0338
    fraction 950
    {\Umathchar"1"0"2211}
```

Figure 5.11 shows what we get when we use different fractions (from 800 upto 1500 with a step of 100). We see that `\overlay` is not always useful.

Normally you can forget about the factor because overlays make most sense for inline math, which uses relative small glyphs, so we can get $\not{x}$ $\not{x}$ $x\not{x}x$ with the following code:

```
$\Umathaccent overlay "0 "0 "0338 {x}$
```

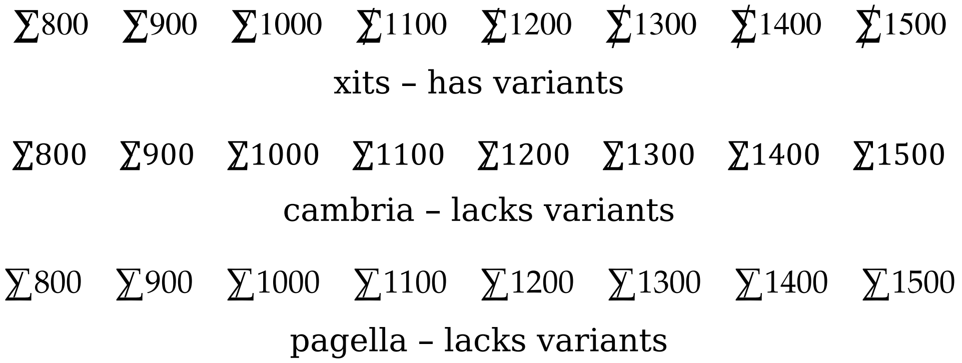
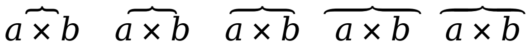


Figure 5.11 Using `overlay` in `\Umathaccent`.

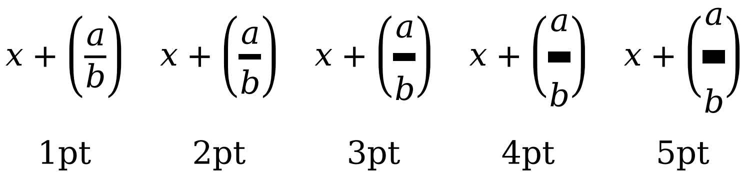
```
$\Umathaccent overlay "0 "0 "0338 {\tf x}$
$\Umathaccent overlay "0 "0 "0338 {\tf xxx}$
```

A normal accent can also be influenced by `fraction`:



5.6 Fractions

A normal fraction has a reasonable thick rule but as soon as you make it bigger you will notice a peculiar effect:



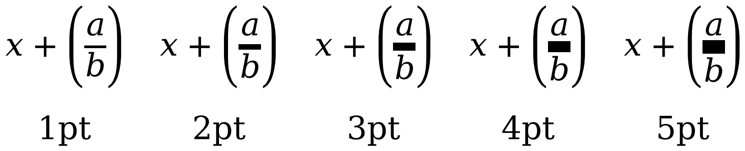
Such a fraction is specified as:

```
x + { {a} \abovewithdelims () 5pt {b} }
```

A new keyword `exact` will nil the excessive spacing:

```
x + { {a} \abovewithdelims () exact 5pt {b} }
```

Now we get:



One way to get consistent spacing in such fractions is to use struts:

```
x + { {\strut a} \abovewithdelims () exact 5pt {\strut b} }
```

Now we get:

$$x+\genfrac{(}{)}{1pt}{0}{a}{b} \quad x+\genfrac{(}{)}{2pt}{0}{a}{b} \quad x+\genfrac{(}{)}{3pt}{0}{a}{b} \quad x+\genfrac{(}{)}{4pt}{0}{a}{b} \quad x+\genfrac{(}{)}{5pt}{0}{a}{b}$$

1pt 2pt 3pt 4pt 5pt

Yet another way to increase the distance between the rule and text a bit is:

```
\Umathfractionnumvgap  \displaystyle4pt
\Umathfractiondenomvgap\displaystyle4pt
```

This looks quite consistent:

$$x+\genfrac{(}{)}{1pt}{0}{a}{b} \quad x+\genfrac{(}{)}{2pt}{0}{a}{b} \quad x+\genfrac{(}{)}{3pt}{0}{a}{b} \quad x+\genfrac{(}{)}{4pt}{0}{a}{b} \quad x+\genfrac{(}{)}{5pt}{0}{a}{b}$$

1pt 2pt 3pt 4pt 5pt

Here we use code like:

```
$\displaystyle x + {{a} \abovewithdelims() exact 2pt {b}}$
```

Using struts, it is best to zero the gap:

$$x+\genfrac{(}{)}{1pt}{0}{a}{b} \quad x+\genfrac{(}{)}{2pt}{0}{a}{b} \quad x+\genfrac{(}{)}{3pt}{0}{a}{b} \quad x+\genfrac{(}{)}{4pt}{0}{a}{b} \quad x+\genfrac{(}{)}{5pt}{0}{a}{b}$$

1pt 2pt 3pt 4pt 5pt

Here we use code like:

```
$\displaystyle x + {{\strut a} \abovewithdelims() exact 2pt {\strut
b}}$
```

5.7 Skewed fractions

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

The syntax used here is:

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

The options can be `noaxis` and `exact`, a combination of them or just nothing. By default we add half the axis to the shifts and also by default we zero the width of the middle character. For Latin Modern the result looks as follows:

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

5.8 Side effects

Not all bugs reported as such are really bugs. Here is one that came from a misunderstanding: In Eijkhout's "TEX by Topic", the rules for handling styles in scripts are described as follows:

- In any style superscripts and subscripts are taken from the next smaller style. Exception: in display style they are taken in script style.
- Subscripts are always in the cramped variant of the style; superscripts are only cramped if the original style was cramped.
- In an `..\over..` formula in any style the numerator and denominator are taken from the next smaller style.
- The denominator is always in cramped style; the numerator is only in cramped style if the original style was cramped.
- Formulas under a `\sqrt` or `\overline` are in cramped style.

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

default	$b_{x=xx}^{x=xx}$
script	$b_{x=xx}^{x=xx}$
crampedscript	$b_{x=xx}^{x=xx}$

This is coded like:

```
$b_{x=xx}^{x=xx}$
```

```
$b_{\scriptstyle x=xx}^{\scriptstyle x=xx}$
$b_{\crampedscriptstyle x=xx}^{\crampedscriptstyle x=xx}$
```

Now we set the following parameters

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

This gives:

default	$b_{x=xx}^{x\quad =x\quad x}$
script	$b_{x\quad =x\quad x}^{x\quad =x\quad x}$
crampedscript	$b_{x=xx}^{x=xx}$

Since the result is not what is expected (visually), we should say:

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

Now we get:

default	$b_{x\quad =x\quad x}^{x\quad =x\quad x}$
script	$b_{x\quad =x\quad x}^{x\quad =x\quad x}$
crampedscript	$b_{x\quad =x\quad x}^{x\quad =x\quad x}$

5.9 Fixed scripts

We have three parameters that are used for anchoring superscripts and subscripts, alone or in combinations.

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

When we set `\mathscriptsmode` to a value other than zero, these are used for calculating fixed positions. This is something that is needed in, for instance, chemical equations. You can manipulate the mentioned variables to achieve different effects, and the logic is shown in the following table. In order to see the differences in more detail, they are enlarged in figure 5.12.

mode	**down**	**up**	
0	dynamic	dynamic	$\mathrm{CH_2 + CH_2^+ + CH_2^2}$

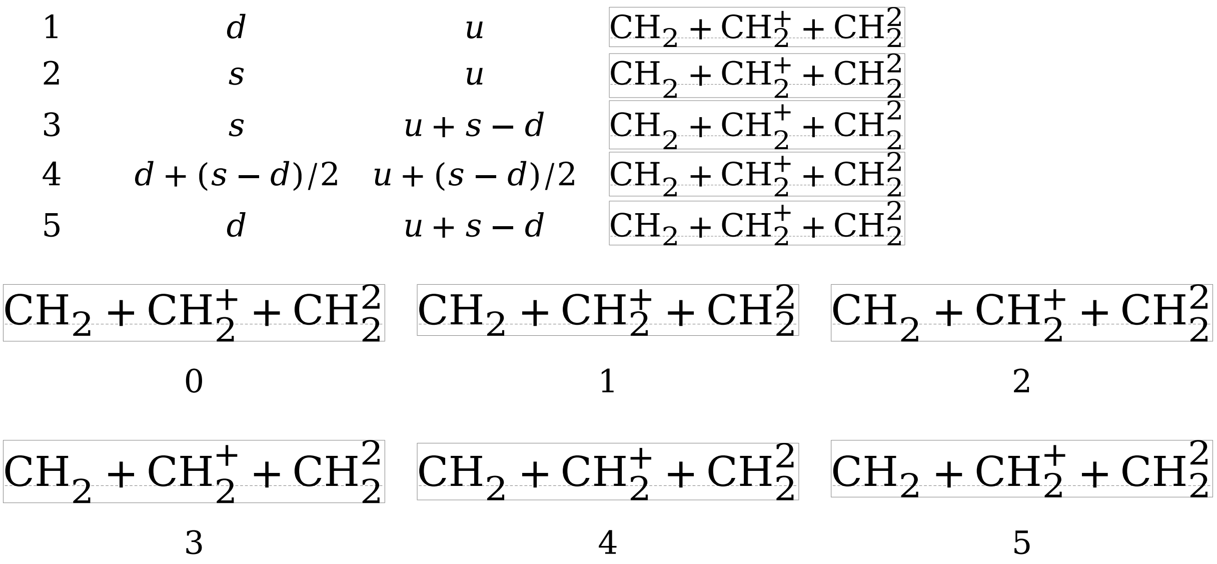


Figure 5.12 The effect of setting `\mathscriptsmode`.

5.10 Remark

The changes that we have made are hopefully not too intrusive. Instead of extending existing commands, new ones were introduced so that compatibility should not really be a problem. To some extend, these extensions violate the principle that extensions should be done in Lua, but TEX being a math renderer and OpenType replacing old font technology, we felt that we should make an exception here. Hopefully, not too many bugs were introduced.

6 Possibly useful extensions

6.1 Introduction

While working on LuaTEX, it is tempting to introduce all kinds of new fancy programming features. Arguments for doing this can be characterized by descriptions like ‘handy’, ‘speedup’, ‘less code’, ‘necessity’. It must be stated that traditional TEX is rather complete, and one can do quite a lot of macro magic to achieve many goals. So let us look a bit more at the validity of these arguments.

The ‘handy’ argument is in fact a valid one. Of course, one can always wrap clumsy code in a macro to hide the dirty tricks, but, still, it would be nicer to avoid needing to employ extremely dirty tricks. I found myself looking at old code wondering why something has to be done in such a complex way, only to realize, after a while, that it comes with the concept; one can get accustomed to it. After all, every programming language has its stronger and weaker aspects.

The ‘speedup’ argument is theoretically a good one too, but, in practice, it’s hard to prove that a speedup really occurs. Say we save 5% on a job. This is nice for multipass on a server where many jobs run at the same time or after each other, but a little bit of clever macro coding will easily gain much more. Or, as we often see: sloppy macro or style writing will easily negate those gains. Another pitfall is that you can measure (say) half a million calls to a macro can indeed be brought down to a fraction of its runtime thanks to some helper, but, in practice, you will not see that gain because saving 0.1 seconds on a 10 second run can be neglected. Furthermore, adding a single page to the document will already make such a gain invisible to the user as that will itself increase the runtime. Of course, many small speedups can eventually accumulate to yield a significant overall gain, but, if the macro package is already quite optimized, it might not be easy to squeeze out much more. At least in ConTEXt, I find it hard to locate bottlenecks that could benefit from extensions, unless one adds very specific features, which is not what we want.

Of course one can create ‘less’ code by using more wrappers. But this can definitely have a speed penalty, so this argument should be used with care. An appropriate extra helper can make wrappers fast and the fewer helpers the better. The danger is in choosing what helpers. A good criterion is that it should be hard otherwise in TEX. Adding more primitives (and overhead) merely because some macro package would like it would be bad practice. I'm confident that helpers for ConTEXt would not be that useful for plain TEX, LATEX, etc., and vice versa.

The 'necessity' argument is a strong one. Many already present extensions from ε-TEX fall into this category: fully expandable expressions (although the implementation is somewhat restricted), better macro protection, expansion control, and the ability to test for a so-called csname (control sequence name) are examples.

In the end, the only valid argument is 'it can't be done otherwise', which is a combination of all these arguments with 'necessity' being dominant. This is why in LuaTEX there are not that many extensions to the language (nor will there be). I must admit that even after years of working with TEX, the number of wishes for more facilities is not that large.

The extensions in LuaTEX, compared to traditional TEX, can be summarized as follows:

- Of course we have the ε-TEX extensions, and these already have a long tradition of proven usage. We did remove the limited directional support.

- From Aleph (follow-up on Omega), part of the directional support and some font support was inherited.

- From pdfTEX, we took most of the backend code, but it has been improved in the meantime. We also took the protrusion and expansion code, but especially the latter has been implemented a bit differently (in the frontend as well as in the backend).

- Some handy extensions from pdfTEX have been generalized; other obscure or specialized ones have been removed. So we now have frontend support for position tracking, resources (images) and reusable content in the core. The backend code has been separated a bit better and only a few backend-related primitives remain.

- The input encoding is now utf-8, exclusively, but one can easily hook in code to preprocess data that enters TEX's parser using Lua. The characteristic catcode settings for TEX can be grouped and switched efficiently.

- The font machinery has been opened wide so that we can use the embedded Lua interpreter to implement any technology that we might want, with the usual control that TEXies like. Some further limitations have been lifted. One interesting point is that one can now construct virtual fonts at runtime.

- Ligature construction, kerning and paragraph building have been separated as a side effect of Lua control. There are some extensions in that area. For instance, we store the language and min/max values in the glyph nodes, and we also store penalties with discretionaries. Patterns can be loaded at runtime, and character codes that influence hyphenation can be manipulated.
- The math renderer has been upgraded to support OpenType math. This has resulted in many new primitives and extensions, not only to define characters and spacing, but also to control placement of superscripts and subscripts and generally to influence the way things are constructed. A couple of mechanisms have gained control options.
- Several Lua interfaces are available making it possible to manipulate the (intermediate) results. One can pipe text to TEX, write parsers, mess with node lists, inspect attributes assigned at the TEX end, etc.

Some of the features mentioned above are rather LuaTEX specific, such as catcode tables and attributes. They are present as they permit more advanced Lua interfacing. Other features, such as utf-8 and OpenType math, are a side effect of more modern techniques. Bidirectional support is there because it was one of the original reasons for going forward with LuaTEX. The removal of backend primitives and thereby separating the code in a better way (see companion article) comes from the desire to get closer to the traditional core, so that most documentation by Don Knuth still applies. It's also the reason why we still speak of 'tokens', 'nodes' and 'noads'.

In the following sections I will discuss a few new low-level primitives. This is not a complete description (after all, we have reported on much already), and one can consult the LuaTEX manual to get the complete picture. The extensions described below are also relatively new and date from around version 0.85, the prelude to the stable version 1 release.

6.2 Rules

For insiders, it is no secret that TEX has no graphic capabilities, apart from the ability to draw rules. But with rules you can do quite a lot already. Add to that the possibility to insert arbitrary graphics or even backend drawing directives, and the average user won't notice that it's not true core functionality.

When we started with LuaTEX, we used code from pdfTEX and Omega (Aleph), and, as a consequence, we ended up with many whatsits. Normal running text has characters, kerns, some glue, maybe boxes, all represented by a limited set of so-called nodes. A whatsit is a kind of escape as it can be anything an extension to TEX needs to wrap up and put in the current list. Examples are (in traditional TEX already) whatsits that write to file (using `\write`) and whatsits that inject code into the backend (using `\special`). The directional mechanism of Omega uses whatsits to indicate direction changes.

For a long time images were also included using whatsits, and basically one had to reserve the right amount of space and inject a whatsit with a directive for the backend to inject something there with given dimensions or scale. Of course, one then needs methods to figure out the image properties, but, in the end, all of this could be done rather easily.

In pdfTEX, two new whatsits were introduced: images and reusable so-called forms, and, contrary to other whatsits, these do have dimensions. As a result, suddenly the TEX code base could no longer just ignore whatsits, but it had to check for these two when dimensions were important, for instance in the paragraph builder, packager, and backend.

So what has this to do with rules? Well, in LuaTEX all the whatsits are now back to where they belong, in the backend extension code. Directions are now first-class nodes, and we have native resources and reusable boxes. These resources and boxes are an abstraction of the pdfTEX images and forms, and, internally, they are a special kind of rule (i.e. a blob with dimensions). Because checking for rules is part of the (traditional) TEX kernel, we could simply remove the special whatsit code and let existing rule-related code do the job. This simplified the code a lot.

Because we suddenly had two more types of rules, we took the opportunity to add a few more.

```
\nohrule width 10cm height 2cm depth 0cm
\novrule width 10cm height 2cm depth 0cm
```

This is a way to reserve space, and it’s nearly equivalent to the following (respectively):

```
{\setbox0\hbox{}\wd0=10cm\ht0=2cm\dp0=0cm\box0\relax}
{\setbox0\vbox{}\wd0=10cm\ht0=2cm\dp0=0cm\box0\relax}
```

There is no real gain in efficiency because keywords also take time to parse, but the advantage is that no Lua callbacks are triggered.[18] Of course, this variant would not have been introduced had we still had just rules and no further subtypes; it was just a rather trivial extension that fit in the repertoire.[19]

So, while we were at it, yet another rule type was introduced, but this one has been made available only in Lua. As this text is about LuaTEX, a bit of Lua code does fit into the discussion, so here we go. The code shown here is rather generic and looks somewhat different in ConTEXt, but it does the job.

First, let’s create a straightforward rectangle drawing routine. We initialize some variables first, then scan properties using the token scanner, and, finally, we construct the rectangle using four rules. The packaged (so-called) hlist is written to TEX.

```
\startluacode
function FramedRule()
    local width     = 0
    local height    = 0
    local depth     = 0
    local linewidth = 0
    --
    while true do
        if token.scan_keyword("width") then
            width = token.scan_dimen()
        elseif token.scan_keyword("height") then
            height = token.scan_dimen()
        elseif token.scan_keyword("depth") then
            depth = token.scan_dimen()
        elseif token.scan_keyword("line") then
            linewidth = token.scan_dimen()
        else
            break
        end
    end
    local doublelinewidth = 2*linewidth
    --
```

---

[18] I still am considering adding variants of `\hbox` and `\vbox` where no callback would be triggered.

[19] This is one of the things I wanted to have for a long time but seems less useful today.

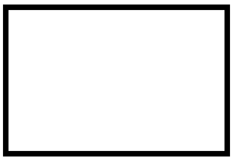
```
    local left    = node.new("rule")
    local bottom  = node.new("rule")
    local right   = node.new("rule")
    local top     = node.new("rule")
    local back    = node.new("kern")
    local list    = node.new("hlist")
    --
    left.width    = linewidth
    bottom.width  = width - doublelinewidth
    bottom.height = -depth + linewidth
    bottom.depth  = depth
    right.width   = linewidth
    top.width     = width - doublelinewidth
    top.height    = height
    top.depth     = -height + linewidth
    back.kern     = -width + linewidth
    list.list     = left
    list.width    = width
    list.height   = height
    list.depth    = depth
    list.dir      = "TLT"
    --
    node.insert_after(left,left,bottom)
    node.insert_after(left,bottom,right)
    node.insert_after(left,right,back)
    node.insert_after(left,back,top)
    --
    node.write(list)
 end
\stopluacode
```

This function can be wrapped in a macro:

```
\def\FrameRule{\directlua{FramedRule()}}
```

and the macro can be used as follows:

```
\FrameRule width 3cm height 1cm depth 1cm line 2pt
```

The result is: 

A different approach follows. Again, we define a rule, but, this time we only set dimensions and assign some attributes to it. Normally, one would reserve some attribute numbers for this purpose, but, for our example here, high numbers are safe enough. Now there is no need to wrap the rule in a box.

```
\startluacode
function FramedRule()
    local width     = 0
    local height    = 0
    local depth     = 0
    local linewidth = 0
    local radius    = 0
    local type      = 0
    --
    while true do
        if token.scan_keyword("width") then
            width = token.scan_dimen()
        elseif token.scan_keyword("height") then
            height = token.scan_dimen()
        elseif token.scan_keyword("depth") then
            depth = token.scan_dimen()
        elseif token.scan_keyword("line") then
            linewidth = token.scan_dimen()
        elseif token.scan_keyword("type") then
            type = token.scan_int()
        elseif token.scan_keyword("radius") then
            radius = token.scan_dimen()
        else
            break
        end
    end
    --
    local r   = node.new("rule")
    r.width   = width
    r.height  = height
    r.depth   = depth
    r.subtype = 4 -- user rule
    r[20000]  = type
    r[20001]  = linewidth
    r[20002]  = radius or 0
    node.write(r)
end
\stopluacode
```

Nodes with subtype 4 (user) are intercepted and passed to a callback function, when set. Here we show a possible implementation:

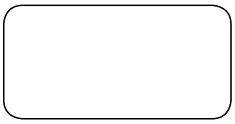
```
\startluacode
local bpfactor = (7200/7227)/65536

local f_rectangle = "%f w 0 0 %f %f re %s"

local f_radtangle = [[
    %f w %f 0 m
    %f 0 l %f %f %f %f y
    %f %f l %f %f %f %f y
    %f %f l %f %f %f %f y
    %f %f l %f %f %f %f y
    h %s
]]

callback.register("process_rule",function(n,h,v)
    local t = n[20000] == 0 and "f" or "s"
    local l = n[20001] * bpfactor -- linewidth
    local r = n[20002] * bpfactor -- radius
    local w = h * bpfactor
    local h = v * bpfactor
    if r > 0 then
        p = string.format(f_radtangle,
            l, r, w-r, w,0,w,r, w,h-r, w,h,w-r,h,
            r,h, 0,h,0,h-r, 0,r, 0,0,r,0, t)
    else
        p = string.format(f_rectangle, l, w, h, t)
    end
    pdf.print("direct",p)
end)
\stopluacode
```

We can now also specify a radius and type, where `0` is a filled and `1` a stroked shape.

```
\FrameRule
```

```
type   1
width  3cm
height 1cm
depth  5mm
line   0.2mm
radius 2.5mm
```

Since we specified a radius we get round corners:

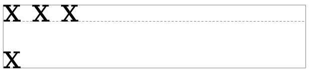


The nice thing about these extensions to rules is that the internals of TEX are not affected much. Rules are just blobs with dimensions and the par builder, for instance, doesn't care what they are. There is no need for further inspection. Maybe future versions of LuaTEX will provide more useful subtypes.

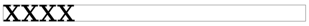
6.3 Spaces

Multiple successive spaces in TEX are normally collapsed into one. But, what if you don't want any spaces at all? It turns out this is rather hard to achieve. You can, of course, change the catcodes, but that won't work well if you pass text around as macro arguments. Also, you would not want spaces that separate macros and text to be ignored, but only those in the typeset text. For such use, LuaTEX introduces `\nospaces`.

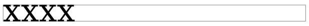
This new primitive can be used to overrule the usual `\spaceskip`-related heuristics when a space character is seen in a text flow. The value `1` specifies no injection, a value of `2` results in injection of a zero skip, and the default `0` gets the standard behavior. Below we see the results for four characters separated by spaces.



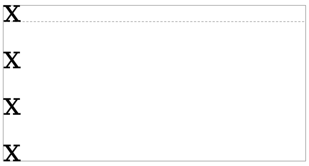
`0 / hsize 10mm`



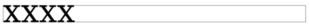
`1 / hsize 10mm`



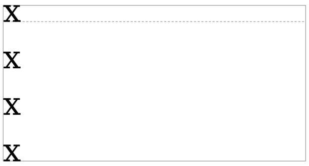
`2 / hsize 10mm`



`0 / hsize 1mm`



`1 / hsize 1mm`



`2 / hsize 1mm`

In case you wonder why setting the space related skips to zero is not enough: even when it is set to zero you will always get something. What gets inserted depends on `\spaceskip`, `\xspaceskip`, `\spacefactor` and font dimensions. I must admit that I always have to look up the details, as, normally, it's wrapped up in a spacing system that you implement once then forget about. In any case, with `\nospaces`, you can completely get rid of even an inserted zero space.

6.4 Token lists

The following four primitives are provided because they are more efficient than macro-based variants: `\toksapp`, `\tokspre`, and `\e...` (expanding) versions of both. They can be used to append or prepend tokens to a token register.

However, don't overestimate the gain that can be brought in simple situations with not that many tokens involved (read: there is no need to instantly change all code that does it the traditional way). The new method avoids saving tokens in a temporary register. Then, when you combine registers (which is also possible), the source gets appended to the target and, afterwards, the source is emptied: we don't copy but combine!

Their use can best be demonstrated by examples. We employ a scratch register `\ToksA`. The examples here show the effects of grouping; in fact, they were written for testing this effect. Because we don't use the normal assignment code, we need to initialize a local copy in order to get the original content outside the group.

```
\ToksA{}
\bgroup
   \ToksA{}
   \bgroup \toksapp\ToksA{!!} [\the\ToksA=!!] \egroup
   [\the\ToksA=]
\egroup
[\the\ToksA=]
```

result: `[!!=!!][=][=]`

```
\ToksA{}
\bgroup
    \ToksA{A}
    \bgroup \toksapp\ToksA{!!} [\the\ToksA=A!!] \egroup
    [\the\ToksA=A]
\egroup
```

```
[\the\ToksA=]
```

result: `[A!!=A!!][A=A][=]`

```
\ToksA{}
\bgroup
    \ToksA{}
    \bgroup
        \ToksA{A} \toksapp\ToksA{!!} [\the\ToksA=A!!]
    \egroup
    [\the\ToksA=]
\egroup
[\the\ToksA=]
```

result: `[A!!=A!!][=][=]`

```
\ToksA{}
\bgroup
    \ToksA{A}
    \bgroup
        \ToksA{} \toksapp\ToksA{!!} [\the\ToksA=!!]
    \egroup
    [\the\ToksA=A]
\egroup
[\the\ToksA=]
```

result: `[!!=!!][A=A][=]`

```
\ToksA{}
\bgroup
    \ToksA{}
    \bgroup
        \tokspre\ToksA{!!} [\the\ToksA=!!]
    \egroup
   [\the\ToksA=]
\egroup
[\the\ToksA=]
```

result: `[!!=!!][=][=]`

```
\ToksA{}
\bgroup
    \ToksA{A}
    \bgroup
        \tokspre\ToksA{!!} [\the\ToksA=!!A]
    \egroup
    [\the\ToksA=A]
\egroup
[\the\ToksA=]
```

result: `[!!A=!!A][A=A][=]`

```
\ToksA{}
\bgroup
    \ToksA{}
    \bgroup
        \ToksA{A} \tokspre\ToksA{!!} [\the\ToksA=!!A]
    \egroup
    [\the\ToksA=]
\egroup
[\the\ToksA=]
```

result: `[!!A=!!A][=][=]`

```
\ToksA{}
\bgroup
    \ToksA{A}
    \bgroup
        \ToksA{} \tokspre\ToksA{!!} [\the\ToksA=!!]
    \egroup
    [\the\ToksA=A]
\egroup
[\the\ToksA=]
```

result: `[!!=!!][A=A][=]`

Here we used `\toksapp` and `\tokspre`, but there are two more primitives, `\etoksapp` and `\etokspre`; these expand the given content while it gets added.

The next example demonstrates that you can also append another token list. In this case the original content is gone after an append or prepend.

```
\ToksA{A}
\ToksB{B}
\toksapp\ToksA\ToksB
```

```
\toksapp\ToksA\ToksB
[\the\ToksA=AB]
```

result: `[AB=AB]`

This is intended behaviour! The original content of the source is not copied but really appended or prepended. Of course, grouping works well.

```
\ToksA{A}
\ToksB{B}
\bgroup
    \toksapp\ToksA\ToksB
    \toksapp\ToksA\ToksB
    [\the\ToksA=AB]
\egroup
[\the\ToksA=AB]
```

result: `[AB=AB][AB=AB]`

6.5 Active characters

We now enter an area of very dirty tricks. If you have read the TEX book or listened to talks by TEX experts, you will, for sure, have run into the term ‘active’ characters. In short, it boils down to this: each character has a catcode and there are 16 possible values. For instance, the backslash normally has catcode zero, braces have values one and two, and normal characters can be 11 or 12. Very special are characters with code 13 as they are ‘active’ and behave like macros. In Plain TEX, the tilde is one such active character, and it’s defined to be a ‘non-breakable space’. In ConTEXt, the vertical bar is active and used to indicate compound and fence constructs.

Below is an example of a definition:

```
\catcode`A=13
\def A{B}
```

This will make the `A` into an active character that will typeset a `B`. Of course, such an example is asking for problems since any `A` is seen that way, so a macro name that uses one will not work. Speaking of macros:

```
\def\whatever
  {\catcode`A=13
```

```
   \def A{B}}
```

This won't work out well. When the macro is read it gets tokenized and stored and at that time the catcode change is not yet done so when this macro is called the A is frozen with catcode letter (11) and the `\def` will not work as expected (it gives an error). The solution is this:

```
\bgroup
\catcode`A=13
\gdef\whatever
  {\catcode`A=13
   \def A{B}}
\egroup
```

Here we make the `A` active before the definition and we use grouping because we don't want that to be permanent. But still we have a hard-coded solution, while we might want a more general one that can be used like this:

```
\whatever{A}{B}
\whatever{=}{{\bf =}}
```

Here is the definition of `whatever`:

```
\bgroup
\catcode`~=13
\gdef\whatever#1#2%
  {\uccode`~=`#1\relax
   \catcode`#1=13
   \uppercase{\def\tempwhatever{~}}%
   \expandafter\gdef\tempwhatever{#2}}
\egroup
```

If you read backwards, you can imagine that `\tempwhatever` expands into an active `A` (the first argument). So how did it become one? The trick is in the `\uppercase` (a `\lowercase` variant will also work). When casing an active character, TEX applies the (here) uppercase and makes the result active too.

We can argue about the beauty of this trick or its weirdness, but it is a fact that for a novice user this indeed looks more than a little strange. And so, a new primitive `\letcharcode` has been introduced, not so much out of necessity but simply driven by the fact that, in my opinion, it looks more natural. Normally the meaning of the active character can be put in its own macro, say:

```
\def\MyActiveA{B}
```

We can now directly assign this meaning to the active character:

```
\letcharcode`A=\MyActiveA
```

Now, when `A` is made active this meaning kicks in.

```
\def\whatever#1#2%
  {\def\tempwhatever{#2}%
   \letcharcode`#1\tempwhatever
   \catcode`#1=13\relax}
```

We end up with less code but, more important, it is easier to explain to a user and, in my eyes, it looks less obscure, too. Of course, the educational gain here wins over any practical gain because a macro package hides such details and only implements such an active character installer once.

6.6 \csname and friends

You can check for a macro being defined as follows:

```
\ifdefined\foo
    do something
\else
    do nothing
\fi
```

which, of course, can be obscured to:

```
do \ifdefined\foo some\else no\fi thing
```

A bit more work is needed when a macro is defined using `\csname`, in which case arbitrary characters (like spaces) can be used:

```
\ifcsname something or nothing\endcsname
    do something
\else
    do nothing
\fi
```

Before ε-TEX, this was done as follows:

```
\expandafter\ifx\csname something or nothing\endcsname\relax
```

```
    do nothing
\else
    do something
\fi
```

The `\csname` primitive will do a lookup and create an entry in the hash for an undefined name that then defaults to `\relax`. This can result in many unwanted entries when checking potential macro names. Thus, ε-TEX's `\ifcsname` test primitive can be qualified as a 'necessity'.

Now take the following example:

```
\ifcsname do this\endcsname
    \csname do this\endcsname
\else\ifcsname do that\endcsname
    \csname do that\endcsname
\else
    \csname do nothing\endcsname
\fi\fi
```

If `do this` is defined, we have two lookups. If it is undefined and `do that` is defined, we have three lookups. So there is always one redundant lookup. Also, when no match is found, TEX has to skip to the `\else` or `\fi`. One can save a bit by uglifying this to:

```
\csname do%
    \ifcsname do this\endcsname this\else
    \ifcsname do that\endcsname that\else
                                nothing\fi\fi
\endcsname
```

This, of course, assumes that there is always a final branch. So let's get back to:

```
\ifcsname do this\endcsname
    \csname do this\endcsname
\else\ifcsname do that\endcsname
    \csname do that\endcsname
\fi\fi
```

As said, when there is some match, there is always one test too many. In case you think this might be slowing down TEX, be warned: it's hard to measure. But as there can be (m)any character(s) involved, including multibyte utf-8 characters or embedded macros, there is a bit of penalty in terms of parsing token lists and converting to utf strings used for the lookup. And, because TEX has to give an error message in case of troubles, the already-seen tokens are stored too.

So, in order to avoid this somewhat redundant operation of parsing, memory allocation (for the lookup string) and storing tokens, the new primitive `\lastnamedcs` is now provided:

```
\ifcsname do this\endcsname
    \lastnamedcs
\else\ifcsname do that\endcsname
    \lastnamedcs
\fi\fi
```

In addition to the (in practice, often negligible) speed gain, there are other advantages: TEX has less to skip, and although skipping is fast, it still isn't a nice side effect (also useful when tracing). Another benefit is that we don't have to type the to-be-looked-up text twice. This reduces the chance of errors. In our example we also save 16 tokens (taking 64 bytes) in the format file. So, there are enough benefits to gain from this primitive, which is not a specific feature, but just an extension to an existing mechanism.

It also works in this basic case:

```
\csname do this\endcsname
\lastnamedcs
```

And even this works:

```
\csname do this\endcsname
\expandafter\let\expandafter\dothis\lastnamedcs
```

And after:

```
\bgroup
\expandafter\def\csname do this\endcsname{or that}
\global\expandafter\let\expandafter\dothis\lastnamedcs
\expandafter\def\csname do that\endcsname{or this}
\global\expandafter\let\expandafter\dothat\lastnamedcs
\egroup
```

We can use `\dothis` that gives `or that` and `\dothat` that gives `or this`, so we have the usual freedom to be able to use something meant to make code clean for the creation of obscure code.

A variation on this is the following:

```
\begincsname do this\endcsname
```

This call will check if `\do this` is defined, and, if so, will expand it. However, when `\do this` is not found, it does not create a hash entry. It is equivalent to:

```
\ifcsname do this\endcsname\lastnamedcs\fi
```

but it avoids the `\ifcsname`, which is sometimes handy as these tests can interfere.

I played with variations like `\ifbegincsname`, but we then quickly end up with dirty code due to the fact that we first expand something and then need to deal with the following `\else` and `\fi`. The two above-mentioned primitives are non-intrusive in the sense that they were relatively easy to add without obscuring the code base.

As a bonus, LuaTEX also provides a variant of `\string` that doesn't add the escape character: `\csstring`. There is not much to explain to this:

```
\string\whatever<>\csstring\whatever
```

This gives: `\whatever<>whatever`.

The main advantage of these several new primitives is that a bit less code is needed and (at least for ConTEXt) leads to a bit less tracing output. When you enable `\tracingall` for a larger document or example, which is sometimes needed to figure out a problem, it's not much fun to work with the resulting megabyte (or sometimes even gigabyte) of output so the more we can get rid of, the better. This consequence is just an unfortunate side effect of the ConTEXt user interface with its many parameters. As said, there is no real gain in speed.

6.7 Packing

Deep down in TEX, horizontal and vertical lists eventually get packed. Packing of an `\hbox` involves:

1. ligature building (for traditional TEX fonts),
2. kerning (for traditional TEX fonts),

3. calling out to Lua (when enabled) and
4. wrapping the list in a box and calculating the width.

When a Lua function is called, in most cases, the location where it happens (group code) is also passed. But say that you try the following:

```
\hbox{\hbox{\hbox{\hbox foo}}}
```

Here we do all four steps, while for the three outer boxes, only the last step makes any sense. And it's not trivial to avoid the application of the Lua function here. Of course, one can assign an attribute to the boxes and use that to intercept, but it's kind of clumsy. This is why we now can say:

```
\hpack{\hpack{\hpack{\hbox foo}}}
```

There are also `\vpack` for a `\vbox` and `\tpack` for a `\vtop`. There can be a small gain in speed when many complex manipulations are done, although in, for instance, ConTEXt, we already have provisions for that. It's just that the new primitives are a cleaner way out of a conceptually nasty problem. Similar functions are available on the Lua side.

6.8 Errors

We end with a few options that can be convenient to use if you don't care about exact compatibility.

```
\suppresslongerror
\suppressmathparerror
\suppressoutererror
\suppressifcsnameerror
```

When entering your document on a paper teletype terminal, starting TEX, and then going home in order to have a look at the result the next day, it does make sense to catch runaway cases, like premature ending of a paragraph (using `\par` or equivalent empty lines), or potentially missing `$$`s. Nowadays, it's less important to catch such coding issues (and be more tolerant) because editing takes place on screen and running (and restarting) TEX is very fast.

The first two flags given above deal with this. If you set the first to any value greater than zero, macros not defined as `\long` (not accepting paragraph endings) will not complain about par tokens in arguments. The second setting permits and ignores empty lines (also pars) in math without reverting to dirty tricks. Both are handy when your content comes from places that are outside of your control. The job will not be aborted (or hang) because of an empty line.

The third setting suppresses the `\outer` directive so that macros that originally can only be used at the outer level can now be used anywhere. It's hard to explain the concept of outer (and the related error message) to a user anyway.

The last one is a bit special. Normally, when you use `\ifcsname` you will get an error when TEX sees something unexpandable or that can't be part of a name. But sometimes you might find it to be quite acceptable and can just consider the condition as false. When the fourth variable is set to non-zero, TEX will ignore this issue and try to finish the check properly, so basically you then have an `\iffalse`.

6.9 Final remarks

I mentioned performance a number of times, and it's good to notice that most changes discussed here will potentially be faster than the alternatives, but this is not always noticeable, in practice. There are several reasons.

For one thing, TEX is already highly optimized. It has speedy memory management of tokens and nodes and unnecessary code paths are avoided. However, due to extensions to the original code, a bit more happens in the engine than in decades past. For instance, Unicode fonts demand sparse arrays instead of fixed-size, 256-slot data structures. Handling utf involves more testing and construction of more complex strings. Directional typesetting leads to more testing and housekeeping in the frontend as well as the backend. More keywords to handle, for instance `\hbox`, result in more parsing and pushing back unmatched tokens. Some of the penalty has been compensated for through the changing of whatsits into regular nodes. In recent versions of LuaTEX, scanning of `\hbox` arguments is somewhat more efficient, too.

In any case, any speedup we manage to achieve, as said before, can easily become noise through inefficient macro coding or user's writing bad styles. And we're pretty sure that not much more speed can be squeezed out. To achieve higher performance, it's time to buy a machine with a faster cpu (and a huge cache), faster memory (lanes), an ssd, and regularly check your coding.

7 The LUATEX PDF backend

7.1 Introduction

The original design of TEX has a clear separation between the frontend and backend code. In principle, shipping out a page boils down to traversing the to-be-shipped-out box and translating the glyph, rule, glue, kern and list nodes into positioning just glyphs and rules on a canvas. The dvi backend is therefore relatively simple, as the dvi output format delegates to other programs the details of font inclusion and such into the final format; it just describes the pages.

Because we eventually want color and images as well, there is a mechanism to pass additional information to post-processing programs. One can insert `\special`s with directives like `insert image named foo.jpg`. The frontend as well as the backend are not concerned with what goes into a special; the dvi post-processor of course is.

The pdf backend, on the other hand, is more complex as it immediately produces the final typeset result and, as such, offers possibilities to insert verbatim code (`\pdfliteral`), images (`\pdfximage` cum suis), annotations, destinations, threads and all kinds of objects, reuse typeset content (`\pdfxform` cum suis); in the end, there are all kinds of `\pdf...` commands. The way these were implemented in LuaTEX prior to 0.82 violates the separation between frontend and backend, an inheritance from pdfTEX. Additional features such as protrusion and expansion add to that entanglement. However, because pdf is an evolving standard, occasionally we need to adapt the related code. A separation of code makes sure that the frontend can become stable (and hopefully frozen) at some point.[20]

In LuaTEX we had already started making this separation of specialized code, such as a cleaner implementation of font expansion, but all these `\pdf...` commands were still pervasive, leading to fuzzy dependencies, checks for backend modes, etc. so a logical step was to straighten all this out. That way we give LuaTEX a cleaner core constructed from traditional TEX, extended with ε-TEX, Aleph/Omega, and LuaTEX functionality.

---

[20] In practice nowadays, the backend code changes little, because the pdf produced by LuaTEX is rather simple and is easily adapted to the changing standard.

7.2 Extensions

A first step, then, was to transform generic (i.e. independent from the backend) functionality which was still (sort of) bound to Aleph and pdfTEX, into core functionality. A second step was to reorganize the backend specific pdf code, i.e. move it out of the core and into the group of extension commands. This extension group is somewhat special and originates in traditional TEX; it is the way to add your own functionality to TEX, the program.

As an example for future programmers, Don Knuth added four (connected) primitives as extensions: `\openout`, `\closeout`, `\write` and `\special`. The Aleph and pdfTEX engines, on the other hand, put some functionality in extensions and some in the core. This arose from the fact that dealing with variables in extensions is often inconvenient, as they are then seen as (unexpandable) commands instead of integers, token lists, etc. That the write-related commands are there is almost entirely due to being the demonstration of the mechanism; everything related to *reading* files is in the core. There is one property that perhaps forces us to keep the writers there, and that's the `\immediate` prefix.[21]

In the process of separating, we reshuffled the code base a bit; the current use of the extensions mechanism still suits as an example and also gives us backward compatibility. However, new backend primitives will not be added there but rather in specific plugins (if needed at all).

7.3 From whatsits to nodes

The pdf backend introduced two new concepts into the core: (reusable) images and (reusable) content (wrapped in boxes). In keeping with good TEX practice, these were implemented as whatsits (a node type for extensions); but this created, as a side effect, an anomaly in the handling of such nodes. Consider looping over a node list where we need to check dimensions of nodes; in Lua, we can write something like this:

```
while n do
    if n.id == glyph then
        -- wd ht dp
    elseif n.id == rule then
```

---

[21] Unfortunately we're stuck with `\immediate` in the backend; a `deferred` keyword would have been handier, especially since other backend-related commands can also be immediate.

```
        -- wd ht dp
    elseif n.id == kern then
        -- wd
    elseif n.id == glue then
        -- size stretch shrink
    elseif n.id == whatsits then
        if n.subtype == pdfxform then
            -- wd ht dp
        elseif n.subtype == pdfximage then
            -- wd ht dp
        end
    end
    n = n.next
end
```

So for each node in the list, we need to check these two whatsit subtypes. But as these two concepts are rather generic, there is no evident need to implement it this way. Of course the backend has to provide the inclusion and reuse, but the frontend can be agnostic about this. That is, at the input end, in specifying these two injects, we only have to make sure we pass the right information (so the scanner might differentiate between backends).

Thus, in LuaTeX these two concepts have been promoted to core features:

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

The index should be considered an arbitrary number set to whatever the backend plugin decides to use as an identifier. These are no longer whatsits, but a special type of rule; after all, TeX is only interested in dimensions. Given this change, the previous code can be simplified to:

```
while n do
    if n.id == glyph then
        -- wd ht dp
    elseif n.id == rule then
        -- wd ht dp
    elseif n.id == kern then
```

```
        -- wd
    elseif n.id == glue then
        -- size stretch shrink
    end
    n = n.next
end
```

The only consequence for the previously existing rule type (which, in fact, is also something that needs to be dealt with in the backend, depending on the target format) is that a normal rule now has subtype 0 while the box resource has subtype 1 and the image subtype 2.

If a package writer wants to retain the pdfTEX names, the previous table can be used; just prefix `\let`. For example, the first line would be (spaces optional, of course):

```
\let\pdfxform\saveboxresource
```

7.4 Direction nodes

A similar change has been made for ``direction'' nodes, which were also previously whatsits. These are now normal nodes so again, instead of consulting whatsit subtypes, we can now just check the id of a node.

It should be apparent that all of these changes from whatsits to normal nodes already greatly simplify the code base.

7.5 Promoted commands

Many more commands have been promoted to the core. Here is an additional list of original pdfTEX commands and their new counterparts (this time with the `\let` included):

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

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

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

\let\ifpdfabsnum       \ifabsnum
\let\ifpdfabsdim       \ifabsdim
\let\ifpdfprimitive    \ifprimitive

\let\pdfprimitive      \primitive

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

\let\pdftexversion     \luatexversion
\let\pdftexrevision    \luatexrevision
\let\pdftexbanner      \luatexbanner

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

\let\pdfpxdimen        \pxdimen

\let\pdfinsertht       \insertht
```

7.6 Backend commands

There are many commands that start with `\pdf` and, over the history of development of pdfTEX and LuaTEX, some have been added, some have been renamed, others removed. Instead of the many, we now have just one: `\pdfextension`. A couple of usage examples:

```
\pdfextension literal {1 0 0 2 0 0 cm}
\pdfextension obj     {/foo (bar)}
```

Here, we pass a keyword that tells for what to scan and what to do with it. A backward-compatible interface is easy to write. Although it delegates a bit more management of these `\pdf` commands to the macro package, the responsibility for dealing with such low-level, error-prone calls is there anyway. The full list of `\pdfextension`s is given here. The scanning after the keyword is the same as for pdfTEX.

```
\protected\def\pdfliteral        {\pdfextension literal }
\protected\def\pdfcolorstack     {\pdfextension colorstack }
\protected\def\pdfsetmatrix      {\pdfextension setmatrix }
\protected\def\pdfsave           {\pdfextension save\relax}
\protected\def\pdfrestore        {\pdfextension restore\relax}
\protected\def\pdfobj            {\pdfextension obj }
\protected\def\pdfrefobj         {\pdfextension refobj }
\protected\def\pdfannot          {\pdfextension annot }
\protected\def\pdfstartlink      {\pdfextension startlink }
\protected\def\pdfendlink        {\pdfextension endlink\relax}
\protected\def\pdfoutline        {\pdfextension outline }
\protected\def\pdfdest           {\pdfextension dest }
\protected\def\pdfthread         {\pdfextension thread }
\protected\def\pdfstartthread    {\pdfextension startthread }
\protected\def\pdfendthread      {\pdfextension endthread\relax}
\protected\def\pdfinfo           {\pdfextension info }
\protected\def\pdfcatalog        {\pdfextension catalog }
\protected\def\pdfnames          {\pdfextension names }
\protected\def\pdfincludechars   {\pdfextension includechars }
\protected\def\pdffontattr       {\pdfextension fontattr }
\protected\def\pdfmapfile        {\pdfextension mapfile }
\protected\def\pdfmapline        {\pdfextension mapline }
\protected\def\pdftrailer        {\pdfextension trailer }
\protected\def\pdfglyphtounicode{\pdfextension glyphtounicode }
```

7.7 Backend variables

As with commands, there are many variables that can influence the pdf backend. The most important one was, of course, that which set the output mode (`\pdfoutput`). Well, that one is gone and has been replaced by `\outputmode`. A value of 1 means that we produce pdf.

One complication of variables is that (if we want to be compatible), we need to have them as real TEX registers. However, as most of them are optional, an easy way out is simply not to define them in the engine. In order to be able to still deal with them as registers (which is backward compatible), we define them as follows:

```
\edef\pdfminorversion        {\pdfvariable minorversion}
\edef\pdfcompresslevel       {\pdfvariable compresslevel}
\edef\pdfobjcompresslevel    {\pdfvariable objcompresslevel}
```

```
\edef\pdfdecimaldigits       {\pdfvariable decimaldigits}

\edef\pdfhorigin             {\pdfvariable horigin}
\edef\pdfvorigin             {\pdfvariable vorigin}

\edef\pdfgamma               {\pdfvariable gamma}
\edef\pdfimageresolution     {\pdfvariable imageresolution}
\edef\pdfimageapplygamma     {\pdfvariable imageapplygamma}
\edef\pdfimagegamma          {\pdfvariable imagegamma}
\edef\pdfimagehicolor        {\pdfvariable imagehicolor}
\edef\pdfimageaddfilename    {\pdfvariable imageaddfilename}
\edef\pdfignoreunknownimages {\pdfvariable ignoreunknownimages}

\edef\pdfinclusioncopyfonts  {\pdfvariable inclusioncopyfonts}
\edef\pdfinclusionerrorlevel {\pdfvariable inclusionerrorlevel}
\edef\pdfpkmode              {\pdfvariable pkmode}
\edef\pdfpkresolution        {\pdfvariable pkresolution}
\edef\pdfgentounicode        {\pdfvariable gentounicode}

\edef\pdflinkmargin          {\pdfvariable linkmargin}
\edef\pdfdestmargin          {\pdfvariable destmargin}
\edef\pdfthreadmargin        {\pdfvariable threadmargin}
\edef\pdfformmargin          {\pdfvariable formmargin}

\edef\pdfuniqueresname       {\pdfvariable uniqueresname}
\edef\pdfpagebox             {\pdfvariable pagebox}
\edef\pdfpagesattr           {\pdfvariable pagesattr}
\edef\pdfpageattr            {\pdfvariable pageattr}
\edef\pdfpageresources       {\pdfvariable pageresources}
\edef\pdfxformattr           {\pdfvariable xformattr}
\edef\pdfxformresources      {\pdfvariable xformresources}
```

You can set them as follows (the values shown here are the initial values):

```
\pdfcompresslevel         9
\pdfobjcompresslevel      1
\pdfdecimaldigits         3
\pdfgamma              1000
\pdfimageresolution      71
\pdfimageapplygamma       0
\pdfimagegamma         2200
\pdfimagehicolor          1
```

```
\pdfimageaddfilename      1
\pdfpkresolution         72
\pdfinclusioncopyfonts    0
\pdfinclusionerrorlevel   0
\pdfignoreunknownimages   0
\pdfreplacefont           0
\pdfgentounicode          0
\pdfpagebox               0
\pdfminorversion          4
\pdfuniqueresname         0

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

Their removal from the frontend has helped again to clean up the code and, by making them registers, their use is still compatible. A call to `\pdfvariable` defines an internal register that keeps the value (of course this value can also be influenced by the backend itself). Although they are real registers, they live in a protected namespace:

```
\meaning\pdfcompresslevel
```

which gives:

```
\count1130
```

It’s perhaps unfortunate that we have to remain compatible because a setter and getter would be much nicer. I am still considering writing the extension primitive in Lua using the token scanner, but it might not be possible to remain compatible then. This is not so much an issue for ConTeXt that always has had backend drivers, but, rather, for other macro packages that have users expecting the primitives (or counterparts) to be available.

7.8 Backend feedback

The backend can report on some properties that were also accessible via `\pdf...` primitives. Because these are read-only variables, another primitive now handles them: `\pdffeedback`. This primitive can be used to define compatible alternatives:

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

The variables are internal, so they are anonymous. When we ask for the meaning of some that were previously defined:

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

we will get, similar to the above:

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

7.9 Removed primitives

Finally, here is the list of primitives that have been removed, with no TEX-level equivalent available. Many were experimental, and they can be easily be provided to TEX using Lua.

```
\knaccode
\knbccode
\knbscode
\pdfadjustinterwordglue
\pdfappendkern
\pdfeachlinedepth
\pdfeachlineheight
\pdfelapsedtime
\pdfescapehex
\pdfescapename
\pdfescapestring
\pdffiledump
\pdffilemoddate
\pdffilesize
\pdffirstlineheight
\pdfforcepagebox
\pdfignoreddimen
\pdflastlinedepth
\pdflastmatch
\pdflastximagecolordepth
\pdfmatch
\pdfmdfivesum
```

`\pdfmovechars`
`\pdfoptionalwaysusepdfpagebox`
`\pdfoptionpdfinclusionerrorlevel`
`\pdfprependkern`
`\pdfresettimer`
`\pdfshellescape`
`\pdfsnaprefpoint`
`\pdfsnapy`
`\pdfsnapycomp`
`\pdfstrcmp`
`\pdfunescapehex`
`\pdfximagebbox`
`\shbscode`
`\stbscode`

7.10 Conclusion

The advantage of a clean backend separation, supported by just the three primitives `\pdfextension`, `\pdfvariable` and `\pdffeedback`, as well as a collection of registers, is that we can now further clean the code base, which remains a curious mix of combined engine code, sometimes and sometimes not converted to C from Pascal. A clean separation also means that if someone wants to tune the backend for a special purpose, the frontend can be left untouched. We will get there eventually.

All the definitions shown here are available in the file `luatex-pdf.tex`, which is part of the ConTEXt distribution.

8 LUATEX going stable

8.1 Introduction

We're closing in on version 1.0 of LuaTEX and at the time of this writing (mid April 2016) we're at version 0.95. The last decade we've reported on a regular basis about progress in user group journals, ConTEXt related documents and the LuaTEX manual and it makes no sense to repeat ourselves.

So where do we stand now? I will not go into details about what is available in LuaTEX, for that you consult the manual but will stick to the larger picture instead.

8.2 What is it

First of all, as the name suggests, LuaTEX has the Lua scripting engine on board. Currently we're still at version 5.2 and the reason for not going 5.3 is mainly because it has a different implementation of numbers and we cannot foresee side effects. We will test this when we move on to LuaTEX version 2.0.

The second part of the name indicates that we have some kind of TEX and we think we managed to remain largely compatible with the traditional engine. We took most of ε-TEX, much of pdfTEX and some from Aleph (Omega). On top of that we added a few new primitives and extended others.

If you look at the building blocks of TEX, you can roughly recognize these:

- an input parser (tokenizer) that includes macro expansion; its working is well described, of course in the TEX book, but more than three decades of availability has made TEX's behaviour rather well documented
- a list builder that links basic elements like characters (tagged with font information), rules, boxes, glue and kerns together in a double linked list of so called nodes (and noads in intermediate math lists)
- a language subsystem that is responsible for hyphenating words using so called patterns and exceptions
- a font subsystem that provides information about glyphs properties, and that also makes it possible to construct math symbols from snippets; it also makes sure that the backend knows what to embed
- a paragraph builder that breaks a long list into lines and a page builder that splits of chunks that can be wrapped into pages; this is all done within given constraints using a model of rewards and penalties
- a first class math renderer that set the standard and has inspired modern math font technology

- mechanisms for dealing with floating data, marking page related info, wrapping stuff in boxes, adding glue, penalties and special information
- a backend that is responsible for wrapping everything typeset in a format that can be printed and viewed

So far we're still talking of a rather generic variant of TEX with Lua as extension language. Next we zoom in on some details.

8.3 Where it differs

Given experiences with discussing extensions to the engine and given the fact that there is never really an agreement about what makes sense or not, the decission was made to not extend the engine any more than really needed but to provide hooks to do that in Lua. And, time has proven that this is a feasible approach. On the one hand we are as good as possible faithful to the original, and at the same time we can deal with todays and near future demands.

Tokenization still happens as before but we can also write input parsers ourselves. You can intercept the raw input when it gets read from file, but you can also create scanners that you can sort of plug into the parser. Both are a compromise between convenience and speed but powerful enough. At the input end we now can group catcode changes (catcodes are properties of characters that control how they are interpreted) into tables so that switching between regimes is fast.

You can in great detail influence how data gets read from files because the io subsystem is opened up. In fact, you have the full power of Lua available when doing so. At the same time you can print back from Lua into the input stream.

The input that makes in into TEX, either or not intercepted and manipulated beforehand, is to be in utf8. What comes out to the terminal and log is also utf8, and internally all codepaths work with wide characters. Some memory constraints have been lifted, and character related commands accept large numbers. This comes at a price, which means that in practice the LuaTEX engine can be several times slower than the 8-bit pdfTEX, but of course in practice performance is mostly determined by the efficiency of macro package, so it might actually be faster in situations that would stress its ancestors.

Node lists travel through TEX and can be intercepted at many points. That way you can add additional manipulations. You can for instance rely on TEX for hyphenation, ligature building and kerning but you can also plug in alternatives. For this purpose these stages are clearly separated and less integrated (deep down) than in traditional TEX. There are helpers for accessing lists of nodes, individual nodes and you can box those lists too (this is called packing). You can adapt, create and destroy node lists at will, as long as you make sure you feed back into TEX something that makes sense.

In order to control (or communicate with) nodes from the TEX end, an attribute mechanism was added that makes it possible to bind properties to nodes when they get added to lists. At the TEX end you can set an attribute that then gets assigned to the currently injected nodes, while at the Lua end you can query the node for these attributes and their values.

The language subsystem is re-implemented and behaves mostly the same as in the original TEX program. It has a few extensions and permits runtime loading of patterns. In addition to language support we also have basic script support, that is: directional information is now part of the stream and contrary to Aleph that wraps this into extension whatsits, in LuaTEX we have directional nodes as core nodes.

The font subsystem is opened up in such a way that you can pass your own fonts to the core. You can even construct virtual fonts. This open approach makes it possible to support OpenType fonts and whatever format will show up in the future. Of course the backend needs to embed the right data in the result file but by then the hard work is already done. This approach fits into the always present wish of users (and package writers) to be able to implement whatever crazy thought one comes up with.

The paragraph builder is a somewhat cleaned up variant of the pdfTEX one, combined with directional and boundary support from Aleph. The protrusion and expansion mechanism have been redone in such a way that the front- and backend code is better separated and is somewhat more efficient now. As one can intercept the paragraph builder, additional functionality can be injected before, after or at some stages in the process.

Of course we have kept the math engine but, because we now need to support OpenType math, alternative code paths have been added to deal with the kind of information that such fonts provide. We also took the opportunity to open up the math machinery a bit so that one can control rendering of some more complex elements and set the spacing between elements. Because TEX users are quite traditional we had to stop somewhere, simply because legacy code has to be dealt with.

Most mentioned auxiliary mechanisms can be accessed via the node lists, for instance you can locate inserts and marks in them. The backend related whatsit nodes can be recognized as well. At any time one can query and set TEX registers and intercept boxed material. Of course some knowledge of the inner working of TEX helps here.

The backend code is as much as possible separated from the frontend code (but there is still some work to do there). As in pdfTEX you can of course inject arbitrary pdf code and make feature rich documents. This flexibility keeps TEX current.

8.4 Extras

Is that all? No, apart from some minor extensions that might help to make programming somewhat easier TEX, there are a few more fundamental additions.

Images and reusable content (boxes) are now part of the core instead of them being wrapped into backend specific whatsits, although of course the backend has to provide support for it. This is more natural in the frontend (and user interface) and also more consistent in the engine itself. All backend functionality is now collected in three primitives that take arguments.

This permits a cleaner separation between front- and backend.

Then there is the MetaPost library, a feature already present for many years now. It provides TEX with some graphic capabilities that, given the origin, fits nicely into the whole. The LuaTEX and mplib project started about the same time and right from the start it was our plan to combine both.

One of the extras is of course Lua. It not only permits us to interface to the internals of TEX, but it also provides the user with a way to manipulate data. Even if you never use Lua to access internals, it might still be found useful for occasionally doing things that are hard to accomplish using the macro langage.

In addition to stock Lua we include the lpeg library, an image reading library (related to the backend) including read access to pdf files via the used poppler library, parsing of pdf content streams, zip compression, access to the file system, the ability to run commands and socket support. Some of this might become external libraries at some point, as we want to keep the expected core functionality lean and mean. A nice extra is that we provide LuajitTEX, a compatible variant that has a faster Lua virtual machine on board.

8.5 Follow up

The interfaces that we have now have to a large extent evolved to what we had in mind. We started with simple experiments: just Lua plus a bit of access to registers. Then the Oriental TEX project (with Idris Samawi Hamid) made it possible to speed up development and conversion to C and opening up took off. After that we gradually moved forward.

That doesn’t mean that we’re done yet. The LuaTEX 1.0 engine will not change much. We might add a few things, and for sure we will keep working on the code base. The move from Pascal to C web (an impressive job by itself), as well as merging functionality of engines (kind of a challenge when you want to remain compatible), opening up via Lua (which possibilities even surprised us), and experimenting (ConTEXt users paid the price for that) took quite some time, also because we played with proofs of concept. It helped that we used the engine exclusively for real typesetting related work ourselves.

We will continue to clean up and document the source and stepwise improve the manual. If you followed the development of ConTEXt, you will have noticed that MkIV is heavily relying on the Lua interface so stability is important (although we can relatively easy adapt to future developments as we did in the past). However, the fact that other packages support LuaTEX means that we also need to keep the 1.0 engine stable. Our challenge is to provide stability on the one hand, but not limit ourselves to much on the other. We’ll keep you posted on what comes next.

Hans, Hartmut, Luigi, Taco