low level

TEX

expansion

Contents

1 Preamble

This short manual demonstrates a couple of properties of the macro language. It is not the in-depth philosophical expose about macro languages, tokens, expansion and such that some TEXies like. I prefer to stick to the practical aspects.

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 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.

A macro can contain references to macros so in practice the input can go several levels down.

```
\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 `a` 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` became five tokens:

a reference to a space token, then three character tokens with property ‘letter’, and finally again a space token.

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

```
a\MyMacroA b
```

In the previous example an `\edef` is used, where the `e` indicates expansion. This time the meaning gets expanded. So we get effectively the same as

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

Characters are easy: they just expand, 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 basically 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: “`1 and 2`”. Later we will see that other engines provide some more expansion tricks. The only way to get a grip on expansion is to just play with it.

The `\expandafter` primitive expands the token (which can be a macro) after the next next one and 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.

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

These macros have the following meanings:

```
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 using an intermediate `\edef`. In ConTEXt the `\expanded` macro does something simpler but it doesn’t expand the first token as it 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:

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

This (when typeset monospaced) is: `this can $ be anything \bgroup`. The detokenizer is like `\string` applied to each token in its argument. Compare this:

```
\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 in the expansion arena. 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 \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`.

There 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.

A second extension is `\immediateassignment` which instead in tokenizing the assignment directive applies it right now.

```
\edef\MyMacroA
  {\scratchcounter 123
   \noexpand\the\scratchcounter}
```

```
\edef\MyMacroB
  {\immediateassignment\scratchcounter 123
   \noexpand\the\scratchcounter}
```

These two macros now have the meaning:

```
macro:->\scratchcounter 123 \the \scratchcounter
macro:->\the \scratchcounter
```

5 LuaMetaTEX primitives

todo