Still expanding LuaTEX: Possibly useful extensions

Hans Hagen

Abstract

New LuaTEX programming features in a variety of areas: rules, spaces, token lists, active characters, `\csname`, packing of lists, and error handling.

1 Introduction

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

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

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

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

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

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

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

- Of course we have the ε-TEX extensions, and these already have a long tradition of proven usage. We did remove the limited directional support.
- From Aleph (follow-up on Omega), part of the directional support and some font support was inherited.
- From pdfTEX, we took most of the backend code, but it has been improved in the meantime. We also took the protrusion and expansion code, but especially the latter has been implemented a bit differently (in the frontend as well as in the backend).
- Some handy extensions from pdfTEX have been generalized; other obscure or specialized ones have been removed. So we now have frontend support for position tracking, resources (images) and reusable content in the core. The backend code has been separated a bit better and only a few backend-related primitives remain.
- The input encoding is now UTF-8, exclusively, but one can easily hook in code to preprocess data that enters TEX's parser using Lua. The characteristic catcode settings for TEX can be grouped and switched efficiently.
- The font machinery has been opened wide so that we can use the embedded Lua interpreter to implement any technology that we might

want, with the usual control that TEXies like. Some further limitations have been lifted. One interesting point is that one can now construct virtual fonts at runtime.

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

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

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

2 Rules

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

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

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

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

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

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

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

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

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

There is no real gain in efficiency because keywords also take time to parse, but the advantage is

that no Lua callbacks are triggered.[1] Of course, this variant would not have been introduced had we still had just rules and no further subtypes; it was just a rather trivial extension that fit in the repertoire.[2]

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

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

```
\startluacode
function FramedRule()
    local width     = 0
    local height    = 0
    local depth     = 0
    local linewidth = 0
    --
    while true do
        if token.scan_keyword("width") then
            width = token.scan_dimen()
        elseif token.scan_keyword("height") then
            height = token.scan_dimen()
        elseif token.scan_keyword("depth") then
            depth = token.scan_dimen()
        elseif token.scan_keyword("line") then
            linewidth = token.scan_dimen()
        else
            break
        end
    end
    local doublelinewidth = 2*linewidth
    --
    local left    = node.new("rule")
    local bottom  = node.new("rule")
    local right   = node.new("rule")
    local top     = node.new("rule")
    local back    = node.new("kern")
    local list    = node.new("hlist")
    --
    left.width    = linewidth
    bottom.width  = width - doublelinewidth
    bottom.height = -depth + linewidth
    bottom.depth  = depth
    right.width   = linewidth
    top.width     = width - doublelinewidth
    top.height    = height
    top.depth     = -height + linewidth
    back.kern     = -width + linewidth
    --
    list.list     = left
    list.width    = width
    list.height   = height
    list.depth    = depth
    list.dir      = "TLT"
    --
    node.insert_after(left,left,bottom)
    node.insert_after(left,bottom,right)
    node.insert_after(left,right,back)
    node.insert_after(left,back,top)
    --
    node.write(list)
 end
\stopluacode
```

This function can be wrapped in a macro:

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

and the macro can be used as follows:

```
\FrameRule width2cm height.5cm depth.5cm line2pt
```

The result is: 

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

```
\startluacode
function FramedRule()
    local width     = 0
    local height    = 0
    local depth     = 0
    local linewidth = 0
    local radius    = 0
    local type      = 0
    --
    while true do
        if token.scan_keyword("width") then
            width = token.scan_dimen()
        elseif token.scan_keyword("height") then
            height = token.scan_dimen()
        elseif token.scan_keyword("depth") then
            depth = token.scan_dimen()
        elseif token.scan_keyword("line") then
            linewidth = token.scan_dimen()
        elseif token.scan_keyword("type") then
            type = token.scan_int()
        elseif token.scan_keyword("radius") then
            radius = token.scan_dimen()
        else
            break
        end
    end
    --
    local r   = node.new("rule")
    r.width   = width
    r.height  = height
```

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

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

```
    r.depth   = depth
    r.subtype = 4    -- user rule
    r[20000]  = type
    r[20001]  = linewidth
    r[20002]  = radius or 0
    node.write(r)
end
\stopluacode
```

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

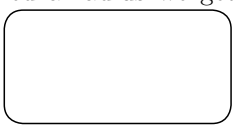
```
\startluacode
local bpfactor = (7200/7227)/65536
local f_rectangle = "%f w 0 0 %f %f re %s"
local f_radtangle = [[
    %f w %f 0 m
    %f 0 l %f %f %f %f y
    %f %f l %f %f %f %f y
    %f %f l %f %f %f %f y
    %f %f l %f %f %f %f y
    h %s
]]

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

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

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

Since we specified a radius we get round corners:

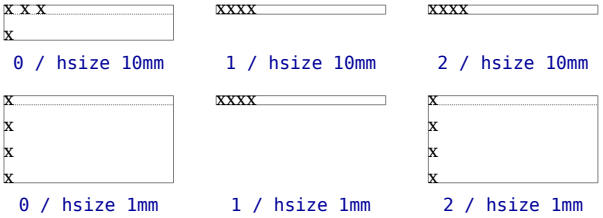


The nice thing about these extensions to rules is that the internals of TEX are not affected much. Rules are just blobs with dimensions and the par builder, for instance, doesn't care what they are. There is no need for further inspection. Maybe future versions of LuaTEX will provide more useful subtypes.

3 Spaces

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

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



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

4 Token lists

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

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

Their use can best be demonstrated by examples. We employ a scratch register `\ToksA`. The examples here show the effects of grouping; in fact, they were

written for testing this effect. Because we don't use the normal assignment code, we need to initialize a local copy in order to get the original content outside the group.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

result: `[AB=AB]`

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

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

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

5 Active characters

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

Below is an example of a definition:

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

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

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

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

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

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

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

Here is the definition of whatever:

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

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

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

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

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

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

Now, when A is made active this meaning kicks in:

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

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

6 \csname and friends

You can check for a macro being defined as follows:

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

which, of course, can be obscured to:

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

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

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

Before ε-TEX, this was done as follows:

```
\expandafter
 \ifx\csname something or nothing\endcsname
 \relax
    do nothing
\else
    do something
\fi
```

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

Now take the following example:

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

If do this is defined, we have two lookups. If it is undefined and do that is defined, we have three lookups. So there is always one redundant lookup.

Also, when no match is found, TEX has to skip to the `\else` or `\fi`. One can save a bit by uglifying this to:

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

This, of course, assumes that there is always a final branch. So let's get back to:

```
\ifcsname do this\endcsname
    \csname do this\endcsname
\else\ifcsname do that\endcsname
    \csname do that\endcsname
\fi\fi
```

As said, when there is some match, there is always one test too many. In case you think this might be slowing down TEX, be warned: it's hard to measure. But as there can be (m)any character(s) involved, including multi-byte UTF-8 characters or embedded macros, there is a bit of penalty in terms of parsing token lists and converting to UTF-8 strings used for the lookup. And, because TEX has to give an error message in case of troubles, the already-seen tokens are stored too.

So, in order to avoid this somewhat redundant operation of parsing, memory allocation (for the lookup string) and storing tokens, the new primitive `\lastnamedcs` is now provided:

```
\ifcsname do this\endcsname
    \lastnamedcs
\else\ifcsname do that\endcsname
    \lastnamedcs
\fi\fi
```

In addition to the (in practice, often negligible) speed gain, there are other advantages: TEX has less to skip, and although skipping is fast, it still isn't a nice side effect (also useful when tracing). Another benefit is that we don't have to type the to-be-looked-up text twice. This reduces the chance of errors. In our example we also save 16 tokens (taking 64 bytes) in the format file. So, there are enough benefits to gain from this primitive, which is not a specific feature, but just an extension to an existing mechanism.

It also works in this basic case:

```
\csname do this\endcsname
\lastnamedcs
```

And even this works:

```
\csname do this\endcsname
\expandafter\let\expandafter\dothis\lastnamedcs
```

And after defining:

```
\bgroup
\expandafter
  \def\csname do this\endcsname{or that}
\global\expandafter
  \let\expandafter\dothis\lastnamedcs
\expandafter
  \def\csname do that\endcsname{or this}
\global\expandafter
  \let\expandafter\dothat\lastnamedcs
\egroup
```

We can use `\dothis` that gives `or that` and `\dothat` that gives `or this`, so we have the usual freedom to be able to use something meant to make code clean for the creation of obscure code.

A variation on this is the following:

```
\begincsname do this\endcsname
```

This call will check if `\do this` is defined, and, if so, will expand it. However, when `\do this` is not found, it does not create a hash entry. It is equivalent to:

```
\ifcsname do this\endcsname\lastnamedcs\fi
```

but it avoids the `\ifcsname`, which is sometimes handy as these tests can interfere.

I played with variations like `\ifbegincsname`, but we then quickly end up with dirty code due to the fact that we first expand something and then need to deal with the following `\else` and `\fi`. The two above-mentioned primitives are non-intrusive in the sense that they were relatively easy to add without obscuring the code base.

As a bonus, LuaTEX also provides a variant of `\string` that doesn't add the escape character: `\csstring`. There is not much to explain to this:

```
\string\whatever<>\csstring\whatever
```

This gives: `\whatever<>whatever`

The main advantage of these several new primitives is that a bit less code is needed and (at least for ConTEXt) leads to a bit less tracing output. When you enable `\tracingall` for a larger document or example, which is sometimes needed to figure out a problem, it's not much fun to work with the resulting megabyte (or sometimes even gigabyte) of output so the more we can get rid of, the better. This consequence is just an unfortunate side effect of the ConTEXt user interface with its many parameters. As said, there is no real gain in speed.

7 Packing of lists

Deep down in TEX, horizontal and vertical lists eventually get packed. Packing of an `\hbox` involves:

1. ligature building (for traditional TEX fonts),
2. kerning (for traditional TEX fonts),
3. calling out to Lua (when enabled) and

4. wrapping the list in a box and calculating the width.

When a Lua function is called, in most cases, the location where it happens (group code) is also passed. But say that you try the following:

```
\hbox{\hbox{\hbox{\hbox foo}}}
```

Here we do all four steps, while for the three outer boxes, only the last step makes any sense. And it's not trivial to avoid the application of the Lua function here. Of course, one can assign an attribute to the boxes and use that to intercept, but it's kind of clumsy. This is why we now can say:

```
\hpack{\hpack{\hpack{\hbox foo}}}
```

There are also `\vpack` for a `\vbox` and `\tpack` for a `\vtop`. There can be a small gain in speed when many complex manipulations are done, although in, for instance, ConTEXt, we already have provisions for that. It's just that the new primitives are a cleaner way out of a conceptually nasty problem. Similar functions are available on the Lua side.

8 Errors

We end with a few options that can be convenient to use if you don't care about exact compatibility.

```
\suppresslongerror
\suppressmathparerror
\suppressoutererror
\suppressifcsnameerror
```

When entering your document on a paper teletype terminal, starting TEX, and then going home in order to have a look at the result the next day, it does make sense to catch runaway cases, like premature ending of a paragraph (using `\par` or equivalent empty lines), or potentially missing `$$`s. Nowadays, it's less important to catch such coding issues (and be more tolerant) because editing takes place on screen and running (and restarting) TEX is very fast.

The first two flags given above deal with this. If you set the first to any value greater than zero, macros not defined as `\long` (not accepting paragraph endings) will not complain about `\par` tokens in arguments. The second setting permits and ignores empty lines (also pars) in math without reverting to dirty tricks. Both are handy when your content comes from places that are outside of your control. The job will not be aborted (or hang) because of an empty line.

The third setting suppresses the `\outer` directive so that macros that originally can only be used at the outer level can now be used anywhere. It's hard to explain the concept of outer (and the related error message) to a user anyway.

The last one is a bit special. Normally, when you use `\ifcsname` you will get an error when TEX sees something unexpandable or that can't be part of a name. But sometimes you might find it to be quite acceptable and can just consider the condition as false. When the fourth variable is set to non-zero, TEX will ignore this issue and try to finish the check properly, so basically you then have an `\iffalse`.

9 Final remarks

I mentioned performance a number of times, and it's good to notice that most changes discussed here will potentially be faster than the alternatives, but this is not always noticeable, in practice. There are several reasons.

For one thing, TEX is already highly optimized. It has speedy memory management of tokens and nodes and unnecessary code paths are avoided. However, due to extensions to the original code, a bit more happens in the engine than in decades past. For instance, Unicode fonts demand sparse arrays instead of fixed-size, 256-slot data structures. Handling UTF involves more testing and construction of more complex strings. Directional typesetting leads to more testing and housekeeping in the frontend as well as the backend. More keywords to handle, for instance `\hbox`, result in more parsing and pushing back unmatched tokens. Some of the penalty has been compensated for through the changing of whatsits into regular nodes. In recent versions of LuaTEX, scanning of `\hbox` arguments is somewhat more efficient, too.

In any case, any speedup we manage to achieve, as said before, can easily become noise through inefficient macro coding or user's writing bad styles. And we're pretty sure that not much more speed can be squeezed out. To achieve higher performance, it's time to buy a machine with a faster CPU (and a huge cache), faster memory (lanes), an SSD, and regularly check your coding.

◇ Hans Hagen
Pragma ADE
http://pragma-ade.com