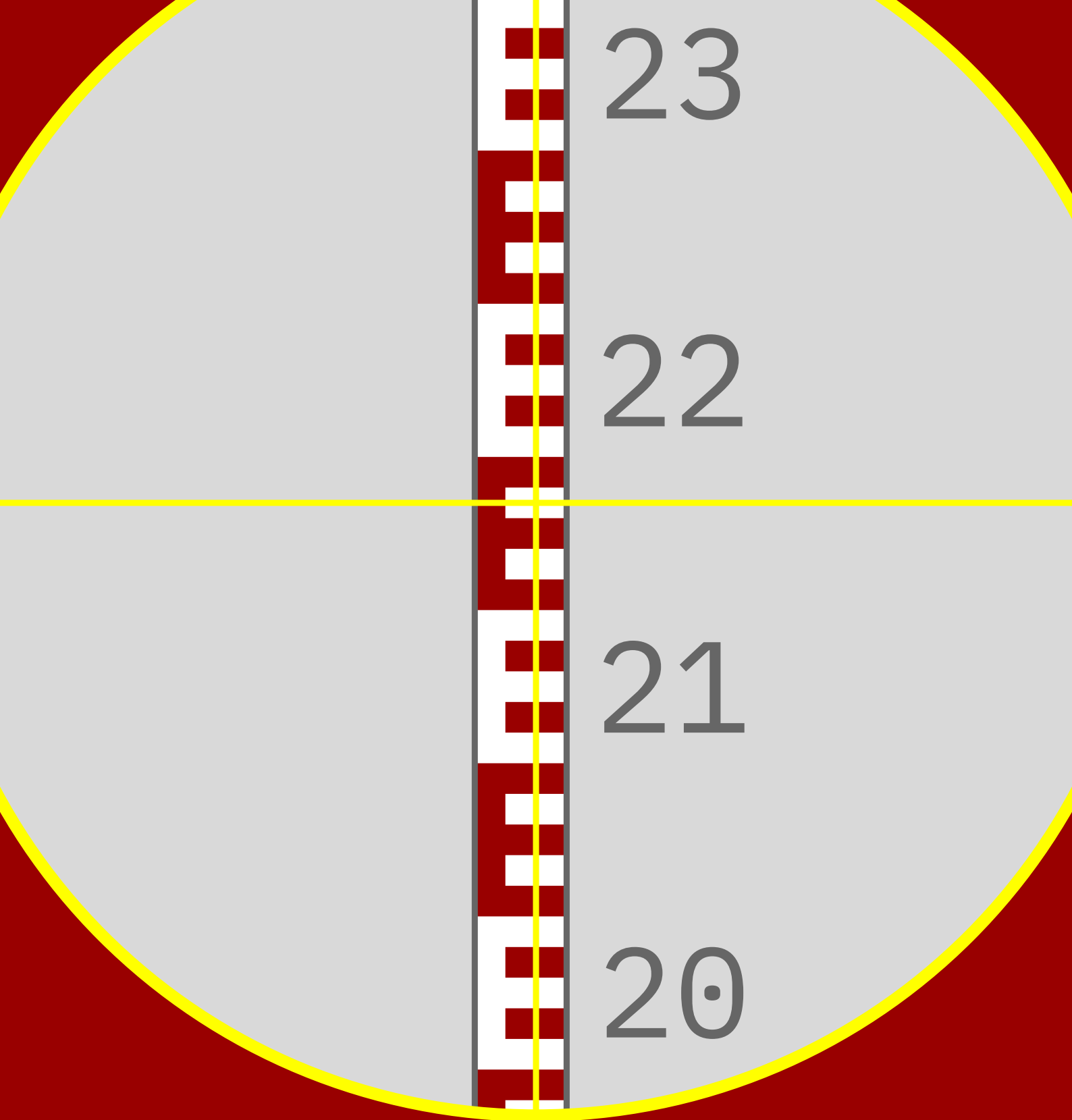


on target

luametatex & context lmtx

Table of contents

1 Introduction

This is the seventh wrapup of the LuaTEX and LuaMetaTEX development cycle. It is dedicated to all those users who kept up with developments and are always willing to test the new features. Without them a project like this would not be possible.

At the time this introduction is written the LuaMetaTEX code base is rather stable and quite a bit of the MkIV code base has been adapted to new situation. But, as usual, there are always new possibilities to explore, so I expect that this document will grow over time as did the others. I'm not going to repeat all that has been done because that's what the previous episodes are about.

As the title suggest, we're still on target. When the LuaMetaTEX project started there actually was no deadline formulated so in fact we're always on target. The core components TEX, MetaPost, and Lua are all long term efforts so we're in no hurry at all. However, this is the year that a fast pace will become a slow pace with respect to the LuaMetaTEX code base. There are still some things on the agenda but in principle the goals are reached. One problem in today's code development is that useability and quality seems to relate to the amount of changes in code. No update can mean old, unusable and uninteresting. It's probably why some sources get this silly yearly copyright year update. However, the update cycle of good old TEX has an decade interval by now while it is still a pretty useable program. It would be nice to end up in such a long term cycle with LuaMetaTEX: bug fixes only.

Although ConTEXt has always adapted early to new developments (color, graphics, pdf, MetaPost, ε-TEX, pdfTEX, LuaTEX, utf, fonts) the effects on the ConTEXt code base are mostly hidden for users. There have been some changes between MkII and MkIV, simply because there has been a shift from specific eight bit encodings to utf and Type1 to OpenType fonts. Both had an impact on important subsystems: input encodings, font definitions and features, language and script support. On other subsystems the impact was hardly noticeable, like for instance backend related features (these have always been kind of abstract). That doesn't mean that these haven't changed deep down, they definitely have. Some mechanisms became better in MkIV, simply because less hackery was needed. My experience is that when users see that it gets better or easier, they are also willing to adapt the few lines in their document source that benefit from it. Of course the impact on the MetaPost integration in ConTEXt had a real large impact, especially in terms of performance.

The upgrade to LMTX, the version of ConTEXt for LuaMetaTEX, is even less visible although already some new mechanisms showed up. This time a couple of engine specific features have been improved and made more flexible. In fact, the whole code base of the engine has been overhauled. This happened stepwise because we had to make sure all things kept working. As a first step code was made independent of the compilation infrastructure and the dependencies, other than a very few small ones, have been removed. The result is a rather lean and mean setup, even when we consider what has been added at the primitive level and traditional subsystems. A benefit is that in the meantime the LuaMetaTEX LMTX combination outperforms LuaTEX with MkIV, something that was not ensured when the built-in pdf backend was removed and delegated to Lua. By binding development closely to ConTEXt we also hope that the code base stays clean of arbitrary extensions.

Because in the end, TEX is also a programming language, there have been extensions that make programming easier. There is already a stable middle layer of auxiliary macros in ConTEXt that help the user who likes to program but doesn't like real low level primitives and dirty tricks, but by extending the primitive repertoire a bit users can now stay closer to the original TEX concepts. Adding more and more layers of indirectness makes no sense if we can improve the bottom programming layer. It

also makes coding a bit more natural (the TEX look), apart from offering performance benefits. This is where you can see differences between the MkIV and LMTX code base which for that reason is now nearly split completely. The MetaPost subsystem has been extended with proper scanners so that we can enhance the interfaces in a natural way and as a result we also have an upgraded code base there. We also moved to Lua5.4 and will keep up as long as compatibility is no issue. Some Lua code is likely to remain common between MkIV and LMTX, for instance font handling and helpers but we'll see where that ends.

The LuaMetaTEX engine provides control over most internals and there are all kind of new interesting features. Decades of ConTEXt development are behind that. Also, in the days that there were discussions about extending TEX, ConTEXt was not that much of influence and on the road to and from user group meetings, Taco and I often discussed what we'd like to see added (and some was actually implemented in `eetex` but that only lived on our machines. One can consider LuaTEX to be a follow up on that, and LuaMetaTEX in turn follows up on that project, which we both liked doing a lot. In some way LuaTEX lowered the boundary for implementing some of the more intrusive extensions in LuaMetaTEX and the follow up on mplib. And once you start along that road small steps become large steps and one can as well be try to be as complete as possible. We've come a long way but eventually arrived at the destination. Personally I think we got there by not being in a hurry.

But even targets that are reached can eventually move,

Hans Hagen
Hasselt NL
August 2021^{++}

2 Eventually 1.0

2.1 Reflection

This is just a short reflection on how we came to version 1.0 of LuaMetaTEX. Much has already been said in articles and history documents. There is nothing in here that is new but I just occasionally like to wrap up the current state. At the time of writing, which happens to be the ConTEXt 2021 meeting, we're somewhere between 0.9 and 1.0 and as usual it reflects a current state of mind.

2.2 Introduction

The development on LuaMetaTEX took a bit more time than I had in planned when I started with it. I presume that it also relates to the way the TEX program is looked at: a finished program that converges to a bugless state. But, with version 1.0 near by it makes sense to reflect on the process. Before I go into details I want to remark that when I wrote ConTEXt I looked at this program from the macro end. I had no real reason to look into the code, and figuring out what happens in a black box is a challenge (and kind of game) in itself. At the time I started using TEX I had done my share of complex and relatively large scale programming in Pascal and Modula so it's not that I was afraid of languages. It was before the Internet took off and not being in academia and connected one had to figure things out anyway. I did have Don's 5 volume TEX series but stuck to the TEX book. Being on msdos I couldn't compile the program anyway, definitely not without the source at hand. I did read the first chapters of the MetaFont book, but apart from being intrigued by it, it was not before I ran into MetaPost that knowing that language took off. Of course I had browsed TEX the program but not in a systematic way.

I was involved with pdfTEX development but stayed at my end of the line: needs, applications, testing and suggestions. With LuaTEX that line got crossed, triggered by the Lua interfaces, but while I focussed on the TEX end, Taco did the C, and we had pleasant and intense daily discussion on how to move forward. I could not get away any longer with the abstraction but had to deal with nodes and such, which was okay as we were hit the boundaries of convenience programming solutions in ConTEXt.

When we started our LuaTEX journey the TEX follow-up most widely used, pdfTEX, did have some ε-TEX extensions but in retrospect only a few of those were of relevance to us, like the concept of `\protected` macros[1] and the larger set of registers. And the ε-TEX project, in spite of occasional discussions, never became a continuous effort. The nts project that was related to ε-TEX and had as objective an extensible successor produced a Java implementation but that one was never useful (as a starter, its performance was such that it could not be used) and I didn't really look forward to spending time on Java anyway. Taco and I played with an extended ε-TEX but lack of time made that one end up in the archive.

There were some programmatic additions to pdfTEX but it's main attributes were protrusion, expansion and a pdf backend (Hàn Thế Thành's thesis subject). Features like position tracking were handy but basically just a built-in variant of a concept we already had come up with at the dvi level (using a postprocessing script that later became `dvipos`). There was Omega with a directional model but this engine was always more of an academic project, not a production system.[2] It was XHTEX that moved the TEX world into the Unicode domain and opened the engine up to new font technologies. Although utf8

[1] In ConTEXt we always had a protection mechanism and from the LuaTEX source I learned that the macro bases solution was basically the same as the one used in the engine.

[2] Aleph was more reliable but never took off, if only because pdfTEX had a backend.

was already doable in earlier engines (which is why ConTEXt used it already for some internals), native support was way more convenient.

It was clear that if we wanted to move on we had to make more fundamental steps, but in such a way that it still fit in with what people expect from TEX. While it started an a playground by embedding the Lua interpreter, it quickly became clear that we could open up the internals in fundamental ways, thereby also getting around the discussion about to what extent TEX could and should be extended: that discussion could be and was postponed by the opening up. Because we already foresaw some of possibilities it was decided to freeze ConTEXt for the older engines. It was around the first ConTEXt meeting that the MkII and MkIV tags showed up, around the same time that LuaTEX became useable. More than a decade later, when LuaTEX basically had become frozen, at another meeting it was decided to move on with LuaMetaTEX: the LuaTEX project was pretty much a ConTEXt projects and that follow up would be even more driven by ConTEXt users and usage. But how does it all feel 15 years later? I'll try to summarize that below. It will also explain why I got more audacious in extending the LuaTEX engine into what is now LuaMetaTEX. This also related to the fact that at some point I realized that progress just demands taking decisions, and it happens that we can make these in the perspective of ConTEXt without side effects for other TEX usage. It is also fun to experiment.

2.3 Extending necessary parts

The pdfTEX program, having a backend built in already supports the usage of wide TrueType but it was XETEX that first provided using them directly in the frontend. But that happened within the concept of traditional TEX, especially when it comes to math. There are some extra primitives to deal with scripts and languages but (and this is personally) I decided that these didn't really fit in the way ConTEXt looks at things so MkII doesn't support anything beyond the fonts. The XETEX program first was available on Apple computers and font support was closely related to its technology as well as technologies that relate to where the program originates. Later other operating systems became supported too.

We decided in LuaTEX to delegate 'everything fonts' to Lua, for a good reason: we didn't want to be platform dependent. And using libraries has the danger of periodical enforced fundamental changes because in these times software politics and fashion have short cycles. The fact that XETEX later changed the font engine proved that this was a good decision. At some point LATEX decided to use a special version of LuaTEX that uses a font library as alternative, which is fine, but that also introduces a dependency (and frequent updating of the binary). The LuaTEX engine has a slim variant of the FontForge library built in for reading various font formats and its backend can embed subsets of OpenType, Type1 and traditional bitmap fonts. At some point ConTEXt switched to its own Lua based font file interpreter and experimented with a Lua based backend that later became exclusive for LuaMetaTEX. It became clear that we could do with less code in the engine and thereby less dependencies.

In this perspective it is also good to notice that the LuaTEX engine has no real concept of Unicode: it just expects utf8 and that's it. All internals provide enough granularity to support Unicode. The rest has to come from the macro package, as we know that each one does it its own way. There are no dependencies on Unicode libraries. You only have to look at what ends up on your system when you install a program that just juggles bytes to notice that by including one library a whole lot gets drawn in, most of which is not relevant to the program and we don't want that. It might start small but who knows where one ends up. If we want users to be able to compile the program, we don't want to end up in dependency hell.

The LuaTEX project was, apart from curiosity and potential usage in ConTEXt, initially also driven by the Oriental TEX project that aimed at high quality bidirectional typesetting. There the focus was on

fonts as well as processing paragraphs. That triggered all kinds of opening up of internals and once ConTEXt started swapping (and adding) mechanisms using Lua more came to fruit. In the end it took a decade to reach version 1.0 and we could have stopped there knowing that we're quite prepared for the future.

Although the whole TEX concept didn't change, there were some fundamental changes. From the documentation by Don Knuth it becomes clear that interpreting is closely interwoven with typesetting: the so called main interpretation loop calls out to font processing, ligature building, hyphenation, kerning, breaking lines, processing pages, etc. In LuaTEX these steps became more independent simply because the processing of fonts (via Lua) came down to feeding a linked list of nodes to a callback function. That list should be hyphenated if needed (a now separated step) and if needed the traditional font processing could be applied (ligature building and kerning). But, although one can say that we already got away from the way TEX works internally, most documentation to the original program still applied, simply because the fundamental approach was the same. We didn't feel too guilty about it and I don't think anyone objected. By the way, the same is true for the math subsystem: we had to adapt it to OpenType parameters and formula construction and although that was inspired by TEX it definitely was different, even to the extend that the math fonts that evolved in the community are now a strange hybrid of old and new.

2.4 Getting around the frozen machinery

So why did the LuaMetaTEX project started at all? There has been plenty written on how LuaTEX evolved and the same is true for LuaMetaTEX so I'm not going to repeat that here. It is enough to know that the demand for a stable and frozen LuaTEX by other users than ConTEXt simply doesn't go well with further experiments and we still had plenty ideas. Because at some point Taco had no time I was already responsible for quite some additions to the LuaTEX program so it was no big deal to switch to a an even more extensive mix of working with "TEX the macro language" and "TEX the program".

The first priorities were with some basic cleanup: remove unused font code, get rid of some ever changing libraries and remove the backend related code. I could do that because I already had a Lua driven backend in MkIV (which was removed later on) and font handling was already all done in Lua. The idea was to go lean and mean, and indeed, even with all kind of extensions, the binary is much smaller than its predecessor, which is nice because it is also a Lua engine. Simplifying the build so that users can easily compile themselves was also of high priority because I considered the rather large and complex setup as a time bomb. And I also had my doubts if we could prevent the LuaTEX engine to evolve over time in a way that made it less useable for ConTEXt.

But, interestingly all this extending and pruning didn't feel like I was violating the concept of a long term stable engine. In fact, original TEX has no backend either, just a simple binary serialization of output (dvi). And by removing some font related frontend code we actually came closer to the original. I suppose that these decisions slowly made me aware of the fact that there was no reason to not consider more drastic extensions. After all, wasn't the ε-TEX project also about extending.[3]

When we look at LuaMetaTEX 1.0 we still see the expected machinery there but many subsystems have been extended. Once I made the decision that it's now or never, each subsystem got evaluated against my long term wish list and usage in ConTEXt. Now, let's be clear: I basically can do all I want in LuaTEX

[3] Although non of the ideas that Taco and I discussed on our numerous trips to meetings all over the world ever made it into that engine.

but that doesn't mean it's always a pretty solution. And to make the ConTEXt code base better to understand for users, even if it is already rather consistent and set up to be readable, is one of my objectives. I spend a lot of time on readability: I cannot stand a bad looking source and over time the look and feel is also determined by the way the ConTEXt interfaces and related syntax highlighting evolved, especially the TEX, MetaPost, Lua mix. This is why LuaMetaTEX has some extensions to the macro language.

So, while some might argue that "It can already be done." I decided to ignore that argument when the actual solutions came too close to "See how well I can do this using dirty tricks!". If we can do better, without harming the system, let's do it: Lua did it, C did it and even Don Knuth switched from Pascal to C. If we want we can put all the extensions under the "TEX is meant to be extended" umbrella, as long as we call it different, which is what we do. But I admit that one has to (emotionally) cross a boundary of feeling comfortable with fundamental additions to a program like TEX. But I've been around long enough to not feel guilty about it.

So in the end that means that for instance marks were extended, inserts got more options, glyphs and boxes have way more properties, (the result and handling of) paragraphs can be better controlled, page breaking got hooks (and might be extended), local boxes got redone, adjustments were extended, the math machinery has been completely opened up, hyphenation became more powerful, the font mechanism got more control and new scaling features, alignments got some extensions, we can do more with boxes, etc. But often I still first had to convince myself that it's okay to do so. After all, none of this had happened before and to my knowledge also has not been considered in ways that resulted in an implementation (but I might be wrong here). It helps that I can test out experiments in production versions of LMTX and that users are quite willing to test.

2.5 Extending the macro language

In the previous section some mechanisms were mentioned, but before TEX even ends up there macros and primitives come into play. The LuaTEX engine already has some handy extras, like ways to prepend and append tokens and a limited so called 'local control' mechanism (think of nested main loops). There are some new look head and expansions related primitives and csname related tricks. There are a few more conditionals too. Details can be found in manual and articles.

In LuaMetaTEX some more got added and some of these mechanism could be improved and the reason again is that I aim at readable code. Most programming languages for instance have conditionals with some kind of continuation (like `elseif`) and so I added that to TEX too `\orelse`. Actually, there are even more new conditionals than in LuaTEX. Yes, we don't really need these, especially because in LuaMetaTEX we can now extend the primitive language via Lua, but I wanted to improve readability deep down in ConTEXt. It also reduces the clutter when logging, although logging itself has been quite a bit overhauled. There is less need for intermediate (often not that natural) intermediate layers when we can do it properly in primitive TEX lingua.

More fundamental was extending the way TEX deals with macro arguments. Although the extensions to parsing them are using specifiers that make them upward compatible I admit that even I have to consult a list of possibilities every now and then but in the end they make things better (performance wise with less code). As a side effect the macro machinery could be optimized a bit (expansion as well as the save stacking).

There are a few more ways to store integers and dimensions (these fit in nicely), there are new into grouping, some primitives have more keywords and therefore scanners have been extended, the ε-TEX expression handlers have alternative variants.

Although this is a sensitive aspect of TEX when it comes to compatibility, at some point I decided that it made no sense to not expose more details about nodes, input, and nesting states. The grouping and input related stacks had been optimized in the meantime so reporting in that area was already not compatible. Improving logging is an ongoing effort and I don't really loose sleep over it not being compatible, as long as it gets better. There is now also some tracing for marks, inserts, math and alignments.

2.6 Refactoring the code base

This is again an emotionally laden decision: what to touch and keep. For sure we keep the original comments but that doesn't make it literate. We started out with a C base that came from converted Pascal web.

The input machinery is a bit different due to the fact that Lua can (and often has to) kick in. In LuaMetaTEX it's even more different because even more goes via Lua. We cannot even run the engine without a basic set of callbacks assigned: if you don't like that, use LuaTEX. Does this violate the TEX concept? Not really, because system dependencies are explicitly mentioned as such in the source code. We have to adapt to the way an operating system sees files anyway (eight bit, utf8, utf16).

We still have many global variables (a practical Knuth thing I guess) but now they are grouped into structures so that we can more clearly see where they belong. This involved quite a but of shuffling and editing but I got there. In LuaMetaTEX all constants (coded in macros) became enumerations, and all hard coded values too which was quite a bit of work too. Probably no one will notice or realize that, but starting from an existing code base is more work than starting from scratch, which is what I always did so far. When possible we use case statements. Most macros became (inline) functions. Complex functions got better variable names. All functions are in name spaces. This was (and is) a stepwise process that takes lots of time, especially because ConTEXt users expect a reasonable stable system and changes like that are sensitive for errors.

Talking of errors, the error and reporting system has been overhauled, so for instance we have now a dedicated string formatter. This all happened in several steps: normalization, consistency, abstraction, formatters, etc. Keep in mind that we not only have the original messages but also new ones. And we have TEX, Lua and MetaPost communicating with the user. Where in LuaTEX we have to conform more to the traditional engine, because that is what other macro packages rely on, In ConTEXt we have more freedom, so we can make it better and more detailed. Of course it could all be controlled by configurations but at some point I decided to kick out variables doing that because it made no sense to complicate the code base.

Memory management has been overhauled (more dynamic) as has dumping to the (more efficient) format file. With what is mentioned in the previous paragraphs we can safely say that in the meantime back porting to LuaTEX (which I had in mind) makes no sense any longer. There is occasionally some pressure to let LuaTEX do the same as other engines (new common features) and that doesn't always fit into the model. There is no need for LuaMetaTEX to follow up on that because often we already have plenty of possibilities. There is of course still work todo, for instance I still have to make some variable names in functions more verbose but that is not fundamental. I also have to go over the documentation in the code. I might make some interfaces more consistent anyway, so that also would demand adaptations. And of course the documentation in general always lags behind.

So far I only mentioned dealing with TEX, but keep in mind that in LuaMetaTEX we also have an upgraded MetaPost: only a Lua backend (we can produce pdf from that other output), no font code, a

couple of extensions, more callbacks, io via Lua. Scanners make extending the language possible and injectors make for efficient piping back to MetaPost. Such extensions are also possible in TEX and the LuaMetaTEX scanning interfaces have been improved and extended too. We have extra callbacks (but some were dropped), more helpers (most noticeable in the node namespace), libraries that improve dealing with binary files, a reworked token library (which in turn lead to a reorganization of command codes in the TEX engine), a few more extensions if Lua file handling and string manipulations. We got decimal math, complex math, new compression libraries, better (Lua) memory management, a few optional library interfaces, etc. Fortunately that all didn't bloat the binary.

So, because in the meantime LuaMetaTEX is quite different from LuaTEX, we can consider the last one to be a prototype for the real deal.

2.7 Simplifying the build

This was one of the first things I did. It was a curious process of removing more and more of the original build (all kind of dependencies) which is not entirely trivial because of the way the LuaTEX build is set up. I admit that I did try to stay within the regular source build concept but after a while I realized that this made no sense so we (Mojca was involved in that) made the move to cmake. Shortly after that I started using Visual Studio as editing environment (which saves time and is rather convenient) and native compilation under MS Windows became possible without any special measures (in fact, setting up the build for arm processors was more work).

A side effect is that right from the start we could provide binaries for various platforms via the compile farm on the ConTEXt garden maintained by Mojca, who also does daily TEX live builds there. On my machine I use the Windows Linux Subsystem for cross compilation but we can also do native builds. And, with my laptop being a robust 2013 old timer I force myself to make sure that LuaMetaTEX keeps performing well.

2.8 Because it just makes sense

So, in the end LuaMetaTEX is likely the engine most different from the Knuthian original but from the above one can conclude that this was a graduate process where I got more audacious over time. In the end the only thing that matters (and I believe that Don Knuth agrees with this) that you like writing the code, feel confident that the code is all right, explore the possibilities, try to improve the quality and understanding and that successive rewrites can reduce obscurity. And in my opinion we didn't loose the TEX look and feel and still can operate well within the established boundaries of the TEX ecosystem. The fact that most ConTEXt users in the meantime use LuaMetaTEX and the related LMTX variant is an indication that they are okay with it, and that is what matters most.

3 A new unit: dk

At the ConTEXt 2021 meeting I mixed my TEX talks with showing some of the (upcoming) LuaMetaTEX source code. One evening we had a extension party where a new unit was implemented, the dk. This event was triggered by a remark Hraban [Ramm] made on the participants list in advance of the meeting, where he pointed to a Wikipedia article from which we quote:

> "In issue 33, Mad published a partial table of the "Potrzebie System of Weights and Measures", developed by 19-year-old Donald E. Knuth, later a famed computer scientist. According to Knuth, the basis of this new revolutionary system is the potrzebie, which equals the thickness of Mad issue 26, or 2.2633484517438173216473 mm [...]."

So, as the result of that session, the source code now has this comment:

> "We support the Knuthian Potrzebie, cf. en.wikipedia.org/wiki/Potrzebie, as the dk unit. It was added on 2021-09-22 exactly when we crossed the season during an evening session at the 15th ConTEXt meeting in Bassenge (Boirs) Belgium. It took a few iterations to find the best numerator and denominator, but Taco Hoekwater, Harald Koenig and Mikael Sundqvist figured it out in this interactive session. The error messages have been adapted accordingly and the scanner in the Lua tex library also handles it. One dk is 6.43985pt. There is no need to make MetaPost aware of this unit because there it is just a numeric multiplier in a macro package."

When compared to the already present units the dk nicely fills a gap:

unit	points	scaled	visual
sp	0.00002	1	
pt	1.0	65536	
bp	1.00374	65781	
dd	1.07	70124	
mm	2.84526	186467	
dk	6.43985	422042	
pc	12.0	786432	
cc	12.8401	841489	
cm	28.45274	1864679	
in	72.26999	4736286	

Deep down, the unit scanner uses a numerator and denominator in order to map the given value onto the internally used scaled points, so the relevant snippet of C is:

```
*num   = 49838; // 152940;
*denom =  7739; //  23749;
return normal_unit_scanned;
```

The impact on performance of scanning an additional unit can be neglected because the scanning code is a bit different from the code in LuaTEX and (probably the) other engines anyway.

Under consideration are a few extra units in the relative_unit_scanned category that we see in css: vw (relative to the \hsize), vh (relative to the \vsize), maybe a percentage (but of what) and ch (width of the current zero digit character). As usual with TEXies, once it's there it will be (ab)used.

13 A new unit: dk

4 Anchoring

4.1 Introduction

It is valid to question what functionality should be in the engine and what can best be implemented using callbacks and postprocessing of lists (and boxes) relying for instance on attributes as signals. In LuaTEX we are rather strict in this and assume that the second method is used. In LuaMetaTEX we still promote this but at the same time some (lightweight) functionality has been added to the engine that helps implementing some features more efficiently. Reasons are that it can be handy to carry (fundamental) properties around that are bound to nodes and that we can set them using primitives, especially for glyphs and boxes. That way they become part of the formal functionality that one can argue should be present in a modern engine. Examples for glyph nodes are scales, offsets and hyphenation, detailed ligature and kerning control. For box nodes we have for instance offsets and orientation. Most of these are always taken into account by core mechanisms like breaking paragraphs into lines, where dimensions matter in which case it really makes sense for them to be part of the engine design.

Some properties are just passed on to for instance a font handler or the backend but still they belong to the core functionality. An example of the later is a (new) simple mechanism for anchoring boxes. This is not really a fundamental feature, because one can just move content around using a combination of kerning and boxing, either or not with offsets. But because we already have features like offsets to boxes it was not that much work to add anchoring as a more fundamental property. The frontend is agnostic to this feature because dimensions are kind of virtual here: the backend however carries the real burden. Because backends are written in Lua it might have a performance hit simply because at least we need to check if this feature is used. Normally that can compensated when this feature *is* used because less work and shuffling around happens in the frontend. And when this feature is no longer experimental (and stays) we can gain some back by using it in existing scenarios. It sounds worse than it is because for orientations we already have to do some usage checking and we can share that check; in most situations nothing needs to be done anyway.

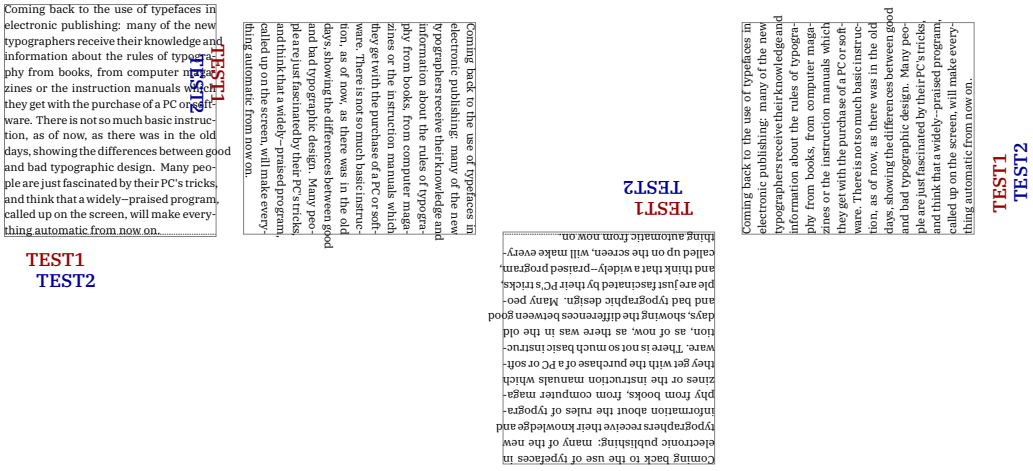
4.2 The low level approach

When we anchor, a box can be a source and/or a target. Both are represented by a number and can be assigned via a keyword. These numbers can be picked up by the backend. Here is an example:

```
\def\TestMe#1{%
    \setbox \scratchbox \ruledvbox
        source 123
        orientation #1
        \bgroup
            \hsize7cm
            \samplefile{zapf}
            \hbox to 0pt
                source 124 target 123
                xoffset 20pt yoffset -30pt
                {\darkred \bfc TEST1}%
            \hbox to 0pt
                source 125 target 124
                xoffset 10pt yoffset -20pt
```

```
                {\darkblue \bfc TEST2}%
        \egroup
    \box \scratchbox
}
```

This example also uses a few offsets. The ‘origin’ is at the left edge of the baseline. Now, we could have passed the source and target as attribute and intercepting an attribute in the backend can work pretty well. However, the code that deals with the final result of the typesetting and thereby flushes it to for instance a pdf file is, at least that is the setup we use in ConTEXt, attribute agnostic. Mixing in attributes at that stage, except for user nodes and whatsits that are effectively plugins, is counter intuitive and all is already pretty complex so a clear separation of functionality makes a lot of sense. Of course the ConTEXt approach is not the only one when it comes to generic engine functionality. Not that many fundamental (conceptual) extensions showed up over the last few decades so no one will bother if in LuaMetaTEX we have new stuff that is only used by ConTEXt. The example code shown here gives:



In order to avoid additional shifting around, which then might involve copying and injecting boxes as well as repackaging, two additional keys are available and these deal with the way boxes get anchored.

```
\vbox
    source 123
    \bgroup
        \offinterlineskip
        \blackrule[width=4cm,height=2cm,depth=0cm,color=darkred]\par
        \blackrule[width=4cm,height=0cm,depth=1cm,color=darkblue]\par
        \setbox\scratchboxtwo\hbox
            anchors "0004 "0001
          % anchor "00040001
            target 123
            orientation 1
            {\blackrule[width=2cm,height=1cm,depth=0cm,color=darkgreen]%
             \hskip-2cm
             \blackrule[width=2cm,height=0cm,depth=1cm,color=darkyellow]}%
          %
        \smash{\box\scratchboxtwo}%
```

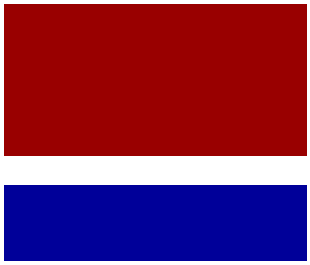
```
\egroup
```

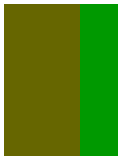
The anchor is just an number but with the plural keyword we can scan it as two because that is a bit easier on usage. The two numbers four byte numbers control the source to target anchoring and there is plenty room for future extensions because not all bits are used.

```
0x001  left origin
0x002  left height
0x003  left depth
0x004  right origin
0x005  right height
0x006  right depth
0x007  center origin
0x008  center height
0x009  center depth
0x00A  halfway total
0x00B  halfway height
0x00C  halfway depth
0x00D  halfway left
0x00E  halfway right
```

The target and source are handled in a way that sort of naturally binds them which involves a little juggling with dimensions in the backend. There is some additional control over this but usage is not advertized here because it might change.

One can set these anchoring related properties with keywords but there are also primitive box manipulators: `\boxanchor`, `\boxanchors`, `\boxsource` and `\boxtarget` that take a box number and value.





There are some helpers at the Lua end but I haven't completely made up my mind about them. Normally that evolves with usage.

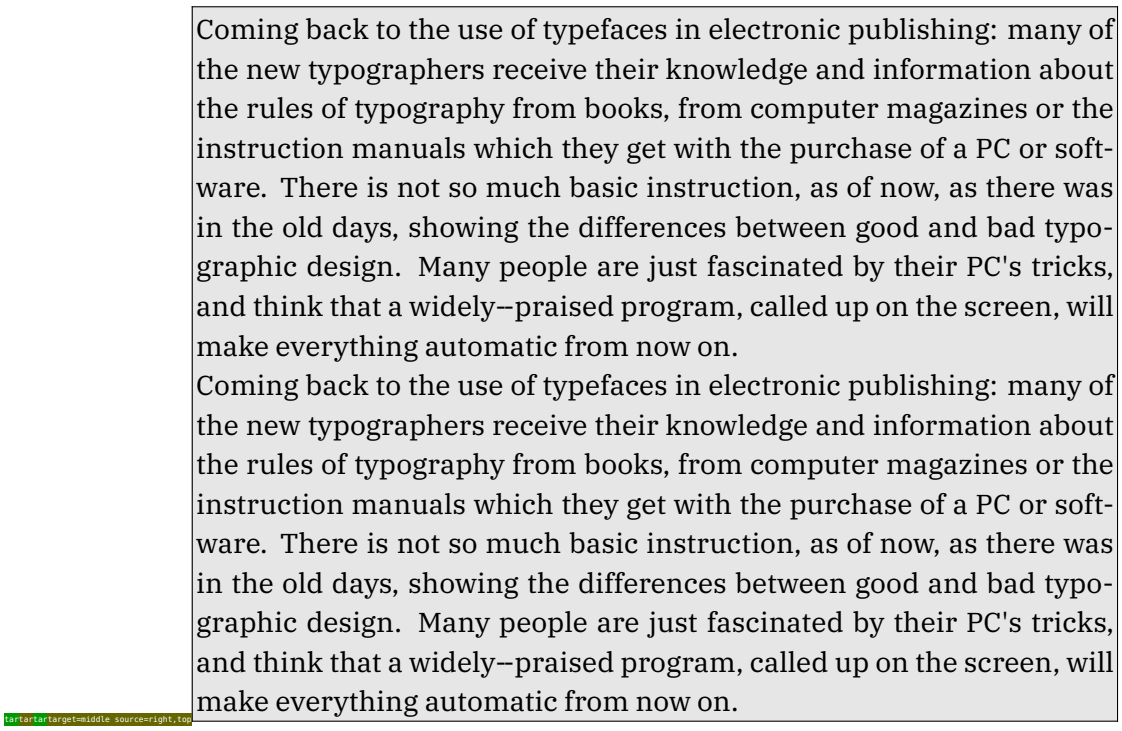
4.3 A first higher level interface

Exploring this here in more detail makes no sense because it is still experimental and also rather ConTEXt specific. As a teaser an interface that hooks into layers is shown:

```
\defineanchorboxoverlay[framed]

\def\DemoAnchor#1#2#3#4%
  {\setanchorbox
     [#1]%
     [target={#3},source={#4}]%
```

```
     \hbox{\backgroundline[#2]{\white\smallinfofont\setstrut\strut target=#3
source=#4}}}

\def\DemoAnchorX#1#2%
  {\DemoAnchor{#1}{darkred}    {#2}{left,top}%
   \DemoAnchor{#1}{darkblue}   {#2}{left,bottom}%
   \DemoAnchor{#1}{darkgreen}  {#2}{right,bottom}%
   \DemoAnchor{#1}{darkyellow}{#2}{right,top}}%

\startsetups framed:demo
    \DemoAnchorX{framed:background}{left,top}%
    \DemoAnchorX{framed:background}{right,top}%
    \DemoAnchorX{framed:background}{left,bottom}%
    \DemoAnchorX{framed:background}{right,bottom}%
    \DemoAnchorX{framed:foreground}{middle}%
\stopsetups

\midaligned\bgroup
    \framed
      [align=normal,
       width=.7\textwidth,
       backgroundcolor=gray,
       background={color,framed:background,foreground,framed:foreground}]
      \bgroup
        \samplefile{zapf}\par
        \directsetup{framed:demo}%
        \samplefile{zapf}%
      \egroup
\egroup
```

Those familiar with ConTEXt will recognize the approach. This one basically is a more low level variant of layers and a high level variant of the primitives. Performance wise (in terms of memory usage and runtime) it sits in a sweet spot.



I played a bit with a mechanism that can store the embedded (to be anchored) content in a more independent way and it actually works okay. However, I'm not entirely sure if that solution is the best so for now it's commented. As usual it is also up to users to come up with demands.

5 A different approach to math spacing

Introduction

The TEX engine is famous for its rendering of math and even after decades there is no real contender. And so there also is no real pressure to see if we can do better. However, when Mikael Sundqvist ran into a Swedish math rendering specification and we started discussing a possible support for that in ConTEXt, it quickly became clear that the way TEX does spacing is a bit less flexible than one wishes for. We already have much of what is needed in place but it also has to work well with how TEX sees things:

1. Math is made from a sequence of atoms: a quantity with a nucleus, superscript subscript.[4] Atoms are spaced by `\thinmuskip`, `\medmuskip` and `\thickmuskip` or nothing, and that is sort of hard coded.
2. Atoms are organized by class and there are seven (or eight, depending on how you look at it) of them visible: binary symbols, relations, etc. The invisible ones, composites like fractions and fenced material (we call them molecules) are at some point mapped onto the core set. Molecules like fences have a different class left and right of the fenced material.
3. In addition the engine itself has all kind of spacing related parameters and these kick in automatically and sometimes have side effects. The same is true for penalties.

The normal approach to spacing other than imposed by the engine is to use correction space, like `\,` and I think that quite some TEX users think that this is how it is supposed to be. The standard way to enter math relates to scientific publishing and there the standards are often chiseled in stone so why should users tweak anyway. However, in ConTEXt we tend to start from the users and not the publishers end so there we can decide to follow different routes. Users can always work around something they don't like but we focus on reliable input giving predictable output. Also, when reading on, it is good to realize that it is all about the user experience here: it should look nice (which then of course makes one become aware of issues elsewhere) and we don't care much about specific demands of publishers in the scientific field: the fact that they often re-key content doesn't go well with users paying attention themselves, let alone the fact that nowadays they can demand word processor formats.

The three mentioned steps are fine for the average case but sometimes make no sense. It was definitely the best approach given time and resources but when LuaTEX went OpenType a lot of parameters were added and at that time we therefore added spacing by class pair. That not only decoupled the relation between the three (configurable) muskip parameters but also made it possible to use plenty of them. Now it must be said that for consistency having these three skips works great but given the tweaking expected from users consistency is not always what comes out.

This situation is very well comparable to the proclaimed qualities of the typesetting of text by TEX. Yes, it can do a great job, and often does, but users can mess up quite well. I remember that when we did tests with hz the outcomes were pretty unimpressive. When you give an audience a set of sample renderings, where each sample is slightly different and each user gets a randomized subset, the sudden lack of being able to compare (and agree) with another TEXie makes for interesting conclusions. They look for the opposites of what is claimed to be perfect. So, two lines with hyphens rate low, even if not doing it would look worse. The same for a few short words in the last line of a paragraph. Excessive spacing is also seen as bad. So, when asked why some paragraphs looked okay noticing (excessive and troublesome) expansion was not seen as a problem; instead it were hyphens that got the attraction.

[4] I suddenly realize why in the engine noads have a nucleus field: they are atoms ... but what does that make super and subscripts.

The same is probably true for math: the input with lots of correction spaces or commands where characters would do can be horrible but it's just the way it is supposed to be. The therefore expected output can only be perfect, right, independent of how one actually messed up spacing. But personally I think that it is often spacing messed up by users that make a TEX document recognizable. It compares to word processor results that one can sometimes identify by multiple consecutive spaces in the typeset text instead of using a glue model like TEX. Reaching perfection is not always trivial, but fortunately we can also find plenty of nice looking documents done with TEX.

The TEXbook has an excellent and intriguing chapter on the fine points of math and it definitely shows why Don Knuth wrote TEX as a tool for his books. He pays a lot of attention to detail and that is also why it all works out so well. If you need to render from unseen sources (as happens in an xml workflow) coming from several authors and have time nor money to check everything, you're off worse. And I'm not even talking of input where invisible Unicode spacing characters are injected. It is the TEX book(s) that has drawn me to this program and believe it or not, in the first project I was involved in that demanded typeset (quantum mechanics) math the ibm typewriter with changing bulbs ruled the scenery. In fact, our involvement was quickly cut off when we dared to show a chapter done in TEX that looked better.

Apart from an occasional tweak, in ConTEXt we never really used this opened up math atom pair spacing mechanism available in LuaTEX extensively. So, when I was pondering how to proceed it stroke me that it would make sense to generalize this mechanism. It was already possible (via a mode parameter) to bypass the second step mentioned above, but we definitely needed more than the visible classes that the engine had. In ConTEXt we already had more classes but those were meant for assigning characters and commands to specific math constructs (think of fences, fractions and radicals) so in the end they were not really classes. Considering this option was made easier by the fact that Mikael would do the testing and help configuring the defaults, which all will result in a new math user manual.

There are extensions introduced in LuaTEX and later LuaMetaTEX that are not discussed here. In this expose we concentrate on the features that were explored, extended and introduced while we worked on updating math support in LMTX.

An example

Before we go into details, let's give an example of unnoticed spacing effects. We use three simple formulas all using fractions:

```
\ruledhbox{$\frac{x^2}{a+1}$}
```

and:

```
\ruledhbox{$x + \frac{x^2}{a+1} = 10$}
```

as well as:

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

$$\frac{x^2}{a+1} \qquad x + \frac{x^2}{a+1} = 10 \qquad \frac{1}{2}\frac{1}{2}x$$

If you look closely you see that the fraction has a little space at the left and right. Where does that come from? Because we normally don't put a tight frame around a fraction, we are not really aware of it. The spacing between what are called ordinary, operator, binary, relation and other classes of atoms is explained in the TEXbook (or "TEX by Topic" if you want a summary) and basically we have a class by class matrix that is built into TEX. The engine looks at successive items and spacing depends on their (perceived) class. Because the number of classes is limited, and because the spacing pairs are hard coded, the engine cheats a little. Depending on what came before or comes next the class of an atom is adapted to suit the spacing matrix. One can say that a "reading mathematician" is built in. And most of the decisions are okay. If needed one can always wrap something in e.g. `\mathrel` but of course that also can interfere with grouping. All this is true for TEX, pdfTEX, XETEX and LuaTEX, but a bit different in LuaMetaTEX as we will see.

The little spacing on both edges of the fraction is a side effect of the way they are built internally: fractions are actually a generalized form of "stuff put on top of other stuff" and they can have left and/or right delimiters: this is driven by primitives that have names like `\atop` and `\atopwithdelims`. The way the components are placed is (especially in the case of OpenType) driven by lots of parameters and I will leave that out of the discussion.

When there are no delimiters, a so called `\nulldelimiterspace` will be injected. That parameter is set to 1.2 points and I have to admit that in ConTEXt I never considered letting that one adapt to the body font size, which means that, as we default to a 12 point body font, the value there should have been 1.44 points: mea culpa. When we set this parameter to zero point, we get this:

$$\frac{x^2}{a+1} \qquad x+\frac{x^2}{a+1}=10 \qquad \frac{1}{2}\frac{1}{2}x$$

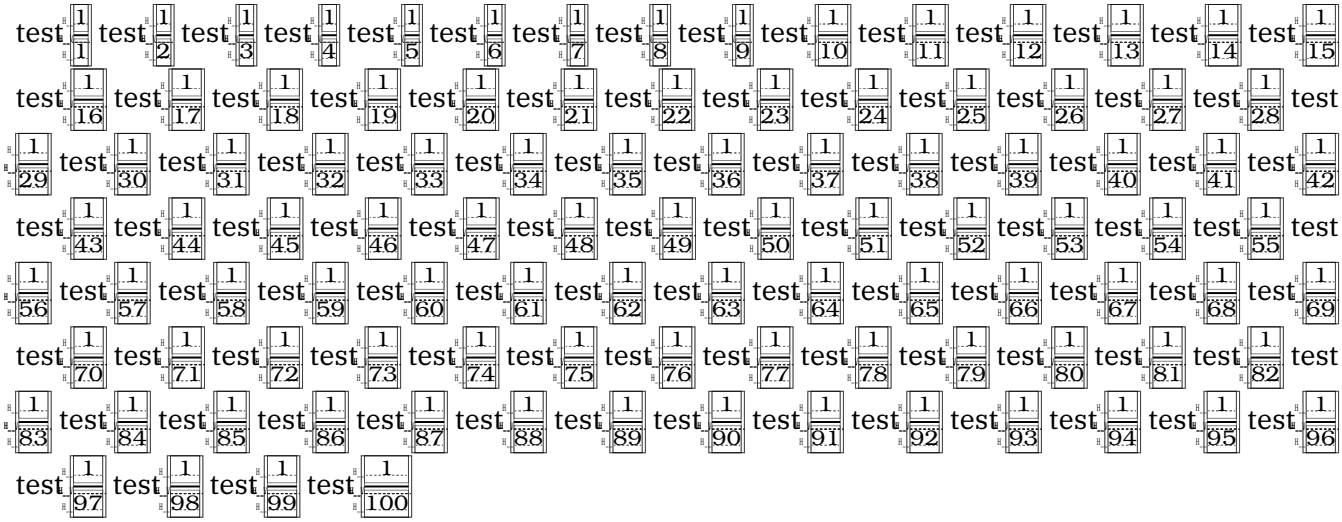
As intermezzo and moment of contemplation I show some examples of fractions mixed into text. When we have the delimiter space set we get this:

test $\frac{1}{1}$ test $\frac{1}{2}$ test $\frac{1}{3}$ test $\frac{1}{4}$ test $\frac{1}{5}$ test $\frac{1}{6}$ test $\frac{1}{7}$ test $\frac{1}{8}$ test $\frac{1}{9}$ test $\frac{1}{10}$ test $\frac{1}{11}$ test $\frac{1}{12}$ test $\frac{1}{13}$ test $\frac{1}{14}$ test $\frac{1}{15}$ test $\frac{1}{16}$ test $\frac{1}{17}$ test $\frac{1}{18}$ test $\frac{1}{19}$ test $\frac{1}{20}$ test $\frac{1}{21}$ test $\frac{1}{22}$ test $\frac{1}{23}$ test $\frac{1}{24}$ test $\frac{1}{25}$ test $\frac{1}{26}$ test $\frac{1}{27}$ test $\frac{1}{28}$ test $\frac{1}{29}$ test $\frac{1}{30}$ test $\frac{1}{31}$ test $\frac{1}{32}$ test $\frac{1}{33}$ test $\frac{1}{34}$ test $\frac{1}{35}$ test $\frac{1}{36}$ test $\frac{1}{37}$ test $\frac{1}{38}$ test $\frac{1}{39}$ test $\frac{1}{40}$ test $\frac{1}{41}$ test $\frac{1}{42}$ test $\frac{1}{43}$ test $\frac{1}{44}$ test $\frac{1}{45}$ test $\frac{1}{46}$ test $\frac{1}{47}$ test $\frac{1}{48}$ test $\frac{1}{49}$ test $\frac{1}{50}$ test $\frac{1}{51}$ test $\frac{1}{52}$ test $\frac{1}{53}$ test $\frac{1}{54}$ test $\frac{1}{55}$ test $\frac{1}{56}$ test $\frac{1}{57}$ test $\frac{1}{58}$ test $\frac{1}{59}$ test $\frac{1}{60}$ test $\frac{1}{61}$ test $\frac{1}{62}$ test $\frac{1}{63}$ test $\frac{1}{64}$ test $\frac{1}{65}$ test $\frac{1}{66}$ test $\frac{1}{67}$ test $\frac{1}{68}$ test $\frac{1}{69}$ test $\frac{1}{70}$ test $\frac{1}{71}$ test $\frac{1}{72}$ test $\frac{1}{73}$ test $\frac{1}{74}$ test $\frac{1}{75}$ test $\frac{1}{76}$ test $\frac{1}{77}$ test $\frac{1}{78}$ test $\frac{1}{79}$ test $\frac{1}{80}$ test $\frac{1}{81}$ test $\frac{1}{82}$ test $\frac{1}{83}$ test $\frac{1}{84}$ test $\frac{1}{85}$ test $\frac{1}{86}$ test $\frac{1}{87}$ test $\frac{1}{88}$ test $\frac{1}{89}$ test $\frac{1}{90}$ test $\frac{1}{91}$ test $\frac{1}{92}$ test $\frac{1}{93}$ test $\frac{1}{94}$ test $\frac{1}{95}$ test $\frac{1}{96}$ test $\frac{1}{97}$ test $\frac{1}{98}$ test $\frac{1}{99}$ test $\frac{1}{100}$

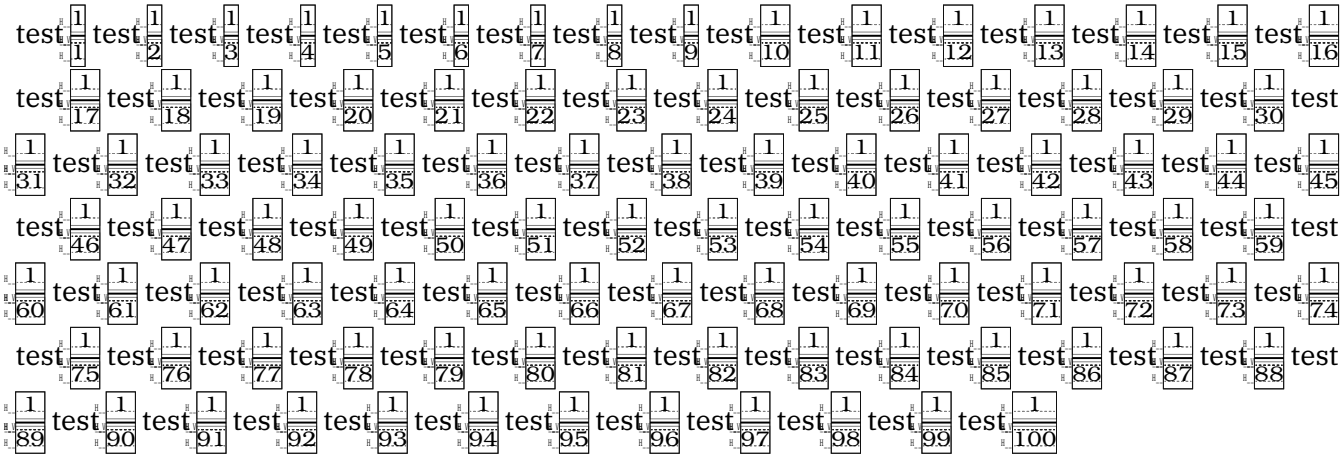
While with zero it looks like this, quite a different outcome:

test $\frac{1}{1}$ test $\frac{1}{2}$ test $\frac{1}{3}$ test $\frac{1}{4}$ test $\frac{1}{5}$ test $\frac{1}{6}$ test $\frac{1}{7}$ test $\frac{1}{8}$ test $\frac{1}{9}$ test $\frac{1}{10}$ test $\frac{1}{11}$ test $\frac{1}{12}$ test $\frac{1}{13}$ test $\frac{1}{14}$ test $\frac{1}{15}$ test $\frac{1}{16}$ test $\frac{1}{17}$ test $\frac{1}{18}$ test $\frac{1}{19}$ test $\frac{1}{20}$ test $\frac{1}{21}$ test $\frac{1}{22}$ test $\frac{1}{23}$ test $\frac{1}{24}$ test $\frac{1}{25}$ test $\frac{1}{26}$ test $\frac{1}{27}$ test $\frac{1}{28}$ test $\frac{1}{29}$ test $\frac{1}{30}$ test $\frac{1}{31}$ test $\frac{1}{32}$ test $\frac{1}{33}$ test $\frac{1}{34}$ test $\frac{1}{35}$ test $\frac{1}{36}$ test $\frac{1}{37}$ test $\frac{1}{38}$ test $\frac{1}{39}$ test $\frac{1}{40}$ test $\frac{1}{41}$ test $\frac{1}{42}$ test $\frac{1}{43}$ test $\frac{1}{44}$ test $\frac{1}{45}$ test $\frac{1}{46}$ test $\frac{1}{47}$ test $\frac{1}{48}$ test $\frac{1}{49}$ test $\frac{1}{50}$ test $\frac{1}{51}$ test $\frac{1}{52}$ test $\frac{1}{53}$ test $\frac{1}{54}$ test $\frac{1}{55}$ test $\frac{1}{56}$ test $\frac{1}{57}$ test $\frac{1}{58}$ test $\frac{1}{59}$ test $\frac{1}{60}$ test $\frac{1}{61}$ test $\frac{1}{62}$ test $\frac{1}{63}$ test $\frac{1}{64}$ test $\frac{1}{65}$ test $\frac{1}{66}$ test $\frac{1}{67}$ test $\frac{1}{68}$ test $\frac{1}{69}$ test $\frac{1}{70}$ test $\frac{1}{71}$ test $\frac{1}{72}$ test $\frac{1}{73}$ test $\frac{1}{74}$ test $\frac{1}{75}$ test $\frac{1}{76}$ test $\frac{1}{77}$ test $\frac{1}{78}$ test $\frac{1}{79}$ test $\frac{1}{80}$ test $\frac{1}{81}$ test $\frac{1}{82}$ test $\frac{1}{83}$ test $\frac{1}{84}$ test $\frac{1}{85}$ test $\frac{1}{86}$ test $\frac{1}{87}$ test $\frac{1}{88}$ test $\frac{1}{89}$ test $\frac{1}{90}$ test $\frac{1}{91}$ test $\frac{1}{92}$ test $\frac{1}{93}$ test $\frac{1}{94}$ test $\frac{1}{95}$ test $\frac{1}{96}$ test $\frac{1}{97}$ test $\frac{1}{98}$ test $\frac{1}{99}$ test $\frac{1}{100}$

A little tracing shows it more clearly:

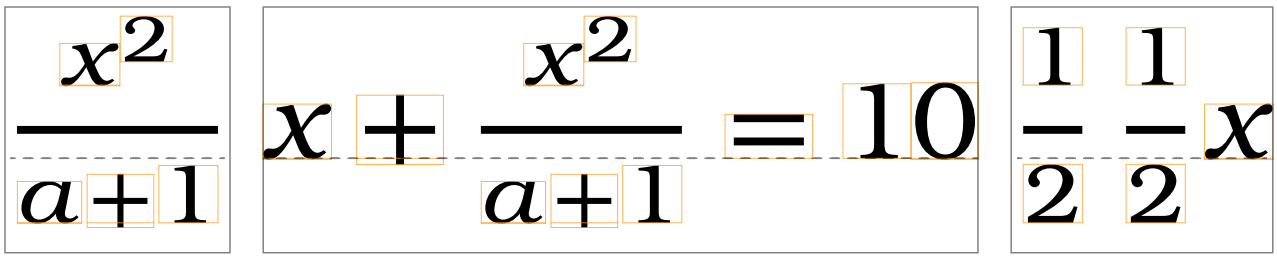


You can zoom in and see where it interferes with margin alignment.

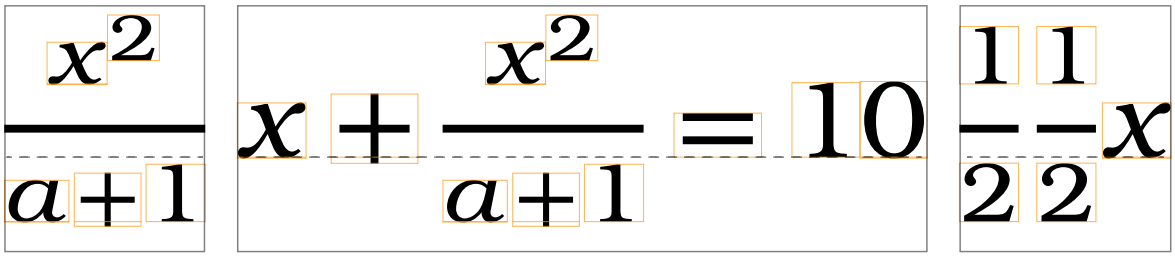


So, if you ever meet a user who claims perfection and superiority of typesetting, check out her/his work which might have inline fractions done the spacy way. It might make other visually typesetting claims less trustworthy. And yes, one can wonder if margin kerning could help here but as this content is wrapped in boxes it is unlikely to work out well (and not worth the effort).

In order to get a better picture of the spacing, two more renderings are shown. This time we show the bounding boxes of the characters too (you might need to zoom in to see it):

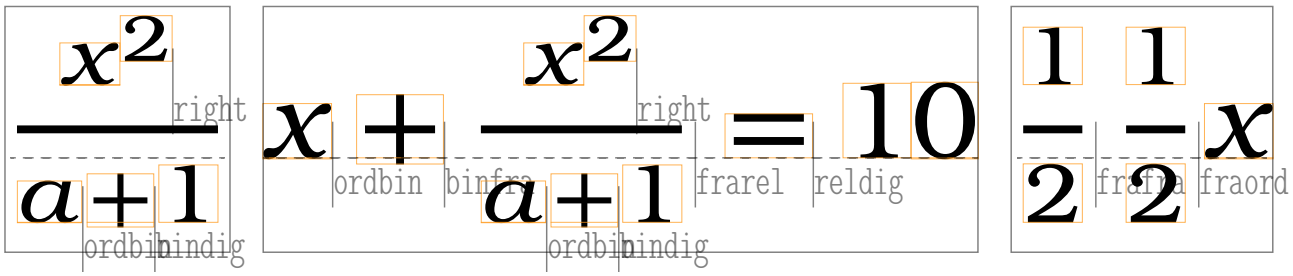


Again we also show the zero case

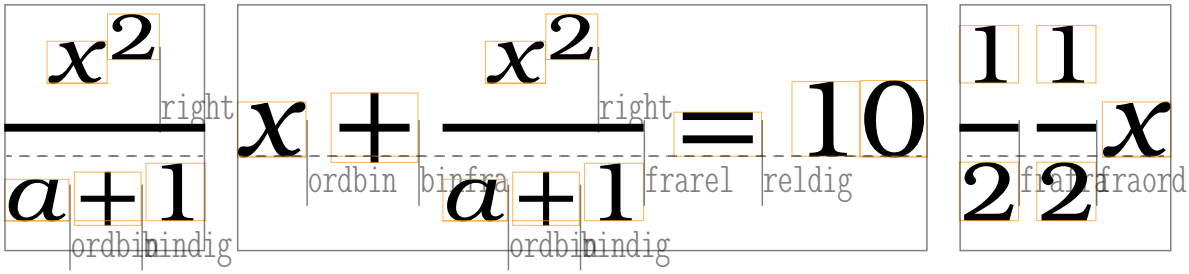


This makes clear why there actually is this extra space around a fraction: regular operators have side bearings and thereby have some added space. And when we put a fraction in front of a symbol we need that little extra space. Of course a proper class pair spacing value could do the job but there is no fraction class. The engine cheats by changing the class depending on what follows or came before and this is why on the average it looks okay. However, these examples demonstrate that there are some assumptions with regard to for instance fonts and this is one of the reasons why the more or less official expected OpenType behavior as dictated by the Cambria font doesn't always work out well for fonts that evolved from the ones used in the TEX community. Also imagine how this interferes with the fact that traditional TEX fonts and the machinery do magic with cheating about width combined with italic correction (all plausible and quite clever but somewhat tricky with respect to OpenType).

Because here we discuss the way LuaMetaTEX and ConTEXt deal with this, the following examples show a probably unexpected outcome. Again first the non-zero case:



And here the zero case:



I will not go into details about the way fractions are supported in the engine because some extensions are already around for quite a while. The main observation here is that in LuaMetaTEX we have alternative primitives that assume forward scanning, as if the numerator and denominator are arguments.

The engine also supports skewed (vulgar) fractions natively where numerator and denominator are raised and lowered relative to the (often) slash. Many aspects of the rendering can be tuned in the so called font goodie files, which is also the place where we define the additional font parameters.

Atom spacing

If you are familiar with traditional TEX you know that there is some built in `ordbin` spacing. But there is no such pair for a fraction and a relation, simply because there is no fraction class. However, in LuaMetaTEX there is one, and we'd better set it up if we zero the margins of a fraction.

It is worth noticing that fractions are sort of special anyway. The official syntax is `n \over m` and numerator and denominator can be sub formulas. This is the one case where the parser sort of has to look back, which is tricky because the machinery is a forward looking one. Therefore, in order to get the expected styling (or avoid unexpected side effects) one will normally wrap all in braces as in: `{ {n} \over{m} }` which of course kind defeats the simple syntax which probably is supported for `1\over2` kind of usage, so a next challenge is to make `1/2` come out right. All this means that in practice we have wrappers like `\frac` which accidentally in LuaMetaTEX can be defined using forward looking primitives with plenty extra properties driven by keywords. It also means that fractions as expected by the engine due to wrapping actually can be a different kind of atom, which can have puzzling side effects with respect to spacing (because the remapping happens unseen).

Interesting is that adapting LuaMetaTEX to a more extensive model was quite doable, also because the code base had already been made more configurable. Of course it involved quite a bit of tedious editing and throwing out already nice and clean code that had taken some effort, but that's the way it is. Of course more classes also means that some storage properties had to be adapted within the available space but by sacrificing families that was possible. With 64 potential classes we now are back to 64 families compared to 7 classes and 256 families in LuaTEX and 7 classes and 16 families in traditional TEX.

Also interesting is that the new implementation is actually somewhat simpler and therefore the binary is a tad smaller too. But does all that mean that there were no pitfalls? Sure there were! It is worth noticing that doing all this reminded me of the early days of LuaTEX development, where Taco and I exchanged binaries and TEX code in a more or less constant way using Skype. For LuaMetaTEX we used good old mail for files and Mojca's build farm for binaries and Mikael and I spent many months exchanging information and testing out alternatives on a daily basis: it is in my opinion the only way to do this and it's fun too. It has been a lot of work but once we got going there was nothing that could stop us. A side effect was that there were no updates during this period, which was something users noticed.

In the spacing matrix there is `inner` and internally there's also some care to be taken of `vcenter`. The `inner` class is actually shared with the `variable` class which is not so much a real class but more a signal to the engine that when an alphabetic or numeric character is included it has to come from a specific family: upright family zero or math italic family one in traditional speak. But, what if we don't have that setup? Well, then one has to make sure that this special class number is not associated (which is no big deal). It does mean that when we extend the repertoire of classes we cannot use slot seven. Always keep in mind that classes (and thereby signals) get assigned to characters (some defaults by the engine, others by the macro package). It is why in ConTEXt we use abstract class numbers, just in case the engine gets adapted.

We also cannot use slot eight because that one is a signal too: for a possible active math character, a feature somewhat complicated by the fact that it should not interfere with passing around such active characters in arguments. In math mode where we have lots of macros passing around content, this special class works around these side effects. We don't need this feature in ConTEXt because contrary to other macro packages we don't handle primes, pseudo superscripts potentially followed by other super and subscripts by making the ' an active character and thereby a macro in math mode. This trickery again closely relates to preferable input, font properties, and limitations of memory and such at the time TEX showed up (much has to fit into 8, 16 or 32 bits, so there is not much room for e.g. more than 8 classes). Since we started with MkIV the way math is dealt with is a bit different than normally done in TEX anyway.

Atom rules

We can now control the spacing between every atom but unfortunately that is not good enough. Therefore, we arrive at yet another feature built into the engine: turning classes into other classes depending on neighbors. And this is precisely why we have certain classes. Let's quote "TEX by Topic": *The cases `*` (in the atom spacing matrix) cannot occur, because a `bin` object is converted to `ord` if it is the first in the list, preceded by `bin`, `op`, `open`, `punct`, `rel`, or followed by `close`, `punct`, or `rel`; also, a `rel` is converted to `ord` when it is followed by `close` or `punct`.*

We can of course keep these hard coded heuristics but can as well make that bit of code configurable, which we did. Below is demonstrated how one can set up the defaults at the TEX end. We use symbolic names for the classes.

```
\setmathatomrule  \mathbegincode          \mathbinarycode       % old
  \allmathstyles    \mathordinarycode       \mathordinarycode   % new

\setmathatomrule  \mathbinarycode         \mathbinarycode
  \allmathstyles    \mathbinarycode         \mathordinarycode
\setmathatomrule  \mathoperatorcode       \mathbinarycode
  \allmathstyles    \mathoperatorcode         \mathordinarycode
\setmathatomrule  \mathopencode           \mathbinarycode
  \allmathstyles    \mathopencode           \mathordinarycode
\setmathatomrule  \mathpunctuationcode    \mathbinarycode
  \allmathstyles    \mathpunctuationcode    \mathordinarycode
\setmathatomrule  \mathrelationcode       \mathbinarycode
  \allmathstyles    \mathrelationcode       \mathordinarycode

\setmathatomrule  \mathbinarycode         \mathclosecode
  \allmathstyles    \mathordinarycode       \mathclosecode
\setmathatomrule  \mathbinarycode         \mathpunctuationcode
  \allmathstyles    \mathordinarycode       \mathpunctuationcode
\setmathatomrule  \mathbinarycode         \mathrelationcode
  \allmathstyles    \mathordinarycode       \mathrelationcode

\setmathatomrule  \mathrelationcode       \mathclosecode
  \allmathstyles    \mathordinarycode       \mathclosecode
\setmathatomrule  \mathrelationcode       \mathpunctuationcode
  \allmathstyles    \mathordinarycode       \mathpunctuationcode
```

Watch the special class with \mathbegincode. This is actually class 62 so you don't need much fantasy to imagine that class 63 is \mathendcode, but that one is not yet used. In a similar fashion we can initialize the spacing itself:[5]

```
\setmathspacing \mathordcode   \mathopcode     \allmathstyles  \thinmuskip
\setmathspacing \mathordcode   \mathbincode    \allsplitstyles \medmuskip
\setmathspacing \mathordcode   \mathrelcode    \allsplitstyles \thickmuskip
\setmathspacing \mathordcode   \mathinnercode  \allsplitstyles \thinmuskip

\setmathspacing \mathopcode    \mathordcode    \allmathstyles  \thinmuskip
\setmathspacing \mathopcode    \mathopcode     \allmathstyles  \thinmuskip
\setmathspacing \mathopcode    \mathrelcode    \allsplitstyles \thickmuskip
\setmathspacing \mathopcode    \mathinnercode  \allsplitstyles \thinmuskip

\setmathspacing \mathbincode   \mathordcode    \allsplitstyles \medmuskip
\setmathspacing \mathbincode   \mathopcode     \allsplitstyles \medmuskip
\setmathspacing \mathbincode   \mathopencode   \allsplitstyles \medmuskip
\setmathspacing \mathbincode   \mathinnercode  \allsplitstyles \medmuskip

\setmathspacing \mathrelcode   \mathordcode    \allsplitstyles \thickmuskip
\setmathspacing \mathrelcode   \mathopcode     \allsplitstyles \thickmuskip
\setmathspacing \mathrelcode   \mathopencode   \allsplitstyles \thickmuskip
\setmathspacing \mathrelcode   \mathinnercode  \allsplitstyles \thickmuskip

\setmathspacing \mathclosecode \mathopcode     \allmathstyles  \thinmuskip
\setmathspacing \mathclosecode \mathbincode    \allsplitstyles \medmuskip
\setmathspacing \mathclosecode \mathrelcode    \allsplitstyles \thickmuskip
\setmathspacing \mathclosecode \mathinnercode  \allsplitstyles \thinmuskip

\setmathspacing \mathpunctcode \mathordcode    \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathopcode     \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathrelcode    \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathopencode   \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathclosecode  \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathpunctcode  \allsplitstyles \thinmuskip
\setmathspacing \mathpunctcode \mathinnercode  \allsplitstyles \thinmuskip

\setmathspacing \mathinnercode \mathordcode    \allsplitstyles \thinmuskip
\setmathspacing \mathinnercode \mathopcode     \allmathstyles  \thinmuskip
\setmathspacing \mathinnercode \mathbincode    \allsplitstyles \medmuskip
\setmathspacing \mathinnercode \mathrelcode    \allsplitstyles \thickmuskip
\setmathspacing \mathinnercode \mathopencode   \allsplitstyles \thinmuskip
\setmathspacing \mathinnercode \mathpunctcode  \allsplitstyles \thinmuskip
\setmathspacing \mathinnercode \mathinnercode  \allsplitstyles \thinmuskip
```

And because we have a few more atom classes this also needs to happen:

```
\letmathspacing  \mathactivecode   \mathordinarycode
```

[5] Constant, engine specific, numbers like these are available in tables at the Lua end so we can change them and users can check that.

```
\letmathspacing  \mathvariablecode \mathordinarycode
\letmathspacing  \mathovercode     \mathordinarycode
\letmathspacing  \mathundercode    \mathordinarycode
\letmathspacing  \mathfractioncode \mathordinarycode
\letmathspacing  \mathradicalcode  \mathordinarycode
\letmathspacing  \mathmiddlecode   \mathopencode
\letmathspacing  \mathaccentcode   \mathordinarycode

\letmathatomrule \mathactivecode   \mathordinarycode
\letmathatomrule \mathvariablecode \mathordinarycode
\letmathatomrule \mathovercode     \mathordinarycode
\letmathatomrule \mathundercode    \mathordinarycode
\letmathatomrule \mathfractioncode \mathordinarycode
\letmathatomrule \mathradicalcode  \mathordinarycode
\letmathatomrule \mathmiddlecode   \mathopencode
\letmathatomrule \mathaccentcode   \mathordinarycode
```

With `\resetmathspacing` we get an all-zero state but that might become more refined in the future. What is not clear from the above is that there is also an inheritance mechanism. The three special muskip registers are actually shortcuts so that changing the register value is reflected in the spacing. When a regular muskip value is (verbose or as register) that value is sort of frozen. However, the `\inherited` prefix will turn references to registers and constants into a delayed value: as with the predefined we now have a more dynamic behavior which means that we can for instance use reserved muskip registers as we can use the predefined. A bonus is that one can also use regular glue or dimensions, just in case one wants the same spacing in all styles (a muskip adapts to the size).

When you look at all of the above you might wonder how users are supposed to deal with math spacing. The answer is that often they can just assume that TEX does the right thing. If something somehow doesn't feel right, looking at solutions by others will probably lead a new user to just copy a trick, like injecting a `\thinmuskip`. But it can be that atoms depend on the already applied (or not) spacing, which in turn depends on values in the atom spacing matrix that probably only a few users have seen. So, in the end it all boils down to trust in the engine and one's eyesight combined with hopefully some consistency in adding space directives and often with TEX it is consistency that makes documents look right. In ConTEXt we have many more classes even if only a few characters fit in, like differential, exponential and imaginary.

Fractions again

We now return to the fraction molecule. With the mechanisms at our disposal we can change the fixed margins to more adaptive ones:

```
\inherited\setmathspacing \mathbinarycode   \mathfractioncode
    \allmathstyles \thickermuskip
\inherited\setmathspacing \mathfractioncode \mathbinarycode
    \allmathstyles \thickermuskip
\nulldelimiterspace\zeropoint
$x + \frac{1}{x+2} + x$
```

Here \thickermuskip is defined as 7mu plus 5mu where the stretch is the same as a \thickmuskip and the width 2mu more. We start out with three variants, where the last two have \nulldelimiterspace set to 0pt and the first one uses the 1.2pt.

$$x + \frac{1}{x+2} + x$$

$$x + \frac{1}{x+2} + x$$

$$x + \frac{1}{x+2} + x$$

When we now apply the new settings to the last one, and overlay them we get the following output: the first and last case are rather similar which is why this effort was started in the first place.

$$x + \frac{1}{x+2} + x$$

Of course these changes are not upward compatible but as they are tiny they are not that likely to change the number of lines in a paragraph. In display mode changes in horizontal dimensions also have little effect.

Penalties

An inline formula can be broken across lines, and for sure there are places where you don't want to break or prefer to break. In TEX line breaks can be influenced by using penalties. At the outer level of an inline math formula, we can have a specific penalty before and after a binary and/or relation. The defaults are such that there are no penalties set, but most macro packages set the so called \relpenalty and \binoppenalty (the op in this name does not relate to the operator class) so a value between zero and 1000. In LuaTEX we also have \pre variants of these, so we have four penalties that can be set, but that is not enough in our new approach.

These penalties are class bound and don't relate to styles, like atom spacing does. That means that while atom spacing involves 64 × 64 × 8 potential values, an amount that we can manage by using the discussed inheritance. The inheritance takes less values because which store 4 style values per class in one number. For penalties we only need to keep 64 × 2 in mind, plus a range of inheritance numbers. Therefore it was decided to also generalize penalties so that each class can have them. The magic commands are shown with some useless examples:

```
\letmathparent \mathdigitcode
    \mathbincode   % pre penalty
    \mathbincode   % post penalty
    \mathdigitcode % options
    \mathdigitcode % reserved
```

By default the penalties are on their own, like:

```
\letmathparent \mathdigitcode
  \mathdigitcode % pre penalty
  \mathdigitcode % post penalty
  \mathdigitcode % options
  \mathdigitcode % reserved
```

The options and reserved parent mapping are not (yet) discussed here. Unless values are assigned they are ignored.

```
\setmathprepenalty  \mathordcode 100
\setmathpostpenalty \mathordcode 600
\setmathprepenalty  \mathbincode 200
\setmathpostpenalty \mathbincode 700
\setmathprepenalty  \mathrelcode 300
\setmathpostpenalty \mathrelcode 800
```

As with spacing, when there is no known value, the parent will be consulted. An unset penalty has a value of 10000.

After discussing the implications of inline math crossing lines, Mikael and I decided there can be two solutions. Both can of course be implemented in Lua, but on the other hand, they make good extensions, also because it sort of standardized it. The first advanced control feature tweaks penalties:

```
\mathforwardpenalties  2  200 100
\mathbackwardpenalties 2  100  50
```

This will add 200 and 100 to the first two math related penalties, and 100 and 50 to the last two (watch out: the 100 will be assigned to the last one found, the 50 to the one before it). As with all things penalty and line break related, you need to have some awareness of how non-linear the badness calculation is as well of the fact that the tolerance and stretch related parameters play a role here.

The second tweak is setting `\maththreshold` to some value. When set to for instance `40pt`, formulas that take less space than this will be wrapped in a `\hbox` and thereby will never break across a page.[6]

[6] A future version might inject severe penalties instead, time will learn.

Actually that second tweak has a variant so we have three tweaks! Say that we have this sample formula wrapped in some bogus text and repeat that snippet a lot of times:

```
x xx xxx xxxx $1 + x$ x xx xxx xxxx
```

Now look at the example on the next page. You will notice that the red and blue text have different line breaks. This is because we have given the threshold some stretch and shrink. The red text has a zero threshold so it doesn't do any magic at all, while the second has this setup:

```
\setupmathematics[threshold=medium]
```

That setting set the threshold to `4em plus 0.75em minus 0.50em` and when the formula size exceeds the four quads the line break code will use the real formula width but with the given stretch and shrink. Eventually the calculated size will be used to repackage the formula. In the future we will also provide a way to define slack more relative to the size and/or number of atoms.

Another way to influence line breaks is to use the two inline math related penalties that have been added at Mikael's suggestion:

```
\setupalign[verytolerant]
{\dorecurse{25}{test $\darkred   #1^{#1} + x_{#1}^{#1}$ test }\blank}
{\preinlinepenalty  500 \postinlinepenalty -500
 \dorecurse{25}{test $\darkgreen #1^{#1} + x_{#1}^{#1}$ test }\blank}
{\postinlinepenalty 500 \preinlinepenalty  -500
 \dorecurse{25}{test $\darkblue  #1^{#1} + x_{#1}^{#1}$ test }\blank}
```

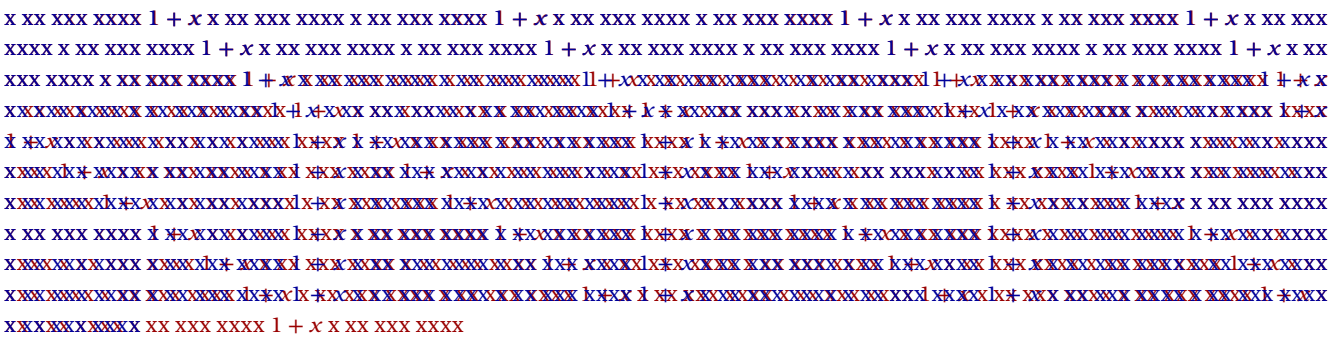
To get an example that shows the effect takes a bit of trial and error because TeX does a very good job in line breaking. This is why we've set the tolerance and also use negative penalties.

In addition to the `\mathsurround` (kern) and `\mathsurroundskip` (glue) parameters this is a property of the nodes that mark the beginning and end of an inline math formula.

test $1^1 + x_1^1$ test test $2^2 + x_2^2$ test test $3^3 + x_3^3$ test test $4^4 + x_4^4$ test test $5^5 + x_5^5$ test test $6^6 + x_6^6$ test test
$7^7 + x_7^7$ test test $8^8 + x_8^8$ test test $9^9 + x_9^9$ test test $10^{10} + x_{10}^{10}$ test test $11^{11} + x_{11}^{11}$ test test $12^{12} + x_{12}^{12}$
test test $13^{13} + x_{13}^{13}$ test test $14^{14} + x_{14}^{14}$ test test $15^{15} + x_{15}^{15}$ test test $16^{16} + x_{16}^{16}$ test test $17^{17} + x_{17}^{17}$
test test $18^{18} + x_{18}^{18}$ test test $19^{19} + x_{19}^{19}$ test test $20^{20} + x_{20}^{20}$ test test $21^{21} + x_{21}^{21}$ test test $22^{22} + x_{22}^{22}$
test test $23^{23} + x_{23}^{23}$ test test $24^{24} + x_{24}^{24}$ test test $25^{25} + x_{25}^{25}$ test

test $1^1 + x_1^1$ test test $2^2 + x_2^2$ test test $3^3 + x_3^3$ test test $4^4 + x_4^4$ test test $5^5 + x_5^5$ test test $6^6 + x_6^6$ test test
$7^7 + x_7^7$ test test $8^8 + x_8^8$ test test $9^9 + x_9^9$ test test $10^{10} + x_{10}^{10}$ test test $11^{11} + x_{11}^{11}$ test test $12^{12} + x_{12}^{12}$
test test $13^{13} + x_{13}^{13}$ test test $14^{14} + x_{14}^{14}$ test test $15^{15} + x_{15}^{15}$ test test $16^{16} + x_{16}^{16}$ test test $17^{17} + x_{17}^{17}$
test test $18^{18} + x_{18}^{18}$ test test $19^{19} + x_{19}^{19}$ test test $20^{20} + x_{20}^{20}$ test test $21^{21} + x_{21}^{21}$ test test $22^{22} + x_{22}^{22}$
test test $23^{23} + x_{23}^{23}$ test test $24^{24} + x_{24}^{24}$ test test $25^{25} + x_{25}^{25}$ test

test $1^1 + x_1^1$ test test $2^2 + x_2^2$ test test $3^3 + x_3^3$ test test $4^4 + x_4^4$ test test $5^5 + x_5^5$ test test $6^6 + x_6^6$ test test
$7^7 + x_7^7$ test test $8^8 + x_8^8$ test test $9^9 + x_9^9$ test test $10^{10} + x_{10}^{10}$ test test $11^{11} + x_{11}^{11}$ test test $12^{12} + x_{12}^{12}$ test
test $13^{13} + x_{13}^{13}$ test test $14^{14} + x_{14}^{14}$ test test $15^{15} + x_{15}^{15}$ test test $16^{16} + x_{16}^{16}$ test test $17^{17} + x_{17}^{17}$ test
test $18^{18} + x_{18}^{18}$ test test $19^{19} + x_{19}^{19}$ test test $20^{20} + x_{20}^{20}$ test test $21^{21} + x_{21}^{21}$ test test $22^{22} + x_{22}^{22}$ test
test $23^{23} + x_{23}^{23}$ test test $24^{24} + x_{24}^{24}$ test test $25^{25} + x_{25}^{25}$ test



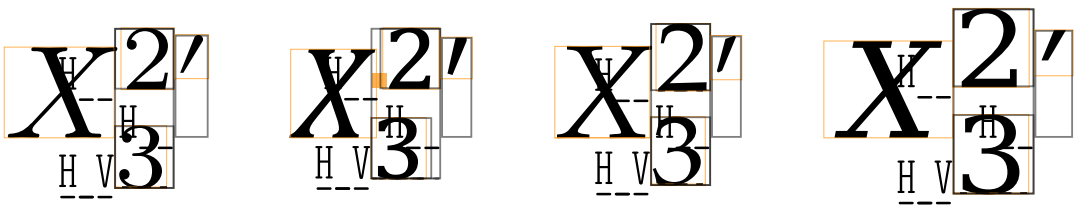
Flattening

The traditional engine has some code for flattening math constructs that in the end are just one character. So in the end, `\tilde{u}` and `\tilde {uu}` become different objects even if both are in fact accents. In fact, when an accent is constructed there is a special code path for single characters so that script placement adapts to the shape of that character.

However because of interaction with primes, which themselves are sort of superscripts and due to the somewhat weird way fonts provide them when it comes to positioning and sizes, in ConTEXt we already are fooling around a bit with these characters. For understandable reasons of memory usage, complexity and eightbitness primes are not a native TEX thing but more something that is handled at the macro level (although not in MkIV and LMTX).

In the end it was script placements on (widely) accented math characters that made us introduce a dedicated `\Umathprime` primitive that adds a prime to a math atom. It permits an uninterupted treatment of scripts while in the final assembly of the molecule the prime, superscript, subscript and maybe even prescripts that prime gets squeezed in. Because the concept of primes is missing in OpenType math an additional font parameter `PrimeTopRaisePercent` has been introduced as well as an `\Umathprimeraise` primitive. In retrospect I should have done that earlier but one tends to stick to the original as much as possible. However, at some point Mikael and I reached a state where we decided that proper (clean) engine extensions make way more sense than struggling with border cases and explaining users why things are so complicated.

The input `$ X \Uprimescript{'} ^2 _3 $` gives this:



Latin Modern Cambria Pagella Dejavu

With `\tracingmath = 1` this nicely traces as:

```
> \inlinemath=
\noad[ord][...]
.\nucleus
..\mathchar[ord] family "0, character "58
.\superscript
..\mathchar[dig] family "0, character "32
```

```
.\subscript
..\mathchar[dig] family "0, character "32
.\primescript
..\mathchar[ord] family "0, character "27
```

Of course this feature can also be used for other prime like ornaments and who knows how it will evolve over time.

You can influence the positioning with `\Umathprimesupshift` which adds some kern between a prime and superscript. The `\Umathextraprimeshift` moves a prime up. The `\Umathprimeraise` is a font parameter that defaults to 25 which means a raise of 25%of the height. These are all (still) experimental parameters.

Fences

Fences can be good for headaches. Because the math that I (or actually my colleague) deal with is mostly school math encoded in presentation MathML (sort or predictable) or some form of sequential ascii based input (often rather messy and therefore unpredictable due to ambiguity) fences are a pain. A TEXie can make sure that left and right fences are matched. A TEXie also knows when something is an inline parenthesis or when a more high level structure is needed, for instance when parentheses have to scale with what they wrap. In that case the `\left` and `\right` mechanism is used. In arbitrary input missing one of those is fatal. Therefore, handling of fences in ConTEXt is one of the more complex sub mechanisms: we not only need to scale when needed, but also catch asymmetrical usage.

A side effect of the encapsulating fencing construct is that it wraps the content in a so called inner (as in `\mathinner`) which means that we get a box, and it is a well known property of boxes that they don't break across lines. With respect to fences, a way out is to not really fence content, but do something like this:

```
\left(\strut\right. x + 1 \left.\strut\right)
```

and hope for the best. Both pairs are coupled in the sense that their sizes will match and the strut is what determines the size. So, as long as there is a proper match of struts all is well, but it is definitely a decent hack. The drawback is in the size of the strut: if a formula needs a higher one, larger struts have to be used. This is why in plain TEX we have these commands:

```
\def\bigl {\mathopen \big } \def\bigm {\mathrel\big } \def\bigr {\mathclose\big }
\def\Bigl {\mathopen \Big } \def\Bigm {\mathrel\Big } \def\Bigr {\mathclose\Big }
\def\biggl{\mathopen \bigg} \def\biggm{\mathrel\bigg} \def\biggr{\mathclose\bigg}
\def\Biggl{\mathopen \Bigg} \def\Biggm{\mathrel\Bigg} \def\Biggr{\mathclose\Bigg}

\def\big #1{{\hbox{$\left#1\vbox to  8.5pt{}\right.\nomathspacing$}}}
\def\Big #1{{\hbox{$\left#1\vbox to 11.5pt{}\right.\nomathspacing$}}}
\def\bigg#1{{\hbox{$\left#1\vbox to 14.5pt{}\right.\nomathspacing$}}}
\def\Bigg#1{{\hbox{$\left#1\vbox to 17.5pt{}\right.\nomathspacing$}}}

\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt} % renamed
```

The middle is kind of interesting because it has relation properties, while the `\middle` introduced in ε-TEX got open properties, but we leave that aside.

In ConTEXt we have plenty of alternatives, including these commands, but they are defined differently. For instance they adapt to the font size. The hard coded point sizes in the plain TEX code relates to the

font and steps available in there (either by next larger or by extensible). The values thereby need to be adapted to the chosen body font as well as the body font size. In MkIV and even better in LMTX we can actually consult the font and get more specific sizes.

But, this section is not about how to get these fixed sizes. Actually, the need to choose explicitly is not what we want, especially because TEX can size delimiters so well. So, take this code snippet:

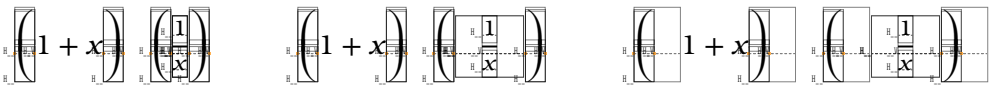
```
$ x = \left( \dorecurse{40}{\frac{x}{x+#1} +} x \right) $
```

When we typeset this inline, as in $x = \left(\frac{x}{x+1} + \frac{x}{x+2} + \frac{x}{x+3} + \frac{x}{x+4} + \frac{x}{x+5} + \frac{x}{x+6} + \frac{x}{x+7} + \frac{x}{x+8} + \frac{x}{x+9} + \frac{x}{x+10} + \frac{x}{x+11} + \frac{x}{x+12} + \frac{x}{x+13} + \frac{x}{x+14} + \frac{x}{x+15} + \frac{x}{x+16} + \frac{x}{x+17} + \frac{x}{x+18} + \frac{x}{x+19} + \frac{x}{x+20} + \frac{x}{x+21} + \frac{x}{x+22} + \frac{x}{x+23} + \frac{x}{x+24} + \frac{x}{x+25} + \frac{x}{x+26} + \frac{x}{x+27} + \frac{x}{x+28} + \frac{x}{x+29} + \frac{x}{x+30} + \frac{x}{x+31} + \frac{x}{x+32} + \frac{x}{x+33} + \frac{x}{x+34} + \frac{x}{x+35} + \frac{x}{x+36} + \frac{x}{x+37} + \frac{x}{x+38} + \frac{x}{x+39} + \frac{x}{x+40} + x \right)$, we get nicely scaled fences but in a way that permits line breaks. The reason is that the engine has been extended with a `fenced` class so that we can recognize later on, when TEX comes to injecting spaces and penalties, that we need to unpack the construct. It is another beneficial side effect of the generalization.

The Plain TEX code can be used to illustrate some of what we discussed before about fractions. In the next code we use excessive delimiter spacing:

```
\def\Bigg#1{%  watch the wrapping in a box
  {%
    \hbox {%
        $\normalleft#1\vbox to 17.5pt{}\normalright.\nomathspacing$%
    }%
  }%
}

\nulldelimiterspace0pt
\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par

\nulldelimiterspace10pt
\def\nomathspacing{\nulldelimiterspace0pt\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par

\nulldelimiterspace10pt
\def\nomathspacing{\mathsurround0pt}

$\Bigg( 1 + x\Bigg) \quad \Bigg( \frac{1}{x}\Bigg)$\par
```

This renders as follows. We explicitly set `\nulldelimiterspace` to values because in ConTEXt it is now zero by default.



0pt with reset at end

10pt with reset at end

10pt without reset at end

Radicals

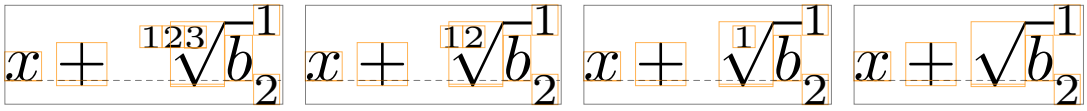
In traditional TEX a radical with degree is defined as macro. That macro does some measurements and typesets the result in four sizes for a choice. The macro typesets the degree in a box that contains the degree as formula. There is a less guesswork going on than with respect to how the radical symbol is shaped but as we're talking plain TEX here it works out okay because the default font is well known.

Radicals are a nice example of a two dimensional 'extender' but only the vertical dimension uses the extension mechanism, which itself operates either horizontally or vertically, although in principle it could go both ways. The horizontal extension is a rule and the fact that the shape is below the baseline (as are other large symbols) will make the rule connect well: the radical shape sticks out a little, so one can think of the height reflecting the rule height.[7] In OpenType fonts there is a parameter and in LuaTEX we use the default rule thickness for traditional fonts, which is correct for Latin Modern. There are more places in the fonts where the design relates to this thickness, for instance fraction rules are supposed to match the minus, but this is a bit erratic if you compare fonts. This is one of the corrections we apply in the goodie files.

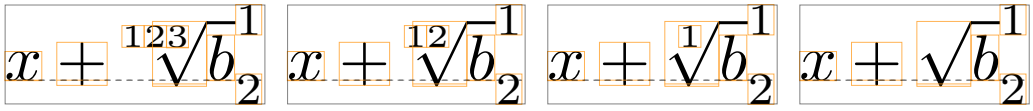
In OpenType the specification of the radical also includes spacing properties of the degree and that is why we have a primitive in LuaTEX that also handles the degree. It is what we used in ConTEXt MkIV. But ... we actually end up with a situation that compares to the already discussed fraction: there is space added before a radical when there is a degree. However, because we now have a radical atom class, we can avoid using that one and use the new pairwise spacing. Some fuzzy spacing logic in the engine could therefore be removed and we assume that `\Umathradicaldegreebefore` is zero. For the record: the `\Umathradicaldegreeafter` sort of tells how much space there is above the low part of the root, which means that we can compensate for multi-digit degrees.

Zeroing a parameter is something that relates to a font which means that it has to happen for each math font which in turn can mean a family-style combination. In order to avoid that complication (or better: to avoid tracing clutter) we have a way to disable a parameter:

```
\ruledhbox{$x + \sqrt[123]{b}^1_2$}
\ruledhbox{$x + \sqrt[12] {b}^1_2$}
\ruledhbox{$x + \sqrt[1]  {b}^1_2$}
\ruledhbox{$x + \sqrt     {b}^1_2$}
```



```
\setmathignore\Umathradicaldegreebefore 0
```



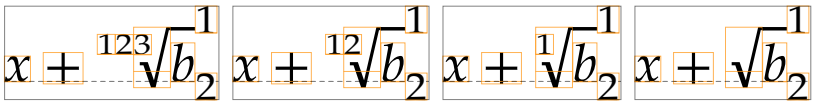
```
\setmathignore\Umathradicaldegreebefore 1
```

Latin Modern

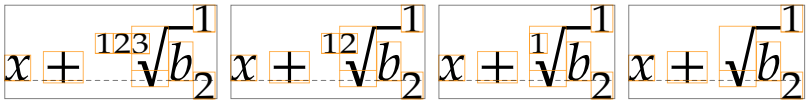
One problem with these spacing parameters is that they are inconsistent across fonts. The Latin Modern has a rather large space before the degree, while Cambria and Pagella have little. That means that

[7] When you zoom in you will notice that this is not always optimal because of the way the slope touched the rule.

when you prototype a mechanism the chosen solution can look great but not so much when at some point you use another font.



```
\setmathignore\Umathradicaldegreebefore 0
```



```
\setmathignore\Umathradicaldegreebefore 1
```

Cambria

More fences

One of the reasons why the MkII and MkIV fence related mechanism is somewhat complex is that we want a clean solution for filtering fences like parenthesis by size, something that in the traditional happens via a fake fence pair that encapsulates a strut of a certain size. In LMTX we use the same approach but have made the sequence more configurable. In practice that means that the values 1 up to 4 are just that but for some fonts we use the sequence `1 3 5 7`. There was no need to adapt the engine as it already worked quite well.

Integrals

The Latin Modern fonts have only one size of big operators and one reason can be that there is no need for more. Another reason can be that there was just no space in the font. However, an OpenType font has plenty slots available and the reference font Cambria has integral signs in sizes as well as extensibles.

In LuaTEX we already have generic vertical extensibles but that only works well with specified sizes. And, cheating with delimiters has the side effect that we get the wrong spacing. In LuaMetaTEX however we have ways to adapt the size to what came or what comes. In fact, it is a mechanism that is available for any atom that we support. However, it doesn't play well with script and this whole `\limits` and `\nolimits` is a bit clumsy so Mikael and I decided that different route had to be followed. For adaptive large operators we provide this interface:

```
$ x + \integral [color=darkred,top={t},bottom={b}] {\frac{1}{x}} = 10 $

$ x + \startintegral [color=darkblue,top={t},bottom={b}]
    \frac{1}{x}
\stopintegral = 10 $

$ x + \startintegral [color=darkgreen,top={t},bottom={b},method=vertical]
    \frac{1}{x}
\stopintegral= 10 $
```

This text is not about the user interface so we won't discuss how to define additional large operators using one-liners.

$$x + \int_b^t \frac{1}{x} = 10 \quad x + \int_b^t \frac{1}{x} = 10 \quad x + \int_b^t \frac{1}{x} = 10$$

The low level LuaMetaTEX implementation handles this input:

```
\Uoperator          \Udelimiter "0 \fam "222B {top} {bottom} {...}
\Uoperator   limits \Udelimiter "0 \fam "222B {top} {bottom} {...}
\Uoperator nolimits \Udelimiter "0 \fam "222B {top} {bottom} {...}
```

plus the usual keywords that fenced accept, because after all, this is just a special case of fencing.

Currently these special left operators are implemented as a special case of fences because that mechanism does the scaling. It means that we need a (bogus) right fence, or need to brace the content (basically create an atom). When no right fence is found one is added automatically. Because there is no real fencing, right fences are removed when processing takes place. When you specify a `class` that one will be used for the left and right spacing, otherwise we have open/close spacing.

Going details

When the next feature was explored Mikael tagged it as math micro typography and the reason is that you need not only to set up the engine for it but also need to be aware of this kind of spacing. Because we wanted to get rid of this script spacing that the font imposes we configured ConTEXt with:

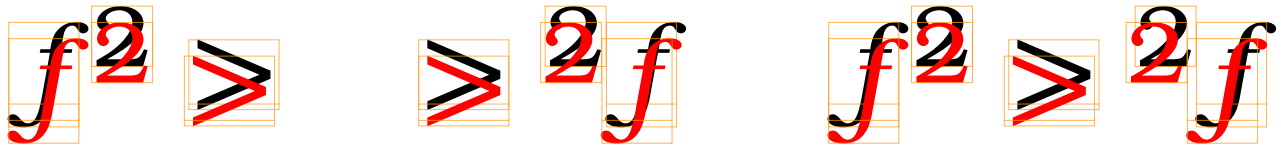
```
\setmathignore\Umathspacebeforescript\plusone
\setmathignore\Umathspaceafterscript \plusone
```

This basically nils all these tiny spaces. But the latest configuration has this instead:

```
% \setmathignore \Umathspacebeforescript\zerocount % default
% \setmathignore \Umathspaceafterscript \zerocount % default

\mathslackmode \plusone

\setmathoptions\mathopcode      \plusthree
\setmathoptions\mathbinarycode  \plusthree
\setmathoptions\mathrelationcode\plusthree
\setmathoptions\mathopencode    \plusthree
\setmathoptions\mathclosecode   \plusthree
\setmathoptions\mathpunctcode   \plusthree
```

This tells the engine to convert these spaces into what we call slack: disposable kerns at the edges. But it also converts these kerns into a glue component when possible. As with all these extensions it complicates the machinery but users will never see that. Now, the last six lines do the magic that made us return to honoring the spaces: we can tell the engine to ignore this slack when there are specific classes at the edges. These options are a bitset and 1 means “no slack left” and 2 means “no slack right” so 3 sets both.

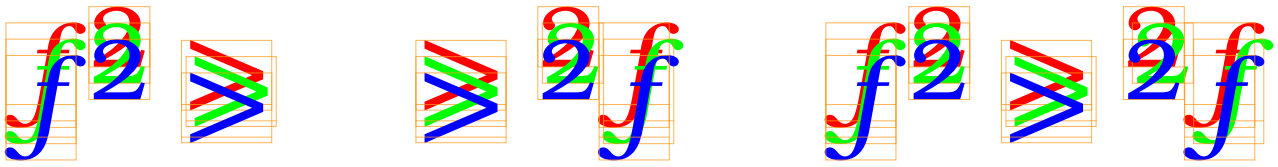
```
\def\TestSlack#1%
  {\vbox\bgroup
      \mathslackmode\zerocount
      \hbox\bgroup
         \setmathignore\Umathspacebeforescript\zerocount
         \setmathignore\Umathspaceafterscript \zerocount
         #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\red
        \setmathignore\Umathspacebeforescript\plusone
        \setmathignore\Umathspaceafterscript \plusone
         #1
      \egroup
   \egroup}

\startcombination[nx=3]
    {\showglyphs\TestSlack{$f^2 >     $}} {}
    {\showglyphs\TestSlack{$    > f^^2$}} {}
    {\showglyphs\TestSlack{$f^2 > f^^2$}} {}
\stopcombination
```



Because this overall removal of slack is not granular enough a while later we introduced a way to set this per class, as is demonstrated in the following example.

```
\def\TestSlack#1%
  {\vbox\bgroup
      \mathslackmode\plusone
      \hbox\bgroup\red
        \setmathignore\Umathspacebeforescript\zerocount
        \setmathignore\Umathspaceafterscript \zerocount
        #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\green
        \setmathoptions\mathrelationcode \zerocount
        #1
      \egroup
      \vskip-.9\lineheight
      \hbox\bgroup\blue
        \setmathoptions\mathrelationcode \plusthree
        #1
      \egroup
   \egroup}
```

```
\startcombination[nx=3]
    {\showglyphs\TestSlack{$f^2 >     $}} {}
    {\showglyphs\TestSlack{$    > f^^2$}} {}
    {\showglyphs\TestSlack{$f^2 > f^^2$}} {}
\stopcombination
```



Of course we need to experiment a lot with real documents and it might take a while before all this is stable (in the engine and in ConTEXt). And as we don't need to conform to the decades old golden TEX math standards we have some degrees of freedom in this: for Mikael and me it is pretty much a visual thing where we look closely at large samples. Of course in practice details get lost when we print at 10 point but that doesn't mean we can't provide the best experience.[8]

As we mention class specific options, we also need to mention the special case where we have for instance simple formulas like single atoms (for instance digits) are preceded by a sign (binary). These special spacing cases are handled by a lookahead flag that can be set `\setmathoptions <class>`, like the slack flags. More options might become available in due time. When set the lookahead will check for the automatically injected end class atom and use that for spacing when found. The mentioned lookahead is one of the hard coded heuristics in the traditional engine but here we need to explicitly configure it.

Ghosts

As plain TEX has macros like `\vphantom` you also find them in macro packages that came later. These create a boxes that have their content removed after the dimensions are set. They take space and are invisible but there's also nothing there.

A variant in the upgraded math machinery are ghosts: these are visible in the sense that they show up but ignored when it comes to spacing. Here is an example. The option bit set here tells the engine that we ghost at the right, so we have ghosts around the relation (it controls where the spacing ends up).

```
$
  x
  \mathatom class \mathghostcode                  {!!}
  >
  \mathatom class \mathghostcode options "00000020 {!!}
  1
  \quad
  x
  \mathatom class \mathghostcode                  {\hbox{\smallinfofont ord}}
  >
  \mathatom class \mathghostcode options "00000020 {\hbox{\smallinfofont dig}}
```

[8] Whenever I look at (my) old (math) school books I realize that Don Knuth had very good reasons to come up with TEX and, it being hard to beat, TEX still sets the standard!

```
  1
$
```

You never know when this comes in handy but it fits in the new, more granular approach to spacing. The code above shows that it's just a class, this time with number 17.

Struts

In order to get consistent spacing the ConTEXt macro package makes extensive use of struts in text mode as well as math mode. The normal way to implement that is either an empty box or a zero width rule, both with a specifically set height and depth. In ConTEXt MkII and MkIV (and for a long time in LMTX too) they were rules so that we could visualize them: they get some width and kerns around them to compensate for that.

In order to not let them interfere with spacing we could wrap them into a ghost atom but it is kind of ugly. Anyway, before we had these ghost atoms an alternative to struts was already implemented: a special kind of rule. The reason is that I wanted a cleaner and more predictable way to adapt struts to the math style uses and sometimes predicting that is fragile. What we want is a delayed assignment of dimensions.

We have two solutions. The first one uses two math parameters that themselves adapt to the style, as do other parameters: `\Umathruleheight` and `\Umathruledepth`. The other solution relates a font (or family) and character with the strut rule which is then used as measure for the height and depth. Just for the record: this also works in text mode, which is why a recent LMTX also does use that for struts now. The optional visualization is just part of the regular visualization mechanism in ConTEXt which already had provisions for struts. A side effect of this is that the rule primitives now accept three more keywords: `font`, `fam` and `char`, in addition to the already present traditional ones `width`, `height` and `depth`, the (backend) margin ones `left` (or `top`) and `right` (or `bottom`) options, as well as `xoffset` and `yoffset`). The command that creates a rule with subtype `strut` is simply `\srule`. Because struts are rather macro package specific I leave it to this.

One positive side effect is that we could simplify the ConTEXt fraction mechanism a bit. Over time control over the (font driven) gaps was introduced but that is not really needed because we zero the gaps anyway. There was also a tolerance mechanism which again was not used. However, for skewed fractions we do use the new tolerance mechanism as well as gap control.

Atoms

Now that we have generic atoms (`\mathatom`) another, sometimes confusing aspect of the math parsing can be solved. Take this:

```
\def\MyBin{\mathbin{\tt mybin}}
$ x ^ \MyBin _ \MyBin $
```

The parser just doesn't like that which means that one has to use

```
\def\MyBin{\mathbin{\tt mybin}}
$ x ^ {\MyBin} _ {\MyBin} $
```

or:

```
\def\MyBin{{\mathbin{\tt mybin}}}
$ x ^ \MyBin _ \MyBin $
```

But the later has side effects: it creates a list that can influence spacing. It is for that reason that we do accept atoms where they were not accepted before. Of course that itself can have side effects but at least we don't get an error message. It fits well into the additional (user) classes model. And, given that in ConTEXt the `\frac` command is actually wrapped as `\mathfrac` the next will work too:

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

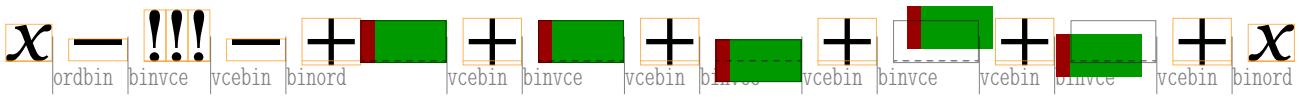
but in practice you should probably use the braced version here for clarity.

The vcenter primitive

Traditionally this primitive is bound to math but it had already been adapted to also work in text mode. As part of the upgrade of math we can now also pass all the options that normal boxed take and we can also cheat with the axis. Here is an example:

```
\def\TEST{\hbox\bgroup
    \darkred   \vrule width  2pt height 4pt
    \darkgreen \vrule width 10pt depth  2pt
\egroup}
$
    x  - \mathatom \mathvcentercode {!!!} -
    + \ruledvcenter                          {\TEST}
    + \ruledvcenter                          {\TEST}
    + \ruledvcenter axis 1                   {\TEST}
    + \ruledvcenter xoffset  2pt yoffset  2pt {\TEST}
    + \ruledvcenter xoffset -2pt yoffset -2pt {\TEST}
    + x
$
```

There was already a vcenter class available before we did this:

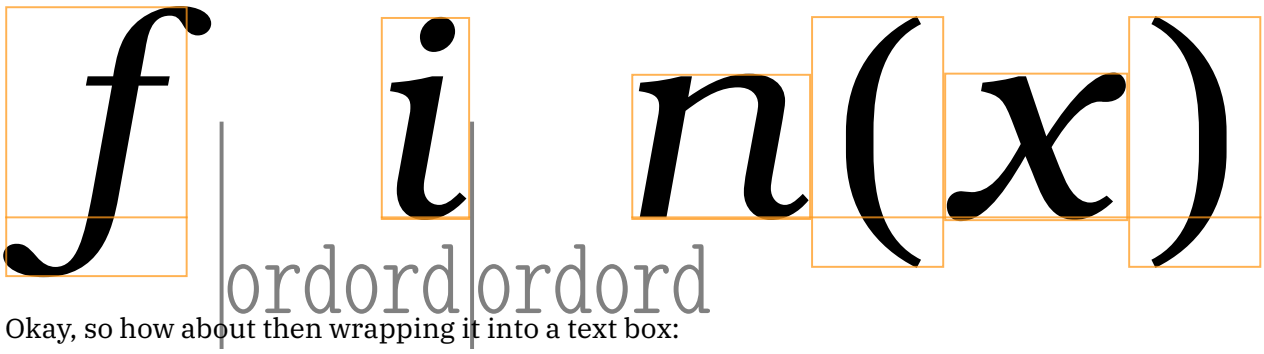


Text

Sometimes you want text in math, for instance `sin` or `cos` but text in math is not really text:

```
$ \setmathspacing\mathordinarycode\mathordinarycode\textstyle 10mu fin(x) $
```

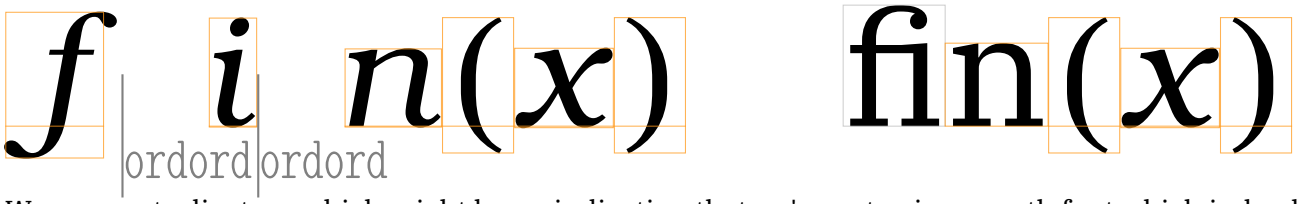
The result demonstrates that what looks like a word actually becomes three math atoms:



Okay, so how about then wrapping it into a text box:

```
$
    \setmathspacing\mathordinarycode\mathordinarycode\textstyle 10mu
    fin(x) \quad \hbox{fin}(x)
$
```

Here we get:



We even get a ligature which might be an indication that we're not using a math font which indeed is the case: the box is typeset in the regular text font.

```
\def\Test#1%
  {\setmathspacing\mathordinarycode\mathordinarycode\textstyle 5mu
   $\showglyphs
   #1% style
   {\tf fin} \quad
   \hbox{fin} \quad
   \mathatom class \mathordinarycode textfont {fin}
   \mathatom class \mathordinarycode textfont {\tf fin}
   \mathatom class \mathordinarycode textfont {\hbox{fin}}
   \mathatom class \mathordinarycode mathfont {\hbox{fin}}
   $}
```

When we feed this macro with the `\textstyle`, `\scriptstyle` and `\scriptscriptstyle` we get:

text

script

scriptscript

Here you see a new atom option action: `textfont` which does as much as setting the font to the current family font and the size to the one used in the style. For the record: you only get ligatures when they are configured and provided by the font (and as math is a script itself it is unlikely to work).[9]

Tracing

I won't discuss the tracing features in ConTEXt here but for sure the visualizer helps a lot in figuring out all this. In LuaMetaTEX we carry a bit more information with the resulting nodes so we can provide more details, for instance about the applied spacing and penalties. Some is shown in the examples. A more recent tracing feature is this:

```
\tracingmath   1
\tracingonline 1
$
    \mathord (
    \mathord {(}
    \mathord \Udelimiter"4 0 `(
    \Udelimiter"4 0 `(
$
```

That gives us on the console (the dots represent detailed attribute info that we omit here):

```
7:3: > \inlinemath=
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathlist
7:3: ...\noad[open][...]
7:3: ....\nucleus
7:3: .....\mathchar[open] family "0, character "28
7:3: \noad[ord][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
7:3: \noad[open][...]
7:3: .\nucleus
7:3: ..\mathchar[open] family "0, character "28
```

A tracing level of 2 will spit out some information about applied spacing and penalties between atoms (when set) and level 3 will show the math list before the first and second pass (a mix of nodes and noads) we well as the result (nodes) plus return some details about rules, spacing and penalties applied.

Is there more?

The engine already provides the option to circumvent the side effect of a change in a parameter acting sort of global: the last value given is also the one that a second pass starts with. The `\frozen` prefix will

[9] The existing mechanisms in ConTEXt already dealt with this but it is nevertheless nice to have it as a clean engine feature.

turn settings into local ones but that's another (already old) topic. There are many such improvements and options not mentioned here but you can find them mentioned and explained in older development stories. A lot has been around for a while but not been applied in ConTEXt yet.

When TEX was written one important property (likely related to memory consumption) is that node lists have only forward pointers. That means that the state of preceding material has to be kept track of: there is no going (or looking) back. In LuaTEX we have double linked lists so in principle we can try to be more clever but so far I decided not to touch the math machinery in that way. But who knows what comes next.

Those italics

Right from the start of LuaTEX it became clear that the fact that TEX assumes the actual width of glyphs to be incremented by the italic correction that then selectively is removed has been an issue. It made for successive attempts to improve spacing in ConTEXt by providing pseudo features. But, when we moved from assembled Unicode math fonts to 'real' ones that became messy: what trick to apply when and even worse where? In the end there are only a very few shapes that actually are affected in the sense that when we don't deal with them it looks bad. It also happens that one of those shapes is the italic 'f', a letter that is used frequently in math. It might even be safe to say that the simple fact that the math italic f has this excessively wrong width and thereby pretty large italic correction is the cause of many problems.

In the LMTX approach Mikael and I settled on patching shapes in the so called font goodie files, aka `lfg` files and only a handful of entries needed a treatment. This makes a good case for removing the traditional font code path from LuaMetaTEX.

`modern`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`cambria`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`pagella`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`termes`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

`bonum`: $a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1$ $\boldsymbol{a_2^1\ b_2^1\ c_2^1\ d_2^1\ e_2^1\ f_2^1\ g_2^1\ h_2^1\ i_2^1\ j_2^1\ k_2^1\ l_2^1\ m_2^1\ n_2^1\ o_2^1\ p_2^1\ q_2^1\ r_2^1\ s_2^1\ t_2^1\ u_2^1\ v_2^1\ w_2^1\ x_2^1\ y_2^1\ z_2^1}$

One of the other very sloped symbol is the integral, although most fonts have them more upright than tex has. Of course there are many variants of these integrals in a math font. Here we also have some font parameters that we can tune, which is what we do.

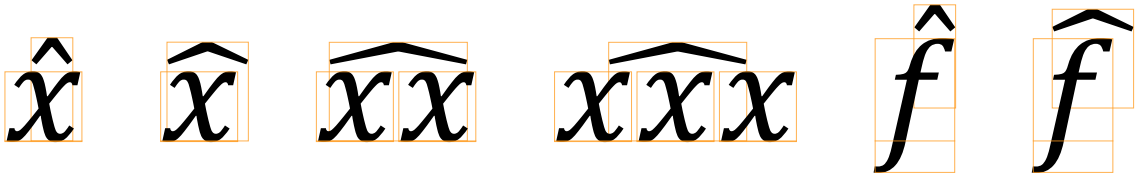
Accents

Accents are common in languages other than English and it's English that TEX was made for. Although the seven bit variant became eight bit handling accents never was sophisticated and one of the main reasons is of course that one could use pre-built composed characters. The OpenType format brought

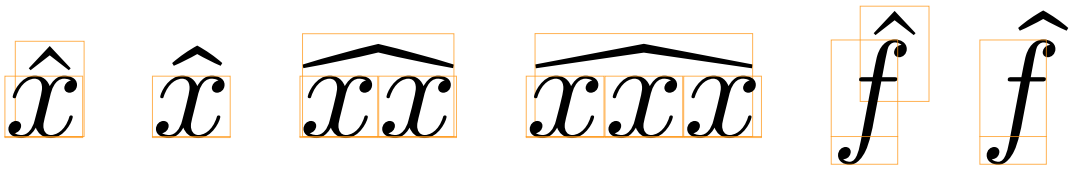
proper anchoring (aka marks) to font formats and when LuaTEX deals with text those kick in. In OpenType math however, anchoring is kind of limited to the top position only. Because the TEX Gyre fonts are based on traditional TEX fonts, their accents have not become better suited.

```
$ \hat{x} \enspace \widehat{x} \enspace \widehat{xx} \enspace \widehat{xxx}
  \enspace \hat{f} \enspace \widehat{f} $
```

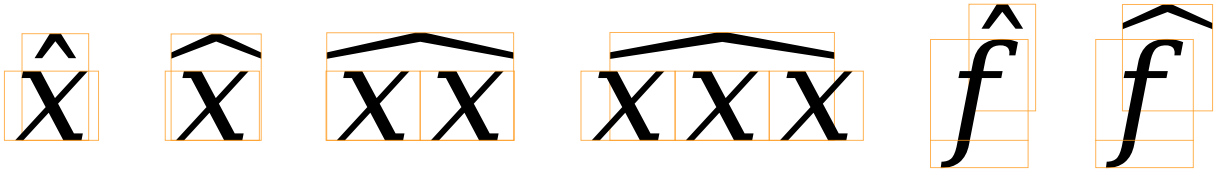
When looking at examples you need to be aware of the fact hat fonts can have been adapted in the goodie files.[10] So, for instance bounding boxes and such can differ from the original. Anyway, the previous code in Cambria looks as follows.



With Latin Modern we get:



And Dejavu comes out as:



As you can see there are some differences. In for instance Latin Modern the shape of the hat and smallest wide hat are different and the first wide one has zero dimensions combined with a negative anchor. When an accented character is followed by a superscript or prime the italic correction of the base kicks in but that cannot be enough to not let this small wide hat overflow into the script. We could compensate for it but then we need to know the dimensions. Of course we can consult the bounding box but it makes no sense to let heuristics enter the machinery here while we're in the process generalization. One option is to have two extra parameters that can be used when the width of the accent comes close to the width of the base (we then assume that zero accent width means that it has base width) we add an additional kern. In the end we settled for a (semi automatic) correction option in the goodie files.

There are actually three categories of extensible accents to consider: those that resemble the ones used in text (like tildes and hats), those wrapping something (like braces and bracket but also arrows) and rules (that in traditional TEX indeed are rules). In ConTEXt we have different interfaces for each of these in order to have a more extensive control. The text related ones are the simplest and closest to what the engine supports out of the box but even there we use tweaked glyphs to get better spacing because (of course) fonts have different and inconsistent spacing in the boundingbox above and below

[10] Extreme examples can be found for Lucida Bright where we not only have to fix the extensible parts of horizontal braces but also have to provide horizontal brackets.

the real shape. This is again some tweak that we moved from being *automatic* to being *under goodie file control*. But this is all too ConTEXt specific to discuss here in more detail.

Decision time

After lots of tests Mikael and I came to the conclusion that we're facing the following situation. When typesetting math most single characters are italic and we already knew from the start of the LuaTEX project that the italics shapes are problematic when it comes to typesetting math. But it looks like even some upright characters can have italic correction: in TexGyreBonum for instance the bold upright `f` has italic correction, probably because it then can (somehow) kern with a following `i`. It anyhow assumes no italic correction to be applied between these characters.

In the end the mixed math font model model got more and more stressed so one decision was to simply assume fonts to be used that are either Cambria like OpenType, or mostly traditional in metrics, or a hybrid of both. It then made more sense to change the engine control options that we have into ones that simply enable certain code paths, independent of the fact if a font is OpenType or not. It then become a bit "crap in, crap out", but because we already tweak fonts in the goodie files it's quite okay. Some fonts have bad metrics anyway or miss characters and it makes no sense to support abandoned fonts either. Also, when a traditional font is assembled one can set up the engine with different flags and we can deal with it as we wish. In the end it is all up to the macro package to configure things right, which is what we tried to do for months when rooting out all the artifacts that fonts bring.[11]

That said, the reason why some (fuzzy) mixed model works out okay (also in LuaTEX) is that proper OpenType fonts use staircase kerns instead of italic correction. They also have no ligatures and kerns. We also suspect that not that much attention is paid to the rendering. It's a bit like these "How many f's do you count in this sentence?" tests where people tend to overlook `of`, `if` and similar short words. Mathematicians loves `f`'s but probably also overlook the occasionally weird spacing and kerning.

A side effect is that mixing OpenType and traditional fonts is also no longer assumed which in turn made a few (newly introduced) state variables obsolete. Once everything is stable (including extensions discussed before) some further cleanup can happen. Another side effect is that one needs to tell the engine what to apply and where, like this:

```
\mathfontcontrol\numexpr \zerocount
   +\overrulemathcontrolcode
   +\underrulemathcontrolcode
   +\fractionrulemathcontrolcode
   +\radicalrulemathcontrolcode
   +\accentskewhalfmathcontrolcode
   +\accentskewapplymathcontrolcode
 % + checkligatureandkernmathcontrolcode
   +\applyverticalitalickernmathcontrolcode
   +\applyordinaryitalickernmathcontrolcode
   +\staircasekernmathcontrolcode
 % +\applycharitalickernmathcontrolcode
 % +\reboxcharitalickernmathcontrolcode
   +\applyboxeditalickernmathcontrolcode
```

[11] In previous versions one could configure this per font but that has been dropped.

```
   +\applytextitalickernmathcontrolcode
   +\checktextitalickernmathcontrolcode
 % +\checkspaceitalickernmathcontrolcode
   +\applyscriptitalickernmathcontrolcode
   +\italicshapekernmathcontrolcode
\relax
```

There might be more control options (also for tracing purposes) and some of the symbolic (ConTEXt) names might change for the better. As usual it will take some years before all is stable but because most users use the latest greatest version it will be tested well.

After this was decided and effective I also decided to drop the mapping from traditional font parameters to the OpenType derives engine ones: we now assume that the latter ones are set. After all, we already did that in ConTEXt for the virtual assemblies that we started out with in the beginning of LuaTEX and MkIV.

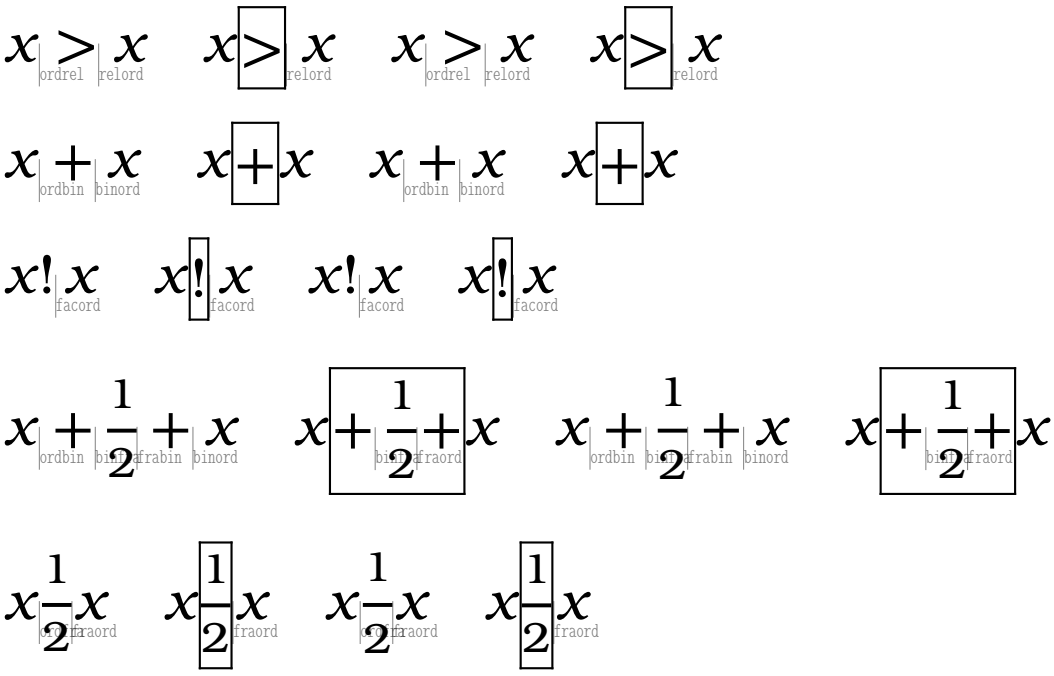
Dirty tricks

Once you start playing with edge cases you also start wondering if some otherwise complex things can be done easier. The next macro brings together a couple of features discussed in previous sections. It also uses two state variables: `\lastleftclass` and `\lastrightclass` that hold the most recent edge classes.

```
\tolerant\permanent\protected\def\NiceHack[#1]#:#2% special argument parsing
  {\begingroup
   \setmathatomrule
     \mathbegincode\mathbincode % context constants
     \allmathstyles
     \mathbegincode\mathbincode
   \normalexpanded
     {\setbox\scratchbox\hpack
        ymove \Umathaxis\Ustyle\mathstyle % an additional box property
        \bgroup
          \framed % a context macro
            [location=middle,#1]
            {$\Ustyle\mathstyle#2$}%
        \egroup}%
   \mathatom
     class 32 % an unused class
     \ifnum\lastleftclass <\zerocount\else leftclass  \lastleftclass\fi
     \ifnum\lastrightclass<\zerocount\else rightclass \lastrightclass\fi
     \bgroup
       \box\scratchbox
     \egroup
   \endgroup}

\def\MyTest#1%
  {$              x #1                          x $\quad
   $              x \NiceHack[offset=0pt]{#1} x $\quad
```

```
   $\displaystyle x #1                          x $\quad
   $\displaystyle x \NiceHack[offset=0pt]{#1} x $}

\scale[scale=2000]{\MyTest{>}}            \blank
\scale[scale=2000]{\MyTest{+}}            \blank
\scale[scale=2000]{\MyTest{!}}            \blank
\scale[scale=2000]{\MyTest{+\frac{1}{2}+}}\blank
\scale[scale=2000]{\MyTest{\frac{1}{2}}}  \blank
```

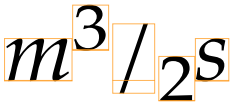
Of course this is not code you immediately come up with after reading this text, also because you need to know a bit of ConTEXt.



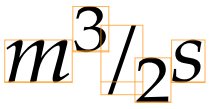
There are a few control options, like `\noatomruling` that can be used to prevent rules being applied to the next atom. We can use these in order to achieve more advanced alignment results, but discussing math alignments would demand many more pages than make sense here.

Tuned kerning

The ConTEXt distribution has dedicated code for typesetting units that dates back to the mid nineties of the previous century but was (code wise) upgraded from MkII to MkIV which made it end up in the physics name space. There is not much reason to redo that code but when we talk new spacing classes it might make sense at some point to see if we can use less code for spacing by using a 'unit' class. When Mikael pointed out that, for instance in Pagella:



doesn't space well the obvious answer is: use the units mechanism because this kind of rendering was why it was made in the first place. However, the question is of course, can we do better anyway. The chosen solution uses a combination of class options and tweaked shape kerning:



An example of a class setup in ConTEXt is:

```
\setmathoptions\mathdivisioncode\numexpr
    \nopreslackclassoptioncode     +\nopostslackclassoptioncode
   +\lefttopkernclassoptioncode    +\righttopkernclassoptioncode
   +\leftbottomkernclassoptioncode +\rightbottomkernclassoptioncode
\relax
```

and, although we don't go into the details of tweaking here, this is the kind if code you will find in the goodie file:

```
{
    tweak = "kerns",
    list  = {
        [0x2F] = {
            topleft     = -0.3,
            bottomright =  0.2,
       }
    }
}
```

where the numbers are a percentage of the width. This specification translates in a math staircase kerning recipe.

More font tweaks

Once you start looking into the details of these fonts you are likely to notice more issues. For instance, in the nice looking Lucida math fonts the relations have inconsistent widths and even shapes. This can partially be corrected by using a stylistic alternate but even that forced us to come up with a mechanism to selectively replace 'bad' shapes because there is not that much granularity in the alternates. And once we looked at these alternates we noticed that the definition of of script versus calligraphic is also somewhat fuzzy and font dependent. That made for yet another tweak where we can swap alphabets and let the math machinery choose the expected shape. In Unicode this is handled by variant selectors which is rather cumbersome. Because these two styles are used mixed in the same document, a proper additional alphabet would have made more sense. As we already support variant selectors it was no big deal to combine that mechanism with a variant selector features over a range of calligraphic or script characters, which indeed is what mathematicians use (Mikael can be very convincing). With this kind of tweaks the engine doesn't really play a role: we always could and did deal with it. It's just that upgrading the engine made us look again at this.

Normalization

Once we had all these spacing related features upgraded it was time to move to other aspects math typesetting. Most of that is not handled in the engine but at the macro level. Examples of this are making sure that math spacing obeys the rules across alignment cells, breaking long formulas into lines with various alignment schemes. The first is accommodated by using the primitives that set the states

at the beginning and end of a formula so that is definitely something that the engine facilitates. The second was already possible in MkIV but is somewhat more transparent now by using tagged boundary nodes.

But for this summary we stick to discussing the more low level features and where most of what we discussed here concerns horizontal spacing we also have some vertical magic like the mentioned scaled fences and operators but they sort of behave as expected given the traditional TEX approach. We have some more:

```
\definemathradical[esqrt][sqrt][height=\maxdimen,depth=\maxdimen]
\definemathradical[ssqrt][sqrt][height=3ex,depth=2ex]

\def\TestSqrt#1%
  {test $ #1{x} + #1{\sin(x)} $ test\quad
   test $ #1{x} + #1{\sin(x)} + #1{\frac{1}{x}} $ test\quad
   test $ #1{x} + #1{x^2} $ test\quad
   test $ \left(#1{x} + #1{x^2} \right) $ test\par}

\TestSqrt \sqrt  \blank
\TestSqrt \esqrt \blank
\TestSqrt \ssqrt \blank
```

test $\sqrt{x} + \sqrt{\sin(x)}$ test test $\sqrt{x} + \sqrt{\sin(x)} + \sqrt{\frac{1}{x}}$ test test $\sqrt{x} + \sqrt{x^2}$ test test $\left(\sqrt{x} + \sqrt{x^2}\right)$ test

test $\sqrt{x} + \sqrt{\sin(x)}$ test test $\sqrt{x + \sqrt{\sin(x)} + \sqrt{\frac{1}{x}}}$ test test $\sqrt{x} + \sqrt{x^2}$ test test $\left(\sqrt{x} + \sqrt{x^2}\right)$ test

test $\sqrt{x + \sqrt{\sin(x)}}$ test test $\sqrt{x + \sqrt{\sin(x) + \sqrt{\frac{1}{x}}}}$ test test $\sqrt{x + \sqrt{x^2}}$ test test $\left(\sqrt{x + \sqrt{x^2}}\right)$ test

In the above example you see that square roots can be made to adapt themselves to other such roots. For this we had to add an additional pass. Originally there are just two passes: a first typesetting pass where the maximum height and depth are collected so that in the second pass the fences can be generated and injected. That second pass also handles the spacing and penalties. In LuaMetaTEX we now have (1) radical body typesetting, (2) radical typesetting, (3) atom typesetting with height and depth analysis, (4) fence typesetting, and finally (5) inject spacing, penalties, remove slack, etc.

In the examples above we set the height and depth and these are passed by keywords to the radical primitive (most atoms and math structures accept keywords that control rendering). Here the special values `\maxdimen` signal that we have to make radicals of equal height and depth.

In MkII we had ways to snap formulas so that we got consistent line spacing. For a while I wondered if the engine could help with that but in the end no specific engine features are needed, but is is definitely an area that I keep an eye on because consistent spacing is important. After all one has to draw aline somewhere and we always have the Lua callback mechanism available.

Final words

One can argue that all these new features can make a document look better. But you only have to look at what Don Knuth produces himself to see that one always could do a good job with TEX, although maybe

at the cost of some extra spacing directives. It is the fact that OpenType showed up as well as many more math fonts, all with their own (sometimes surprising) special effects, that made us adapt the engine. Of course there are also new possibilities that permit better and more robust macro support. The TEXbook has a chapter on “the fine points of mathematics typesetting” for a reason.

There has never been an excuse to produce bad looking documents. It is all about care. For sure there is a category of users who are forced to use TEX, so they are excused. There are also those who have no eye for typography and rely on the macro package, so there we can to some extent blame the authors of those packages. And there are of course the sloppy users, those who don't enter a revision loop at all. They could as well use any system that in some way can handle math. One can also wonder in what way massive remote editing as well as collaborative working on documents make things better. It probably becomes less personal. At meetings and platforms TEX users like to bash the alternatives but in the end they are part of the same landscape and when it comes to math they dominate. Maybe there is less to bragg about then we like: just do your thing and try to do it as good as possible. Rely on your eyes and pay attention to the details, which is possible because the engine provided the means. The previous text shows a few things to pay attention to.

Once all the basics that have to do with proper dimensions, spacing, penalties and logic are dealt with, we will move on to the more high level constructs. So, expect more.

6 The binary

This is a very short chapter. Because LuaMetaTEX is also a script runner, I want to keep it lean and mean. So, when the size exceeded 3MB after we'd extended the math engine, I decided to (finally) let all the MetaPost number interfaces pass pointers which brought down the binary 100K and below the 3MB mark again.

I then became curious about how much of the binary actually is taken by MetaPost, and a bit of calculation indicated that we went from 20.1% down to 18.3%. Here is the state per May 13, 2022:

component	pct	bytes	comment
liblua	11.8	349158	lua core, tex interfaces
libluaoptional	2.4	70263	framework, several small interfaces, cerf
libluarest	1.9	55911	general helper libraries
libluasocket	2.4	71640	helper that interfaces to the os libraries
libmimalloc	4.1	121186	memory management partial
libminiz	1.2	34962	minimalistic core
libmp	18.3	540615	mp graphic core, number libraries, lua interfacing
libpplib	7.4	220386	pdf reading core, encryption helpers
libtex	50.5	1495970	extended tex core
luametatex		**2960091**	2022-05-13

It is clear that the TEX core is good for half of the code (50.5%) with the accumulated Lua stuff (18.5%) and MetaPost being a good second (18.3%) and third and the pdf interpreting library a decent fourth (7.4%) place.

7 To the point

In the 2022 ntg Maps 53 there is a visual very attractive article about generative graphics with MetaPost by Fabrice Larribe. These graphics actually use very little MetaPost code that use randomized paths and points and the magic is in getting the parameters right. This means that one has to process them a lot to figure out what looks best. Here is an example of such a graphic definition. I will show more variants so a rendering happens later on.

```
\startMPdefinitions
    vardef agitate_a(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel, rlength ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := for i=1 upto nbpoints:
                point (i/nbpoints) along R
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

I will not explain the working of this because there is the article. Instead, I will focus on something that came up when the Maps was prepared: performance. Not only are these graphics large (which is no real problem) but they also take a while to render (which is something that does matter when one wants to find the best a parameters). For the first few variants we keep the same names of variables as in the article.

In figure 7.1 we show the (kind of) graphic that we are dealing with. Such an agitator is used in a loop so that we agitate multiple circles, where we go from large to small with for instance 4868, 4539, 4221, 3892, 3564, 3245, 2917, 2599, 2270, 1941, 1623, 1294, 966, 647 and 319 points. The article uses a definition like below for the graphic where you can see the agitator being applied to each of the circles.

```
path P ; numeric NbCircles, S, nzero, fn, tzero, ft ;

randomseed   := 10 ;
defaultscale := .05 ;

NbCircles := 15 ; S := 10 ; nzero := 10 ; fn := 1.3 ; tzero := 5 ; ft := 0.8 ;

for c = NbCircles downto 1 :
    P := fullcircle scaled (c*6.5) scaled 3 ;
    P := agitate_a(P, S, nzero, fn, tzero, ft) ;
    eofill P
        withcolor transparent(1,4/NbCircles,col) ;
```

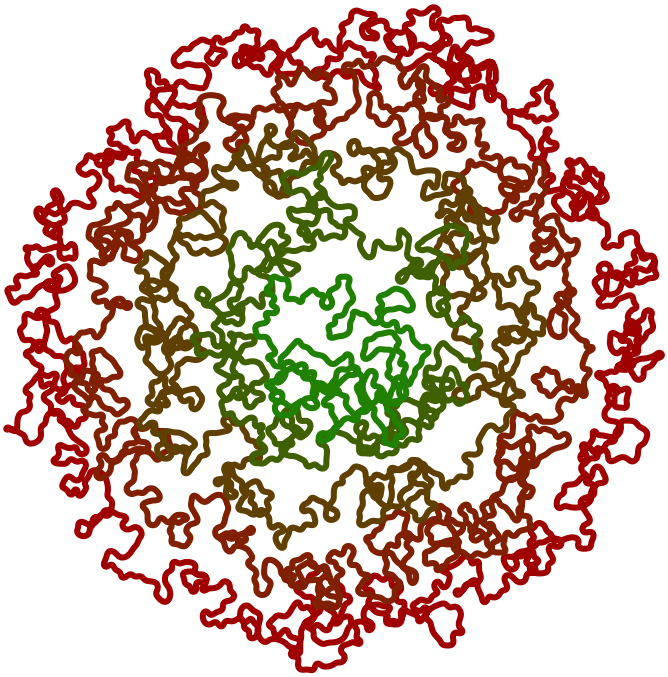


Figure 7.1 Fabrice's agitated circles, with reduced properties to keep this file small (see source).

```
    draw P
        withpen pencircle scaled 0.1
        transparent(1,4/NbCircles,.90[black,col]) ;
endfor ;
```

The first we noticed is that the graphics processes faster when double mode is used: we gain 40–50% and the reason for this is that modern processors are very good at handling doubles while MetaPost in scaled mode has to do a lot of juggling with pseudo fractions. In the timings shown later we leave that improvement out. Also, because of this observation ConTEXt LMTX now defaults its MetaPost instances to method double.

When I stared at the agitator code I noticed that the `along` macro was used. That macro returns a point at given percentage along a path. In order to do that the macro calculates the length of the path and then locates that point. The primitive operations involved are `arclength`, `arctime of` and `point of` and each these takes some time to complete. A first improvement is to inline the `along` and hoist the length calculation outside the loop.

```
\startMPdefinitions
    vardef agitate_b(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel, rlength ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            rlength := (arclength R) / nbpoints;
            R := for i=1 upto nbpoints:
                (point (arctime (i * rlength) of R) of R)
                    randomized noiselevel
```

```
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

There is not that much that we can improve here but because Mikael Sundqvist and I had just extended MetaPost with some intersection improvements, it made sense to see what we could do in the engine. In the next variant the `arcpoint` combines `arctime of` and `point of`. The reason this is much faster is that we are already on the right spot when we got the time, and we save a sequential `point of` lookup, something that takes more time when paths are longer.

```
\startMPdefinitions
    vardef agitate_c(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel, rlength ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            rlength := (arclength R) / nbpoints;
            R := for i=1 upto nbpoints:
                (arcpoint (i * rlength) of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

At that stage we wondered if we could come up with a primitive like `intersectiontimelist` for these points; here a list refers to a path in which we collect the points. Now, as with the intersection primitives, MetaPost loops over the segments of a path and works within such a segment. That is why the following variant has an explicit start at point zero: we can now use offsets (discrete points).

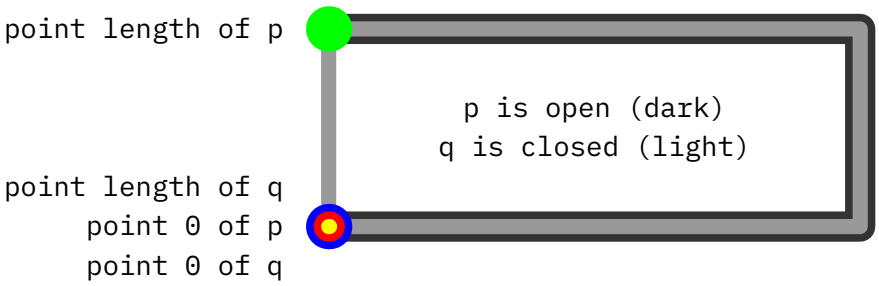
```
\startMPdefinitions
    vardef agitate_d(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel, rlength ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            rlength := (arclength R) / nbpoints;
            R := for i=1 upto nbpoints:
                (arcpoint (0, i * rlength) of R)
                    randomized noiselevel
```

```
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

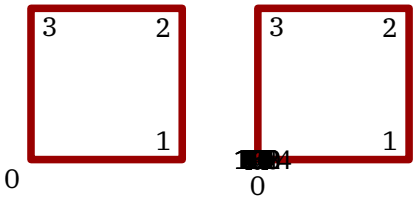
During an evening zooming Mikael and I figured out, by closely looking at the source, how the arc functions work and how we could indeed come up with a list primitive. The main issue was to use the right information. Mikael sat down to make a pure MetaPost variant and I hacked the engine. Mikael came up with a first variant similar to the following, where we use a new primitive `subarclength`.

```
\startMPdefinitions
    vardef arcpoints_a(expr thepath, cnt) =
        save len, seg, tot, tim, stp, acc ;
        numeric len ; len := length thepath ;
        numeric seg ; seg := 0 ;
        numeric tot ; tot := 0 ;
        numeric tim ; tim := 0 ;
        %
        numeric acc[] ; acc[0] := 0 ;
        for i = 1 upto len:
            acc[i] := acc[i-1] + subarclength (i-1,i) of thepath ;
        endfor;
        %
        numeric stp ; stp := acc[len] / cnt;
        %
        point 0 of thepath
        for tot = stp step stp until acc[len] :
            hide(
                forever :
                    exitif ((tim < tot) and (tot < acc[seg+1])) ;
                    seg := seg + 1 ;
                    tim := acc[seg] ;
                endfor ;
            )
            -- (arcpoint (seg,tot-tim) of thepath)
        endfor if cycle thepath : -- cycle fi
    enddef ;
\stopMPdefinitions
```

Getting points of a path is somewhat complicated by the fact that the length of a closed path is different from that of an open path even if they have the same number of so-called knots. Internally a path is always a closed loop. That way, when MetaPost runs over a path, it can easily access the first point when it's at the end, something that is handy when that point has to be taken into account. Therefore, the end condition of a loop over a path is the arrival at the beginning. In the next graphic we show a bit how these first (zero) and last points are located. One reason why the previous macros start at point one and not at zero is that `arclength` can overflow due to the randomly growing path otherwise.



The difference between starting at zero or one for a cycle is show below, we get more and more points!



In the next variants we will not loop over points but step to the `arclength`. Watch the new `subarclength` primitive that starts at an offset. This is much faster than taking a `subpath of`. We can move the accumulator loop into the main loop:

```
\startMPdefinitions
    vardef arcpoints_b(expr thepath, cnt) =
        save len, aln, seg, tot, tim, stp, acc ;
        numeric len ; len := length thepath ;
        numeric aln ; aln := arclength thepath ;
        numeric seg ; seg := 0 ;
        numeric tot ; tot := 0 ;
        numeric tim ; tim := 0 ;
        numeric stp ; stp := aln / cnt;
        numeric acc ; acc := subarclength (0,1) of thepath ;
        %
        point 0 of thepath
        for tot = stp step stp until aln :
            hide(
                forever :
                    exitif tot < acc ;
                    seg := seg + 1 ;
                    tim := acc ;
                    acc := acc + subarclength (seg,seg+1) of thepath ;
                endfor ;
            )
            -- (arcpoint (seg,tot-tim) of thepath)
        endfor if cycle thepath : -- cycle fi
    enddef ;
\stopMPdefinitions
```

If you don't like the `hide` the next variant also works okay:

```
\startMPdefinitions
    vardef mfun_arc_point(text tot)(text thepath) =
```

```
        forever :
            exitif tot < acc ;
            seg := seg + 1 ;
            tim := acc ;
            acc := acc + subarclength (seg,seg+1) of thepath ;
        endfor ;
        (arcpoint (seg,tot-tim) of thepath)
    enddef ;

    vardef arcpoints_c(expr thepath, cnt) =
        save len, aln, seg, tot, tim, stp, acc ;
        numeric len ; len := length thepath ;
        numeric aln ; aln := arclength thepath ;
        numeric seg ; seg := 0 ;
        numeric tot ; tot := 0 ;
        numeric tim ; tim := 0 ;
        numeric stp ; stp := aln / cnt;
        numeric acc ; acc := subarclength (0,1) of thepath ;
        %
        point 0 of thepath
        for tot = stp step stp until aln :
            -- mfun_arc_point(tot)(thepath)
        endfor if cycle thepath : -- cycle fi
    enddef ;
\stopMPdefinitions
```

This got applied in three test agitators

```
\startMPdefinitions
    vardef agitate_e_a(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := arcpoints_a(R, nbpoints) ; % original Mikael
            R := for i=0 upto length R:
                (point i of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;

    vardef agitate_e_b(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel ;
        path R ; nbpoints := n ; noiselevel := t ;
```

```
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := arcpoints_b(R, nbpoints) ; % merged Mikael
            R := for i=0 upto length R:
                (point i of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;

    vardef agitate_e_c(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := arcpoints_c(R, nbpoints) ; % split Mikael
            R := for i=0 upto length R:
                (point i of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
        R
    enddef ;
\stopMPdefinitions
```

The new engine primitive shortens these agitators:

```
\startMPdefinitions
    vardef agitate_e_d(expr thepath, S, n, fn, t, ft) =
        save R, nbpoints, noiselevel ;
        path R ; nbpoints := n ; noiselevel := t ;
        R := thepath ;
        for s=1 upto S :
            nbpoints := nbpoints * fn ;
            noiselevel := noiselevel * ft ;
            R := arcpointlist nbpoints of R;
            R := for i=0 upto length R:
                (point i of R)
                    randomized noiselevel
                ..
            endfor cycle ;
        endfor ;
```

```
        R
    enddef ;
\stopMPdefinitions
```

So are we done? Did we get rid of all bottlenecks? The answer is no! We still loop over the list in order to randomize the points. For each point we start at the beginning of the list. Let's first rewrite the agitator a little:

```
\startMPdefinitions
    vardef agitate_f_a(expr pth, iterations, points, pointfactor, noise,
      noisefactor) =
        save currentpath, currentpoints, currentnoise ; path currentpath ;
        currentpath   := pth ;
        currentpoints := points ;
        currentnoise  := noise ;
        for step = 1 upto iterations :
            currentpath   := arcpointlist currentpoints of currentpath ;
            currentnoise  := currentnoise  * noisefactor ;
            currentpoints := currentpoints * pointfactor ;
            if currentnoise <> 0 :
                currentpath :=
                    for i = 0 upto length currentpath:
                        (point i of currentpath) randomized currentnoise ..
                    endfor
                cycle ;
            fi
        endfor ;
        currentpath
    enddef ;
\stopMPdefinitions
```

One of the LuaMetaFun extensions is a fast path iterator. In the next variant the `inpath` macro sets up an iterator (regular loop) with the length as final value. In the process the given path gets passed to Lua where we can access it as array. The `pointof` macro (again a Lua call) injects a pair. You will be surprised that even with passing the path to Lua and calling out to Lua to inject the pair this is way faster than the built-in `point of`.

```
\startMPdefinitions
    vardef agitate_f_b(expr pth, iterations, points, pointfactor, noise,
      noisefactor) =
        save currentpath, currentpoints, currentnoise ; path currentpath ;
        currentpath   := pth ;
        currentpoints := points ;
        currentnoise  := noise ;
        for step = 1 upto iterations :
            currentnoise  := currentnoise  * noisefactor ;
            currentpoints := currentpoints * pointfactor ;
            currentpath   := arcpointlist currentpoints of currentpath ;
            if currentnoise <> 0 :
```

```
                currentpath :=
                    for i inpath currentpath :
                        (pointof i) randomized currentnoise ..
                    endfor
                cycle ;
            fi
        endfor ;
        currentpath
    enddef ;
\stopMPdefinitions
```

It was tempting to see if a more native solution pays of. One problem there is that a path is not really suitable for that as we currently don't have a data type that represents a point. Okay, actually we sort of have because we can use the transform record that has six points but that is something I will look into later (it just got added to the todo list).

The `i within pth` iterator is no conceptual beauty but does the job. Just keep in mind that it is just means for this kind of applications: run over a path point by point. The `i` has the current point number. Because we run over a path following the links we only run forward.

```
\startMPdefinitions
    vardef agitate_f_c(expr pth, iterations, points, pointfactor, noise,
      noisefactor) =
        save currentpath, currentpoints, currentnoise ; path currentpath ;
        currentpath   := pth ;
        currentpoints := points ;
        currentnoise  := noise ;
        for step = 1 upto iterations :
            currentnoise  := currentnoise  * noisefactor ;
            currentpoints := currentpoints * pointfactor ;
            currentpath   := arcpointlist currentpoints of currentpath ;
            if currentnoise <> 0 :
                currentpath :=
                    for i within currentpath :
                        pathpoint
                        randomized currentnoise
                        ..
                    endfor
                cycle ;
            fi
        endfor ;
        currentpath
    enddef ;
\stopMPdefinitions
```

Any primitive solution more complex than this, like first creating a fast access data structure, of having a double linked list, or using some iterator larger than a simple numeric is very likely to have no gain over the super fast Lua variant.

We show the average runtime for three runs. Here we don't render the paths, which takes about one second, including conversion to pdf. Of course measurements like this can change a bit over time. To these times you need to add about a second for the draw and fill operations as well as conversion to a pdf stream with transparencies. The improvement in runtime makes it possible to use agitators like this at runtime especially because normally one will not use such (combinations of) large paths.

agitate_a	776.26	**agitate_e_a**	291.99	**agitate_f_a**	10.82
agitate_b	276.43	**agitate_e_b**	76.06	**agitate_f_b**	2.55
agitate_c	259.89	**agitate_e_c**	77.27	**agitate_f_c**	2.17
agitate_d	260.41	**agitate_e_d**	18.67		

The final version of the agitator is slightly different because it depends if we start at zero or one but gives similar results and adapt the noise before or after the loop.

```
\startMPdefinitions
    vardef agitator(expr pth, iterations, points, pointfactor, noise,
      noisefactor) =
        save currentpath, currentpoints, currentnoise ; path currentpath ;
        currentpath   := pth ;
        currentpoints := points ;
        currentnoise  := noise ;
        for step = 1 upto iterations :
            currentpath := arcpointlist currentpoints of currentpath ;
            if currentnoise <> 0 :
                currentpath :=
                    for i within currentpath :
                        pathpoint
                        randomized currentnoise
                        ..
                    endfor
                cycle ;
            fi
            currentnoise  := currentnoise  * noisefactor ;
            currentpoints := currentpoints * pointfactor ;
        endfor ;
        currentpath
    enddef ;
\stopMPdefinitions
```

We use a similar example as in the mentioned article but coded a bit differently:

```
\startMPcode
  path pth ;
  nofcircles  := 15 ; iterations  := 10 ;
  points      := 10 ; pointfactor := 1.3 ;
  noise       :=  5 ; noisefactor := 0.8 ;

  nofcircles  :=  5 ; iterations  := 10 ;
  points      :=  5 ; pointfactor := 1.3 ;

% for c = nofcircles downto 1 :
```

```
%   pth    := fullcircle scaled (c * 6.5) scaled 3 ;
%   points := floor(arclength(pth) * 0.5) ;
%   pth    := agitator(pth, iterations, points, pointfactor, noise, noisefactor)
  ;
%   eofill pth
%     withcolor darkred
%     withtransparency(1,4/nofcircles) ;
%   draw pth
%     withpen pencircle scaled 0.1
%     withtransparency(1,4/nofcircles) ;
% endfor ;

% currentpicture := currentpicture xsized TextWidth ;

  for c = nofcircles downto 1 :
    pth    := fullcircle scaled (c * 6.5) scaled 3 ;
    points := floor(arclength(pth) * 0.5) ;
    pth    := agitator(pth, iterations, points, pointfactor, noise, noisefactor)
      ;
    draw pth
      withpen pencircle scaled 1
      withcolor (c/nofcircles)[darkgreen,darkred] ;
  endfor ;

  currentpicture := currentpicture xsized .5TextWidth ;
\stopMPcode
```

For Mikael and me, who both like MetaPost, it was a nice distraction from working months on extending math in LuaMetaTEX, but it also opens up the possibilities to do more with rendering (math) functions and graphics, so in the end we get paid back anyway.

8 Not all makes sense

The development of ConTEXt is to a large extend driven by users with a wide variety of background and usage. I can safely say that much time spent on ConTEXt qualifies as hobby (or maybe even more by curiosity). Of course I do use it myself but personally I never make advanced documents. I'm not a writer, nor an artist, nor a typesetter. I do like challenges so that's why we get mechanisms that can do tricky things and some stay sort of hidden because the practical usage is limited, although you will be surprised to see what users find in the source and use anyway. My colleague uses ConTEXt for large scale, mostly complex and demanding xml documents where one source is rendered in different ways with different parts used. Many features in ConTEXt relate to workflows.

I like to visualize things so that's part of the development cycle. I never start from some 'typographical' point of view, if only because in my experience much design is arbitrary and personal. The output should look okay on the average, and on reasonable simple documents there should be no need for manual intervention. I am quite willing to accept an occasional less optimal looking page and don't loose sleep over it. A next time, when a sentence gets added, it might be better and the problem can be moved further down the pages. Also, given what one runs into nowadays the average job that TEX does is pretty good (but users can of course mess up). It is boundary conditions that determine in what direction a style or solution goes. The more abstract one argues about typesetting and possible solutions, the less interested I often become simply because there are no perfect solutions for every case. There are always those last few % points that need manual intervention or some trickery and most users get that. It is also what makes using TEX fun.

As mentioned, the TEX engine does a pretty good job on average but that didn't prevent me from extending it: the mix of TEX, MetaPost and Lua is even more fun. But what is the development agenda there? Again, it is very much driven by what users want me to solve, but there's also the curiosity element. A recent example of extending is the math sub system. It was already made more configurable and some features where added but now it is really flexible. This was doable because the heuristics in the engine are clear. It was could be done because I had a dedicated partner in this journey.[12] Other parts are more difficult but have nevertheless been extended, to mention a few: alignments, par building and page building. However the last two use some heuristics that are hard to make more flexible. For instance the badness calculation combined with the loop that tries to find breakpoints is already quite good and the somewhat special values involved in the calculations have been optimized stepwise by Don Knuth during the development of TEX.

Does that mean that one cannot add some options to influence that tuning? For sure one can. The source has this comment:

> "When looking for optimal line breaks, TEX creates a 'break node' for each break that is *feasible*, in the sense that there is a way to end a line at the given place without requiring any line to stretch more than a given tolerance. A break node is characterized by three things: the position of the break (which is a pointer to a `glue_node`, `math_node`, `penalty_node`, or `disc_node`); the ordinal number of the line that will follow this breakpoint; and the fitness classification of the line that has just ended, i.e., `tight_fit`, `decent_fit`, `loose_fit`, or `very_loose_fit`."

The book TEX by Topic (by Eijkhout) gives a good explanation of the way lines are broken so there is no need to go into detail here. The code involved is not that trivial anyway. The criteria for deciding what is bad are as follows:

[12] In another chapter I summarize what Mikael Sundqvist and I did in this context.

verdict	effect	badness
very loose	stretch	>= 100
loose	stretch	>= 13
decent		<= 12
tight	shrink	>= 13

When the difference between two lines is more than one, they are considered to be visually incompatible. Then, if the badness of any line exceeds `pretolerance` a second pass is triggered, When `pretolerance` is negative the first pass is skipped. When the badness of any line exceeds `tolerance` a third pass is triggered and `emergencystretch` is used to make things fit.

Where in traditional TEX a lot of parsing, hyphenation, font handling and par building is combined, in LuaMetaTEX we always work with completely hyphenated and font readied lists. In traditional TEX the first pass works on the original non-hyphenated lists.

In the source there is an old note that one day I will play with a plugged in badness calculation but it also says that there might be a performance impact as well as all kind of unforeseen side effects because TEX makes sure that the heuristics lead to values that don't result in overflow and such.

Another note concerns more fitness values. Doing that will increase the runtime a little but on a modern machine that is not really an issue. Shortly after I upgraded my laptop to a somewhat newer one I decided to play with this and therefore any performance hit would go unnoticed anyway. The following snippet from the source shows the idea:

```
typedef enum fitness_value {
    very_loose_fit, /*tex lines stretching more than their stretchability */
    loose_fit,      /*tex lines stretching 0.5 to 1.0 of their stretchability */
    semi_loose_fit,
    decent_fit,     /*tex for all other lines */
    semi_tight_fit,
    tight_fit,      /*tex lines shrinking 0.5 to 1.0 of their shrinkability */
    n_of_finess_values
} fitness_value;
```

This means that when we loop over `very_loose_fit` upto `tight_fit` we have two more classes to take into account: the semi ones. Playing with that and associating them with magic numbers quickly learned that we enter the area of 'random improvements'. You can render variants and because some will look better and others worse one can argue for any case. And as usual, once a user (unaware of what we are doing) looks at it, things like successive hyphens, wider spaces, rivers and such are seen as the main difference. Of course spacing is the direct result of this kind of messing, but because the effects are actually mostly noticeable on non-justified texts it then is the end-of-line spacing that influences the verdict.[13]

In the end this kind of extensions make little sense. One can of course play science and introduce all kind of imaginary cases where it might work but that is why I started this summary by explaining

[13] When hz showed up in pdfTEX we did experiments with random samples of its usage and TEXies at user group meetings and the results were such that one could only draw the conclusion that on the average a user has no clue if something is good or bad for what reason. The strong emphasis in the TEX community on hyphenation makes that an eye-catching criterium. So having two in a successive lines even when there is really no better solution is what draws the attention and users then tend to think that what a survey is about is "The quality of hyphenation related to breaking paragraphs into lines."

what drives developments: users and constraints. Playing science for the sake of it is pseudo science. And, as with much science related to typesetting (probably with the exception of Don's work) most has therefore little practical value.

So, do we keep this feature or not? We actually do, if only to be able to demonstrate the fuzziness of this. We have an undocumented magic parameter:

```
\linebreakcriterium"0C0C0C63
```

Actually the value is zero but when one of the four byte pairs is zero it will default to `"0C` (12) or `"63` (99). The values concern `semitight`, `decent`, `semiloose`, and `loose`. After some trial and error I got to the examples on the next two pages. You need to zoom in to see the differences (the black one is the original). In setting used are:

	`\hsize`	`\setupalign`
1	12em	normal, stretch, tolerant
2	18em	flushleft

As mentioned, one can look at specific expected properties and draw conclusions but when TEX cannot find a good solution using its default, it is unlikely that alternative settings help you out, unless you do that on a per-paragraph basis.


We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.
\linebreakcriteria="00000000
\linebreakcriteria="00001C00
\linebreakcriteria="00002C00
\linebreakcriteria="00003C00
\linebreakcriteria="00004C00
\linebreakcriteria="00005C00
\linebreakcriteria="00000000
\linebreakcriteria="00000000
\linebreakcriteria="00000000
\linebreakcriteria="00000000
\linebreakcriteria="00000000
\linebreakcriteria="00000000

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00000000`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00001C00`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00002C00`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00003C00`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00004C00`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00005C00`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00000000`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00001000`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00002000`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00003000`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00004000`

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the chaff and separate the sheep from the goats.

`\linebreakcriterium="00005000`