Extensions Related to Programming Macros

Hans Hagen

1. Introduction

Sometimes you can read (or hear) comments about TEX not being a real programming language or the wish for it to be more like a typical procedural language. A discussion about this is somewhat pointless because it relates to experiences and preferences. Also, when we mention TEX, we are talking about an interpreter, a language, a set of macros and in practice, about an ecosystem, simply because all kinds of resources are involved—especially since the ecosystem is the one reason why a successor has not shown up.

So, when we discuss the language aspect, it concerns a macro language. This is for a good reason: one can mix content and operations on that content in one document source. That source is interpreted and processed as it goes. This is contrary to a procedural language, where one explicitly has to push content into some procedure. There are some languages that are mixed, e. g., webpage templates where some elements are snippets of programs and a preprocessor assembles the result.

```
\def\MyMacroA#1{This or #1!}
\def\MyMacroB{that}

\MyMacroA{\MyMacroB}
```

Here, the last line will result in “This or that!” in the output. But it must be noted that `\MyMacroB` is passed as a token, and only in the body of the macro does it get expanded into “that”.

```
\edef\MyMacroC{\MyMacroB}
```

The code above defines a new macro with the expanded text as the body. To expand or not, that is often the question. Now compare this code with the following:

```
function MyFunctionA(one)
    return "This or " .. one .. "!"
end
function MyFunctionB()
    return "that"
end
function MyFunctionC(one)
    return "This or " .. one() .. "!"
end
```

```
MyFunctionA("that")
MyFunctionA(MyFunctionB())
MyFunctionC(MyFunctionB)
```

The first function expects a string and returns a concatenation. The second function returns a string. The first call gets a string passed and the second one too because we call that function. But the third call passes the function itself, which is why the third function has to call it explicitly in the function body. It is this property that, in my opinion, complicates matters when you want to do typesetting in such a language: the more you nest, the more dangers there are for asynchronous side effects. This can be understood from the following example:

```
function MyFunctionA(one)
    print("A")
    return "This or " .. one .. "!"
end
function MyFunctionB()
    print("B")
    return "that"
end

MyFunctionA(MyFunctionB())
```

Here we print `B` before we print `A`. Now, one can certainly argue that despite this, functions are easier to understand than macros (which can also have surprising side effects). Indeed, when one works on an abstract document tree where content is fetched from, say, a database, that might be true but most TEX users mix content and operations.

In the following sections I will introduce some of the additional features that LuaMetaTEX provides. These are the result of many years of experience in writing macros and the wish to come up with readable code using native features of the language wherever possible. Of course, in ConTEXt we have a high level interface for dealing with typographical constructs and properties, but deep down the code looks less clear. Putting layer upon layer doesn't help much either, so we are not taking that route. Using funny characters like `!?@_:` doesn't make things look better either. We do have lots of so-called low-level macros but it doesn't make much sense to come up with a pseudo-programming layer while in fact the engine could make better facilities available; so this is the route we follow. After decades it had become clear that none of the successor TEX variants had filled in the gaps in this way, so at some point I decided that LuaMetaTEX should do it (at least for ConTEXt).

While ConTEXt MkII was written for the more traditional engines pdfTEX and XƎTEX, MkIV targets LuaTEX. It resulted in a rewrite of many components and a freeze of MkII. It made no sense to cripple ourselves, but in the end we went further than we had originally intended. Then, when LuaMetaTEX development started, again a rewrite happened, but this time the reason was to make the code base a bit more

efficient (less indirectness) by using extended native functionality. Apart from other benefits of this new engine, it gave us a cleaner code base with fewer layers. This is why ConTEXt LMTX (a.k.a. MkXL) has been split-off from the MkIV code base to protect it from harmful changes. All that said, I do admit that lacking other TEX challenges, it is also fun to explore new avenues.

2. Conditions

It must be said that when one goes even a little beyond simple TEX programming, one could indeed wish for a bit more comfort. Take this:[1]

```
\def\MyMacro#1#2%
  {\ifdim\dimexpr#1\relax<\dimexpr#2\relax
     less%
   \else\ifdim\dimexpr#1\relax=\dimexpr#2\relax
     equal%
   \else
     more%
   \fi\fi}
```

One needs to keep track of the nesting here in order to have the right number of `\fi`s.

```
\def\doifelse#1#2#3#4%
  {\edef\a{#1}\edef\b{#1}%
   \ifx\a\b#3\else#4\fi}
```

The temporary macros are needed in order to be able to compare the expanded meanings. But when `#3` and `#4` are macros that look ahead you can imagine that when they see `\else` or `\fi` things can get confused. Compare this to:

```
function doifelse(a,b,c,d)
    if a == b then
        c()
    else
        d()
    end
end
```

Here the compiler creates code that calls either `c()` or `d()` without them having to bother about leaving the condition. In TEX-speak we would need to have something like this:

[1] We use a `\dimexpr` because we cannot use a terminal percentage or space if we want to be fully expandable and don't want spaces to creep in after one token arguments.

```
\def\firstoftwoarguments #1#2{#1}
\def\secondoftwoarguments#1#2{#2}
\def\doifelse#1#2#3#4%
  {\edef\a{#1}\edef\b{#1}%
   \ifx\a\b
     \expandafter\firstoftwoarguments
   \else
     \expandafter\secondoftwoarguments
   \fi}
```

And when you try that with the first example where we had a nested condition, you can imagine that it quickly starts looking complex. Another aspect of the last macro is that it uses two temporary macros that must have names that don't clash, meaning that the ones we have chosen here are pretty bad. I will come back to dealing with this later.

One gets accustomed to this complexity, and often this kind of code is hidden from the user so only macro writers are victims here. But, being one myself, the question is, can we make the code cleaner?

Let's redo the first example with LuaMetaTEX:

```
\def\MyMacro#1#2%
  {\ifdim\dimexpr#1\relax<\dimexpr#2\relax
     less%
   \orelse\ifdim\dimexpr#1\relax=\dimexpr#2\relax
     equal%
   \else
     more%
   \fi}
```

Many programming languages have something like `elseif` but because TEX has quite a number of different tests, `\elseifdim` makes no sense but the more generic `\orelse` does. We can even think of:

```
\def\MyMacro#1#2%
  {\ifcmpdim\dimexpr#1\relax\dimexpr#2\relax
     less%
   \or
     equal%
   \else
     more%
   \fi}
```

And because LuaMetaTEX provides this test, one obstacle is gone. We leave it to the reader to come up with a traditional TEX implementation of this:

```
\def\MyMacro#1#2%
  {\ifcmpdim\dimexpr#1\relax\dimexpr#2\relax
     \expandafter\firstofthreearguments
   \or
     \expandafter\secondofthreearguments
   \else
     \expandafter\thirdofthreearguments
   \fi}
```

And how nice it would be to be able to do this:

```
\def\doifelse#1#2%
  {\iftok{#1}{#2}%
     \expandafter\firstoftwoarguments
   \else
     \expandafter\secondoftwoarguments
   \fi}
```

And so, LuaMetaTEX has such a primitive test. Keep in mind that defining `\iftok` as a macro is possible here but that won't work well nested, even with `\orelse`:

```
\iftok{.}{.}
\orelse\iftok{..}{..}
\orelse\iftok{...}{...}
\fi
```

When a condition succeeds or fails, TEX enters fast scanning mode to skip over the branch that is not used. For that it needs to know if a token is a test, which is why defining `\iftok` as a macro is of no help. We could flag a macro as a test (and I have actually played with this), but it means that we need to test a macro property independent of the current condition handler, and that is something for later. As an intermediate solution, we have an `\ifcondition` primitive that is seen as a condition when fast scanning happens and as a no-op when a condition is expected. In this case the following macro has to expand to a condition itself. Something like this:

```
\ifcondition\mytest{.}{.}
\orelse\ifcondition\mytest{..}{..}
\orelse\ifcondition\mytest{...}{...}
\fi
```

Because we have Lua, there are also ways to let Lua functions behave like 'if' tests but that is beyond this overview since it goes beyond the scope of the macro language. In ConTEXt we use this feature to implement some bitwise operations and tests.

In the engine, we provide the following repertoire of tests: \if, \ifcat, \ifnum, \ifdim, \ifodd, \ifvmode, \ifhmode, \ifmmode, \ifinner, \ifvoid, \ifhbox, \ifvbox, \ifx, \iftrue, \iffalse, \ifcase, \ifdefined, \ifcsname, \iffontchar, \ifincsname, \ifabsnum, \ifabsdim, \ifchknum, \ifchkdim, \ifcmpnum, \ifcmpdim, \ifnumval, \ifdimval, \iftok, \ifcstok, \ifcondition, \ifflags, \ifempty, \ifrelax, \ifboolean, \ifmathparameter, \ifmathstyle, \ifarguments, \ifparameters, \ifparameter, \ifhastok, \ifhastoks and \ifhasxtoks.

Some of these are variants of \ifcase and are needed when there are more than two possible outcomes. In addition, there are \unless, \else, \or, \orelse and \orunless. The new primitives are discussed in the documents that come with the ConTEXt distribution.

With respect to testing arguments, you can also use the pseudo-counter \lastarguments (watch the 'last' in the name) and the somewhat less efficient but more reliable \parametercount, as these are indicators of the number of passed commands.

3. Protection

In the previous section we mentioned that using auxiliary macros is tricky because they can clash with existing macros. In fact, this is true for any macro! I therefore decided to do what has been on the agenda for a while: add a mechanism that protects against overloading. This is still experimental and the impact on users can only be tested after most ConTEXt users have switched to LMTX, which may take a while. This also means that it will take a while before the related primitives are considered stable (although I'm sure not much will change). Let's take a previous example:

```
\permanent\def\firstoftwoarguments #1#2{#1}
\permanent\def\secondoftwoarguments#1#2{#2}
\permanent\protected\def\doifelse#1#2%
  {\iftok{#1}{#2}%
     \expandafter\firstoftwoarguments
   \else
     \expandafter\secondoftwoarguments
   \fi}
```

Here the three macros are defined as permanent. The test itself is protected against expansion (which it has always been, so we keep it). Depending on the value of the \overloadmode variable (discussed below) a user will get a warning or fatal error. By default there is no checking (but I might give the \immutable prefix, also discussed below, an "always check for it" property).

The whole repertoire of prefixes related to overload protection is given in the following table.

`frozen`	a macro that has to be redefined in a managed way
`permanent`	a macro that had better not be redefined
`primitive`	a primitive that will not normally be adapted
`immutable`	a macro or quantity that cannot be changed; it is a constant
`mutable`	a macro that can be changed no matter how well protected it is
`instance`	a macro marked (for instance) for generation by the user interface
`overloaded`	when permitted the flags will be adapted
`enforced`	all is permitted (but only in zero mode or 'initex' mode)
`aliased`	the macro gets the same flags as the original

For the first five, the primitive states have no related prefix primitive; it is set by the engine itself. Maybe someday I will decide to permit defining primitives, which would take hardly any code to implement. Permanent macros are (as shown) those that we don't want users to redefine, and frozen ones are mildly protected. They can be redefined when the `\overloaded` prefix is used. A mutable macro can always be redefined (think of temporary macros), while an immutable can never be redefined. The instance property is just a signal that we're dealing with an instance, which can be handy when we are tracing a macro's execution. The `\aliased` prefix will copy properties, so this:

```
\aliased\let\forgetaboutit\relax
```

makes `\forgetaboutit` a reference to the current meaning of `\relax` (because that is what `\let` does) but also protects it like a primitive (because that is what `\relax` is).

The `\enforced` prefix is special. It only has a meaning inside a macro body or token register and it gets converted in a (hidden) `\always` prefix when in so-called ini mode (when the format is made). This permits system macros to overload in spite of heavy protection against it. Think of macros like `\NC` where the meaning can differ depending on the kind of table mechanism used, or `\item` which can differ by environment. We can protect these against overloading by the user but still redefine them. Of course, when the overload mode is zero, all can be redefined.

The value of `\overloadmode` determines to what extent a user will be annoyed when an existing macro is redefined, as shown in the table below. Such a macro can also be an instance defined by commands like `\definehighlight` although these normally are just `\frozen \instance` which means that a low level of protection only issues a warning.

		immutable	permanent	primitive	frozen	instance
1	warning	⋆	⋆	⋆		
2	error	⋆	⋆	⋆		
3	warning	⋆	⋆	⋆	⋆	
4	error	⋆	⋆	⋆	⋆	
5	warning	⋆	⋆	⋆	⋆	⋆
6	error	⋆	⋆	⋆	⋆	⋆

The even values (except zero) will abort the run. A value of 255 will freeze this parameter. At level five and above the instance flag is also checked but no drastic action takes place. We use this to signal to the user that a specific instance is redefined (of course the definition macros can check for that too).

4. Alignments

In ConTEXt, many commands are defined using the prefix `\protected`, which is handy when they are used in a context where expansion would not work out well, like writing to file or inside an `\edef`. However, this is impossible when we use the alignment mechanism. This has to do with the fact that the parser looks ahead to see if we have (for instance) a `\noalign` primitive. And since the parser doesn't look inside a `\protected` macro, the following fails:

```
\protected\def\MyMacro{\noalign{\vskip 10pt}}
```

It also works out badly for macros that look for arguments. A dirty trick is:

```
\def\MyMacroA{\noalign\bgroup\MyMacroB}
\def\MyMacroB{\dosingleempty\MyMacroC}
\def\MyMacroC[#1]{....\egroup}
```

This somewhat over-the-top approach can now (in LuaMetaTEX) be simplified to the following. Let's also go crazy with prefixes here:

```
\noaligned\permanent\tolerant\protected\def\MyMacroA[#1]%
  {\noalign\bgroup....\egroup}
```

For the record: in LuaMetaTEX the `\noalign` construct can be nested which again simplifies some (ConTEXt) code. Keep in mind that until now we could do whatever we wanted in traditional TEX speak, apart from making such macros `\protected`.

5. Definitions

From the perspective of the above, it will become clear that in a system like ConTEXt quite a number of definitions are candidates for being flagged. You also need to think of symbolic character names or math symbols. For instance, dimensions defined by `\dimendef` also get a permanent status. This means that one cannot redefine `\scratchcounter` but its value can still be changed. At this moment I see

no reason to have a flag for preventing this (also because it would add overhead), but it might become an option some day.

However, there are often quantities that need overload protection, such as constant values. This is why we have:

```
\immutable \integerdef    \plusone   1
\immutable \dimensiondef  \onepoint  1pt
\immutable \gluespecdef   \zeroskip  0pt plus 0pt minus 0pt
\immutable \mugluespecdef \onemuskip 1mu
```

These will never change and are a macro-like variant of registers but with an efficient storage model and that behave like a register. However, one cannot use the operators like `\advance` on them. Their intended usage is as a constant.

Another definition-related extension involves `\csname`. In LuaTEX we introduced more robust handling of `\ifcsname` as well as an extra accessor:

```
\ifcsname f o o\endcsname
    \lastnamedcs % reference to the constructed \cs
\fi
```

as well as:

```
\begincsname f o o\endcsname
```

which doesn't define `\f o o` as a 'relaxed' macro when it doesn't already exist. Both `\begincsname` and `\lastnamedcs` avoid a second name construction, as in:

```
\ifcsname f o o\endcsname
    \csname f o o\endcsname
\fi
```

Keep in mind that these additions are a side effect of control sequences being in UTF-8 format so we want to avoid unnecessary construction of temporary strings and related expansion.

Original TEX only had `\csname`; ε-TEX and LuaTEX added some companion primitives to this, and LuaMetaTEX once again extends the repertoire:

```
\letcsname  f o o\endcsname\relax
\defcsname  f o o\endcsname{...}
\edefcsname f o o\endcsname{...}
\gdefcsname f o o\endcsname{...}
\xdefcsname f o o\endcsname{...}
```

This saves passing some arguments to a helper like `\setvalue`, which is a bit more efficient, and it also saves a token. (The ConTEXt format file became quite a bit smaller when the extensions discussed here were applied.) The `\ifcsname` primitive has been made somewhat more efficient by honoring macros that were defined as `\protected` which (we think) means: don't expand me in those cases where it

makes no sense. So here we have an (in my opinion) acceptable downward incompatibility with engines that conform to ε-TEX.

There are a few more definition-related new primitives, like:

```
\glet\MyMacroA\MyMacroB              % shortcut for \global\let
\swapcsvalues\MyMacroA\MyMacroB      % also works for registers
\futuredef\DoWhatever\MyMacro{...}
\expand\MyProtectedMacro             % \protected like this
```

6. Arguments

Let's start with a teaser. A previous definition needed a helper to gobble one of two arguments. The following does the same but it just gobbles and doesn't store the argument, which is why we use #1 in both cases. This avoids storing token lists for the unused arguments.

```
\permanent\def\firstoftwoarguments #1#-{#1}
\permanent\def\secondoftwoarguments#-#1{#1}
```

Because anything other than a digit after a # triggers an error, I saw no reason not to support some more characters: it doesn't hurt downward compatibility, unless you use TEX to generate error messages. Here is the full list of extensions, of which I will discuss a few (more can be found in the ConTEXt distribution and source code).

+	keep the braces
-	discard and don't count the argument
/	remove leading and trailing spaces and pars
=	braces are mandatory
_	braces are mandatory and kept
^	keep leading spaces
1-9	an argument
0	discard but count the argument
*	ignore spaces
:	pick up scanning here
;	quit scanning

We have a few useful characters left, such as < and > so who knows what future extensions might show up?

Delimited arguments are used frequently in ConTEXt; take this:

```
\def\MyMacro[#1][#2]{...}
```

Here the call is rather sensitive, for instance this will fail:

```
\MyMacro[A] [B]
```

We can cheat and define:

```
\def\MyMacro[#1]#2[#3]{...}
```

in which case #2 gets what sits between the brackets. But still these two arguments have to be given. So, in MkII and MkIV you will find indirectness like the following:

```
\def\MyMacro{\dodoubleempty\doMyMacro}
\def\doMyMacro[#1][#2]{}
```

However, in LMTX you can find this alternative:

```
\tolerant\def\MyMacro[#1]#*[#2]{...}
```

The `\tolerant` will make the parser quit when no match can be made and the #* will gobble spaces. In fact, we often do this:

```
\tolerant\protected\def\MyMacro[#1]#*[#2]{...}
```

and if we want overload protection:

```
\permanent\tolerant\protected\def\MyMacro[#1]#*[#2]{...}
```

The combination of `\tolerant` and `\protected` with either expansion or not of a macro gives four variants of low-level macro commands: *normal*, *tolerant normal*, *protected* and *tolerant protected*. In LuaTEX that protection against expansion is implemented in a more indirect way, just like in ε-TEX. There we also have `\long` and `\outer` properties so we have *normal*, *long normal*, *outer normal* and *long outer normal*. Making *protected against expansion* a native command would have given another four command codes. Combining this with `\tolerant` would again double it so we then would end up with 16 command codes. But in LuaMetaTEX we dropped the `\long` and `\outer` properties. In ConTEXt we never used `\outer` and always want `\long` anyway.

The reason for mentioning these details is to make clear that the introduced overhead is negligible when we compare it to LuaTEX. Apart from the fact that we gain from the expansion protection being a first class feature, macros without arguments are being stored more efficiently, the parser is a little better optimized, and so on.

But of course the biggest benefit is that, when we look at the example above, we avoid indirectness. It looks nicer. It gives less clutter in tracing. It takes fewer tokens in the format (where each token takes eight bytes). It runs a little faster. It demands no trickery. Take your choice. For the record: you don't want to know what the set of `\dodoubleempty` macros looks like, as they themselves use indirectness and are highly optimized for performance.

The list of possible features has more than skipping spaces. Here's another example:

```
\tolerant\def\MyMacro[#1]#;(#2){<#1#2>}
```

Here `\MyMacro` accepts `[A]` and then quits or, when not seen, checks for `(A)` and when not found is still happy. So, either `#1` or `#2` has a value. How do we know which arguments got grabbed? There are several ways to find out:

```
\tolerant\def\MyMacro[#1]#;(#2)%
  {\ifarguments
     % zero arguments
   \or
     % one argument
   \else
     % two arguments
   \fi}
```

This test uses the count from the last expansion so if any macro expansion happens before the test, you can get the wrong value! The next test provides feedback about which argument received a value:

```
\tolerant\def\MyMacro[#1]#;(#2)%
  {\ifparameters
     % all empty
   \or
     % first has value
   \else
     % second has value
   \fi}
```

But this still may not be enough so we can also explicitly test for a parameter. Again be aware of the nesting:

```
\tolerant\def\MyMacro[#1]#;(#2)%
  {\ifparameter#1\or
     % first has value
   \fi
   \ifparameter#2\or
     % second has value
   \fi}
```

This is pretty robust but it expands the arguments in the test:

```
\tolerant\def\MyMacro[#1]#;(#2)%
  {\unless\iftok{#1}{}%
     % first has value
   \fi
   \unless\iftok{#2}{}%
     % second has value
   \fi}
```

When we use a colon instead of a semicolon the parser knows where to pick up after a match fails:

```
\tolerant\def\MyMacro[#1]#:#2{...}
```

So, the argument between brackets is optional and the single token or braced second argument (turned into a token list) is mandatory.

The other extensions more or less speak for themselves: they grab arguments and discard or keep braces and such, in cases where TEX would treat them specially when storing or passing them on.

Speaking of braces, in spite of what one might expect (assuming that braces are more a TEX thing than brackets), the following two definitions perform equally well:

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

but:

```
\def\oof[#1]{}
\def\foo{\dosingleempty\oof}
```

performs more that five times worse than this:

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

So, the added overhead in the argument parser (and there is some, also because we keep track of more) gets compensated well by the fact that we can avoid indirectness. The impact on an average document would probably go unnoticed.

As with much in TEX, you need to be aware of (intentional) side effects. Take for instance:

```
\tolerant\def\foo#1[#2]#*[#3]{\edef\ofo{#1}}
\def\oof{\foo{oeps}}
```

This will probably not do what you expect. It has to do with how TEX interprets spaces in the context of argument parsing: they can become part of the argument (here #1) so anything before the first seen left bracket becomes the argument's value.

```
\tolerant\def\foo#1#*[#2]#*[#3]{\edef\ofo{#1}}
\def\oof{\foo{oeps}}
```

This works because the first #* directive stops scanning for the first argument. It then gobbles spaces when they are seen before continuing to look for the bracketed arguments. So TEX's charm is still there.

7. Introspection

Because macros have more properties and variation in arguments the `\meaning` command has a companion `\meaningfull` that displays what prefixes were applied. The `\meaningless` variant only shows the body.

Quite some effort went into normalizing the so-called command codes. Primitives are grouped into categories with similar treatments in order to keep the main loop efficient. These codes also determine the expansion contexts (think of usage in an `\edef`, how they get serialized, for instance in messages etc.). The char codes (called such because in most cases tokens represent characters of some kind) distinguish commands in these groups. Think of `\def` and `\edef` being call commands with a different code. This rather intrusive (internal) regrouping of primitives was needed in order to get a more consistent Lua token interface. So, for instance the codes are now in consecutive ranges, registers are split into internal and user variants, etc.

Also, memory management has been overhauled so we have a more dynamic allocation of various data structures (stacks, equivalents, tokens, nodes, etc.) and we use the whole 64-bit memory word to save some memory in places too. All this is the reason why it is unlikely that much will get backported to LuaTEX. Also, in ConTEXt we now have a special version for LuaMetaTEX: LMTX.

8. There is more

Here we've discussed only the primitives that make the source look better while also making it more convenient. But it is worth mentioning that there are primitives like `\toksapp` and `\etokspre` that append and prepend tokens to a register (there are eight variants). There are ways to collect tokens for just before or after a group ends. There are some new expansion-related primitives like `\expandtoken` that can be used to inject a token with some specific catcode, just like one can define active characters without the need for dirty uppercase tricks.

The typesetting department also has extensions. We can freeze paragraph properties, adjust math parameters locally, normalize lines so that at the Lua end we know what to expect (think of consistent presence of left and right skip, left and right shape related properties, left and right parfill skips, indentation being glue, etc.). Hyphenation can be controlled in more detail too, and left and right side ligatures and kerns can be influenced in the running text and go with glyphs. Talking of glyphs, there are advanced scaling options as well as support for influencing placement in the running text, which permits more efficient font handling. Boxes have more properties too: they can have offsets, an orientation, etc. which makes implementing vertical typesetting a bit easier. Rules also have shifts. We can register actions to be expanded at the end of a paragraph. All this evolved over time and has been tested in ConTEXt but will be applied more frequently after the complete code split between MkIV and LMTX. That process goes hand-in-hand with adapting to the new situation, removing old (obsolete) variants, removing still present experimental code, etc.

There is more but hopefully this gives an impression of how substantial the LuaMetaTEX engine differs (in added functionality) with its predecessors. Maybe it looks a bit over the top, but I did actually reject some ideas after experimenting with them. On the other hand, there are still some ideas on the agenda. For instance, the engine can migrate and carry around so-called 'deeply buried inserts' pretty well now. However, dealing with inserts could be made a bit easier (think of columns), so we're not done yet.

It should be noted that, contrary to what one might expect, the code base is still quite okay and the binary stays well below 3 MB. In the meantime, memory management has also improved and the format file gotten smaller. A lot of the internal reorganization relates to the fact that we have a Lua interface which exposes internals, thereby demanding consistency, avoidance of (often clever) tricks, more abstraction, etc.

It is also worth noting that we can only do such a massive operation because users are willing to test intermediate versions (sometimes on very large projects) and because all changes in the code base are meticulously checked by Wolfgang Schuster who knows TEX and ConTEXt inside out. And of course we have Mojca Miklavec's compile farm to keep them available for all relevant platforms, where we use a mix of `gcc` (also with cross compilation), `clang` and `msvc` for various platforms. It definitely helps that compilations are fast (due to the refactored code base) and that I can use Visual Studio to work with the code.

In this summary I have only covered some of the aspects of TEX. Another important set of extensions concerns the MetaPost library, where token scanners are exposed, more advanced Lua calls are possible and where obsolete bits of code have been removed. And we use the latest and greatest Lua 5.4—but discussing the implications of these is for another article.

This article was first published in the spring 2021 issue of TugBoat and then in MAPS No.51.
Many thanks to Karl Berry who improved the English while copy-editing the text.