# low level TEX

## expansion

# Contents



## 1 Preamble

This short manual demonstrates a couple of properties of the macro language. It is not an in-depth philosophical expose about macro languages, tokens, expansion and such that some TEXies like. I prefer to stick to the practical aspects. Occasionally it will be technical but you can just skip those paragraphs (or later return to them) when you can't follow the explanation. It's often not that relevant. I won't talk in terms of mouth, stomach and gut the way the TEXbook does and although there is no way to avoid the word 'token' I will do my best to not complicate matters by too much token speak. Examples show best what we mean.

## 2 TEX primitives

The TEX language provides quite some commands and those built in are called primitives. User defined commands are called macros. A macro is a shortcut to a list of primitives and/or macro calls. All can be mixed with characters that are to be typeset somehow.

```
\def\MyMacro{b}

a\MyMacro c
```

When TEX reads this input the `a` gets turned into a glyph node with a reference to the current font set and the character `a`. Then the parser sees a macro call, and it will enter another input level where it expands this macro. In this case it sees just an `b` and it will give this the same treatment as the `a`. The macro ends, the input level decrements and the `c` gets its treatment.

Before we move on to more examples and differences between engines, it is good to stress that `\MyMacro` is not a primitive command: we made our command here. The `b` actually can be seen as a sort of primitive because in this macro it gets stored as so

called token with a primitive property. That primitive property can later on be used to determine what to do. More explicit examples of primitives are `\hbox`, `\advance` and `\relax`. It will be clear that ConTEXt extends the repertoire of primitive commands with a lot of macro commands. When we typeset a source using module `m-scite` the primitives come out dark blue.

The amount of primitives differs per engine. It all starts with TEX as written by Don Knuth. Later ε-TEX added some more primitives and these became official extensions adopted by other variants of TEX. The pdfTEX engine added quite some and as follow up on that LuaTEX added more but didn't add all of pdfTEX. A few new primitives came from Omega (Aleph). The LuaMetaTEX engine drops a set of primitives that comes with LuaTEX and adds plenty new ones. The nature of this engine (no backend and less frontend) makes that we need to implement some primitives as macros. But the basic set is what good old TEX comes with.

Internally these so called primitives are grouped in categories that relate to their nature. They can be directly expanded (a way of saying that they get immediately interpreted) or delayed (maybe stored for later usage). They can involve definitions, calculations, setting properties and values or they can result in some typesetting. This is what makes TEX confusing to new users: it is a macro programming language, an interpreter but at the same time an executor of typesetting instructions.

A group of primitives is internally identified as a command (they have a `cmd` code) and the sub commands are flagged by their `chr` code. This sounds confusing but just thing of the fact that most of what we input are characters and therefore they make up most sub commands. For instance the 'letter `cmd`' is used for characters that are seen as letters that can be used in the name of user commands, can be typeset, are valid for hyphenation etc. The letter related `cmd` can have many `chr` codes (all of Unicode). I'd like to remark that the grouping is to a large extend functional, so sometimes primitives that you expect to be similar in nature are in different groups. This has to do with the fact that TEX needs to be a able to determine efficiently if a primitive is operating (or forbidden) in horizontal, vertical and/or math mode.

There are more than 150 internal `cmd` groups. if we forget about the mentioned character related ones, some, have only a few sub commands (`chr`) and others many more (just consider all the OpenType math spacing related parameters). A handful of these commands deal with what we call macros: user defined combinations of primitives and other macros, consider them little programs. The `\MyMacro` example above is an example. There are differences between engines. In standard TEX there are `\outer` and `\long` commands, and most engines have these. However, in LuaMetaTEX the later to be discussed `\protected` macros have their own specific 'call `cmd`'. Users don't need to bother about this.

So, when from now on we talk about primitives, we mean the built in, hard coded commands, and when we talk about macros we mean user commands. Although internally there are less `cmd` categories than primitives, from the perspective of the user they are all unique. Users won't consult the source anyway but when they do they are warned. Also, when in LuaMetaTEX you use the low level interfacing to TEX you have to figure out these subtle aspects because there this grouping does matter.

Before we continue I want to make clear that expansion (as discussed in this document) can refer to a macro being expanded (read: its meaning gets injected into the input, so the engine kind of sidetracks from what is was doing) but also to direct consequences of running into a primitive. However, users only need to consider expansion in the perspective of macros. If a user has `\advance` in the input it immediately gets done. But when it's part of a macro definition it only is executed when the macro expands. A good check in (traditional) TEX is to compare what happens in `\def` and `\edef` which is why we will use these two in the upcoming examples. You put something in a macro and then check what `\meaning` or `\show` reports.

Now back to user defined macros. A macro can contain references to macros so in practice the input can go several levels up and some applications push back a lot so this is why your TEX input stack can be configured to be huge.

```
\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}

a\MyMacroA b
```

When `\MyMacroB` is defined, its body gets three so called tokens: the character token `1` with property 'other', a token that is a reference to the macro `\MyMacroB`, and a character token `2`, also with property 'other' The meaning of `\MyMacroA` is five tokens: a reference to a space token, then three character tokens with property 'letter', and finally a space token.

```
\def \MyMacroA{ and }
\edef\MyMacroB{1\MyMacroA 2}

a\MyMacroA b
```

In the second definition an `\edef` is used, where the `e` indicates expansion. This time the meaning gets expanded immediately. So we get effectively the same as in:

```
\def\MyMacroB{1 and 2}
```

Characters are easy: they just expand to themselves or trigger adding a glyph node,

but not all primitives expand to their meaning or effect.

```
\def\MyMacroA{\scratchcounter = 1 }
\def\MyMacroB{\advance\scratchcounter by 1}
\def\MyMacroC{\the\scratchcounter}

\MyMacroA a
\MyMacroB b
\MyMacroB c
\MyMacroB d
\MyMacroC
```

a b c d 4

```
macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:\the \scratchcounter
```

Let's assume that `\scratchcounter` is zero to start with and use `\edef`'s:

```
\edef\MyMacroA{\scratchcounter = 1 }
\edef\MyMacroB{\advance\scratchcounter by 1}
\edef\MyMacroC{\the\scratchcounter}

\MyMacroA a
\MyMacroB b
\MyMacroB c
\MyMacroB d
\MyMacroC
```

a b c d 0

```
macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:0
```

So, this time the third macro has its meaning frozen, but we can prevent this by applying a `\noexpand` when we do this:

```
\edef\MyMacroA{\scratchcounter = 1 }
\edef\MyMacroB{\advance\scratchcounter by 1}
\edef\MyMacroC{\noexpand\the\scratchcounter}
```

```
\MyMacroA a
\MyMacroB b
\MyMacroB c
\MyMacroB d
\MyMacroC
```

a b c d 4

```
macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:\the \scratchcounter
```

Of course this is a rather useless example but it serves its purpose: you'd better be aware what gets expanded immediately in an `\edef`. In most cases you only need to worry about `\the` and embedded macros (and then of course their meanings).

You can also store tokens in a so-called token register. Here we use a predefined scratch register:

```
\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks {\MyMacroA}
```

The content of `\scratchtoks` is: "`\MyMacroA`", so no expansion has happened here.

```
\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroA}
```

Now the content of `\scratchtoks` is: " `and` ", so this time expansion has happened.

```
\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroB}
```

Indeed the macro gets expanded but only one level: "`1\MyMacroA 2`". Compare this with:

```
\def\MyMacroA{ and }
\edef\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroB}
```

The trick is to expand in two steps with an intermediate `\edef`: "`1 and 2`". Later we will

see that other engines provide some more expansion tricks. The only way to get some grip on expansion is to just play with it.

The `\expandafter` primitive expands the token (which can be a macro) standing after the next next one and then injects its meaning into the stream. So:

```
\expandafter \MyMacroA \MyMacroB
```

works okay. In a normal document you will never need this kind of hackery: it only happens in a bit more complex macros. Here is an example:

```
\scratchcounter 1
\bgroup
\advance\scratchcounter 1
\egroup
\the\scratchcounter

\scratchcounter 1
\bgroup
\advance\scratchcounter 1
\expandafter
\egroup
\the\scratchcounter
```

The first one gives 1, while the second gives 2.

## 3 ε-TEX primitives

In this engine a couple of extensions were added and later on pdfTEX added some more. We only discuss a few that relate to expansion. There is however a pitfall here. Before ε-TEX showed up, ConTEXt already had a few mechanism that also related to expansion and it used some names for macros that clash with those in ε-TEX. This is why we will use the `\normal` prefix here to indicate the primitive.[1].

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\edef\MyMacroABC{\MyMacroA\MyMacroB\MyMacroC}
```

These macros have the following meanings:

[1] In the meantime we no longer have a low level `\protected` macro so one can use the primitive

```
macro:a
macro:b
protected  macro:c
macro:ab\MyMacroC
```

In ConTEXt you will use the `\unexpanded` prefix instead, because that one did something similar in older versions of ConTEXt. As we were early adopters of ε-TEX, this later became a synonym to the ε-TEX primitive.

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\normalexpanded{\scratchtoks{\MyMacroA\MyMacroB\MyMacroC}}
```

Here the wrapper around the token register assignment will expand the three macros, unless they are protected, so its content becomes "`ab\MyMacroC`". This saves either a lot of more complex `\expandafter` usage or the need to use an intermediate `\edef`. In ConTEXt the `\expanded` macro does something simpler but it doesn't expand the first token as this is meant as a wrapper around a command, like:

```
\expanded{\chapter{....}} % a ConTeXt command
```

where we do want to expand the title but not the `\chapter` command (not that this would happen actually because `\chapter` is a protected command.)

The counterpart of `\normalexpanded` is `\normalunexpanded`, as in:

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\normalexpanded {\scratchtoks
    {\MyMacroA\normalunexpanded {\MyMacroB}\MyMacroC}}
```

The register now holds "`a\MyMacroB \MyMacroC`": three tokens, one character token and two macro references.

Tokens can represent characters, primitives, macros or be special entities like starting math mode, beginning a group, assigning a dimension to a register, etc. Although you can never really get back to the original input, you can come pretty close, with:

```
\detokenize{this can $ be anything \bgroup}
```

This (when typeset monospaced) is: `this can $ be anything \bgroup`. The detok-

enizer is like \string applied to each token in its argument. Compare this to:

```
\normalexpanded {
    \normaldetokenize{10pt}
}
```

We get four tokens: 10pt.

```
\normalexpanded {
    \string 1\string 0\string p\string t
}
```

So that was the same operation: 10pt, but in both cases there is a subtle thing going on: characters have a catcode which distinguishes them. The parser needs to know what makes up a command name and normally that's only letters. The next snippet shows these catcodes:

```
\normalexpanded {
    \noexpand\the\catcode`\string 1 \noexpand\enspace
    \noexpand\the\catcode`\string 0 \noexpand\enspace
    \noexpand\the\catcode`\string p \noexpand\enspace
    \noexpand\the\catcode`\string t \noexpand
}
```

The result is "12 12 11 11": two characters are marked as 'letter' and two fall in the 'other' category.

# 4 LuaTEX primitives

This engine adds a little to the expansion repertoire. First of all it offers a way to extend token lists registers:

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{b}
\scratchtoks{\MyMacroA\MyMacroB}
```

The result is: "\MyMacroA \MyMacroB".

```
\toksapp\scratchtoks{\MyMacroA\MyMacroB}
```

We're now at: "\MyMacroA \MyMacroB \MyMacroA \MyMacroB \MyMacroA \MyMacroB".

```
\etoksapp\scratchtoks{\MyMacroA\space\MyMacroB\space\MyMacroC}
```

The register has this content: "\MyMacroA \MyMacroB \MyMacroA \MyMacroB a b \MyMacroC a b \MyMacroC", so the additional context got expanded in the process, except of course the protected macro \MyMacroC.

There is a bunch of these combiners: \toksapp and \tokspre for local appending and prepending, with global companions: \gtoksapp and \gtokspre, as well as expanding variant: \etoksapp, \etokspre, \xtoksapp and \xtokspre.

These are not beforehand more efficient that using intermediate expanded macros or token lists, simply because in the process TEX has to create tokens lists too, but sometimes they're just more convenient to use. In ConTEXt we actually do benefit from these.

## 5 LuaMetaTEX primitives

We already saw that macro's can be defined protected which means that

```
           \def\TestA{A}
\protected \def\TestB{B}
          \edef\TestC{\TestA\TestB}
```

gives this:

```
\TestC : A\TestB
```

One way to get \TestB expanded it to prefix it with \expand:

```
           \def\TestA{A}
\protected \def\TestB{B}
          \edef\TestC{\TestA\TestB}
          \edef\TestD{\TestA\expand\TestB}
```

We now get:

```
\TestC : A\TestB
\TestD : AB
```

There are however cases where one wishes this to happen automatically, but that will also make protected macros expand which will create havoc, like switching fonts.

```
               \def\TestA{A}
\protected     \def\TestB{B}
```

```
\semiprotected \def\TestC{C}
               \edef\TestD{\TestA\TestB\TestC}
               \edef\TestE{\normalexpanded{\TestA\TestB\TestC}}
               \edef\TestF{\semiexpanded  {\TestA\TestB\TestC}}
```

This time `\TestC` looses its protection:

```
\TestA : A
\TestB : B
\TestC : C
\TestD : A\TestB \TestC
\TestE : A\TestB \TestC
\TestF : A\TestB C
```

Actually adding `\fullyexpanded` would be trivial but it makes not much sense to add the overhead (at least not now). This feature is experimental anyway so it might go away when I see no real advantage from it.

When you store something in a macro or token register you always need to keep an eye on category codes. A dollar in the input is normally treated as math shift, a hash indicates a macro parameter or preamble entry. Characters like 'A' are letters but '[' and ']' are tagged as 'other'. The TEX scanner acts according to these codes. If you ever find yourself in a situation that changing catcodes is no option or cumbersome, you can do this:

```
\edef\TestOA{\expandtoken\othercatcode `A}
\edef\TestLA{\expandtoken\lettercatcode`A}
```

In both cases the meaning is `A` but in the first case it's not a letter but a character flagged as 'other'.

A whole new category of commands has to do with so called local control. When TEX scans and interprets the input, a process takes place that is called tokenizing: (sequences of) characters get a symbolic representation and travel through the system as tokens. Often they immediately get interpreted and are then discarded. But when for instance you define a macro they end up as a linked list of tokens in the macro body. We already saw that expansion plays a role. In most cases, unless TEX is collecting tokens, the main action is dealt with in the so-called main loop. Something gets picked up from the input but can also be pushed back, for instance because of some lookahead that didn't result in an action. Quite some time is spent in pushing and popping from the so-called input stack.

When we are in Lua, we can pipe back into the engine but all is collected till we're

back in TEX where the collected result is pushed into the input. Because TEX is a mix of programming and action there basically is only that main loop. There is no real way to start a sub run in Lua and do all kind of things independent of the current one. This makes sense when you consider the mix: it would get too confusing.

However, in LuaTEX and even better in LuaMetaTEX, we can enter a sort of local state and this is called 'local control'. When we are in local control a new main loop is entered and the current state is temporarily forgotten: we can for instance expand where one level up expansion was not done. It sounds complicated an indeed it is complicated so examples have to clarify it.

```
1 \setbox0\hbox to 10pt{2} \count0=3 \the\count0 \multiply\count0 by 4
```

This snippet of code is not that useful but illustrates what we're dealing with:

- The `1` gets typeset. So, characters like that are seen as text.
- The `\setbox` primitive triggers picking up a register number, then goes on scanning for a box specification and that itself will typeset a sequence of whatever until the group ends.
- The `count` primitive triggers scanning for a register number (or reference) and then scans for a number; the equal sign is optional.
- The `the` primitive injects some value into the current input stream and it does so by entering a new input level.
- The `multiply` primitive picks up a register specification and multiplies that by the next scanned number. The `by` is optional.

We now look at this snippet again but with an expansion context:

```
\def \TestA{1 \setbox0\hbox{2} \count0=3 \the\count0}

\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}
```

These two macros have a slightly different body. Make sure you see the difference before reading on.

| **control sequence: TestA** | | | | | | |
|---|---|---|---|---|---|---|
| 501992 | 12 | 49 | other char | 1 | U+00031 | |
| 214118 | 10 | 32 | spacer | | | |
| 351830 | 116 | 0 | set box | | | setbox |

| | | | | | | |
|---|---|---|---|---|---|---|
| 386699 | 12 | 48 | other char | 0 | U+00030 | |
| 501919 | 30 | 10 | make box | | | hbox |
| 499872 | 1 | 123 | left brace | | | |
| 450367 | 12 | 50 | other char | 2 | U+00032 | |
| 501945 | 2 | 125 | right brace | | | |
| 501760 | 10 | 32 | spacer | | | |
| 499875 | 109 | 0 | register | | | count |
| 501966 | 12 | 48 | other char | 0 | U+00030 | |
| 502253 | 12 | 61 | other char | = | U+0003D | |
| 260185 | 12 | 51 | other char | 3 | U+00033 | |
| 502173 | 10 | 32 | spacer | | | |
| 509449 | 129 | 0 | the | | | the |
| 30523 | 109 | 0 | register | | | count |
| 499870 | 12 | 48 | other char | 0 | U+00030 | |

**control sequence: TestB**

| | | | | | | |
|---|---|---|---|---|---|---|
| 501909 | 12 | 49 | other char | 1 | U+00031 | |
| 501761 | 10 | 32 | spacer | | | |
| 501916 | 116 | 0 | set box | | | setbox |
| 502146 | 12 | 48 | other char | 0 | U+00030 | |
| 82910 | 30 | 10 | make box | | | hbox |
| 450456 | 1 | 123 | left brace | | | |
| 450436 | 12 | 50 | other char | 2 | U+00032 | |
| 501774 | 2 | 125 | right brace | | | |
| 30540 | 10 | 32 | spacer | | | |
| 502304 | 109 | 0 | register | | | count |
| 502158 | 12 | 48 | other char | 0 | U+00030 | |
| 502122 | 12 | 61 | other char | = | U+0003D | |
| 339751 | 12 | 51 | other char | 3 | U+00033 | |
| 158336 | 10 | 32 | spacer | | | |
| 501832 | 12 | 49 | other char | 1 | U+00031 | |

We now introduce a new primitive `\localcontrolled`:

```
\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}

\edef\TestC{1 \setbox0\hbox{2} \localcontrolled{\count0=3} \the\count0}
```

Again, watch the subtle differences:

| control sequence: TestB | | | | | | |
|---|---|---|---|---|---|---|
| 502044 | 12 | 49 | other char | 1 | U+00031 | |
| 502094 | 10 | 32 | spacer | | | |
| 94730 | 116 | 0 | set box | | | setbox |
| 186448 | 12 | 48 | other char | 0 | U+00030 | |
| 30483 | 30 | 10 | make box | | | hbox |
| 386727 | 1 | 123 | left brace | | | |
| 30480 | 12 | 50 | other char | 2 | U+00032 | |
| 501844 | 2 | 125 | right brace | | | |
| 450449 | 10 | 32 | spacer | | | |
| 501764 | 109 | 0 | register | | | count |
| 214115 | 12 | 48 | other char | 0 | U+00030 | |
| 30545 | 12 | 61 | other char | = | U+0003D | |
| 502033 | 12 | 51 | other char | 3 | U+00033 | |
| 502291 | 10 | 32 | spacer | | | |
| 502197 | 12 | 49 | other char | 1 | U+00031 | |

| control sequence: TestC | | | | | | |
|---|---|---|---|---|---|---|
| 386736 | 12 | 49 | other char | 1 | U+00031 | |
| 502131 | 10 | 32 | spacer | | | |
| 502155 | 116 | 0 | set box | | | setbox |
| 502204 | 12 | 48 | other char | 0 | U+00030 | |
| 497886 | 30 | 10 | make box | | | hbox |
| 509452 | 1 | 123 | left brace | | | |
| 502132 | 12 | 50 | other char | 2 | U+00032 | |
| 507020 | 2 | 125 | right brace | | | |
| 501914 | 10 | 32 | spacer | | | |
| 30554 | 10 | 32 | spacer | | | |
| 502006 | 12 | 51 | other char | 3 | U+00033 | |

Another example:

```
\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}

\edef\TestD{\localcontrolled{1 \setbox0\hbox{2} \count0=3 \the\count0}}
```

1 3 ← Watch how the results end up here!

| control sequence: TestB |
|---|

| | | | | | | |
|---|---|---|---|---|---|---|
| 502281 | 12 | 49 | other char | 1 | U+00031 | |
| 502335 | 10 | 32 | spacer | | | |
| 386722 | 116 | 0 | set box | | | setbox |
| 225583 | 12 | 48 | other char | 0 | U+00030 | |
| 501837 | 30 | 10 | make box | | | hbox |
| 502421 | 1 | 123 | left brace | | | |
| 177416 | 12 | 50 | other char | 2 | U+00032 | |
| 186451 | 2 | 125 | right brace | | | |
| 497894 | 10 | 32 | spacer | | | |
| 497805 | 109 | 0 | register | | | count |
| 386730 | 12 | 48 | other char | 0 | U+00030 | |
| 30510 | 12 | 61 | other char | = | U+0003D | |
| 501790 | 12 | 51 | other char | 3 | U+00033 | |
| 502014 | 10 | 32 | spacer | | | |
| 502205 | 12 | 51 | other char | 3 | U+00033 | |

control sequence: TestD

<no tokens>

We can use this mechanism to define so called fully expandable macros:

```
\def\WidthOf#1%
  {\beginlocalcontrol
   \setbox0\hbox{#1}%
   \endlocalcontrol
   \wd0 }

\scratchdimen\WidthOf{The Rite Of Spring}

\the\scratchdimen
```

104.72021pt

When you want to add some grouping, it quickly can become less pretty:

```
\def\WidthOf#1%
  {\dimexpr
      \beginlocalcontrol
        \begingroup
          \setbox0\hbox{#1}%
          \expandafter
```

```
      \endgroup
    \expandafter
    \endlocalcontrol
    \the\wd0
  \relax}

\scratchdimen\WidthOf{The Rite Of Spring}

\the\scratchdimen
```

104.72021pt

A single token alternative is available too and its usage is like this:

```
 \def\TestA{\scratchcounter=100 }
\edef\TestB{\localcontrol\TestA \the\scratchcounter}
\edef\TestC{\localcontrolled{\TestA} \the\scratchcounter}
```

The content of `\TestB` is '`100`' and of course the `\TestC` macro gives '` 100`'.

We now move to the Lua end. Right from the start the way to get something into TEX from Lua has been the print functions. But we can also go local (immediate). There are several methods:

- via a set token register
- via a defined macro
- via a string

Among the things to keep in mind are catcodes, scope and expansion (especially in when the result itself ends up in macros). We start with an example where we go via a token register:

```
\toks0={\setbox0\hbox{The Rite Of Spring}}
\toks2={\setbox0\hbox{The Rite Of Spring!}}

\startluacode
tex.runlocal(0) context("[1: %p]",tex.box[0].width)
tex.runlocal(2) context("[2: %p]",tex.box[0].width)
\stopluacode
```

[1: 104.72021pt][2: 109.14062pt]

We can also use a macro:

```
\def\TestA{\setbox0\hbox{The Rite Of Spring}}
\def\TestB{\setbox0\hbox{The Rite Of Spring!}}

\startluacode
tex.runlocal("TestA") context("[3: %p]",tex.box[0].width)
tex.runlocal("TestB") context("[4: %p]",tex.box[0].width)
\stopluacode
```

[3: 104.72021pt][4: 109.14062pt]

A third variant is more direct and uses a (Lua) string:

```
\startluacode
tex.runstring([[\setbox0\hbox{The Rite Of Spring}]])

context("[5: %p]",tex.box[0].width)

tex.runstring([[\setbox0\hbox{The Rite Of Spring!}]])

context("[6: %p]",tex.box[0].width)
\stopluacode
```

[5: 104.72021pt][6: 109.14062pt]

A bit more high level:

```
context.runstring([[\setbox0\hbox{(Here \bf 1.2345)}]])
context.runstring([[\setbox0\hbox{(Here \bf   %.3f)}]],1.2345)
```

Before we had `runstring` this was the way to do it when staying in Lua was needed:

```
\startluacode
token.setmacro("TestX",[[\setbox0\hbox{The Rite Of Spring}]])
tex.runlocal("TestX")
context("[7: %p]",tex.box[0].width)
\stopluacode
```

[7: 104.72021pt]

```
\startluacode
tex.scantoks(0,tex.ctxcatcodes,[[\setbox0\hbox{The Rite Of Spring!}]])
tex.runlocal(0)
context("[8: %p]",tex.box[0].width)
\stopluacode
```

[8: 109.14062pt]

The order of flushing matters because as soon as something is not stored in a token list or macro body, TEX will typeset it. And as said, a lot of this relates to pushing stuff into the input which is stacked. Compare:

```
\startluacode
context("[HERE 1]")
context("[HERE 2]")
\stopluacode
```

[HERE 1][HERE 2]

with this:

```
\startluacode
tex.pushlocal() context("[HERE 1]") tex.poplocal()
tex.pushlocal() context("[HERE 2]") tex.poplocal()
\stopluacode
```

[HERE 2][HERE 1]

You can expand a macro at the Lua end with `token.expandmacro` which has a peculiar interface. The first argument has to be a string (the name of a macro) or a userdata (a valid macro token). This macro can be fed with parameters by passing more arguments:

| | |
|---|---|
| string | serialized to tokens |
| true | wrap the next string in curly braces |
| table | each entry will become an argument wrapped in braces |
| token | inject the token directly |
| number | change control to the given catcode table |

There are more scanner related primitives, like the $\varepsilon$-TEX primitive `\detokenize`:

```
[\detokenize {test \relax}]
```

This gives: `[test \relax ]` . In LuaMetaTEX we also have complementary primitive(s):

```
[\tokenized   catcodetable \vrbcatcodes {test {\bf test} test}]
[\tokenized                             {test {\bf test} test}]
[\retokenized             \vrbcatcodes {test {\bf test} test}]
```

The \tokenized takes an optional keyword and the examples above give: [test {\bf test} test] [test **test** test] [test {\bf test} test] . The LuaTEX primitive \scantextokens which is a variant of ε-TEX's \scantokens operates under the current catcode regime (the last one honors \everyeof). The difference with \tokenized is that this one first serializes the given token list (just like \detokenize).[2]

With \retokenized the catcode table index is mandatory (it saves a bit of scanning and is easier on intermixed \expandafter usage. There often are several ways to accomplish the same:

```
\def\MyTitle{test {\bf test} test}
\detokenize                \expandafter{\MyTitle}: 0.46\crlf
\meaningless                            \MyTitle : 0.47\crlf
\retokenized               \notcatcodes{\MyTitle}: 0.87\crlf
\tokenized   catcodetable  \notcatcodes{\MyTitle}: 0.93% \crlf
```

test {\bf test} test: 0.46
test {\bf test} test: 0.47
test {\bf test} test: 0.87
test {\bf test} test: 0.93

Here the numbers show the relative performance of these methods. The \detokenize and \meaningless win because they already know that a verbose serialization is needed. The last two first serialize and then reinterpret the resulting token list using the given catcode regime. The last one is slowest because it has to scan the keyword.

There is however a pitfall here:

```
\def\MyText {test}
\def\MyTitle{test \MyText\space test}
\detokenize                \expandafter{\MyTitle}\crlf
\meaningless                            \MyTitle \crlf
\retokenized               \notcatcodes{\MyTitle}\crlf
\tokenized   catcodetable  \notcatcodes{\MyTitle}\crlf
```

The outcome is different now because we have an expandable embedded macro call. The fact that we expand in the last two primitives is also the reason why they are 'slower'.

[2] The \scan *tokens primitives now share the same helpers as Lua, but they should behave the same as in LuaTEX.

test \MyText \space test
test \MyText \space test
test test test
test test test

To complete this picture, we show a variant than combines much of what has been introduced in this section:

```
\semiprotected\def\MyTextA {test}
\def\MyTextB {test}
\def\MyTitle{test \MyTextA\space \MyTextB\space test}
\detokenize                \expandafter{\MyTitle}\crlf
\meaningless                            \MyTitle \crlf
\retokenized               \notcatcodes{\MyTitle}\crlf
\retokenized               \notcatcodes{\semiexpanded{\MyTitle}}\crlf
\tokenized   catcodetable \notcatcodes{\MyTitle}\crlf
\tokenized   catcodetable \notcatcodes{\semiexpanded{\MyTitle}}
```

This time compare the last four lines:

test \MyTextA \space \MyTextB \space test
test \MyTextA \space \MyTextB \space test
test \MyTextA test test
test test test test
test \MyTextA test test
test test test test

Of course the question remains to what extend we need this and eventually will apply in ConTEXt. The `\detokenize` is used already. History shows that eventually there is a use for everything and given the way LuaMetaTEX is structured it was not that hard to provide the alternatives without sacrificing performance or bloating the source.

# 6 Dirty tricks

When I was updating this manual Hans vd Meer and I had some discussions about expansion and tokenization related issues when combining of xml processing with TEX macros where he did some manipulations in Lua. In these mixed cases you can run into catcode related problems because in xml you want for instance a # to be a hash mark (other character) and not an parameter identifier. Normally this is handled well in ConTEXt but of course there are complex cases where you need to adapt.

Say that you want to compare two strings (officially we should say token lists) with mixed catcodes. Let's also assume that you want to use the normal `\if` construct (which was part of the discussion). We start with defining a test set. The reason that we present this example here is that we use commands discussed in previous sections:

```
               \def\abc{abc}
\semiprotected \def\xyz{xyz}
              \edef\pqr{\expandtoken\notcatcodes`p%
                        \expandtoken\notcatcodes`q%
                        \expandtoken\notcatcodes`r}

1: \ifcondition\similartokens{abc} {def}YES\else NOP\fi (NOP) \quad
2: \ifcondition\similartokens{abc}{\abc}YES\else NOP\fi (YES)

3: \ifcondition\similartokens{xyz} {pqr}YES\else NOP\fi (NOP) \quad
4: \ifcondition\similartokens{xyz}{\xyz}YES\else NOP\fi (YES)

5: \ifcondition\similartokens{pqr} {pqr}YES\else NOP\fi (YES) \quad
6: \ifcondition\similartokens{pqr}{\pqr}YES\else NOP\fi (YES)
```

So, we have a mix of expandable and semi expandable macros, and also a mix of catcodes. A naive approach would be:

```
\permanent\protected\def\similartokens#1#2%
  {\iftok{#1}{#2}}
```

but that will fail on some cases:

```
1: NOP(NOP)   2: YES(YES)
3: NOP(NOP)   4: NOP(YES)
5: YES(YES)   6: NOP(YES)
```

So how about:

```
\permanent\protected\def\similartokens#1#2%
  {\iftok{\detokenize{#1}}{\detokenize{#2}}}
```

That one is even worse:

```
1: NOP(NOP)   2: NOP(YES)
3: NOP(NOP)   4: NOP(YES)
5: YES(YES)   6: NOP(YES)
```

We need to expand so we end up with this:

```
\permanent\protected\def\similartokens#1#2%
  {\normalexpanded{\noexpand\iftok
     {\noexpand\detokenize{#1}}
     {\noexpand\detokenize{#2}}}}
```

Better:

```
1: NOP(NOP)   2: YES(YES)
3: NOP(NOP)   4: NOP(YES)
5: YES(YES)   6: YES(YES)
```

But that will still not deal with the mildly protected macro so in the end we have:

```
\permanent\protected\def\similartokens#1#2%
  {\semiexpanded{\noexpand\iftok
     {\noexpand\detokenize{#1}}
     {\noexpand\detokenize{#2}}}}
```

Now we're good:

```
1: NOP(NOP)   2: YES(YES)
3: NOP(NOP)   4: YES(YES)
5: YES(YES)   6: YES(YES)
```

Finally we wrap this one in the usual `\doifelse...` macro:

```
\permanent\protected\def\doifelsesimilartokens#1#2%
  {\ifcondition\similartokens{#1}{#2}%
     \expandafter\firstoftwoarguments
   \else
     \expandafter\secondoftwoarguments
   \fi}
```

so that we can do:

```
\doifelsesimilartokens{pqr}{\pqr}{YES}{NOP}
```

A companion macro of this is `\wipetoken` but for that one you need to look into the source.