Notes On Programming in TEX

Dr. Christian FeuersÃ¤nger

cfeuersaenger@users.sourceforge.net

Revision 1.18.1 (2021/05/15)

Abstract

This document contains notes which are intended for those who are interested in TEX programming. It is valuable for beginners as a first start with a lot of examples, and it is also valuable 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 **2**

2 Programming in TEX **2**
- 2.1 Variables in Registers . . . 2
  - 2.1.1 Allocating Registers . . . 3
  - 2.1.2 Using More than 256 Registers . . . 4
- 2.2 Arithmetics in TEX . . . 4
- 2.3 Expansion Control . . . 5
  - 2.3.1 Macros . . . 6
  - 2.3.2 Token Registers . . . 10
  - 2.3.3 Summary of macro definition commands . . . 11
  - 2.3.4 Debugging Tools – Understanding and Tracing What TEX Does . . . 12
- 2.4 The Scope of a Variable . . . 12
  - 2.4.1 Global Variables . . . 13
  - 2.4.2 Transporting Changes to an Outer Group . . . 14
- 2.5 Branching . . . 15
  - 2.5.1 Boolean Variables . . . 16
  - 2.5.2 Special Cases for Conditionals . . . 16
- 2.6 Loops . . . 17
  - 2.6.1 Counting loops . . . 17
  - 2.6.2 Loops over list of items . . . 19
- 2.7 More On TEX . . . 19

3 Survey of Key–Value Handling using pgfKeys **19**
- 3.1 The Low–Level (Direct) API of PGFKeys . . . 20
- 3.2 The Standard API of PGFKeys . . . 22
  - 3.2.1 The Current Key Path . . . 24
  - 3.2.2 Key Filtering . . . 24

4 Special Tricks **24**
- 4.1 Handling # in Arguments . . . 24

Index **25**

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.

`\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. It is discarded.
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 1pt=65536sp=2^{16}sp. 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.

The value is now abcDEF.

```
\toks0={abc}%
\toks1={DEF}%
The value is now \the\toks0 \the\toks1.
```

Note the white space after \the\toks0: its purpose is to stop the number parsing when TEX scans for 0. The white space is discarded.

Token registers can also contain the special token # which would typically have a special meaning inside of macros:

```
\toks0={#1}%
\message{Meaning is \the\toks0}%
```

This outputs "Meaning is ##1" in your log file.

Token registers are mainly useful when it comes to fine grained expansion control and are discussed in more depth in Section 2.3.

2.1.1 Allocating Registers

There is a very limited number of registers. Consequently, one has to think carefully how to allocate them. Typical use–cases for registers are temporary variables (like some intermediate result) and long–living resources which are to be accumulated while the document or some part of it is to be generated.

It is clearly a bad idea to carelessly overwrite a register.

TEX comes with a single way to "allocate" registers:

\newdimen⟨*macroname*⟩
\newcount⟨*macroname*⟩
\newtoks⟨*macroname*⟩

These macros allocate a new register which is then accessable as ⟨*macroname*⟩.

The value is now 42.0pt.

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

```
\newdimen\variable

\variable=42pt

The value is now \the\variable.
```

The resulting ⟨*macroname*⟩ can now be used in the same way as if you used the register directly. In fact, it is often simpler because you do not need to worry about the register's number.

The allocation relies on some global integer variable which is increased for each allocation. This ensures that variables stored in such allocated variables do not accidentally overwrite the contents of some other variable.

Note that deallocation is impossible.

While it is perfectly reasonable to allocate long–living variables, one should avoid the allocation of a new variable just because one needs a "new" temporary variable.

It makes sense to allocate a couple of named variables like `\tempa`, `\tempb`, or something like that and reuse these values for every temporary evaluation. Clearly, care needs to be taken to avoid unintended overwrites.

It is also possible to use token registers as explained above. However, the usage should be protected by means of groups:

toks3 inside of group: Value inside of group
toks3 outside of group: Value outside of group

```
\toks3={Value outside of group}

\begingroup
\toks3={Value inside of group}
toks3 inside of group: \the\toks3
\endgroup

toks3 outside of group: \the\toks3
```

Groups constitute TEX's concept of "scope" and are explained somewhere else in this document.

2.1.2 Using More than 256 Registers

TEX on its own is restricted to 256 registers. However, you can manually activate "extended TEX mode" by using

```
\usepackage{etex}
```

early in your preamble. This is actually a very good idea: it allows access to 65536 registers. Today's documents which involve lots of packages actually need `etex`.

Note that even `etex` does not justify wild and uncontrolled allocated of registers just to store temporary variables.

If you want almost unlimited temporary variables, you should store the temporaries in macros. This, of course, involves conversion from numbers to string, but it is the only save way which avoids the limited number of 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.
```

It is impossible to divide by some non–integer number.

\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.
```

This is actually all that TEX allows. One needs powerful macro packages like PGF with its \pgfmathparse{⟨*expression*⟩} to do some "real" arithmetics.

Note that the limited number range of these registers also applies to the result of any numerical operation.

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

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 occurrence 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 occurrence 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'.

```
\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 execute 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 execute 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 enclosed 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:

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’}
```

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

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.

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

A shortcut for \global\def. It defines \macroname globally, independent 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, independent 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'.
```

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

[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 occurrence where a backslash would be inserted by TeX commands.

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.

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 unmatched 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`.

`\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}
}
```

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

\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 changes again after the end of the scope. All of them can be implemented 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 \expandafters, 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 Branching

Here we discuss some of the available branching constructions of TEX, with emphasis on conditions involving numbers and tokens.

ifnum⟨*count/integer number*⟩=⟨*count/integer number*⟩⟨*true-block*⟩\else⟨*false-block*⟩\fi

`\ifnum` compare integer numbers or integer registers (`\count` registers) and contains two branches, one is executed in the true case, the other in the case of false:

This is shown if above results to false.

```
\ifnum1=2 % this space is important.
This is shown if above were true.
\else
This is shown if above results to false.
\fi
```

Note that the `\else` with its ⟨*false-block*⟩ is optional.

ifdim⟨*dimen/fixed point number*⟩=⟨*dimen/fixed point number*⟩⟨*true-block*⟩\else⟨*false-block*⟩\fi

Similar to `\ifnum`, `\ifdim` compares two fixed point numbers or `\dimen` registers. The numbers must have a unit.

This is shown if above results to false.

```
\ifdim1pt=2pt % this space is important.
This is shown if above were true.
\else
This is shown if above results to false.
\fi
```

ifx⟨*token1*⟩⟨*token2*⟩⟨*true-block*⟩\else⟨*false-block*⟩\fi

`\ifx` is a bit more complex: It compares two *tokens* up to their first-level expansion.

This is shown if the two tokens have equal expansion.

```
\def\empty{\empty}
\ifx\empty\empty %
This is shown if the two tokens have equal expansion.
\else
This is shown if the two tokens expand to something different.
\fi
```

Here, we have defined a token `\empty` to be a replacement for `\empty` and subsequently have compared whether these two tokens are equal in first-level expansion. Note that the definition is actually nonsense. If TEX ever were to go through the whole expansion – i.e. we would put `\empty` somewhere else – it would do so indefinitely. However, with `\ifx` only first-level expansion is done and compared. Hence, the statement evaluates to true.

Have a look at the following example:

This is shown if the two tokens expand to something different.

```
\def\empty{\relax}
\ifx\empty\relax %
This is shown if the two tokens have equal expansion.
\else
This is shown if the two tokens expand to something different.
\fi
```

On first glance, this should evaluate to true: `\empty` is defined as a replacement for `\relax`. But it does not. Why?

`\empty` is expanded to `\relax`, however `\relax` expanded has a different meaning, namely stop scanning and not `\relax` anymore. Hence, they are different and the statement is false! If the expansion in `\ifx` were to be taken till maximum, both would be equal but not in the case of a comparison on first-level expansion only.

if⟨*token1*⟩⟨*token2*⟩⟨*true-block*⟩\else⟨*false-block*⟩\fi

The `\if` comparison is closely related to the `\ifx` conditional, with one major exception: it expands tokens until it finds the next two unexpandable tokens. If these two tokens are the same, it expands to the ⟨*true-block*⟩, otherwise to the ⟨*false-block*⟩.

The `\if` conditional should be handled with care as it might produce undesirable effects. Use it only if you know what you do.

A useful example is if you *know* that a macro contains at most one character, and you want to test for a particular one:

This is shown for all other choices.

```
\def\choice{a}
\if b\choice
  This is shown for the ‘b’ choice.
\else
  This is shown for all other choices.
\fi
```

iftrue⟨*true-block*⟩\else⟨*false-block*⟩\fi

A “conditional” which always invokes the ⟨*true-block*⟩.

iffalse⟨*true-block*⟩\else⟨*false-block*⟩\fi

A “conditional” which always invokes the ⟨*false-block*⟩.

2.5.1 Boolean Variables

`\newif`⟨*if-name*⟩

You can declare a new “boolean variable ‘`\ifsupermanmode` by means of `\newif\ifsupermanmode`. Afterwards, you can use the `\supermanmodetrue` and `\supermanmodefalse` switches to assign the boolean and `\ifsupermanmode` to check it.

The ⟨*if-name*⟩ has to start with `\if` (to support scans for nested `\if...\fi` pairs, see below).

2.5.2 Special Cases for Conditionals

Whenever you work with `\if`... and friends, you should know the following features:

1. `\if...\else...\fi` is expandable (including each of the single macros `\if...`, `\else` and `\fi`), which means you can even use it inside of `\edef`:

   We have now `temp=macro:->The choice is ‘a’`.

   ```
   \def\choice{a}
   \edef\temp{The choice is \if a\choice ‘a’\else not ‘a’\fi}
   We have now \texttt{\string temp=\meaning\temp}.
   ```

   The next token is ‘2’.3

   ```
   \def\shownexttoken#1{The next token is ‘\texttt{\string#1}’.}
   \def\mymacro{%
        \ifnum1=1 %
            \expandafter\shownexttoken%
        \fi%
   }%
   \mymacro 23
   ```

   This example is tricky. What would have happened without the `\expandafter`!? Well, `\shownexttoken` would be invoked with `#1=\fi`. This would lead to an error because the `\fi` would be missing, and it would spoil the effect since we do not want the `\fi` to be seen – we expected `#1=2`. The `\expandafter` first expands `\fi` (which simply removes the `\fi` without further effect) such that `\shownexttoken` will see the `2` token in our example above. This would also have worked if there was an `\else` branch instead of `\fi`.

2. You should generally make sure that the matching `\else` or `\fi` tokens are "directly reachable", i.e. without token expansion.

   The background here is that TEX works on a token–based level: Whenever it encounters an `\if...` statement, it evaluates it and scans tokens to find the matching end part (either an `\else` or an `\fi` token). But it will not expand tokens during this scan, although it will count nested `\if...\fi` pairs! Thus, if you are careless, it might become confused and your conditional will go awry.

2.6 Loops

As you have seen, in TEX we have a very specific control over token expansion. This makes it possible to construct even loops via means of recursion. In essence, a loop consist of the following parts:

- counter or, more generally, list of items
- incrementor, or more generally, a next item picker
- threshold or, more generally, an end list marker
- a check of the threshold or end marker, respectively

Reafing through the sections above, we realize that all of this is actually in place: We do know about counters, we do know about branching. Only the specifics of how to create these loops is still to be made clear. We will show both cases, the counting loop and the loop over a list of items in the following in detail.

In general, for a loop done via a recursion we need two definitions: One for the loop start and another for the loop step.

2.6.1 Counting loops

For a counting loop, we need a counter `\count0`, an incrementor `\advance`, a threshold `3` and a check `\ifnum\count=10` if the threshold has been reached.

The current value is '0'
The current value is '1'
The current value is '2'
The current value is '3'

```
\long\def\countingloop#1 in #2:#3#4{%
    #1=#2 %
    \loopcounter{#1}{#3}{#4}%
}

\long\def\loopcounter#1#2#3{%
    #3%
    \ifnum#1=#2 %
    \else%
        \advance#1 by1 %
        \loopcounter{#1}{#2}{#3}%
    \fi%
}
\countingloop{\count0} in 0:{3}{%
    The current value is '\the\count0'\par
}
```

There are some subtleties to the above example:

- We put a lot of % in the example. Why? Note that whenever TEX scans for a number – e. g. as in the case of `#1=#2` – it will continue scanning token by token, that is digit by digit, till he is sure that the number has ended, even over white space, and even expanding macros in case they themselves might not represent numbers again. Hence, % tells TEX to stop scanning. It is generally good practice to place % to tell TEX to stop scanning for more digits. However, there are some exceptions to it as well: In case of `\advance#1 by1` one should keep a white space in between, as well as in the case of `\ifnum#1=#2`.
- We placed the threshold `3` in `\countingloop{\count0} in 0:{3}` in curly brackets. Why? TEX otherwise will recognize only the token `1` if a threshold of e g. `10` is given and stumble over the now remnant 'extra' argument `0`. That is because a single letter represents a token to TEX. Hence, two

letters are two tokens and – ungrouped – become two arguments. Here, we have to group the threshold to make clear what we mean.

- One last thing becomes clear first when debugging is activated: As loops are done by recursion, i. e. by expansion followed by expansion till some threshold is reached, we will end with a lot of `\fi`s in the above case. If we place `\tracingmacros=2 \tracingcommands=2` before the `\countingloop` call and inspect the log file this will become apparent. This is bad because TEX will keep a stack frame open for each `\if`. . . `\fi` sequence. If we now have a loop over 10.000 items . . .
- It is not good practice to use one of the system counters, here `\count0`, because one can never be sure that is not used for something else or changed somewhere else. E. g. when the page is full, TEX will interrupt the current sequence of tokens to deal with creating a new page and finishing the old one, in this course changing `\count0`. Hence, we should also create our own counter.

Hence, we modify the example as follows:

The current value is '0'
The current value is '1'
The current value is '2'
The current value is '3'

```
\long\def\countingloop#1 in #2:#3#4{%
    #1=#2 %
    \loopcounter{#1}{#3}{#4}%
}

\long\def\loopcounter#1#2#3{%
    #3%
    \ifnum#1=#2 %
        \let\next=\relax%
    \else
        \advance#1 by1 %
        \def\next{\loopcounter{#1}{#2}{#3}}%
    \fi
    \next
}
\newcount\ourcounter

\countingloop{\ourcounter} in 0:{3}{%
    The current value is '\the\ourcounter'\par
}
```

Principally, nothing has changed in terms of the output. However, notice that we have introduced the macro `\next` which either recurses into the next level – but after the `\fi` statement has been given – or ends the recursion by simply containing `\relax`. Also, we have declared a new counter called `\ourcounter` that is safe from harm.

Finally, let us briefly summarize what happens in detail:

1. `\countingloop`. . . is expanded to an assignment `#1=#2` and another macro `\loopcounter`. . . .
2. The assignment is done: `\ourcounter` is set to the starting value `0`.
3. The actual loop macro is expanded to the command block – printing the current value – and an if statement.
4. The current value is printed.
5. `\ourcounter` is compared to the threshold `3` and . . .
   - False, i. e. the if statement is expanded to an `\advance` statement followed by defining `\next` to be another call of the same macro loop.
   - True, i. e. `\next` is set to be just `\relax`.
6. The statement is still false: `\advance` will increase `\ourcounter` by one, it is now `1`. `\next` is set to the loop macro.
7. The loop macro is again expanded, go to step 3. `\ourcounter` is . . . `2` . . . `\ourcounter` is `3`.
8. Now the statement is true: `\next` is expanded to `\relax` and nothing happens.

2.6.2 Loops over list of items

Looping over a list of items is very similar, only we will need `\ifx` in place of `\ifnum` and we need some end marker instead of the threshold value. However, how do we specify the list itself? Let's make some comma-separated list, e. g. `{a,b,c,d}` and call the end marker `\listingloopENDMARKER`.

The current item is 'a'
The current item is '
b'
The current item is '
c'
The current item is '
'
The current item is '
d'
The current item is '
e'

```
\def\listingloopENDMARKER{\par \listingloopENDMARKER}
\long\def\listingloop#1in#2#3{%
    \looppicker{#1}{#3}#2,\listingloopENDMARKER,%
}%
\long\def\looppicker#1#2#3,{%
    \def\tempitem{#3}%
    \ifx\tempitem\listingloopENDMARKER
        \let\next=\relax%
    \else
        \def#1{#3}%
        #2%
        \def\next{\looppicker{#1}{#2}}%
    \fi
    \next
}%
\listingloop\x in{a,b,c,,d,e}{%
    The current item is '\x'
}
```

Again, we make clear the subtleties contained therein:

- We have defined `\listingloopENDMARKER` to replace itself. This is possible because `\ifx` will only compare first-level expansion, see Section 2.5.
- We seem to miss a white space in ... `#1in#2`.... However, tokens are always ending with an additional white space as `\xin` is not equal to `\x in`. Hence, none is needed here and more than one white space would probably get gobbled.
- The definition `\looppicker#1#2#3,`... has three arguments but the recursive call `\looppicker{#1}{#2}` only gives two arguments!? This is the actual magic making this type of list possible! TeX is actually scanning beyond the scope of the given token to obtain the third argument. In effect, we are biting off piece by piece, list item by list item off the given list. All because we have stated an additional `,` – comma being the item separator – in the definition of the `\looppicker` macro. The expansion of the loop macro will always pick up one more item from the list concatenated to its end until it has reached the `\ENDMARKER`. This is added to the list's very end on the loop's start, and there it stops.

2.7 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.

3 Survey of Key–Value Handling using pgfKeys

One of the most important things for every TeX package is key–value input. There is a good overview and survey over different key–value packages, among them `xkeyval` and `pgfkeys`, in [5].

In addition to the paper mentioned above and the extensive reference manual for `pdfkeys` in [4], I give a brief survey over `pgfkeys` here. The addressed audience is primarily package writers or macro programmers. This section should allow you to define your own user interfaces and styles for PGFPLOTS and for PGF. It should also improve the understanding of `pgfkeys` and how it is to be used. I also address the topic of key filtering which is mainly useful for package writers.

The package `pgfkeys` is available as stand–alone package `\usepackage{pgfkeys}`. However, I believe that you never need to load it explicitly as PGF will be loaded anyway and PGF always loads `pgfkeys`.

It comes with two user interfaces. I believe that it is a best–practice to use the best of both worlds; although it might be sufficient to use just one of them. Consequently, I discuss both of them and propose a best–practices afterwards.

3.1 The Low–Level (Direct) Api of pgfKeys

Let us start with the low–level API of PGFKeys. It consists of a couple of macros which allow to define keys, assign values, and get their values back.

`\pgfkeyssetvalue{`⟨*/key path/key name*⟩`}{`⟨*value*⟩`}`

This macro (re)defines a key.

It is (almost) equivalent to a macro definition of sorts

`\expandafter\def\csname key@`⟨*/key path/key name*⟩`\endcsname{`⟨*value*⟩`};`

i.e. it stores ⟨*value*⟩ into a new macro such that the key can be looked up in constant time in TEX's hash map. Note that in contrast to other key–value packages like `xkeyval`, the low–level macro name which is used to store the value is *not* part of the PGFKeys API[7] – use `\pgfkeysgetvalue` and its friends to access the value.

The only limit for the number of possible keys is the size of TEX's hash map (which is very large).

You may have wondered what the slash '`/`' means. Users which are accustomed to PGF/PGFPLOTS know that there exists some kind of "key path" which qualifies ⟨*key name*⟩. The ⟨*key path*⟩ has the purpose of providing a name space such that many many keys with the same name can exist piecefully without ever touching another – provided the correct ⟨*key path*⟩ has been used. It can be seen as a (unix) file path: you can have many files with the same name, provided the files reside in different directories (i.e. have different paths).

You should *always* provide a key path, and it is highly recommended to use a different key path than just '`/`'.

The ⟨*value*⟩ can be anything; it is just stored. It can even contain `#`.

`\pgfkeysgetvalue{`⟨*/key path/key name*⟩`}{`⟨*\macro*⟩`}`

As you might have guessed, this macro allows to retrieve the value for some key and store it into ⟨*\macro*⟩.

Now that we have read about `\pgfkeyssetvalue` and `\pgfkeysgetvalue`, we can also provide an example:

The value of key `/notes/key` is 'abc'.

```
\pgfkeyssetvalue{/notes/key}{abc}

\pgfkeysgetvalue{/notes/key}\temp

The value of key \texttt{/notes/key} is '\temp'.
```

There is few magic around these two keys; it is just like a hashmap access with some special naming convention for the keys (due to the key path). Note that since "hashmap access" is what TEX does all the time when it handles macros, we could have replaced the pair `\pgfkeyssetvalue`/`\pgfkeysgetvalue` by `\def` and suitable `\let` commands, perhaps combined with `\csname...\endcsname`. The advantage of PGFKeys comes into play as soon as we inspect the high–level user interface in the next section.

Note that since `\pgfkeyssetvalue` is essentially the same as a suitable `\def`, the assignment is local to the current TEX group. In other words: the assignment will be undone by the next closing curly brace, or the next `\endgroup`, or the next `\end{`⟨*environment*⟩`}`.

[7]Note that `key@` is unrelated to PGFKeys.

\pgfkeyslet{⟨/key path/key name⟩}{⟨\macro⟩}

This is essentially the same as \pgfkeyssetvalue, except that the key's value is already available inside of ⟨\macro⟩:

The value of key **/notes/key** is 'abc'.

```
\def\something{abc}
\pgfkeyslet{/notes/key}{\something}

\pgfkeysgetvalue{/notes/key}\temp

The value of key \texttt{/notes/key} is `\temp'.
```

Just like \pgfkeyssetvalue boils down to \def, \pgfkeyslet boils down to \let.

\pgfkeysvalueof{⟨/key path/key name⟩}

This is essentially the same as \pgfkeysgetvalue{⟨/key path/key name⟩}{⟨\macro⟩} ⟨\macro⟩; i.e. it expands to the value stored in a key.

The value of key **/notes/key** is 'abc'.

```
\pgfkeyssetvalue{/notes/key}{abc}

The value of key \texttt{/notes/key} is `\pgfkeysvalueof{/notes/key}'.
```

However, this key has one major advantage: it can be used inside of an \edef (because it is fully expandable):

The value of key **/notes/key** along with dashes is — abc —.

```
\pgfkeyssetvalue{/notes/key}{abc}

\edef\temp{--- \pgfkeysvalueof{/notes/key} ---}

The value of key \texttt{/notes/key} along with dashes is \temp.
```

It boils down to a suitable \csname ... \endcsname. Consequently, it expands to \relax if the key happens to be undefined (see \pgfkeysifdefined below).

\pgfkeysdef{⟨/key path/key name⟩}{⟨macro body⟩}

This is a variant of \pgfkeyssetvalue. However, it has a substantial difference which appears to be unmotivated as long as we discuss the low–level API. It defines a so–called code–key.

Code–keys are executable macros. They take an argument, and they do something with it. "Assigning values" to such a key is equivalent to invoking ⟨*macro body*⟩ in a "suitable" way.

The result of this macro call is a new key named ⟨*/key path/key name/*⟩.@cmd. That key, in turn, is stored as executable macro. The macro is equivalent to the following definition (up to the name, of course):

\def\macro#1\pgfeov{⟨*macro body*⟩}.

This macro is stored (using \pgfkeyslet) under ⟨*/key path/key name/*⟩.@cmd.

We can use \pgfkeysgetvalue and/or \pgfkeysvalueof to access this special key[8], even though its use becomes more apparent later in this document:

We "assign a value" or "execute the code key" (which is equivalent):

Expansion with value abc—X.

```
\pgfkeysdef{/notes/code key}{Expansion with value #1---X.}%

We ``assign a value'' or ``execute the code key'' (which is equivalent):
\pgfkeysvalueof{/notes/code key/.@cmd}abc\pgfeov
```

Note that in this case, we *have* to use \pgfeov to terminate the argument list. We could have placed our argument into curly braces, but we have to provide \pgfeov; just as we had to add the suffix /.@cmd.

[8]Note that the suffix /.@cmd is part of the public API of PGFKeys, so it is no hackery to make use of it.

\pgfkeysifdefined{⟨/key path/key name⟩}{⟨true case⟩}{⟨false case⟩}
\pgfkeysifassignable{⟨/key path/key name⟩}{⟨true case⟩}{⟨false case⟩}

These keys provide conditionals based on existance or type of a key. Please refer to the reference manual in [4] for details.

3.2 The Standard Api of pgfKeys

Now that we have seen how things defined by PGFKeys can be accessed at a rather low level of abstraction, we will repeat the same using a higher level. This section explains the standard API of PGFKeys; this is how Keys can be defined and maintained easily, and it is also the end user interface.

PGFKeys addresses a couple of use–cases with its standard API:

1. simple key–value storage (i.e. put and get),
2. code–keys which can do some (complex) operation whenever the key is used,
3. configuration and modification of the key–value tool.

All of these items are possible with the same macro:

\pgfkeys{⟨comma–separated key–value pairs⟩}

This key constitutes the public API of PGFKeys. It accepts any number of key–value pairs, separated by commas.

We start with an example:

The value of key `/notes/key` is 'abc'.The value of key `/notes/key` is 'efg'.

```
% key definition:
\pgfkeys{
    /notes/key/.initial=abc,
}

The value of key \texttt{/notes/key} is '\pgfkeysvalueof{/notes/key}'.

% key usage:
\pgfkeys{
    /notes/key =  efg  ,
}

The value of key \texttt{/notes/key} is '\pgfkeysvalueof{/notes/key}'.
```

There are some items which appear to be clear, and I will briefly confirm that it really is clear: white spaces before and after the key name and before and after the value are stripped away. Furthermore, trailing commas are ignored. Note that trailing commas are a best–practice: always insert trailing commas. This simplifies the addition of further keys significantly (I can't remember how often I added a key and wondered why it was not properly recognised until I found the missing comma). Just add the trailing comma as a habit. Another good practice is to indent code properly, i.e. to insert a tab stop for every new line. It is also a good idea to provide one key per line, although all that stuff is optional.

The first thing which is strange when inspecting the actual code is the suffix '`/.initial`'. This is, in fact, a consistent new system of PGFKeys: these suffixes allow to configure and modify the keys to which them apply. They are called "key handlers". Whenever you encounter ⟨*key path/key name*⟩ followed by '`/.`⟨*handler*⟩', you can safely assume that ⟨*key path/key name*⟩ is about to be reconfigured or modified.

Knowledge of key handlers means control over PGFKeys. In the following, I will briefly discuss the most important handlers.

Key handler ⟨*key*⟩`/.initial`={⟨*value*⟩}

The key handler `/.initial` defines a new ⟨*key*⟩ and assigns its initial ⟨*value*⟩.

As such, it is equivalent to `\pgfkeyssetvalue`{⟨*key*⟩}{⟨*value*⟩}.

That a key which has been defined by means of `/.initial` can be set at any time later using a simple value assignment (see the example above).

Consequently, the first definition needs the suffix, all following assignments need to assignment to set the value.

Key handler ⟨*key*⟩/.code={⟨*body*⟩}

This key handler defines a new code–key ⟨*key*⟩ with ⟨*body*⟩ as result.

Execute the key using the simple API:Expansion with value abc—X.... execute the key using assignment in the standard API:Expansion with value abc—X.

```
\pgfkeys{
    /notes/code key/.code={Expansion with value #1---X.},
}%

Execute the key using the simple API:
\pgfkeysvalueof{/notes/code key/.@cmd}abc\pgfeov

... execute the key using assignment in the standard API:
\pgfkeys{/notes/code key=abc}
```

We see that assignment of a code key means to executing ⟨*body*⟩ where `#1` is set to the value assigned in the API.

A key defined by means of `/.code` is equivalent to one defined by means of `\pgfkeysdef`.

Note that the argument ⟨*body*⟩ can be surrounded by curly braces, but it does not need to be:

```
\pgfkeys{
    /notes/code key/.code={Expansion with value #1---X.},
    /notes/code key/.code=Expansion with value #1---X.,
}%
```

This is a common feature of PGFKeys: any kind of value assignment can use braces, but it does not need to. You only need to use curly braces if the assigned argument (in our ⟨*body*⟩) contains control characters of PGFKeys (i.e. `=` or `,`).

Key handler ⟨*key*⟩/.style={⟨*option list*⟩}

This key handler defines a new code–key ⟨*key*⟩ which sets all options in ⟨*option list*⟩ whenever it is assigned (used).

Styles are defined in a simple way: they simply invoke `\pgfkeys` with ⟨*option list*⟩ (well, almost – see below). However, they are very expressive in any kind of application.

Definition has been done. Assigning the style:OK. Value of A=42, value of B=42.

```
\pgfkeys{
    /notes/A/.initial=,
    /notes/B/.initial=,
    /notes/my style/.style={
        /notes/A={#1},
        /notes/B={#1},
    },
}%

Definition has been done. Assigning the style:
\pgfkeys{
    /notes/my style=42
}
OK. Value of A=\pgfkeysvalueof{/notes/A}, value of B=\pgfkeysvalueof{/notes/B}.
```

Our example is a very simple application of a style: it sets a bunch of other options.

Note that ⟨*option list*⟩ can depend on `#1`.

So far, this document did always provide fully qualified key paths. However `/.style` explicitly supports the notion of a "current key path": if a "current key path" is in effect, ⟨*option list*⟩ will be set in a context which also makes use of the same current key path. Technically, this means that `/.style` uses `\pgfkeysalso` to set ⟨*option list*⟩, i.e. it does not use `\pgfkeys` as claimed above.

\pgfkeysalso{⟨*comma–separated key–value pairs*⟩}

This macro is *almost* the same as `\pgfkeys{`⟨*comma–separated key–value pairs*⟩`}`. In fact, if any assignments in its argument use fully–qualified paths (as we did so far in this document), both invocations are equivalent.

The difference is how they treat keys which are *relative* to some current key path, a concept which will be explained in the next subsection.

Here is the difference between the macros: `\pgfkeys` resets the current key path to `/` before processing its argument whereas `\pgfkeysalso` does not change the current key path. Consequently, `\pgfkeysalso` is only useful inside of the body of some code–key (like `/.style`).

3.2.1 The Current Key Path

tbd

3.2.2 Key Filtering

tbd

4 Special Tricks

4.1 Handling # in Arguments

More than once, I encountered the following difficulty: I wanted to collect an argument which contains the hash sign, '`#`'. That's not particularly difficult, but it can lead to a lot of strange error messages when the resulting argument shall be processed! Consider

```
\def\collectargument#1{%
    \def\collectedcontent{#1}%
    \ifx\collectedcontent\empty
        It is empty.
    \else
        It is not empty, executing it: #1.
    \fi
}%

\collectargument{}% works

\collectargument{something}% works

\collectargument{% does not work!
    \def\something#1{which depends on #1}
}%
```

The code in this example is relatively simple: the `\collectargument` macro expects one argument and checks if it is empty (using `\ifx`, which is a common and reliable check for emptiness). It is is not empty, it executes it. The `\collectargument` macro works in most circumstances. More precisely: it works as long as there is *no* hash sign in its argument! In our example, the third call fails with "Illegal parameter number in definition of `\collectedcontent`." which occurs during the `\def\collectedcontent{#1}` line (and TEX has reasons for this message due to the special meaning of the parameter expansion).

The cure: redefine the `\collectargument` macro using

```
\def\collectargument#1{%
    \toks0={#1}%
    \edef\collectedcontent{\the\toks0}%
    \ifx\collectedcontent\empty
        It is empty.
    \else
        It is not empty, executing it: #1.
    \fi
}%
```

(you may want to allocate a temporary token register for this task). What is the difference? Well, the `\toks0={#1}` assignment introduces no special meaning for the hash sign `#`, and `\the\toks0` neither. Note, however, that this requires `\edef\collectedcontent` instead of `\def\collectedcontent` since the `\the` statement needs to be expanded. Everything works as expected.

Index

\advance, 4

\begingroup, 13
\bgroup, 13

.code handler, 23
\count, 2
\csname, 11

\def, 6, 11
\dimen, 3, 5
\divide, 5

\edef, 9, 11
\egroup, 13
\endgroup, 13
\expandafter, 7

\gdef, 11
\global, 13
\globaldefs, 14

.initial handler, 22

Key handlers
- .code, 23
- .initial, 22
- .style, 23

\let, 11

\meaning, 6, 12
\message, 12
\multiply, 5

\newcount, 3
\newdimen, 3
\newif, 16
\newtoks, 3
\noexpand, 10

\pgfkeys, 22
\pgfkeysalso, 23
\pgfkeysdef, 21
\pgfkeysgetvalue, 20
\pgfkeysifassignable, 22
\pgfkeysifdefined, 22
\pgfkeyslet, 21
\pgfkeyssetvalue, 20
\pgfkeysvalueof, 21

\relax, 2

\string, 12
.style handler, 23

\toks, 3
\tracingcommands, 12
\tracingmacros, 12
\tracingrestores, 12

\xdef, 11

References

[1] C. Feuersänger. PGFPLOTS manual, May 15, 2021.

[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`. $v. \geq 2.00$.

[5] J. Wright and C. Feuersänger. Implementing keyval input: an introduction. `http://pgfplots.sourceforge.net` as `.pdf`, 2008.