Notes On Programming in TeX

Christian Feuersänger*
Institut für Numerische Simulation
Universität Bonn, Germany

March 31, 2010

Abstract

This document contains notes which are intended for those who are interested in TeX programming. It is valueable for beginners as a first start with a lot of examples, and it is also valueable for experienced TeXnicians who are interested in details about TeX programming. However, it is neither a complete reference, nor a complete manual of TeX.

Contents

1 Introduction **1**

2 Programming in TeX **1**
2.1 Variables in Registers . . . 1
2.1.1 Allocating Registers . . . 3
2.1.2 Using More than 256 Registers . . . 3
2.2 Arithmetics in TeX . . . 3
2.3 Expansion Control . . . 4
2.3.1 Macros . . . 4
2.3.2 Token Registers . . . 8
2.3.3 Summary of macro definition commands . . . 9
2.3.4 Debugging Tools – Understanding and Tracing What TeX Does . . . 11
2.4 The Scope of a Variable . . . 11
2.4.1 Global Variables . . . 12
2.4.2 Transporting Changes to an Outer Group . . . 12
2.5 More On TeX . . . 13

Index **14**

1 Introduction

This document is intended to provide a direct start with TeX programming (not necessarily TeX typesetting). The addressed audience consists of people interested in package or library writing.

At the time of this writing, this document is far from complete. Nevertheless, it might be a good starting point for interested readers. Consult the literature given below for more details.

2 Programming in TeX

2.1 Variables in Registers

TeX provides several different variables and associated registers which can be manipulated freely.

*http://wissrech.ins.uni-bonn.de/people/feuersaenger

\count⟨*num*⟩

There are 256 Integer registers which provide 32 Bit Integer arithmetics. The registers can be used for example with `\count0=42` or `\count7=\macro` where `\macro` expands to a number.

The value of a register can be typeset using `\the`⟨*register*⟩.

The value is now '42'.

The value is now '-123456'.

```
\count0=42
The value is now '\the\count0'.
\def\macro{-123456}
\count0=\macro
The value is now '\the\count0'.
```

The '`=`' sign is optional and can be omitted. One thing is common among the registers: an assignment of the form `\count0=`⟨···⟩ expands everything which follows until the expansion doesn't need more numbers – even more than one following macro.

The value is now '123456789'.

```
\def\firstmacro{123}
\def\secondmacro{456}
\def\thirdmacro{789}
\count0=\firstmacro\secondmacro\thirdmacro
The value is now '\the\count0'.
```

The precise rules can be found in [2], but it should be kept in mind that care needs to be taken here. More than once, my code failed to produce the expected result because TEX kept expanding macros and the registers got unexpected results. Here is the correct method:

1. The value is now '42'.

2. The following code will absorb the '3' of '3.':

. The value is now '12343'.

4. Use \relax after an assignment to end scanning:

5. The value is now '1234'.

```
1. \count0=42 %  a white space after the number aborts the reading process.
The value is now '\the\count0'.
2. The following code will absorb the '3' of '3.':
\def\macro{1234}
\count0=\macro %  a white space after a macro will be absorbed by TeX, so this is wrong.
3. The value is now '\the\count0'.
4. Use \textbackslash relax after an assignment to end scanning:
\count0=\macro\relax
5. The value is now '\the\count0'.
```

The command `\relax` tells TEX to "relax": it stops scanning for tokens, but `\relax` doesn't expand to anything.

\dimen⟨*num*⟩

There are also 255 registers for fixed point numbers which are used pretty much in the same way as the `\count` registers – but `\dimen` register assignments require a unit like '`cm`' or '`pt`'.

String access with '`\the`' works in exactly the same way as for `\count` registers.

The value is now 1.0pt.

The value is now 0.0001pt.

The value is now 1234.5678pt.

```
\dimen0=1pt
The value is now \the\dimen0.
\dimen0=0.0001pt
The value is now \the\dimen0.
\def\macro{1234.5678}
\dimen0=\macro pt
The value is now \the\dimen0.
```

The same rules with expansion of macros after assignments apply here as well.

The `\dimen` registers perform their arithmetics internally with 32 bit scaled integers, so called 'scaled point' with unit 'sp'. It holds `1sp=65536pt`$=2^{16}$`pt`. One of the 32 bits is used as sign. The total number range in `pt` is $[-(2^{30}-1)/2^{16}, (2^{30}-1)/2^{16}] = [-16383.9998, +16383.9998]$[1].

`\toks`⟨*number*⟩

There are also 255 token registers which can be thought of as special string variables. Of course, every macro assignment `\def\macro{`⟨*content*⟩`}` is also some kind of string variable, but token registers are special: their contents won't be expanded when used with `\the\toks`⟨*number*⟩. This can be used for fine grained expansion control, see section 2.3 below.

2.1.1 Allocating Registers

2.1.2 Using More than 256 Registers

2.2 Arithmetics in TEX

`\advance`⟨*register*⟩ `by`⟨*quantity*⟩

The value is now 52.

```
\count0=42
\advance\count0 by 10
The value is now \the\count0.
```

The value is now 11.0pt.

```
\dimen0=1pt
\advance\dimen0 by 10pt
The value is now \the\dimen0.
```

`\multiply`⟨*register*⟩ `by`⟨*integer*⟩

The value is now -420.

```
\count0=42
\multiply\count0 by -10
The value is now \the\count0.
```

The value is now 10.0pt.

```
\dimen0=0.5pt
\multiply\dimen0 by 20
The value is now \the\dimen0.
```

`\divide`⟨*register*⟩ `by`⟨*integer*⟩

This allows integer division by ⟨*integer*⟩ with truncation.

The value is now 2.

```
\count0=5
\divide\count0 by 2
The value is now \the\count0.
```

Scaling of `\dimen` registers:

The value is now 0.5pt.

```
\dimen0=10pt
\divide\dimen0 by 20
The value is now \the\dimen0.
```

[1] Please note that this does not cover the complete range of a 32 bit integer, I do not know why.

\dimen⟨*number*⟩=⟨*fixed point number without unit*⟩\dimen⟨*number*⟩

This allows fixed point multiplication in \dimen registers.

The value is now 30.0003pt.

```
\dimen1=50pt
\dimen0=0.6\dimen1
The value is now \the\dimen0.
```

2.3 Expansion Control

Expansion is what TEX does all the time. Thus, expansion control is a key concept for understanding how to program in TEX.

The first thing to know is: TEX deals the input as a long, long sequence of "tokens". A token is the smallest unit which is understood by TEX. Each character becomes a token the first time it is seen by TEX. Every macro becomes a (single!) token the first time it is seen by TEX.

The second thing to know is what characters are *before* TEX has seen them. Although this knowledge is rarely needed in every day's life, it is nevertheless important. The characters which are in the input document are nothing but characters at first. Even the characters known to have a special meaning like '`%`', '\' or the braces '`{}`' are *not* special – until they have been converted to a token. This happens when TEX encounters them the first time during its linear processing of the character stream. A token stays a token - and it will remain the same token forever. If you manage to tell TEX that '\' is a normal character and TEX sees just one backslash, this backslash will be a normal character token – even if the meaning of all following backslashes is again special.

Now, we are given a very long list of tokens ⟨*token1*⟩⟨*token2*⟩⟨*token3*⟩⟨*token4*⟩⟨*token5*⟩· · · . TEX processes these tokens one-by-one in linear sequence. If ⟨*token1*⟩ is a character token like '`a`', it is typeset. This is not what I want to write about here now; my main point is how to program in TEX[2]. So, the interesting thing in these notes is when ⟨*token1*⟩ is a macro.

2.3.1 Macros

We have already seen some applications of macros above. Actually, most users who are willing to read notes about TEX programming will have seen macros and may have written some on their own – for example using `\newcommand` (`\newcommand` is a "more high–level" version of \def used only in LATEX).

A macro has a name and is treated as an elementary token in TEX (even if the name is very long). A macro has replacement text. As soon as TEX encounters a macro, it replaces its occurance with the replacement text. Furthermore, a macro can consume one or more of the following tokens as arguments.

Executing it: 'This here is actually the replacement text.'.

```
\def\macro{This here is actually the replacement text.}
Executing it: '\macro'.
```

Invoking it: replacement with first argument=hello!.

```
\def\macro#1{replacement with first argument=#1}
Invoking it: \macro{hello!}.
```

This here is not really a surprise. What might come as a surprise is that the accepted arguments can be pretty much anything.

Invoking it: replacement with arguments: 'a' and 'sign'.

```
\def\macro#1-#2.{replacement with arguments: '#1' and '#2'.}
Invoking it: \macro a-sign.
```

The last example `\macro` runs through the token list which follows the occurance of `\macro`. This token list is "`a-sign.`". Macro expansion is greedy, that means the first matching pattern is used. Now, our `\macro` expected something, then a minus sign '`-`', then another (possibly long) argument, then a period '`.`'. The argument between `\macro` and the minus sign is available as `#1` and the tokens between the minus sign and the period as `#2`.

I found arguments '42', '43' and '44'.

[2]Of course, typesetting is an art in itsself and there is a lot to read about it. Just not here in these notes.

```
\def\macro(#1,#2,#3){I found arguments ‘#1’, ‘#2’ and ‘#3’.}
\macro(42,43,44)
```

As we have seen, macros can be used to manipulate the input tokens by expansion: they take some input arguments (maybe none) away and insert other tokens into the input token list. These tokens will be the next to process. We will soon learn more about that.

There is a command which helps to understand what TeX does here:

`\meaning`⟨*macro*⟩

This command expands to the contents of ⟨*macro*⟩ as it is seen by TeX.

```
\def\macro{Replacement \textmacro text  \count0=42 \the\count0.}
\message{Debug message: '\meaning\macro'}
```

As result, the log file and terminal output will contain

`Debug message:  'macro:->Replacement \textmacro text \count 0=42 \the \count 0.'`

The last example already shows something about `\def`: the replacement text can still contain other macros.

`\def`⟨*macroname*⟩⟨*argument pattern*⟩`{`⟨*replacement text*⟩`}`

A new macro named ⟨*macroname*⟩ will be defined (or re-defined). The `{`⟨*replacement text*⟩`}` is the macro body, whenever the macro is executed, it expands to `{`⟨*replacement text*⟩`}`. The `{`⟨*replacement text*⟩`}` is a token list which can contain other macros. On the time of the definition, TeX does *not* process (expand) the `{`⟨*replacement text*⟩`}`.

The `{`⟨*replacement text*⟩`}` will only be expanded if the macro is executed. This does also apply to any macros which are inside of `{`⟨*replacement text*⟩`}`.

Now, I execute it: Macro two contains This is macro one..

Now, I exectute the second macro again: Macro two contains Redefined macroone..

```
\def\macroone{This is macro one}
\def\macrotwo{Macro two contains \macroone.}
Now, I execute it: \macrotwo.
\def\macroone{Redefined macroone}
Now, I exectute the second macro again: \macrotwo.
```

Macros can be defined almost everywhere in a TeX document. They can also be invoked almost everywhere.

The ⟨*argument pattern*⟩ is a token list which can contain simple strings or macro parameters ‘`#`⟨*number*⟩’ or other macro tokens. The ⟨*number*⟩ of the first parameter is always 1, the second must have 2 and so on up to at most 9. Valid argument patterns are ‘`#1#2#3`’, ‘`(#1,#2,#3)`’ or ‘`---\relax`’. If TeX executes a macro, it searches for ⟨*argument pattern*⟩ in the input token list until the first match is found. If no match can be found, it aborts with a (more or less helpful) error message.

Got ‘g’

```
\def\macroone abc{\macrotwo}
\def\macrotwo def{\macrothree}
\def\macrothree#1{Got ‘#1’}
\macroone abcdefg
```

The last example contains three macro definitions. Then, TeX encounters `\macroone`. The input token list is now

‘`\macroone abcdefg`’.

The space(s) following `\macroone` are ignored by TeX, they delimit the ⟨*macroname*⟩. Now, TeX attempts to find matches for ⟨*argument pattern*⟩. It expects ‘`abc`’ – and it finds ‘`abc`’. These three tokens are *removed* from the input token list, and TeX inserts the replacement text of `\macroone` which is `\macrotwo`. At that time, the input token list is

‘`\macrotwo defg`’.

Now, the same game continues with \macrotwo: TEX searches for the expected {⟨*argument pattern*⟩} which is 'def', erases these tokens from the input token list and inserts the replacement text of \macrotwo instead. This yields

'\macrothree g'.

Finally, \macrothree expects one parameter token (or a token list enclosen in parenthesis). The next token is 'g', which is consumed from the input token list and the replacement text is inserted – and '#1' is replaced by 'g'. Then, the token list is

'Got 'g''.

This text is finally typeset (because it doesn't expand further).

What we have seen now is how TEX macros can be used to modify the token list. It should be noted explicitly that macro expansion does is in no way limited to those tokens provided inside of {⟨*replacement text*⟩} – if the last argument in {⟨*replacement text*⟩} is a macro which requires arguments, these arguments will be taken from the following tokens. Using nested macros, one can even process a complete part of the token list, in a manner of loops (but we don't know yet how to influence macro expansion conditionally, that comes later).

Let's try to solve the following task. Suppose you have a macro named \point with ⟨*argument pattern*⟩ '(#1,#2)', i.e.

\def\point(#1,#2){we do something with #1 and #2}.

Suppose furthermore that you want to invoke \point with the contents which is stored in another macro. After all, macros are some kind of string variables – it makes sense to accumulate or generate string variables which will then be used as input for other macros. Let's assume we have \temp and \temp contains '(42,1234)'. A first choice to invoke \point would be to use \point\temp. But: \point searches for an argument pattern which starts with '(', not with \temp! The invocation fails.

\expandafter⟨*token*⟩⟨*next token*⟩

The \expandafter command is an – at first sight confusing – method to alter the input token list. But: it solves our problem with \point\temp!

we do something with 42 and 1234

```
\def\point(#1,#2){we do something with #1 and #2}
\def\temp{(42,1234)}
\expandafter\point\temp
```

Why did that work!? The command \expandafter scans for the token after \expandafter in the input token list. This is \point in our case. Then, it scans for the next token which is \temp in our case (remember: macros are considered to be elementary tokens, just like characters 'a' or so). The two scanned arguments are removed from the input token list. Then, \expandafter *expands* the ⟨*next token*⟩ one time. In our case, ⟨*next token*⟩ is \temp. The first level of expansion of \temp is '(42,1234)'. Then, \expansion inserts the (unexpanded) ⟨*token*⟩ followed by the (expanded) contents of ⟨*next token*⟩ back into the input token list. In single steps:

1. \expandafter\point\temp
2. Expand \expandafter: next two tokens are '\point\temp'.
3. Use \point as ⟨*token*⟩ and \temp as ⟨*next token*⟩.
4. Expand \temp once, which leads to the tokens '(42,1234)'.
5. re-insert ⟨*token*⟩ and the expansion of ⟨*next token*⟩ back into the input token list. The list is then '\point(42,1234)'.
6. Expand \point as next token.

A further example: suppose we want to invoke \theimportantmacro{⟨*argument*⟩}. However, {⟨*argument*⟩} is contained in another macro! Furthermore, \theimportantmacro is defined to take exactly one parameter and our desired argument may have more than one token (which means we need to surround it with braces). This can be solved by the listing below.

I got the pre-assembled argument 'xyz' here.

```
\def\theimportantmacro#1{I got the pre-assembled argument '#1' here.}
\def\temp{xyz}
\expandafter\theimportantmacro\expandafter{\temp}
```

Now, what happens here? Let's apply the rules step by step again:

1. After the initial definitions, the token list is `\expandafter\theimportantmacro\expandafter{\temp}`.
2. TeX expands `\expandafter`, using `\theimportantmacro` as ⟨*token*⟩ and the second `\expandafter` as ⟨*next token*⟩.
3. According to the rules, TeX expands ⟨*next token*⟩ once. But: ⟨*next token*⟩ is again a macro, namely `\expandafter`! Does that make a difference? No:
   (a) The token list after the second `\expandafter` is '`{\temp}`' (3 tokens).
   (b) The ⟨*token*⟩ is thus '`{`' and ⟨*next token*⟩ is '`\temp`'.
   (c) The expansion of ⟨*next token*⟩ is '`xyz`'.
   (d) The second `\expandafter` re-inserts its ⟨*token*⟩ and expanded ⟨*next token*⟩, which is '`{xyz`'.
   Note that the closing brace '`}`' has not been touched at all, TeX hasn't even seen it so far.

   We come back from the recursion. Remember: ⟨*token*⟩ is `\theimportantmacro` and the top-level expansion of ⟨*next token*⟩ is – as we have seen above – '`{xyz`'.
4. TeX re-inserts ⟨*token*⟩ and the expansion of ⟨*next token*⟩ to the input token list, which leads to '`\theimportantmacro{xyz}`'.
   The closing brace '`}`' has not been touched, it simply resides in the input token list.
5. TeX expands `\theimportantmacro`.

The ⟨*next token*⟩ is expanded exactly once. We have already seen that if ⟨*next token*⟩ is a macro which does substitutions on its own, these substitutions will be performed recursively. But what means 'once' exactly? We will need to use `\meaning` to check that (or the `\tracingmacros` tools) because we need to see what TeX does.

So far, nothing has been typeset. But now: 4[This is macro one –2–].

```
\def\macroone{This is macro one \macrotwo}
\def\macrotwo{--2--}
\def\macrothree#1{\def\macrofour{4[#1]}}
\expandafter\macrothree\expandafter{\macroone}%
So far, nothing has been typeset. But now: \macrofour.
\message{We have macrofour = \meaning\macrofour}%
```

The logfile (and terminal) will now contain

'`We have macrofour = macro:->4[This is macro one \macrotwo ]`'.

What happened? We can proceed as in the last example. After the two `\expandafter` expansions, TeX finds the input token list

'`\macrothree{This is macro one \macrotwo}`'

which, after execution, defines `\macrofour` to be '`This is macro one \macrotwo`'. The top-level expansion of `\macroone` has not expanded the nested call to `\macrotwo`.

So, `\expandafter` is a normal macro which can be expanded – and it is even possible to expand an `\expandafter` by another `\expandafter`.

What we have seen so far is

1. the `\def` command which stores *unexpanded* arguments in a macro variable and
2. the `\expandafter` which allows control over top-level expansion of macros (it expands one time).

TeX provides two more features for expansion control: the `\edef` macro and token registers.

`\edef`⟨*macroname*⟩⟨*argument pattern*⟩`{`⟨*replacement text*⟩`}`

The `\edef` command is the same as `\def` insofar as it defines a new macro. However, it expands `{`⟨*replacement text*⟩`}` until only unexpandable tokens remain (`\edef` = expanded definition).

```
\def\a{3}
\def\b{2\a}
\def\c{1\b}
\def\d{value=\c}
\message{Macro ‘d’ is defined to be ‘\meaning\d’}
\edef\d{value=\c}
\message{Macro ‘d’ is e-defined to be ‘\meaning\d’}
\expandafter\def\expandafter\d\expandafter{\c}
\message{Macro ‘d’ is defined to be ‘\meaning\d’ using expandafter}
```

This listing results in the log-file output

`Macro ‘d’ is defined to be ‘macro:->value=\c ’`

`Macro ‘d’ is e-defined to be ‘macro:->value=123’`

`Macro ‘d’ is defined to be ‘macro:->1\b ’ using expandafter`

So, `\def` does not expand at all, `\edef` expands until it can't expand any further and the `\expandafter` construction expands `\c` one time and defines `\d` to be the result of this expansion.

Although possible, it might not occur too often to specify ⟨*argument pattern*⟩ for an `\edef` because the expansion is immediate in contrast to `\def`. But it works in the same way: the positional arguments `#1`, `#2`, . . . , `#9` will be replaced with their arguments.

The expansion of `{`⟨*replacement text*⟩`}` happens in the same way as the expansion the main token list of TeX.

Now, what exactly does "expands until only unexpandable tokens remain" mean? Our example indicates that the three tokens `1`, `2` and `3` are not expandable while the macros `\c`, `\b` and `\a` could be expanded. There is one large class of TeX commands which can't be expanded: any assignment operation. The example

```
\edef\d{\count0=42}
\message{Macro ‘d’ is defined to be ‘\meaning\d’}
\def\a{1234}
\edef\d{\advance\count0 by\a}
\message{Macro ‘d’ is defined to be ‘\meaning\d’}
```

yields the log-messages

`Macro ‘d’ is defined to be ‘macro:->\count 0=42’` and

`Macro ‘d’ is defined to be ‘macro:->\advance \count 0 by1234’`.

So, assignment and arithmetics operations are *not* expandable, they remain as executable tokens in the newly defined macro. This does also hold for `\let` and other assignment operations.

Interestingly, conditional expressions using `\if` · · · `\fi` *are* expandable, but we will come to that later.

There is also a method to convert a macro temporarily into an unexpandable token: the `\noexpand` macro.

`\noexpand`⟨*expandable token*⟩

The `\noexpand` command is only useful inside of the `{`⟨*replacement text*⟩`}` of an `\edef` command. As soon as `\edef` encounters the `\noexpand`, the `\noexpand` will be removed and the ⟨*expandable token*⟩ will be converted into an unexpandable token. Thus, the code

```
\edef\d{Invoke \noexpand\a another macro}
\message{Macro ‘d’ is defined to be ‘\meaning\d’}
```

yields the terminal output

`Macro ‘d’ is defined to be ‘macro:->Invoke \a another macro’`

because `\noexpand\a` yields the token '`\a`' (unexpanded)[3].

2.3.2 Token Registers

Now, we turn to token registers. As we have already seen in section 2.1, a token register stores a token list. A macro does also store a token list in its `{`⟨*replacement text*⟩`}`, so where is the difference? There are two differences:

[3]The `\noexpand` key is actually used to implement the LaTeX command `\protect`: LaTeX's concept of moveable arguments is implemented with `\edef`.

1. Token registers are faster.
2. The contents of token registers will *never* be expanded.

I can't give numbers for the first point – I have just read it in [2]. But the second point allows expansion control. While \edef allows "infinite" expansion, token registers allow only top–level expansion, just like \expandafter. But they can be used in a more flexible (and often more efficient) way than \expandafter.

The following examples demonstrates the second point.

```
\toks0={A \token list \a \b \count0=42 will never be expanded}
\edef\d{\the\toks0 }% the space token is important!
\message{Macro 'd' is defined to be '\meaning\d'}
```

Executing this code fragment yields the log output

`Macro 'd' is defined to be 'macro:->A \token list \a \b \count 0=42 will never be expanded'.`

So, the contents of \toks0 has been copied unexpanded into \d, although we have just \edef. Note that the space token after \the\toks0 is indeed important! TEX uses it to delimit the integer 0. Without the space token, it would have continued scanning, even beyond the boundaries of the replacement text of \edef (see section 2.1 for details about this scanning).

The example is very simple, and we could have done the same with \expandafter as before. But let's try something more difficult: we want to assemble a new macro which consists of different pieces. Each piece is stored in a macro, and for whatever reason, we only want top-level expansion of the single pieces. And: the pieces won't be adjacent to each other. We can assemble the target macro using the following example listing.

```
\def\piecea{\a{xyz}}
\def\pieceb{\count0=42 }
\def\piecec{string \b}
\toks0=\expandafter{\piecea}
\toks1=\expandafter{\pieceb}
\toks2=\expandafter{\piecec}
\edef\d{I have \the\toks0 and \the\toks1 and \the\toks2}
\message{Macro 'd' is defined to be '\meaning\d'}
```

The first three lines define our pieces. Each of the macros \piecea, \pieceb and \piecec contains tokens which should not be expanded during the definition of \d. The three following lines assign the top-level expansion of our pieces into token registers. Since \toks0={\piecea} would have stored '\piecea' into the token register, we need to use \expandafter here[4]. Then, we use \the\toks⟨*number*⟩ to insert the contents of a token list somewhere – in our case, into the expanded replacement text of our macro \d. Thus, the complete example yields the log–output

`Macro 'd' is defined to be 'macro:->I have \a {xyz}and \count 0=42 and string \b '.`

It *is* possible to get exactly the same result using (a lot of) \expandafters. Don't try it.

2.3.3 Summary of macro definition commands

Besides \def and \edef, there are some more commands which allow to define macros (although the main functionality is covered by \def and \edef). Here are the remaining definition commands.

\def⟨*macroname*⟩⟨*argument pattern*⟩{⟨*replacement text*⟩}

Defines a new macro named \macroname without expanding {⟨*replacement text*⟩}, see above.

\edef⟨*macroname*⟩⟨*argument pattern*⟩{⟨*replacement text*⟩}

Defines a new macro named \macroname, expanding {⟨*replacement text*⟩} completely (see above).

\let⟨*newmacro*⟩=⟨*token*⟩

Defines or redefines \newmacro to be an equivalent to ⟨*token*⟩. For example, \let\a=\b will create a new copy of macro \b. The copy is named \a, and it will have exactly the same {⟨*replacement text*⟩} and ⟨*argument pattern*⟩ as \b.

It is also possible that ⟨*token*⟩ is something different than a macro, for example a named register or a single character.

[4]We could have eliminated the \piece* macros by writing everything into token registers directly. But I think this example is more realistic.

\gdef⟨\macroname⟩⟨*argument pattern*⟩{⟨*replacement text*⟩}

A shortcut for \global\def. It defines \macroname globally, independant of the current scope.

You should avoid macros which exist in both, the global namespace and a local scope, with different meanings. Section 2.4 explains more about scoping.

\xdef⟨\macroname⟩⟨*argument pattern*⟩{⟨*replacement text*⟩}

A shortcut for \global\edef. It defines \macroname globally, independant of the current scope.

You should avoid macros which exist in both, the global namespace and a local scope, with different meanings. Section 2.4 explains more about scoping.

\csname⟨*expandable tokens*⟩\endcsname

This command is not a macro definition, it is a definition of a macro's *name*. The “cs” means “control sequence”. The \csname, \endcsname pair defines a control sequence name (a macro name) using ⟨*expandable tokens*⟩. The control sequence character ‘\’ will be prepended automatically by \csname.[5]

This here is normal usage: ‘Content’.

This here uses csname: ‘Content’.

```
\def\macro{Content}
This here is normal usage: ‘\macro’.
This here uses csname: ‘\csname macro\endcsname’.
```

The example demonstrates that \csname⟨*expandable tokens*⟩\endcsname is actually the same as if you had written \⟨*expandable tokens*⟩ directly – but the \csname construction allows much more tokens inside of macro names:

I use a strange macro. Here is it: ‘Content’.

```
\expandafter\def\csname a01macro with.strange.chars\endcsname{Content}
I use a strange macro. Here is it: ‘\csname a01macro with.strange.chars\endcsname’.
```

The example uses \expandafter to expand \csname one time. The top–level expansion of \csname is a single token, namely the control sequence name. Then, \def is used to define a macro with the prepared macro name.

When \csname is expanded, it parses all tokens up to the next \endcsname. Those tokens will be expanded until only unexpandable tokens remain (as in \edef). The resulting string will be used to define a macro name (with the control sequence character ‘\’ prepended). The fact that ⟨*expandable tokens*⟩ is expanded allows to use “indirect” macro names:

I have just defined “macroonetwothree

with replacement text ‘Content’.

```
\def\macro{onetwothree}
\expandafter\def\csname macro\macro\endcsname{Content}
I have just defined \expandafter\string\csname macro\macro\endcsname
with replacement text ‘\csname macro\macro\endcsname’.
```

I suppose the example is self-explaining, up to the \string command which is described below.

Due do this flexibility, \csname is used to implement all (?) of the available key–value packages in TEX.

\string⟨*macro*⟩

This command does not define a macro. Instead, it returns a macro's name as a sequence of separate tokens, including the control sequence token ‘\’.

I have just defined ‘“macro’ using ‘“def’.

```
\def\macro{Content}
I have just defined ‘\string\macro’ using ‘\string\def’.
```

You can also use \string on other tokens – for example characters. That doesn't hurt, the character will be returned as-is.

[5]In fact, the contents of \escapechar will be used here. If its value is -1, no character will be prepended. The same holds for any occurance where a backslash would be inserted by TEX commands.

2.3.4 Debugging Tools – Understanding and Tracing What TEX Does

\message{⟨*tokens*⟩}

\meaning⟨*macro*⟩

\tracingmacros=2

\tracingcommands=2

\tracingrestores=1

2.4 The Scope of a Variable

Each programming language knows the concept of a scope: they limit the effect of variables or routines. However, TEX's scoping mechanisms have not been designed for programming – TEX is a typesetting language. Many programming languages like `C`, `C++`, `java` or a lot of scripting languages define the scope of a variable using the place where the variable has been defined. For example, the `C` fragment

```
int i = 42;

{
    ++i;
    int i = 5;
}
```

changes the value of the outer `i` to `43`. The inner `i` is `5`, but it will be deleted as soon as the closing brace is encountered. It may even be possible to access both, the value of the inner `i` variable and the value of the outer `i` variable, at the same time.

In TEX, braces are also used for scopes. But: while TEX will also destroy any variables (macros) defined inside of a scope at the end of that scope, it will *also* undo any change which has been applied inside of that scope.

The value of \i is now 42.

```
\def\i{42}
{
    \def\i{43}
    \def\b{2}
}
The value of \textbackslash i is now \i.
```

The listing above defines `\i`, enters a local scope (a TEX "group") and changes `\i`. However, due to TEX's scoping rules, the old program state will be restored *completely* after returning from the local group! Neither the change to `\i` nor the definition of `\b` will survive. The same holds for register changes or other assignments.

TEX groups can be created in one of three ways: using curly braces[6], using `\begingroup` or using `\bgroup`. Curly braces are seldom used to delimit TEX groups because the other commands are more flexible. If one uses curly braces, they need to match up – it is forbidden to have unmatches curly braces.

`\begingroup`

Starts a new TEX group (a local scope). The scope will be active until it will be closed by `\endgroup`. The `\endgroup` command can occur later in the main token list.

`\endgroup`

Ends a TEX group which has been opened with `\begingroup`.

`\bgroup`

A special variant of `\begingroup` which can also be used to delimit arguments to `\hbox` or `\vbox` (i.e. it avoids the necessity to provide matched curly braces in this context).

The `\bgroup` macro is also useful to test whether the next following character is an opening brace (see `\futurelet`).

If one just needs to open a TEX group, one should prefer `\begingroup`.

[6] Or other tokens with the correct category code, compare [2].

\egroup

Closes a preceding \bgroup.

TEX does not know how to write into macros of an outer scope – except for the topmost (global) scope. This restriction is quite heavy if one needs to write complex structures: local variables should be declared inside of local groups, but changes to the structure should be written to the outer group. There is no direct possibility to do such a thing (except global variables).

2.4.1 Global Variables

TEX knows only "global" variables and "local" variables. A local variable will be deleted at the end of the group in which it has been declared. All values assigned locally will also be restored to their old value at the end of the group.

A global variable, on the other hand, maintains the same value throughout *every* scope. Usually, the topmost scope is the same as the one used for global variables: if you define anything in your TEX document, you add commands on global scope. It is also possible to explicitly make assignments or definitions in the global scope.

\global⟨*definition or assignment*⟩

The definition which follows \global immediately will be done globally.

```
{
    \global\def\a{123}
    \global\advance\count0 by3
    \global\toks0={34}
}
```

\globaldefs=-1|0|1 (initially 0)

I cite from [2]: "If the \globaldefs parameter is positive at the time of an assignment, a prefix of \global is automatically implied; but if \globaldefs is negative at the time of the assignment, a prefix of \global is ignored. If \globaldefs is zero (which it usually is), the appearance of nonappearance of \global determines whether or not a global assignment is made."

2.4.2 Transporting Changes to an Outer Group

There are a couple of methods to "transport" changes to an outer scope. Some are copy operations, some require to redo the the changes again after the end of the scope. All of them can be realized using expansion control.

Let's start with macro definitions which should be carried over the end of the group. I see the following methods:

- Copy the macro into a global, temporary variable (or even token register) and get that value after the scope.

```
\def\initialvalue{0}
{
    %  do something:
    \def\initialvalue{42}
    \global\let\myglobaltemporary=\initialvalue
}
\let\initialvalue=\myglobaltemporary
```

  The idea is that \myglobaltemporary is only used temporary; its value is always undefined and can be overwritten at any time. This allows to use a local variable \initialvalue.

  Please note that you should not use variables both globally and locally. This confuses TEX and results in a slow-down at runtime.

- "Smuggle" the result outside of the current group. I know this idea from the implementation of [4] written by Mark Wibrow and Till Tantau. The idea is to use several \expandafters and a \def to redefine the macro directly after the end of the group:

```
\def\smuggle#1\endgroup{%
    \expandafter\endgroup\expandafter\def\expandafter#1\expandafter{#1}%
}

\begingroup
    \def\variable{12}
    \edef\variable{\variable34}
    \edef\variable{\variable56}
    \smuggle\variable
\endgroup
```

The technique relies on groups started with `\begingroup` and ended with `\endgroup` because unmatched braces are not possible with `\def`. The effect is that after all those `\expandafter`s, TeX encounters the token list

`\endgroup\def\variable{123456}`

at the end of the group.

- Use the aftergroup stack. TeX has a special token stack of limited size which can be used to re-insert tokens after the end of a group. However, this does only work efficiently if the number of tokens which need to be transported is small and constant (say, at most three). It works by prefixing every token with `\aftergroup`, compare [2] for details.

Sometimes one needs to copy other variables outside of a scope. The trick with a temporary global variable works always, of course. But it is also possible to define a macro which contains commands to apply any required changes and transport that macro out of the scope.

2.5 More On TeX

This document is far from complete. I recommend reading about conditional expressions in [3] (german, online version) or [2] (bounded book). Hints about loops can be found in the manual of PGFPLOTS, [1] and the manual of PGF, [4]. Moreover, PGFPLOTS and PGF come with a whole lot of utility functions which are documented in the source `.code.tex` files.

Index

\advance, 3

\begingroup, 11
\bgroup, 11

\count, 2
\csname, 10

\def, 5, 9
\dimen, 2, 4
\divide, 3

\edef, 7, 9
\egroup, 12
\endgroup, 11
\expandafter, 6

\gdef, 10
\global, 12
\globaldefs, 12

\let, 9

\meaning, 5, 11
\message, 11
\multiply, 3

\noexpand, 8

\relax, 2

\string, 10

\toks, 3
\tracingcommands, 11
\tracingmacros, 11
\tracingrestores, 11

\xdef, 10

References

[1] C. Feuersänger. PGFPLOTS manual, March 31, 2010.

[2] D. Knuth. *Computers & Typesetting*. Addison Wesley, 2000.

[3] N. Schwartz. *Einführung in TEX (german!)*. Addison Wesley, 1991. Also available online at http://www.ruhr-uni-bochum.de/www-rz/schwanbs/TeX/ as .pdf.

[4] T. Tantau. Ti*k*Z and PGF manual. http://sourceforge.net/projects/pgf. $v. \geq 2.00$.