low level TEX

macros

Contents

1 Preamble

This chapter overlaps with other chapters but brings together some extensions to the macro definition and expansion parts. As these mechanisms were stepwise extended, the other chapters describe intermediate steps in the development.

Now, in spite of the extensions discussed here the main ides is still that we have TEX act like before. We keep the charm of the macro language but these additions make for easier definitions, but (at least initially) none that could not be done before using more code.

2 Definitions

A macro definition normally looks like like this:[1]

```
\def\macro#1#2%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

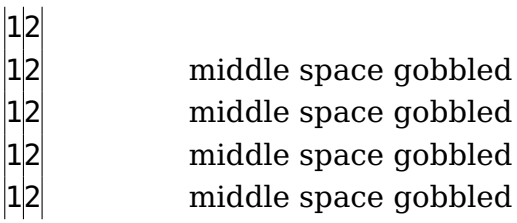
Such a macro can be used as:

```
\macro {1}{2}
\macro {1} {2}  middle space gobbled
\macro 1 {2}    middle space gobbled
\macro {1} 2    middle space gobbled
```

[1] The `\dontleavehmode` command make the examples stay on one line.

```
\macro 1 2      middle space gobbled
```

We show the result with some comments about how spaces are handled:



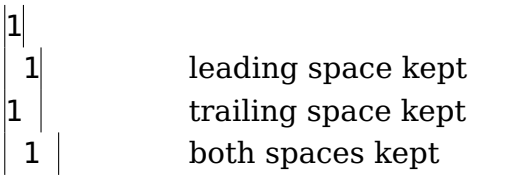
A definition with delimited parameters looks like this:

```
\def\macro[#1]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\hss}}
```

When we use this we get:

```
\macro [1]
\macro [ 1]    leading space kept
\macro [1 ]    trailing space kept
\macro [ 1 ]   both spaces kept
```

Again, watch the handling of spaces:



Just for the record we show a combination:

```
\def\macro[#1]#2%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

With this:

```
\macro [1]{2}
\macro [1] {2}
\macro [1] 2
```

we can again see the spaces go away:

12

A definition with two separately delimited parameters is given next:

```
\def\macro[#1#2]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

When used:

```
\macro [12]
\macro [ 12]     leading space gobbled
\macro [12 ]     trailing space kept
\macro [ 12 ]    leading space gobbled, trailing space kept
\macro [1 2]     middle space kept
\macro [ 1 2 ]   leading space gobbled, middle and trailing space kept
```

We get ourselves:

12	
12	leading space gobbled
12	trailing space kept
12	leading space gobbled, trailing space kept
1 2	middle space kept
1 2	leading space gobbled, middle and trailing space kept

These examples demonstrate that the engine does some magic with spaces before (and therefore also between multiple) parameters.

We will now go a bit beyond what traditional TEX engines do and enter the domain of LuaMetaTEX specific parameter specifiers. We start with one that deals with this hard coded space behavior:

```
\def\macro[#^#^]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

The #^ specifier will count the parameter, so here we expect again two arguments but the space is kept when parsing for them.

```
\macro [12]
\macro [ 12]
\macro [12 ]
\macro [ 12 ]
\macro [1 2]
```

```
\macro [ 1 2 ]
```

Now keep in mind that we could deal well with all kind of parameter handling in ConTEXt for decades, so this is not really something we missed, but it complements the to be discussed other ones and it makes sense to have that level of control. Also, availability triggers usage. Nevertheless, some day the #^ specifier will come in handy.

```
|1|2|
|  |12|
|12|  |
|  |12|  |
|1  |2|
|  |1 2|  |
```

We now come back to an earlier example:

```
\def\macro[#1]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\hss}}
```

When we use this we see that the braces in the second call are removed:

```
\macro [1]
\macro [{1}]
```

```
|1|  |1|
```

This can be prohibited by the #+ specifier, as in:

```
\def\macro[#+]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\hss}}
```

As we see, the braces are kept:

```
\macro [1]
\macro [{1}]
```

Again, we could easily get around that (for sure intended) side effect but it just makes nicer code when we have a feature like this.

```
|1|  |{1}|
```

Sometimes you want to grab an argument but are not interested in the results. For this we have two specifiers: one that just ignores the argument, and another one that keeps counting but discards it, i.e. the related parameter is empty.

```
\def\macro[#1][#0][#3][#-][#4]%
  {\dontleavehmode\hbox spread 1em
     {\vl\type{#1}\vl\type{#2}\vl\type{#3}\vl\type{#4}\vl\hss}}
```

The second argument is empty and the fourth argument is simply ignored which is why we need #4 for the fifth entry.

```
\macro [1][2][3][4][5]
```

Here is proof that it works:

|1|3|5|

The reasoning behind dropping arguments is that for some cases we get around the nine argument limitation, but more important is that we don't construct token lists that are not used, which is more memory (and maybe even cpu cache) friendly.

Spaces are always kind of special in TEX, so it will be no surprise that we have another specifier that relates to spaces.

```
\def\macro[#1]#*[#2]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

This permits usage like the following:

```
\macro [1][2]
\macro [1] [2]
```

|1|2| |1|2|

Without the optional ‘grab spaces’ specifier the second line would possibly throw an error. This because TEX then tries to match `][` so the `] [` in the input is simply added to the first argument and the next occurrence of `][` will be used. That one can be someplace further in your source and if not TEX complains about a premature end of file. But, with the #* option it works out okay (unless of course you don't have that second argument `[2]`.

Now, you might wonder if there is a way to deal with that second delimited argument being optional and of course that can be programmed quite well in traditional macro code. In fact, ConTEXt does that a lot because it is set up as a parameter driven system with optional arguments. That subsystem has been optimized to the max over years and it works quite well and performance wise there is very little to gain. However, as soon as you enable tracing you end up in an avalanche of expansions and that is no fun.

This time the solution is not in some special specifier but in the way a macro gets defined.

```
\tolerant\def\macro[#1]#*[#2]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

The magic `\tolerant` prefix with delimited arguments and just quits when there is no match. So, this is acceptable:

```
\macro [1][2]
\macro [1] [2]
\macro [1]
\macro
```

|1|2| |1|2| |1| |

We can check how many arguments have been processed with a dedicated conditional:

```
\tolerant\def\macro[#1]#*[#2]%
  {\ifarguments 0\or 1\or 2\or ?\fi: \vl\type{#1}\vl\type{#2}\vl}
```

We use this test:

```
\macro [1][2] \macro [1] [2] \macro [1] \macro
```

The result is: 2: |1|2| 2: |1|2| 1: |1|0: | which is what we expect because we flush inline and there is no change of mode. When the following definition is used in display mode, the leading `n=` can for instance start a new paragraph and when code in `\everypar` you can loose the right number when macros get expanded before the `n` gets injected.

```
\tolerant\def\macro[#1]#*[#2]%
  {n=\ifarguments 0\or 1\or 2\or ?\fi: \vl\type{#1}\vl\type{#2}\vl}
```

In addition to the `\ifarguments` test primitive there is also a related internal counter `\lastarguments` set that you can consult, so the `\ifarguments` is actually just a shortcut for `\ifcase\lastarguments`.

We now continue with the argument specifiers and the next two relate to this optional grabbing. Consider the next definition:

```
\tolerant\def\macro#1#*#2%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

With this test:

```
\macro {1} {2}
\macro {1}
\macro
```

We get:

```
|1|2|   |1|\macro|
```

This is okay because the last `\macro` is a valid (single token) argument. But, we can make the braces mandate:

```
\tolerant\def\macro#=#*#=%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Here the #= forces a check for braces, so:

```
\macro {1} {2}
\macro {1}
\macro
```

gives this:

```
|1|2|   |1|  |
```

However, we do loose these braces and sometimes you don't want that. Of course when you pass the results downstream to another macro you can always add them, but it was cheap to add a related specifier:

```
\tolerant\def\macro#_#*#_%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Again, the magic `\tolerant` prefix works will quit scanning when there is no match. So:

```
\macro {1} {2}
\macro {1}
\macro
```

leads to:

```
|{1}|{2}|   |{1}|  |
```

When you're tolerant it can be that you still want to pick up some argument later on. This is why we have a continuation option.

```
\tolerant\def\foo       [#1]#*[#2]#:#3{!#1!#2!#3!}
```

```
\tolerant\def\oof[#1]#*[#2]#:(#3)#:#4{!#1!#2!#3!#4!}
\tolerant\def\ofo      [#1]#:(#2)#:#3{!#1!#2!#3!}
```

Hopefully the next example demonstrates how it works:

```
\foo{3} \foo[1]{3} \foo[1][2]{3}
\oof{4} \oof[1]{4} \oof[1][2]{4}
\oof[1][2](3){4} \oof[1](3){4} \oof(3){4}
\ofo{3} \ofo[1]{3}
\ofo[1](2){3} \ofo(2){3}
```

As you can see we can have multiple continuations using the `#:` directive:

!!!3! !1!!3! !1!2!3!
!!!!4! !1!!!4! !1!2!!4!
!1!2!3!4! !1!!3!4! !!!3!4!
!!!3! !1!!3!
!1!2!3! !!2!3!

The last specifier doesn't work well with the `\ifarguments` state because we no longer know what arguments were skipped. This is why we have another test for arguments. A zero value means that the next token is not a parameter reference, a value of one means that a parameter has been set and a value of two signals an empty parameter. So, it reports the state of the given parameter as a kind if `\ifcase`.

```
\def\foo#1#2{ [\ifparameter#1\or(ONE)\fi\ifparameter#2\or(TWO)\fi] }
```

Of course the test has to be followed by a valid parameter specifier:

```
\foo{1}{2} \foo{1}{} \foo{}{2} \foo{}{}
```

The previous code gives this:

[(ONE)(TWO)] [(ONE)] [(TWO)] []

A combination check `\ifparameters`, again a case, matches the first parameter that has a value set.

We could add plenty of specifiers but we need to keep in ind that we're not talking of an expression scanner. We need to keep performance in mind, so nesting and backtracking are no option. We also have a limited set of useable single characters, but here's one that uses a symbol that we had left:

```
\def\startfoo[#/]#/\stopfoo{ [#1](#2) }
```

The slash directive removes leading and trailing so called spacers as well as tokens that represent a paragraph end:

```
\startfoo [x ] x \stopfoo
\startfoo [ x ] x \stopfoo
\startfoo [ x] x \stopfoo
\startfoo [ x] \par x \par \par \stopfoo
```

So we get this:

[x](x) [x](x) [x](x) [x](x)

The next directive, the quitter `#;`, is demonstrated with an example. When no match has occurred, scanning picks up after this signal, otherwise we just quit.

```
\tolerant\def\foo[#1]#;(#2){/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par

\tolerant\def\foo[#1]#;#={/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par

\tolerant\def\foo[#1]#;#2{/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par

\tolerant\def\foo[#1]#;(#2)#;#={/#1/#2/#3/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

/1// /2// /3//
//1/ //2/ //3/
/1// /2// /3//
//1/ //2/ //3/
/1// /2// /3//
//1/ //2/ //3/
/1/// /2/// /3///
//1// //2// //3//

///1/ ///2/ ///3/

I have to admit that I don't really need it but it made some macros that I was redefining behave better, so there is some self-interest here. Anyway, I considered some other features, like picking up a detokenized argument but I don't expect that to be of much use. In the meantime we ran out of reasonable characters, but some day #? and #! might show up, or maybe I find a use for #< and #>. A summary of all this is given here:

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

The last two have not been discussed and were added later. The period directive gobbles space and par tokens and discards them in the process. The comma directive is like * but it pushes back a space when the matching quits.

```
\tolerant\def\foo[#1]#;(#2){/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par

\tolerant\def\foo[#1]#;#={/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par

\tolerant\def\foo[#1]#;#2{/#1/#2/}

\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par

\tolerant\def\foo[#1]#;(#2)#;#={/#1/#2/#3/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
/1//   /2//   /3//
//1/   //2/   //3/
/1//   /2//   /3//
//1/   //2/   //3/
/1//   /2//   /3//
//1/   //2/   //3/
/1///   /2///   /3///
//1//   //2//   //3//
///1/   ///2/   ///3/
```

Gobbling spaces versus pushing back is an interface design decision because it has to do with consistency.

3 Runaway arguments

There is a particular troublesome case left: a runaway argument. The solution is not pretty but it's the only way: we need to tell the parser that it can quit.

```
\tolerant\def\foo[#1=#2]%
  {\ifarguments 0\or 1\or 2\or 3\or 4\fi:\vl\type{#1}\vl\type{#2}\vl}
```

The outcome demonstrates that one still has to do some additional checking for sane results and there are alternative way to (ab)use this mechanism. It all boils down to a clever combination of delimiters and `\ignorearguments`.

```
\dontleavehmode \foo[a=1]
\dontleavehmode \foo[b=]
\dontleavehmode \foo[=]
\dontleavehmode \foo[x]\ignorearguments
```

All calls are accepted:

```
2:|a|1|
2:|b|
2:|
1:|x]|
```

Just in case you wonder about performance: don't expect miracles here. On the one hand there is some extra overhead in the engine (when defining macros as well as when

collecting arguments during a macro call) and maybe using these new features can sort of compensate that. As mentioned: the gain is mostly in cleaner macro code and less clutter in tracing. And I just want the ConTEXt code to look nice: that way users can look in the source to see what happens and not drown in all these show-off tricks, special characters like underscores, at signs, question marks and exclamation marks.

For the record: I normally run tests to see if there are performance side effects and as long as processing the test suite that has thousands of files of all kind doesn't take more time it's okay. Actually, there is a little gain in ConTEXt but that is to be expected, but I bet users won't notice it, because it's easily offset by some inefficient styling. Of course another gain of loosing some indirectness is that error messages point to the macro that the user called for and not to some follow up.

4 Introspection

A macro has a meaning. You can serialize that meaning as follows:

```
\tolerant\protected\def\foo#1[#2]#*[#3]%
  {(1=#1) (2=#3) (3=#3)}

\meaning\foo
```

The meaning of `\foo` comes out as:

> tolerant protected macro:#1[#2]#*[#3]->(1=#1) (2=#3) (3=#3)

When you load the module `system-tokens` you can also say:

```
\luatokentable\foo
```

This produces a table of tokens specifications:

tolerant protected macro:#1[#2]#*[#3]->(1=#1) (2=#3) (3=#3)

tolerant protected control sequence: foo					
371885	19	49	match		argument 1
578588	12	91	other char	[	U+0005B
579231	19	50	match		argument 2
566670	12	93	other char	]	U+0005D
578093	19	42	match		argument *
578696	12	91	other char	[	U+0005B
32433	19	51	match		argument 3

566713	12	93	other char	]	U+0005D
578957	20	0	end match		
579246	12	40	other char	(	U+00028
118762	12	49	other char	1	U+00031
578896	12	61	other char	=	U+0003D
579109	21	1	parameter reference		
579238	12	41	other char	)	U+00029
32429	10	32	spacer		
544370	12	40	other char	(	U+00028
578092	12	50	other char	2	U+00032
32434	12	61	other char	=	U+0003D
578728	21	3	parameter reference		
32430	12	41	other char	)	U+00029
32437	10	32	spacer		
579274	12	40	other char	(	U+00028
544974	12	51	other char	3	U+00033
279993	12	61	other char	=	U+0003D
544361	21	3	parameter reference		
579276	12	41	other char	)	U+00029

A token list is a linked list of tokens. The magic numbers in the first column are the token memory pointers. and because macros (and token lists) get recycled at some point the available tokens get scattered, which is reflected in the order of these numbers. Normally macros defined in the macro package are more sequential because they stay around from the start. The second and third row show the so called command code and the specifier. The command code groups primitives in categories, the specifier is an indicator of what specific action will follow, a register number a reference, etc. Users don't need to know these details. This macro is a special version of the online variant:

```
\showluatokens\foo
```

That one is always available and shows a similar list on the console. Again, users normally don't want to know such details.

5 nesting

You can nest macros, as in:

```
\def\foo#1#2{\def\oof##1{<#1>##1<#2>}}
```

At first sight the duplication of # looks strange but this is what happens. When TEX scans the definition of `\foo` it sees two arguments. Their specification ends up in the preamble that defines the matching. When the body is scanned, the `#1` and `#2` are turned into a parameter reference. In order to make nested macros with arguments possible a # followed by another # becomes just one #. Keep in mind that the definition of `\oof` is delayed till the macro `\foo` gets expanded. That definition is just stored and the only thing that get's replaced are the two references to a macro parameter

control sequence: foo						
556897	19	49	match		argument 1	
544819	19	50	match		argument 2	
312583	20	0	end match			
578624	124	1	def			def
544685	144	0	tolerant call			oof
578537	6	35	parameter			
544072	12	49	other char	1	U+00031	
578704	1	123	left brace			
388257	12	60	other char	<	U+0003C	
579163	21	1	parameter reference			
578831	12	62	other char	>	U+0003E	
578832	6	35	parameter			
579044	12	49	other char	1	U+00031	
544929	12	60	other char	<	U+0003C	
577237	21	2	parameter reference			
566350	12	62	other char	>	U+0003E	
312593	2	125	right brace			

Now, when we look at these details, it might become clear why for instance we have 'variable' names like `#4` and not `#whatever` (with or without hash). Macros are essentially token lists and token lists can be seen as a sequence of numbers. This is not that different from other programming environments. When you run into buzzwords like 'bytecode' and 'virtual machines' there is actually nothing special about it: some high level programming (using whatever concept, and in the case of TEX it's macros) eventually ends up as a sequence of instructions, say bytecodes. Then you need some machinery to run over that and act upon those numbers. It's something you arrive at naturally when you play with interpreting languages.[2]

[2] I actually did when I wrote an interpreter for some computer assisted learning system, think of a kind of interpreted Pascal, but later realized that it was a a bytecode plus virtual machine thing. I'd just applied what I learned when playing with eight bit processors that took bytes, and interpreted opcodes and such.

So, internally a #4 is just one token, a operator-operand combination where the operator is “grab a parameter” and the operand tells “where to store” it. Using names is of course an option but then one has to do more parsing and turn the name into a number[3], add additional checking in the macro body, figure out some way to retain the name for the purpose of reporting (which then uses more token memory or strings). It is simply not worth the trouble, let alone the fact that we loose performance, and when TEX showed up those things really mattered.

It is also important to realize that a # becomes either a preamble token (grab an argument) or a reference token (inject the passed tokens into a new input level). Therefore the duplication of hash tokens ## that you see in macro nested bodies also makes sense: it makes it possible for the parser to distinguish between levels. Take:

```
\def\foo#1{\def\oof##1{#1##1#1}}
```

Of course one can think of this:

```
\def\foo#fence{\def\oof#text{#fence#text#fence}}
```

But such names really have to be unique then! Actually ConTEXt does have an input method that supports such names, but discussing it here is a bit out of scope. Now, imagine that in the above case we use this:

```
\def\foo[#1][#2]{\def\oof##1{#1##1#2}}
```

If you're a bit familiar with the fact that TEX has a model of category codes you can imagine that a predictable “hash followed by a number” is way more robust than enforcing the user to ensure that catcodes of ‘names’ are in the right category (read: is a bracket part of the name or not). So, say that we go completely arbitrary names, we then suddenly needs some escaping, like:

```
\def\foo[#{left}][#{right}]{\def\oof#{text}{#{left}#{text}#{right}}}
```

And, if you ever looked into macro packages, you will notice that they differ in the way they assign category codes. Asking users to take that into account when defining macros makes not that much sense.

So, before one complains about TEX being obscure (the hash thing), think twice. Your demand for simplicity for your coding demand will make coding more cumbersome for

---

There's nothing spectacular about all this and I only realized decades later that the buzzwords describes old natural concepts.

[3] This is kind of what MetaPost does with parameters to macros. The side effect is that in reporting you get `text0`, `expr2` and such reported which doesn't make things more clear.

the complex cases that macro packages have to deal with. It's comparable using TEX for input or using (say) mark down. For simple documents the later is fine, but when things become complex, you end up with similar complexity (or even worse because you lost the enforced detailed structure). So, just accept the unavoidable: any language has its peculiar properties (and for sure I do know why I dislike some languages for it). The TEX system is not the only one where dollars, percent signs, ampersands and hashes have special meaning.

6 Prefixes

Traditional TEX has three prefixes that can be used with macros: `\global`, `\outer` and `\long`. The last two are no-op's in LuaMetaTEX and if you want to know what they do (did) you can look it up in the TEXbook. The ε-TEX extension gave us `\protected`.

In LuaMetaTEX we have `\global`, `\protected`, `\tolerant` and overload related prefixes like `\frozen`. A protected macro is one that doesn't expand in an expandable context, so for instance inside an `\edef`. You can force expansion by using the `\expand` primitive in front which is also something LuaMetaTEX.

Frozen macros cannot be redefined without some effort. This feature can to some extent be used to prevent a user from overloading, but it also makes it harder for the macro package itself to redefine on the fly. You can remove the lock with `\unletfrozen` and add a lock with `\letfrozen` so in the end users still have all the freedoms that TEX normally provides.

```
                \def\foo{foo} 1: \meaning\foo
          \frozen\def\foo{foo} 2: \meaning\foo
     \unletfrozen    \foo      3: \meaning\foo
\protected\frozen\def\foo{foo} 4: \meaning\foo
     \unletfrozen    \foo      5: \meaning\foo
```

1: macro:foo
2: macro:foo
3: macro:foo
4: protected macro:foo
5: protected macro:foo

This actually only works when you have set `\overloadmode` to a value that permits redefining a frozen macro, so for the purpose of this example we set it to zero.

A `\tolerant` macro is one that will quit scanning arguments when a delimiter cannot be matched. We saw examples of that in a previous section.

These prefixes can be chained (in arbitrary order):

```
\frozen\tolerant\protected\global\def\foo[#1]#*[#2]{...}
```

There is actually an additional prefix, `\immediate` but that one is there as signal for a macro that is defined in and handled by Lua. This prefix can then perform the same function as the one in traditional TEX, where it is used for backend related tasks like `\write`.

Now, the question is of course, to what extent will ConTEXt use these new features. One important argument in favor of using `\tolerant` is that it gives (hopefully) better error messages. It also needs less code due to lack of indirectness. Using `\frozen` adds some safeguards although in some places where ConTEXt itself overloads commands, we need to defrost. Adapting the code is a tedious process and it can introduce errors due to mistypings, although these can easily be fixed. So, it will be used but it will take a while to adapt the code base.

One problem with frozen macros is that they don't play nice with for instance `\futurelet`. Also, there are places in ConTEXt where we actually do redefine some core macro that we also want to protect from redefinition by a user. One can of course `\unletfrozen` such a command first but as a bonus we have a prefix `\overloaded` that can be used as prefix. So, one can easily redefine a frozen macro but it takes a little effort. After all, this feature is mainly meant to protect a user for side effects of definitions, and not as final blocker.[4]

A frozen macro can still be overloaded, so what if we want to prevent that? For this we have the `\permanent` prefix. Internally we also create primitives but we don't have a prefix for that. But we do have one for a very special case which we demonstrate with an example:

```
\def\FOO % trickery needed to pick up an optional argument
  {\noalign{\vskip10pt}}

\noaligned\protected\tolerant\def\OOF[#1]%
  {\noalign{\vskip\iftok{#1}\emptytoks10pt\else#1\fi}}

\starttabulate[|l|l|]
    \NC test \NC test \NC \NR
    \NC test \NC test \NC \NR
```

[4] As usual adding features like this takes some experimenting and we're now at the third variant of the implementation, so we're getting there. The fact that we can apply such features in large macro package like ConTEXt helps figuring out the needs and best approaches.

```
    \FOO
    \NC test \NC test \NC \NR
    \OOF[30pt]
    \NC test \NC test \NC \NR
    \OOF
    \NC test \NC test \NC \NR
\stoptabulate
```

When TEX scans input (from a file or token list) and starts an alignment, it will pick up rows. When a row is finished it will look ahead for a `\noalign` and it expands the next token. However, when that token is protected, the scanner will not see a `\noalign` in that macro so it will likely start complaining when that next macro does get expanded and produces a `\noalign` when a cell is built. The `\noaligned` prefix flags a macro as being one that will do some `\noalign` as part of its expansion. This trick permits clean macros that pick up arguments. Of course it can be done with traditional means but this whole exercise is about making the code look nice.

The table comes out as:

test test
test test

test test

test test

test test

One can check the flags with `\ifflags` which takes a control sequence and a number, where valid numbers are:

1	frozen	2	permanent	4	immutable	8	primitive
16	mutable	32	noaligned	64	instance		

The level of checking is controlled with the `\overloadmode` but I'm still not sure about how many levels we need there. A zero value disables checking, the values 1 and 3 give warnings and the values 2 and 4 trigger an error.

7 Arguments

The number of arguments that a macro takes is traditionally limited to nine (or ten if one takes the trailing # into account). That this is enough for most cases is demonstrated by

the fact that ConTEXt has only a handful of macros that use #9. The reason for this limitation is in part a side effect of the way the macro preamble and arguments are parsed. However, because in LuaMetaTEX we use a different implementation, it was not that hard to permit a few more arguments, which is why we support upto 15 arguments, as in:

```
\def\foo#1#2#3#4#5#6#7#8#9#A#B#C#D#E#F{...}
```

We can support the whole alphabet without much trouble but somehow sticking to the hexadecimal numbers makes sense. It is unlikely that the core of ConTEXt will use this option but sometimes at the user level it can be handy. The penalty in terms of performance can be neglected.

```
\tolerant\def\foo#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=%
  {(#1)(#2)(#3)(#4)(#5)(#6)(#7)(#8)(#9)(#A)(#B)(#C)(#D)(#E)(#F)}

\foo{1}{2}
```

In the previous example we have 15 optional arguments where braces are mandate (otherwise we the scanner happily scoops up what follows which for sure gives some error).

8 Constants

The LuaMetaTEX engine has lots of efficiency tricks in the macro parsing and expansion code that makes it not only fast but also let is use less memory. However, every time that the body of a macro is to be injected the expansion machinery kicks in. This often means that a copy is made (pushed in the input and used afterwards). There are however cases where the body is just a list of character tokens (with category letter or other) and no expansion run over the list is needed.

It is tempting to introduce a string data type that just stores strings and although that might happen at some point it has the disadvantage that one need to tokenize that string in order to be able to use it, which then defeats the gain. An alternative has been found in constant macros, that is: a macro without parameters and a body that is considered to be expanded and never freed by redefinition. There are two variants:

```
\cdef      \foo          {whatever}
\cdefcsname foo\endcsname{whatever}
```

These are actually just equivalents to

```
\edef      \foo          {whatever}
```

```
\edefcsname foo\endcsname{whatever}
```

just to make sure that the body gets expanded at definition time but they are also marked as being constant which in some cases might give some gain, for instance when used in csname construction. The gain is less then one expects although there are a few cases in ConTEXt where extreme usage of parameters benefits from it. Users are unlikely to use these two primitives.

Another example of a constant usage is this:

```
\lettonothing\foo
```

which gives `\foo` an empty body. That one is used in the core, if only because it gives a bit smaller code. Performance is no that different from

```
\let\foo\empty
```

but it saves one token (8 bytes) when used in a macro. The assignment itself is not that different because `\foo` is made an alias to `\empty` which in turn only needs incrementing a reference counter.

9 Passing parameters

When you define a macro, the #1 and more parameters are embedded as a reference to a parameter that is passed. When we have four parameters, the parameter stack has four entries and when an entry is eventually accessed a new input level is pushed and tokens are fetched from that list. This has some side effects when we check a parameter. This can happen multiple times, depending on how often we access a parameter. Take the following:

```
\def\oof#1{#1}

\tolerant\def\foo[#1]#*[#2]%
  {1:\ifparameter#1\or Y\else N\fi\quad
   2:\ifparameter#2\or Y\else N\fi\quad
   \oof{3:\ifparameter #1\or Y\else N\fi\quad
        4:\ifparameter #2\or Y\else N\fi\quad}%
   \par}

\foo \foo[] \foo[][] \foo[A] \foo[A][B]
```

This gives:

```
1:N  2:N  3:N  4:N
1:N  2:N  3:N  4:N
1:N  2:N  3:N  4:N
1:Y  2:N  3:Y  4:N
1:Y  2:Y  3:Y  4:Y
```

as you probably expect. However the first two checks are different from the embedded checks because they can check against the parameter reference. When we expand `\oof` its argument gets passed to the macro as a list and when the scanner collects the next token it will then push the parameter content on the input stack. So, then, instead of a reference we get the referenced parameter list. Internally that means that in 3 and 4 we check for a token and not for the length of the list (as in case 1 & 2). This means that

```
\iftok{#1}\emptytoks Y\else N\fi
\ifparameter#1\or    Y\else N\fi
```

are different. In the first case we have a proper token list and nested conditionals in that list are okay. In the second case we just look ahead to see if there is an `\or`, `\else` or other condition related command and if so we decide that there is no parameter. So, if `\ifparameter` is a suitable check for empty depends on the need for expansion.

When you define macros that themselves call macros that should operate on the arguments of its parent you can easily pass these:

```
\def\foo#1#2%
  {\oof{#1}{#2}{P}%
   \oof{#1}{#2}{Q}%
   \oof{#1}{#2}{R}}

\def\oof#1#2#3%
  {[#1][#1]%
   #3%
   [#2][#2]}
```

Here the nested call to `\oof` involved three passed parameters. You can avoid that as follows:

```
\def\foo#1#2%
  {\def\MyIndexOne{#1}%
   \def\MyIndexTwo{#2}%
   \oof{P}\oof{Q}\oof{R}}
```

```
\def\oof#1%
  {(\MyIndexOne)(\MyIndexOne)%
   #1%
   (\MyIndexTwo)(\MyIndexTwo)}
```

You can also do this:

```
\def\foo#1#2%
  {\def\oof##1%
     {/#1/#2/%
     ##1%
     /#1//#2/}%
   \oof{P}\oof{Q}\oof{R}}
```

These parameters indicated by # in the macro body are in fact references. When we call for instance `\foo{1}{2}` the two parameters get pushed on a parameter stack and the embodied references point to these stack entries. By the time that body gets expanded TEX bumps the input level and pushes the parameter list onto the input stack. It then continues expansion. The parameter is not copied, because it can't be changed anyway. The only penalty in terms of performance and memory usage is the pushing and popping of the input. So how does that work out for these three cases?

When in the first case the `\oof{#1}{#2}{P}` is seen, TEX starts expanding the `\oof` macro. That one expects three arguments. The #1 reference is seen and in this case a copy of that parameter is passed. The same is true for the other two. Then, inside `\oof` expansion happens on the parameters on the stack and no copies have to be made there.

The second case defines two macros so again two copies are made that make the bodies of these macros. This comes at the cost of some runtime and memory. However, this time with `\oof{P}` only one argument gets passed and instead expansion of the macros happen in there.

Normally macro arguments are not that large but there can be situations where we really want to avoid useless copying. This not only saves memory but also can give a bit better performance. In the examples above the second variant is some 10%faster than the first one. We can gain another 10%with the following trick:

```
\def\foo#1#2%
  {\parameterdef\MyIndexOne\plusone % 1
   \parameterdef\MyIndexTwo\plustwo % 2
   \oof{P}\oof{Q}\oof{R}\norelax}
```

```
\def\oof#1%
  {<\MyIndexOne><\MyIndexOne>%
   #1%
   <\MyIndexTwo><\MyIndexTwo>}
```

Here we define an explicit parameter reference that we access later on. There is the overhead of a definition but it can be neglected. We use that reference (abstraction) in `\oof`. Actually you can use that reference in any call down the chain.

When applied to `\foo{1}{2}` the four variants above give us:

```
[1][1]P[2][2][1][1]Q[2][2][1][1]R[2][2]
(1)(1)P(2)(2)(1)(1)Q(2)(2)(1)(1)R(2)(2)
/1/2/P/1//2//1/2/Q/1//2//1/2/R/1//2/
<1><1>P<2><2><1><1>Q<2><2><1><1>R<2><2>
```

Before we had `parameterdef` we had this:

```
\def\foo#1#2%
  {\integerdef\MyIndexOne\parameterindex\plusone % 1
   \integerdef\MyIndexTwo\paramet\plustwo % 2
   \oof{P}\oof{Q}\oof{R}\norelax}

\def\oof#1%
  {<\expandparameter\MyIndexOne><\expandparameter\MyIndexOne>%
   #1%
   <\expandparameter\MyIndexTwo><\expandparameter\MyIndexTwo>}
```

It involves more tokens, is a bit less abstract, but as it is a cheap extension we kept it. It actually demonstrates that one can access parameters in the stack by index, but it one then needs to keep track of where access takes place. In principle one can debug the call chain this way.

To come back to performance and memory usage, when the arguments become larger the fourth variant with the `\parameterdef` quickly gains over the others. But it only shows in exceptional usage. This mechanism is more about abstraction: it permits us to efficiently turn arguments into local variables without the overhead involved in creating macros.

10 Nesting

We also have a few preamble features that relate to nesting. Although we can do without (as shown for years in LMTX) they do have some benefits. They are discussed as group

here and because they are only useful for low level programming we stick to simple examples. The #L and #R use the following token as delimiters. Here we use [and] but they can be a \cs as well. Nested delimiters are handled well.

The #S grabs the argument till the next final square bracket] but in the process will grab nested with it sees a [. The #P does the same for parentheses and #X for angle brackets. In the next examples the #* just gobbles optional spaces but we've seen that one already.

The #G argument just registers the next token as delimiter but it will grab multiple of them. The #M gobbles more: in addition to the delimiter spaces are gobbled.

```
\tolerant\def\fooA               [#1]{(#1)}
\tolerant\def\fooB          [#L[#R]#1{(#1)}
\tolerant\def\fooC               #S#1{(#1)}
\tolerant\def\fooE              #S#1,{(#1)}
\tolerant\def\fooF         #S#1#*#S#2{(#1/#2)}
\tolerant\def\fooG [#1]#S[#2]#*#S[#3]{(#1/#2/#3)}
\tolerant\def\fooH [#1][#S#2]#*[#S#3]{(#1/#2/#3)}
\tolerant\def\fooI           #1=#2#G,{(#1=#2)}
\tolerant\def\fooJ           #1=#2#M,{(#1=#2)}
```

```
\fooA[x]           (x)           (x)
\fooB[x]           (x)           (x)
\fooC[1[2]3[4]5]   ([1[2]3[4]5]) (1[2]3[4]5)
\fooE X[,]X,       (X[,]X)       (X[,]X)
\fooF[A] [B]       ([A]/[B])     (A/B)
\fooF[] []         ([]/[])       (/)
\fooG[a][b][c]     (a/b/c)       (a/b/c)
\fooG[a][b]        (a/b/)        (a/b/)
\fooG[a]           (a//)         (a//)
\fooG[a][x[x]x][c] (a/x[x]x/c)   (a/x[x]x/c)
\fooH[a][x[x]x][c] (a/x[x]x/c)   (a/x[x]x/c)
\fooI X=X,,        (X=X)         (X=X)
\fooJ X=X, , ,     (X=X)         (X=X)
```

These features make it possible to support nested setups more efficiently and also makes it possible to accept values that contain balanced brackets in setup commands without additional overhead. Although it has never been an issue to let users specify:

```
\defineoverlay[whatever][{some \command[withparameters] here}]
```

```
\setupfoo[before={\blank[big]}]
```

it might be less confusing to permit:

```
\defineoverlay[whatever][some \command[withparameters] here]

\setupfoo[before=\blank[big]]
```

as well, if only because occasionally users get hit by this.

11 Duplicate hashes

In TEX every character has a so called category code. Most characters are classified as 'letter' (they make up words) or as 'other'. In Unicode we distinguish symbols, punctuation, and more, but in TEX these are all of category 'other'. In math however we can classify them differently but in this perspective we ignore that. The backslash has category 'escape' and it starts a control sequence. The curly braces are (internally) of category 'left brace' and 'right brace' aka 'begin group' and 'end group' but, no matter what they are called, they begin and end something: a group, argument, token list, box, etc. Any character can have those categories. Although it would loook strange to a TEX user, this can be made valid:

```
!protected !gdef !weird¶1
B
    something: ¶1
E
!weird BhereE
```

In such a setup spaces can be of category 'invisible'. The paragraph symbol takes the place of the hash as parameter identifier. The next code shows how this is done. Here we wrap all in a macro so that we don't get catcode interference in the document source.

```
\def\NotSoTeX
  {\begingroup
   \catcode `B \begingroupcatcode
   \catcode `E \endgroupcatcode
   \catcode `¶ \parametercatcode
   \catcode `! \escapecatcode
   \catcode 32 \ignorecatcode
   \catcode 13 \ignorecatcode
   % this buffer has a definition:
   \getbuffer
```

```
   % which is now known globally
   \endgroup}
\NotSoTeX
\weird{there}
```

This results in:

something:here
something:there

In the first line the `!`, `B` and `E` are used as escape and argument delimiters, in the second one we use the normal characters. When we show the `\meaningasis` we get:

```
\protected \def \weird #1{something:#1}
```

or in more detail:

```
protected control sequence: weird
556878  19   49  match                   argument 1
580381  20    0  end match
580393  11  115  letter                s  U+00073
580032  11  111  letter                o  U+0006F
566673  11  109  letter                m  U+0006D
580375  11  101  letter                e  U+00065
578812  11  116  letter                t  U+00074
580002  11  104  letter                h  U+00068
580004  11  105  letter                i  U+00069
578868  11  110  letter                n  U+0006E
566719  11  103  letter                g  U+00067
577164  12   58  other char           :  U+0003A
578086  21    1  parameter reference
```

So, no matter how we set up the system, in the end we get some generic representation. When we see `#1` in 'print' it can be either two tokens, `#` (catcode parameter) followed by `1` with catcode other, or one token referring to parameter `1` where the character `1` is the opcode of an internal 'reference command'. In order to distinguish a reference from the two token case, parameter hash tokens get shown as doubles.

```
\def\test #1{x#1x##1x####1x}
\def\tset ¶1{x¶1x¶¶1x¶¶¶¶1x}
```

And with `\meaning` we get, consistent with the input:

```
macro:#1->x#1x##1x####1x
macro:#1->x#1x¶¶1x¶¶¶¶1x
```

These are equivalent, apart from the parameter character in the body of the definition:

control sequence: test					
578814	19	49	match		argument 1
566675	20	0	end match		
578079	11	120	letter	x	U+00078
580316	21	1	parameter reference		
416903	11	120	letter	x	U+00078
545011	6	35	parameter		
579106	12	49	other char	1	U+00031
580299	11	120	letter	x	U+00078
580301	6	35	parameter		
580258	6	35	parameter		
580117	12	49	other char	1	U+00031
578451	11	120	letter	x	U+00078

control sequence: tset					
579126	19	49	match		argument 1
573330	20	0	end match		
579119	11	120	letter	x	U+00078
580500	21	1	parameter reference		
580282	11	120	letter	x	U+00078
579026	6	182	parameter		
566730	12	49	other char	1	U+00031
578871	11	120	letter	x	U+00078
580279	6	182	parameter		
566696	6	182	parameter		
577236	12	49	other char	1	U+00031
577170	11	120	letter	x	U+00078

Watch how every ‘parameter’ is just a character with the Unicode index of the used input character as property. Let us summarize the process. When a single parameter character is seen in the input, the next characer determines how it will be interpreted.

Duplicate hashes

If there is a digit then it becomes a reference to a parameter in the preamble, and when followed by another parameter character it will be appended to the body of the macro and that second one is dropped. So, two parameter characters become one, and four become two. One parameter character becomes a reference and from that you can guess what three in a row become. However, when TEX is showing the macro definition (using `meaning`) the hashes get duplicated in order to distinguish parameter references from parameter characters that were kept (e.g. for nested definitions). One can make an argument for `\parameterchar` as we also have `\escapechar` but by now this convention is settled and it doesn't look that bad anyway.

We now come to the more tricky part with respect to the doubling of hashes. When TEX was written its application landscape looked a bit different. For instance, fonts were limited and therefore it was natural to access special characters by name. Using `\#` to get a hash in the text was not that problematic, if one needed that character at all. The same can be said for the braces, backslash and even the dollar (after all TEX is free software).

But what if we have more visualization and/or serialization than meanings and tracing? When we opened op the internals in LuaTEX and even more in LuaMetaTEX the duplicating of hashes became a bit of a problem. There we don't need to distinguish between a parameter reference and a parameter character because by that time these references are resolved. All hashes that we encounter are just that: hashes. And this is why in LuaMetaTEX we disable the duplication for those cases where it serves no purpose.

When the engine scans a macro definition it starts with pickin g up the name of the macro. Then it starts scanning the preamble upto the left brace. In the preamble of a macro the scanner converts hashes followed by another token into single match token. Then when the macro body is scanned single hashes followed by a number become a reference, while double hashes become one hash and get interpreted at expansion time (possibly triggering an error when not followed by a valid specifier like a number). In traditional TEX we basically had this:

```
\def\test#1{#1}
\def\test#1{##}
\def\test#1{#X}
\def\test#1{##1}
```

There can be a traling # in the preamble for special purposes but we forget about that now. The first definition is valid, the second definition is invalid when the macro is expanded and the third definition triggers an error at definition time. The last definition will again trigger an error at expansion time.

However, in LuaMetaTEX we have an extended preamble where the following preamble parameters are handled (some only in tolerant mode):

`#n`	parameter	index 1 upto E
`#0`	throw away parameter	increment index
`#-`	ignore parameter	keep index
`#*`	gobble white space	
`#+`	keep (honor) the braces	
`#.`	ignore pars and spaces	
`#,`	push back space when no match	
`#/`	remove leading and trailing spaces and pars	
`#=`	braces are mandate	
`#^`	keep leading spaces	
`#_`	braces are mandate and kept (obey)	
`#@`	par delimiter	only for internal usage
`#:`	pick up scanning here	
`#;`	quit scanning	
`#L`	left delimiter token	followed by token
`#R`	right delimiter token	followed by token
`#G`	gobble token	followed by token
`#M`	gobble token and spaces	followed by token
`#S`	nest square brackets	only inner pairs
`#X`	nest angle brackets	only inner pairs
`#P`	nest parentheses	only inner pairs

As mentioned these will become so called match tokens and only when we show the meaning the hash will show up again.

```
\def\test[#1]#*[*S#2]{.#1.#2.}
```

```
control sequence: test
580400  12  91  other char          [  U+0005B
32395   19  49  match                  argument 1
579215  12  93  other char          ]  U+0005D
578043  19  42  match                  argument *
```

```
577166  12  91  other char          [  U+0005B
166772  12  42  other char          *  U+0002A
578745  11  83  letter              S  U+00053
578104  19  50  match                 argument 2
566707  12  93  other char          ]  U+0005D
580251  20   0  end match
-------------------------------------------------
118735  12  46  other char          .  U+0002E
578567  21   1  parameter reference
579225  12  46  other char          .  U+0002E
579200  21   2  parameter reference
580482  12  46  other char          .  U+0002E
-------------------------------------------------
```

This means that in the body of a macro you will not see #* show up. It is just a directive that tells the macro parser that spaces are to be skipped. The #S directive makes the parser for the second parameter handle nested square bracket. The only hash that we can see end up in the body is the one that we entered as double hash (then turned single) followed by (in traditional terms) a number that when all gets parsed with then become a reference: the sequence ##1 internally is #1 and becomes 'reference to parameter 1' assuming that we define a macro in that body. If no number is there, an error is issued. This opens up the possibility to add more variants because it will only break compatibility with respect to what is seen as error. As with the preamble extensions, old documents that have them would have crashed before they became available.

So, this means that in the body, and actually anywhere in the document apart from preambles, we now support the following general parameter specifiers. Keep in mind that they expand in an expansion context which can be tricky when they overlap with preamble entries, like for instance #R in such an expansion. Future extensions can add more so *any* hashed shortcut is sensitive for that.

#I	current iterator	\currentloopiterator
#P	parent iterator	\previousloopiterator 1
#G	grandparent iterator	\previousloopiterator 2
#H	hash escape	#
#S	space escape	
#T	tab escape	\t
#L	newline escape	\n
#R	return escape	\r
#X	backslash escape	\

#N	nbsp	U+00A0 (under consideration)
#Z	zws	U+200B (under consideration)

Some will now argue that we already have ^^ escapes in TEX and ^^^^ and ^^^^^^ in LuaTEX and that is true. However, these can be disabled, and in ConTEXt they are, where we instead enable the prescript, postscript, and index features in mathmode and there type ^ and _ are used. Even more: in ConTEXt we just let ^, _ and & be what they are. Occasionally I consider $ to be just that but as I don't have dollars I will happily leave that for inline math. When users are not defining macros or are using the alternative definitions we can consider making the # a hash. An excellent discussion of how TEX reads it's input and changes state accordingly can be found in Victor Eijkhouts “TEX By Topic”, section 2.6: when ^^ is followed by a character with $\nu < 128$ the interpreter will inject a character with code $\nu - 64$. When followed by two (!) lowercase hexadecimal characters, the corresponding character will be injected. Anyway, it not only looks kind of ugly, it also is somewhat weird because what follows is interpreted mixed way. The substitution happens early on (which is okay). But, how about the output? Traditional TEX serializes special characters with a similar syntax but that has become optional when eight bit mode was added to the engines, it is configurable in LuaTEX and has been dropped in LuaMetaTEX: we operate in a utf universum.

11 Colofon

Author	Hans Hagen
ConTEXt	2023.08.24 19:58
LuaMetaTEX	211.0
Support	www.pragma-ade.com
	contextgarden.net