TEX in a nutshell

Petr Olšák

The pure TEX features are described here, no features provided by macro extensions. Only the last section gives a summary of plain TEX macros.

The main goal of this document is its brevity. So features are described only roughly and sometimes inaccurately here. If you need to know more then you can read free available books, for example TEX by topic or TEXbook naruby. Try to type `texdoc texbytopic` in your system.

OpTEX manual supposes that the user knows the basics principles of TEX itself. If you are converting from LATEX to OpTEX for example[1] then you may welcome a summary document that presents these basic principles because LATEX manuals typically don't distinguish between TEX features and features specially implemented by LATEX macros.

Table of contents

1 Terminology

The main principle of TEX is that its input files can be a mix of the material which could be printed and *control sequences* which gives a setting for build-in algorithms of TEX or gives a special message to TEX what to do with the inputted material.

Each control sequence (typically a word prefixed by a backslash) has its *meaning*. There are four types of meanings of control sequences:

- the control sequence can be a *register*, it means it represents a variable which is able to keep a value. There are *primitive registers*. Their values influence behavior of build-in algorithm (e.g. `\hsize`, `\parindent`, `\hyphenpenalty`). On the other hand *declared registers* are used by macros (e.g. `\medskipamount` used in plain TEX or `\ttindent` used by OpTEX).
- the control sequence can be a *primitive command*, it runs a build-in algorithm (e.g. `\def` declares a macro, `\halign` runs algorithm for tables, `\hbox` creates a box in typesetting output).
- the control sequences can be a *character constant* (declared by `\chardef` or `\mathchardef` primitive command) or a font selector (declared by `\font` primitive command).
- the control sequence can be a *macro*. When it is read then it is replaced by its *replacement text* in the input queue. If there are more macros in the replacement text, all macros are replaced

[1] Congratulations to your decision:-)

too. It is so called *expansion process* which ends on the level of text to be printed or primitive commands or registers or character constants or font selectors.

Example. When TEX reads:

```
\def\TeX{T\kern-.1667em\lower.5ex\hbox{E}\kern-.125emX}
```

in a macro file then the `\def` primitive command saves the information that `\TeX` is a control sequence with meaning "macro", the replacement text is declared here and it is a mix of a material to be typeset: `T`, `E` and `X` and primitive commands `\kern`, `\lower`, `\hbox` with their parameters in given syntax. Each primitive command has declared syntax, for example, `\kern` must be followed by dimension specification in the format "decimal number followed by a unit". More about this primitive syntax is in the sections 11, 12 and 13.

When a control sequence `\TeX` with meaning "macro" occurs in the input stream then it is *expanded* to its replacement text, i.e the sequence of typesetting material and primitive commands in our case of `\TeX` macro. The logo TEX is printed as a result of this processing.

None of the control sequences have their definitive meaning. The control sequence could change its meaning by re-defining it as a new macro (using `\def`), redeclaring it as an arbitrary object in TEX (using `\let`) etc. When you re-define a primitive control sequence then the access to its value or build-in algorithm is lost. This is a reason why OpTEX macros duplicate all primitive sequences (`\hbox` and `\_hbox`) with the same meaning and use only "private" control sequences (prefixed by `_`. So, user can re-define `\hbox` without the loss of the primitive command `\_hbox`.

2 Formats, engines

TEX is able to start without any macros preloaded in so-called *ini-TEX state* (the `-ini` option on the command line must be used). It knows only Cca 300 primitive registers and primitive commands at this state.[2] When ini-TEX reads macro files then new control sequences are declared as macros, declared registers, character constants or font selectors. The primitive command `\dump` saves the binary image of the TEX memory (with newly declared control sequences) to the *format file* (`.fmt` extension).

The original intention of existence format files was to prepare a collection of macro declarations, register settings and to load default fonts and to dump this information to a file for later use. Such collection typically declares macros for the markup of documents and for typesetting design. This is the reason why we call these files *format files*: they give a format of documents on the output side and declares markup rules for document source files.

When TEX is started without `-ini` option then it tries to load a prepared format file into its memory and to continue with reading more macros or a real document (or both). The starting point is at the place where `\dump` was processed during ini-TEX state. If the format file is not specified explicitly (by `-fmt` option on the command line) then TEX tries to read the format file with the same name which is used for running TEX. For example `tex document` runs TEX, it loads the format `tex.fmt` and reads the `document.tex`. Or `latex document` runs TEX, it loads the format `latex.fmt` and reads the `document.tex`.

The `tex.fmt` is format file dumped when *plain TEX macros*[3] were read and `latex.fmt` is format file dumped when *LATEX macros* were read. This is typically done when a TEX distribution is installed without any user intervention. So, the user can run `tex document` or `latex document` without worry that these typical format files exist.

From this point of view, LATEX is nothing more than a format of TEX, i.e. a collection of macro declarations and register settings.

[2] Roughly speaking, if you know all these 300 primitive objects and all syntax of primitive commands and all build-in algorithms then you know all about TEX. But starting to produce common documents from this primitive level without macro support is nearly impossible.

[3] Plain TEX macros were made by Donald Knuth, the author of TEX. It is a set of basic macros and settings which is used (more or less) as a subset of all other macro packages.

A typical TEX distributions have four common *TEX engines*, i.e. programs they implement classical TEX algorithms with various extensions:

- TEX – only classical TEX algorithms by Donald Knuth,
- pdfTEX– an extension supporting PDF output directly and micro-typographical features,
- XeTEX – an extension supporting Unicode and PDF output,
- luaTEX – an extension supporting Lua programming, Unicode, micro-typographical features and PDF output.

Each of them are able to run in ini-TEX state or with a format file. For example the command `luatex -ini macros.ini` starts luaTEX at ini-TEX state, read `macros.ini` file and final `\dump` command is supposed here to create a format `macros.fmt`. Then user can use the command `luatex -fmt macros document` to load `macros.fmt` and to process the `document.tex`. Or the command `luatex document` processes luaTEX with `document.tex` and with `luatex.fmt` which is a little extension of plain TEX macros. Another example: `lualatex document` runs luaTEX with `lualatex.fmt`. It is a format with LATEX macros for luaTEX engine. Final example: `optex document` runs LuaTEX with `optex.fmt` which is a format with OpTEX macros.

3 Searching data

If TEX needs to read something from the file system (for example the primitive command `\input` ⟨*file name*⟩ or `\font` ⟨*font selector*⟩ `=` ⟨*file name*⟩ is used) then the rule "first wins" is applied. TEX looks at the current directory first or somewhere to the TEX installation second. The behavior in the second step depends on the used TEX distribution. For example TEXlive programs are linked with a *kpathsea* library and they do the following: Search the given file in the current directory, then in `~/texmf` tree (data are saved by the user here), then in `texmf-local` tree (data are saved by the system administrator here, they are not removed when TEX distribution is upgraded), then in `texmf-var` tree (data are saved automatically by programs from TEX distribution here) and then in `texmf-dist` tree (data from TEXlive distribution). Each directory tree can be divided into sub-trees: first level `tex`, `fonts`, `doc` etc., second level is divided by TEX engines or font types etc., more levels are typically organized to keep a clarity. New files in the current directory on in `~/texmf` tree are found without doing something more, but new files in other places have to be registered by `texhash` program (TEX distribution does automatically this during its installation).

4 Processing the input

The lines from input files are transformed by the *tokenizer* first. It reads input lines and generates a sequence of tokens. The main goals of tokenizer is

- It converts each control sequence to a single token characterized by its name.
- Another input material is tokenized as "one token per character".
- The continuous sequence of more spaces is transformed into one space token.
- The end of the line is transfromed into space token, so the paragraph text can continue in a next input line and one space token is between the last word on the previous line and the first word on the next line.
- Comment character `%` is ignored and all the text after it to the end of line is ignored too.
- Spaces from the begining of each line are ignored. Thus, you can use arbitrary indentation in your source file without change of the result.
- Each empty line (or line with only spaces) is transformed to the token `\par`. This token has primitive meaning: "finalize the current paragraph". This implies the general rule in TEX source files: paragraphs are terminated by empty lines.

The behavior of the tokenizer is not definitive. Tokenizer works with a table of category codes. Any change of category codes of characters (done by `\catcode`\⟨character⟩=⟨code⟩` primitive command) influences tokenizer processing. For example, the verbatim environment is declared using setting all characters to normal meaning.

By default, there are the following characters with special meaning. Tokenizer converts them or sets them as special tokens used in syntactical rules in TEX later. The corresponding category codes are mentioned here as an index of the character.

- `\`$_0$ – starts completion of a control sequence by the tokenizer.
- `{`$_1$ and `}`$_2$ – opens and close group or have special syntactical meaning. The main syntactical rule is: each subsequence of tokens treated by macros or primitive commands must have these pairs o tokens balanced. There is no exception. Tokenizer treats them as special tokens with meaning "opening character" and "closing character".
- `%`$_{14}$ – comment character, removed by the tokenizer.
- `$`$_3$, `&`$_4$, `#`$_6$, `^`$_7$, `_`$_8$, `~`$_{13}$ – tokenizer treats them as a special tokens with meaning: "math-mode selector$_3$", "table separator$_4$", "parameter prefix for macros$_6$", "superscript prefix in math$_7$", prefix "subscript prefix in math$_8$" "active character$_{13}$" (the active character `~` is defined as no-breakable space in all typical formats).
- Letters and other characters are tokenized as "letter character$_{11}$" or "other character$_{12}$".

If you need to print these special characters you can use `\%`, `\&`, `\$`, `\#` or `\_`. These five control sequences are declared as "print this character" in all typical TEX formats. Another possibility is to use a verbatim environment (it depends on used format) Last alternative: you can use `\csstring\`⟨*character*⟩ in luaTEX, because luaTEX disposes with the primitive command `\csstring` which converts `\`⟨*character*⟩ to ⟨*character*⟩$_{12}$.

The "active character$_{13}$" can be declared by `\catcode`\⟨character⟩=13`. Such ⟨*character*⟩ behaves like ⟨*control sequence*⟩. For example, you can define it by `\def`⟨*character*⟩`{...}`. If a ⟨*control sequence*⟩ is listed in this document, the active character is meant too.

Each control sequence is built by the tokenizer starting from `\`$_0$. Its name is a continuous sequence of letters$_{11}$ finalized by first non-letter. Note that OpTEX sets `_` as letter$_{11}$, thus control sequence names can include this character. LATEX sets the `@` as letter$_{12}$ when reading styles and macro files. You can look to such files and you will see many such characters inside private control sequence names declared by LATEX macros.

If the first character after `\`$_0$ is not letter$_{\neq 11}$ then control sequence is finalized with only this character in its name. So called *one-character control sequence* is created. Other control sequences are *multiletter control sequences*.

Spaces ␣$_{10}$ after multi-letter control sequences are ignored, so the space can be used as a terminating character of the control sequence. Another characters used immediately after a control sequences are not ignored. So `\TeX !` and `\TeX!` gives the same result: the control sequence `\TeX` followed immediately by `!`$_{12}$.

Tokenizer's output (a sequence of tokens) goes to *expand processor* and its output goes to *main processor* of TEX. The expand processor performs expansions of macros or a primitive command which is working at expand processor level. See a summary of such commands in the section 12.

Main processor performs assignment of registers, declares macros by `\def` primitive command and runs all primitive commands at the main processor level. Moreover, it creates the typesetting output as described in the next section.

The very important difference between TEX and other programs is that there are no strings, only sequences of tokens. We can return to the example `\def\TeX{...}` above in the section 1. The token `\def` is a control sequence with meaning "declare a macro". It gets the following token `\TeX` and declares it as a macro with replacement text, which is the sequence of tokens:

T	\kern	-	.	1	6	6	7	e	m	\lower	.	5	e	x	\hbox	{	E	}	\kern	-	.	1	2	5	e	m	X
11		12	12	12	12	12	12	11	11		12	12	11	11		1	11	2		12	12	12	12	12	11	11	11

If you are thinking like TEX then you must forget the term "string" because all texts in TEX are preprocessed by the tokenizer when input lines are read and only sequences of tokens are manipulated inside TEX.

Tokenizer converts two ^7^7 characters followed by ASCII uppercase letter to the Ctrl-letter ASCII code. For example `^^M` is Ctrl-M (carriage return). It converts two ^7^7 followed by two hexadecimal digits (`0123456789abcdef`) to one-byte code, for example, `^^0d` is Ctrl-M too because it has code 13. Moreover, tokenizer of XETEX or luaTEX converts ^7^7^7^7 followed by four hexadecimal digits or ^7^7^7^7^7^7 followed by six hexadecimal digits to one character with given Unicode.

5 Vertical and horizontal modes

When the main processor creates the typesetting output then it alternates between vertical and horizontal mode. It starts in *vertical mode*: all materials are put vertically below in this mode. For example `\hbox{a}\hbox{b}\hbox{c}` creates c below b and b below a in vertical mode.

If there is something incompatible with vertical mode principle: a special command working only in horizontal mode or a character itself then main processor switches to *horizontal mode*: it opens an unlimited horizontal data line for typesetting material and puts material next to each other. For example `\hbox{a}\hbox{b}\hbox{c}` creates abc in horizontal mode.

When an empty line is scanned then tokenizer creates `\par` token here and if the main processor is in horizontal mode, the `\par` command finalizes the paragraph. More exactly it returns to the vertical mode, it breaks the horizontal data line filled in previous horizontal mode to parts with the `\hsize` width. These parts are completed as *boxes* and they are put one below second in the vertical mode. So, a paragraph of `\hsize` width is created.

Repeatedly: if there is something incompatible with current vertical mode (typically a character), then the horizontal mode is opened and all characters (and spaces between them) are put to the horizontal data line. When empty line is scanned then `\par` command is started and the horizontal data line is broken to the lines of `\hsize` width and next paragraph is completed.

In the vertical mode, the material is cumulated in a vertical data column called *main vertical list*. If the height of this material is greater than `\vsize` then its part with maximal `\vsize` height is completed as a *page box* and shipped to *output routine*. A programmer or designer can declare a design of pages using macros in the output routine: header, footer, pagination, the position of the main page box etc. The output routine completes the main page box with another material declared in the output routine and the result is shipped out as one page of the document. The main processor continues in the vertical mode with the rest of the unused material in the main vertical list. Then it can switch to the horizontal mode if a character occurs etc...

The plain TEX macro `\bye` (or primitive command `\end`[4]) starts the last `\par` command, finalizes the last paragraph (if any), completes the last page box, puts it to the output routine, finalizes the last page in it and TEX is terminated.

There are *internal vertical mode* and *internal horizontal mode*. They are activated when the main processor is typesetting a material inside `\vbox{...}` or `\hbox{...}` primitive commands. More about boxes is in sections 7 and 13.

Understanding of switching between modes is very important for TEX users. There are primitive commands which are context dependent on the current mode. For example, `\par` primitive command (generated by an empty line) does nothing in vertical mode but it finalizes paragraph in horizontal mode and it causes an error in math mode. Or `\kern` primitive command creates a vertical space in vertical mode or horizontal space in horizontal mode.

The following primitive commands used in the vertical mode starts the horizontal mode: the first character of the paragraph (most common situation) or `\indent`, `\noindent`, `\hskip` (and its alternatives), `\vrule`[5] and plain TEX macro `\leavevmode`. When the horizontal mode

[4] LATEX format re-defines this primitive control sequence `\end` to another meaning which follows the logic of LATEX's markup rules.

[5] The list is not fully completed, but most important commands are mentioned here.

is opened, the indentation of the \parindent width is included. The exception is only if the horizontal mode is started by the \noindent, then the paragraph has no indentation.

The following primitive commands used in the horizontal mode finalizes the paragraph and returns to the vertical mode: \par, \vskip (and its alternatives), \hrule, \end and plain TEX macro \bye.

6 Groups in TEX

Each assignment to registers, declaration macros or font selecting is local in groups. When the current group ends then the assignments done inside the group are forgotten and the values when this group was opened are restored. The groups can be delimited by { and } pair or by \begingroup and \endgroup primitive commands or by \bgroup and \egroup control sequences declared by plain TEX. For example, plain TEX declares the macros \rm (selects roman font), \bf (selects bold font) and \it (selects italics) and it initializes by \rm font. User can write:

```
The roman font is here {\it here is italics} and the roman font continues.
```

Not only fonts but all registers are set locally inside a group. The macro designer can declare a special environments with font selecting and with more special typographical parameters in groups.

The following example is a test of understanding vertical and horizontal modes switching.

```
{\hsize=5cm This is the first paragraph which should be formatted
           to 5 cm width.}

But it is not true...
```

Why the example above does not create the paragraph with a 5 cm width? The empty line (\par) command is placed *after* the group is finished, so the \hsize parameter has its previous value at the time when the paragraph is completed, no value 5 cm. The value of \hsize register[6] is used when the paragraph is completed, no at the beginning of the paragraph. This is the reason why macro programmers put explicitly \par command to macros before the local environment is finished by the end of the group. Our example should look like

```
{\hsize=5cm This is the first ... to 5 cm width.\par}
```

7 Box, kern, penalty, glue

You can look at one character, say the y. It is represented by three dimensions: height (above baseline), depth (below baseline) and width. Suppose that there are more characters printed in horizontal mode and completed to a line of a paragraph. This line has its height equal to the maximal height of characters inside it, it has the depth equal to maximal depth of all characters inside it and it has its width. Such a sequence of characters encapsulated to one typesetting element with its height, depth and width is called *box*. Boxes are placed next to each other (from left to right[7]) in horizontal mode or one above second in vertical mode.

The boxes can include individual characters or spaces or boxes. The boxes can include more boxes. Paragraph lines are boxes. The page box includes paragraph lines (boxes). The finalized page with a header, page box, pagination, etc. is a box and it is shipped out to the PDF page. Understanding boxes is necessary for macro programmers and designers.

You can create an individual box by the primitive command \hbox{⟨*horizontal material*⟩} or \vbox{⟨*vertical material*⟩}. The ⟨*horizontal material*⟩ is completed in internal horizontal mode

[6] and about twenty other registers which declare the paragraph design

[7] There is an exception for special languages.

and ⟨*vertical material*⟩ in internal vertical mode. Both cases open a group, create the material in a specified mode and closes the group, where all settings are local.

The ⟨*horizontal material*⟩ can include individual characters, boxes, horizontal *glues* or *kerns*. The glue is a special term for stretchable or shrinkable and possible breakable spaces and the kern is a term used for fixed nonbreakable spaces.

The ⟨*vertical material*⟩ can include boxes, vertical glues or kerns. No individual characters. If you put an individual character in the vertical mode (for example in the `\vbox`) then the horizontal mode is opened. At the end of `\vbox`[8] or when `\par` command is invoked, the opened paragraph is finished (with current `\hsize` width) and the resulting lines are vertically placed inside the `\vbox`.

The completed boxes are unbreakable and they are treaded as a single object in the surrounding printed material.

The line boxes of the paragraph have the fixed width `\hsize`, so there must be something stretchable or shrinkable in order to get desired fixed width of lines. Typically the spaces between words have this feature[9] These spaces have declared their *default size*, their *stretchability* and their *shrinkability* in the font metric data of currently used font.

You can place such glue explicitly by the primitive command:

`\hsize` ⟨*default size*⟩ `plus`⟨*stretchability*⟩ `minus`⟨*shrinkability*⟩
`for example:`
`\hsize 10pt plus5pt minus2.5pt`

This example places the glue with 10 pt default size, stretchable to 15 pt[10] and shrinkable to 7.5 pt as its minimal size. All glues in one line are stretched or shrank equally but with weights given from their stretchability/shrinkability values.

You can do experiments of this feature if you say `\hbox to`⟨*size*⟩`{...}`. Then the `\hbox` is created with a given width. Probably, the glues inside this `\hbox` must be stretched or shrank. You can see in the log that the total *badness* is calculated, it represents the amount of a "force" used to all glue included in such `\hbox`.

An infinitely stretchable (to an arbitrary positive value) or shrinkable (to an arbitrary negative value) glue can exist. This glue is stretched/shrank and other glues with finite amounts of stretching or shrinking keep their default size in such case. You can put infinitely stretchable/shrinkable glue using the reserved unit `fil` in `\hskip` command, for example the command `\hskip 0pt plus 1fil` means zero default size but infinitely stretchable. There is a shortcut for such glue: `\hfil`. When you type `\hbox to\hsize{\hfil` ⟨*text*⟩`\hfil}` then the ⟨*text*⟩ is centered. But if the ⟨*text*⟩ is wider than `\hsize` then TEX reports an `overfull \hbox`. If you want to center a wide ⟨*text*⟩ too, you can use `\hss` instead `\hfil`. The `\hss` primitive command is equal to `\hskip 0pt plus1fil minus1fil`. The ⟨*text*⟩ printed by `\hbox to\hsize{\hss`⟨*text*⟩`\hss}` is now centered in its arbitrary size.

A glue created with `fill` stretchability or shrinkability (double ell) is infinitely more stretchable or shrinkable than glues with only `fil` unit. So, glue with `fill` are stretched or shrank and glues with only `fil` in the same box keep their default size. For example, a macro declares centering a ⟨*text*⟩ by `\hbox to\hsize{\hss` ⟨*text*⟩`\hss}` and user can create the ⟨*text*⟩ in the form `\hfill` ⟨*real text*⟩. Then ⟨*real text*⟩ is printed flushed right because `\hfill` is a shortcut to `\hskip0pt plus1fill` and has greater priority than glues with only `fil` unit.

Common usage is `\hbox to0pt{`⟨*text*⟩`\hss}` or `\hbox to0pt{\hss`⟨*text*⟩`}`. The box with zero width is created and the text overlaps this dimension to right (first example) or to left (second example). Plain TEX declares macros for these cases: `\rlap{`⟨*text*⟩`}` or `\llap{`⟨*text*⟩`}`.

[8] before the `\vbox` group is closed

[9] When the microtypographical feature `\pdfadjustspacing` is activated, then not only spaces are stretchable and shrinkable but individual characters are slightly deformed (by invisible amount) too.

[10] It can be stretchable ad absurdum (more than 15 pt) but with very considerable *badness* calculated by TEX whenever glues are stretched or shrank.

The last line of each paragraph is finalized by a glue of type `\hfil` by default. When you write `\hfill` ⟨*object*⟩ in vertical mode (⟨*object*⟩ is something like a table, image or whatever else in the box) then ⟨*object*⟩ is flushed right, because the paragraph is started by the `\hfill` space but finalized only by `\hfil` space. If you type `\noindent\hfil` ⟨*object*⟩ then the ⟨*object*⟩ is centered. And putting only ⟨*object*⟩ places it to the left side because the common left side is default placement rule in the vertical mode.

The same principles concerned with horizontal glues are applicable for vertical modes where glues are created by `\vskip` commands instead `\hskip`. You can write `\vbox to`⟨*size*⟩`{...}` and do experiments.

If a glue is put to the horizontal data line in horizontal mode and paragraph breaking algorithm decides about the suitable breakpoints for creating lines with desired width `\hsize`, then each glue is potentially breakable point. Each glue can be preceded by a *penalty* value (created by `\penalty` primitive) in the typical range -10000 to 10000. The paragraph breaking algorithm gets a penalty if it decides to break line in the glue with given penalty value preceded. If no penalty is declared for a given glue, then it is the same as a penalty equal to zero. The penalty value 10000 or more means "impossible to break". The negative penalty means a bonus for paragraph breaking algorithm. The penalty -10000 or less means "you must break here".

The paragraph breaking algorithm tries to find an optimum of breakpoints positions concerning all penalties, to all badnesses of all created lines and to many more values not mentioned here in this brief document. The analogical optimal breakpoint is found in the vertical material when TEX breaks it into pages.

The concept "box, penalty, glue" with the optimum-fit breaking algorithms makes TEX unique between many other typesetting software.

8 Syntactical rules

The primitive commands can get their parameters written after it. These parameters must suit syntactical rules given for each primitive command. Some parameters are optional. For example `\hskip`⟨*dimen*⟩ `plus`⟨*stretchability*⟩ `minus`⟨*shrinkability*⟩ means that the parameter ⟨*dimen*⟩ must follow (it must suit syntactical rules for dimensions, see section 11) then the optional parameter prefixed by keyword `plus` can follow and then the optional parameter prefixed by `minus` can follow. We denote the optional parameters by underline in this document.

The *keywords* (typically prefixes some parameters) may have optional spaces around them.

If the syntactical rule mentions the pair `{`, `}` then these characters are not definitive: other characters may be tokenized with this special meaning but it is not common. The text between this pair must be *balanced* with respect to this pair. For example the syntactic rule `\message{`⟨*text*⟩`}` supposes that ⟨*text*⟩ must not be `ab{cd` but `ab{c{}}d` is allowed for instance.

By default, all parameters read by primitive commands are got from input stream tokenized and fully expanded by the expand processor. But sometimes, when TEX reads parameters for a primitive command, the expand processor is deactivated. We denote these parameters by red color. For example `\let`⟨*control sequence*⟩`=`⟨*token*⟩ means that these parameters processed by `\let` command are not expanded.

Whenever a syntactical rule mentions the `=` character (see the previous example with `\let` command, then this is the equal sign tokenized as a normal character and it is optional. The syntactical rule allows to omit it. Optional spaces are allowed around this equal sign.

The concept of the optional parameters of primitive commands (terminated if something different from the keyword follows) may bring troubles if a macro programmer forget to terminate an uncomplete parameter text by `\relax` command (`\relax` does nothing but it can terminate a list of optional parameters of the previous command). Suppose, for example, that `\mycoolspace` is defined by `\def\mycoolspace{\penalty42\hskip2mm}`. If an user write `first\mycoolspace plus second` then TEX reports the error `missing number, treated as zero` in the position of `s` character and appends: `<to be read again> s`. An user who is unfa-

miliar with TEX primitive commands and their parameters is totally lost. The correct definition looks like: `\def\mycoolspace{\penalty42\hskip2mm\relax}`.

9 Principles of macros

Macros can be declared by `\def` primitive command (or `\edef`, `\gdef`, `\xdef` commands, see below). The syntax is `\def` ⟨*control sequence*⟩ ⟨*parameters*⟩ `{` ⟨*replacement text*⟩ `}`.

The ⟨*parameters*⟩ are a sequence of formal parameters of the declared macro written in the form `#1`, `#2`, etc. They must be numbered from one and incremented by one. The maximal number of declared parameters is nine. These parameters can be used in the ⟨*replacement text*⟩. This specifies the place where the real parameter is positioned when the macro is expanded. For example:

```
\def\test #1{here is "#1".}
\test A        % expands to: here is "A".
\def\swap #1#2{#2#1}
\swap AB       % expands to: BA
\test {param}  % expands to: here is "param".
\swap A{param} % expands to: paramA
```

Note that there are two possibilities of how to write real macro parameters when a macro is in use. The parameter is one token by default but if there is `{` ⟨*something*⟩ `}` then the parameter is ⟨*something*⟩. The braces here are delimiters for the real parameter.

The example above shows a declaration of *unseparated parameters*. The parameters were declared by `#1` or `#1#2` with no text appended to such declaration. But there is another possibility. Each formal parameter can have a text appended in its declaration, so the general syntax of declaration of formal parameters is `#1` ⟨*text1*⟩ `#2` ⟨*text2*⟩ `;` etc. If such ⟨*text*⟩ is appended then we say that the parameter is *separated* or *delimited* by text. The same delimiter must be used when the macro is in use. For example

```
\def\Test #1#2..#3 {first "#1", second "#2", third "#3".}
\Test ABC..DEF G % expands to: first "A", second "BC", third "DEF".
                 % the letter G follows after expansion.
```

In the example above the `#1` parameter is unseparated (one token is read as a real parameter if not used the syntax `{` ⟨*parameter*⟩ `}`). The `#2` parameter is delimited by two dots and the `#3` parameter is delimited by space.

There should be a ⟨*text0*⟩ immediately before `#1` in the parameter declaration. This means that the declared macro must be used with the same ⟨*text0*⟩ immediately appended. If not, TEX reports the error. General rule for declaration a macro with three parameters should be: `\def` ⟨*control sequence*⟩ ⟨*text0*⟩ `#1` ⟨*text1*⟩ `#2` ⟨*text2*⟩ `#3` ⟨*text3*⟩ `{` ⟨*replacement text*⟩ `}`.

The rule "everything must be balanced" is applied to separated parameters too. It means that `\Test AB{C..DEF G}.. H` from the example above reads `B{C..DEF G}` to the `#2` parameter and the `#3` parameter is empty because the space (the delimiter of `#3` parameter) immediately follows two dots.

The separated parameter could bring a potential problem when the user forgot the delimiter or the delimiter is specified incorrectly. Then TEX reports the error. This error is reported when the first `\par` is scanned to the parameter (probably generated from an empty line). If you really want to scan to the parameter more paragraphs including `\par` between them then you can use `\long` prefix before `\def`. For example `\long\def\scan#1\stop{...}` reads the parameter of the `\scan` macro to the `\stop` control sequence and this parameter could include more paragraphs. If the delimiter is missing when `\long` defined macro is processed then TEX reports the error at the end of the file.

When a real parameter of a macro is scanned then the expand processor is deactivated. When the ⟨*replacement text*⟩ is processed then the expand processor works normally. It means that if parameters are used in ⟨*replacement text*⟩ then they are expanded here.

If a macro declaration is used inside ⟨*replacement text*⟩ of another macro then the number of # must be doubled for inner declaration. Example:

```
\def\defmacro#1#2{%
   \def#1##1 ##2 {##1 says: #1 ##2.}%
}
\defmacro \hello  {hello}  % expands to \def\hello#1 #2 {#1 says: hello #2.}
\defmacro \goobye {good bye}
\hello   Jane Eric         % expands to: Jane says: hello Eric.
\goodbye Eric John         % expands to: Eric says: good bye John.
```

Note the % characters used in the \defmacro definition. They mask the end of lines. If you don't use them then the space tokens are here (generated by tokenizer at the end of each line). The ⟨*replacement text*⟩ of \defmacro will be ⟨*space*⟩\def#1...{...}⟨*space*⟩ in such case. Each usage of \defmacro generates two unwanted spaces. It is not a problem if \defmacro is used in the vertical mode because spaces are ignored in this mode. But if \defmacro is used in horizontal mode then these spaces are printed[11].

The macro declaration behaves as another assignment, so the information about such declaration is lost if it is used in a group and the group is left. But you can use \global prefix before \def or the primitive \gdef. Then the assignment is global regardless of groups.

When \def or \gdef is processed then ⟨*replacement text*⟩ is read with deactivated expand processor. We have alternatives \edef (expanded def) and \xdef (global expanded def) which read their ⟨*replacement text*⟩ expanded by expand processor. The summary of \def syntax is:

```
\def ⟨control sequence⟩ ⟨parameters⟩{⟨replacement text⟩}   % local assignment
\gdef ⟨control sequence⟩ ⟨parameters⟩{⟨replacement text⟩} % global assignment
\edef ⟨control sequence⟩ ⟨parameters⟩{⟨replacement text⟩} % local assignment
\xdef ⟨control sequence⟩ ⟨parameters⟩{⟨replacement text⟩} % global assignment
```

You can set \tracingmacros=2 and you can see to log file how the macros are expanded.

10 Math modes

The $_3$⟨*math text*⟩$_3$ specifies a math formula inside the line of the paragraph. It processes the ⟨*math text*⟩ in a group and in *internal math mode*. The $_3$$_3$⟨*math text*⟩$_3$$_3$ generates a separate line with math formula(s). It processes the ⟨*math text*⟩ in a group and in *dislay math mode*.

The fonts in math mode are selected by very specific manner which is independent on current text font. Six different math object are automatically detected in math mode: \mathord (normal material), \mathop (big operators), \mathbin (binary operators), \mathrel (relations), \mathopen (open brackets), \mathclose (close brackets), \mathpunct (punctuations). They can be processed in four styles \displaystyle (default in the display mode), \textstyle (default in the internal math mode), \scriptstyle (used for indexes or exponents, more small text) and \scriptscriptstyle (used in indexes of indexes, smaller text).

The math algorithms were implemented to TEX by its author with very care. All typographical traditions of math typesetting were taken into account. There are three chapters about math typesetting in his TEXbook. Moreover, there is the detailed appendix G with exact specification of generating math formulas here. This topic is unfortunately out of the frame of this short text.

There is a good a piece of news: all formats (including LATEX) take default TEX syntax for ⟨*math text*⟩. So, LATEX manuals or LATEX documents serves a good source if you want to get to know the rules of math typesetting by TEX. There is only one significant difference. Fractions are constructed at the primitive level by the \over primitive: {⟨*nominator*⟩\over⟨*denominator*⟩} but LATEX uses a macro \frac in the syntax \frac{⟨*nominator*⟩}{⟨*denominator*⟩}. Plain TEX users (including the author of TEX) prefer the syntax which follows the principle "how a human reads

[11] More precisely, they are transformed into horizontal glues used between words.

the formula". On the other hand, the \frac syntax is derived from machine languages. You can define the \frac macro by \def\frac#1#2{{#1\over#2}} if you want.

11 Registers

There are four types of registers used in TEX:

- *Counters*, their values are integer numbers. Counters are declared by \newcount ⟨*register*⟩ [12] or they are primitive registers (\linepenalty for example). TEX interprets primitive commands which represent an integer from an internal table as counter type register too (examples: \catcode`A, \lccode`A).
- *Dimen type*, their values are dimensions. They are declared by \newdimen ⟨*register*⟩ or they are primitive registers (\hsize, for example). TEX interprets primitive commands which represent a dimension value as dimen type register too (example: \wd0).
- *Glue type*, their values are triples like in general \hskip parameters. They can be declared by \newskip ⟨*register*⟩ or they are primitive registers (\abovedisplayskip for example).[13]
- *Tokens lists*, their values are sequences of tokens. They are declared by \newtoks ⟨*register*⟩ or they are primitive registers (\everypar for example).

The following example shows, how registers are declared, how the value is saved to the register and how to print the value of the register.

```
\newcount  \mynumber
\newdimen  \mydimension
\newskip   \myskip
\newtoks   \mytoks
\mynumber = 42
\mydimension = -13cm
\myskip = 10mm plus 12mm minus1fil
\mytoks = {abCd ef}
To print these vaules use the primitive command "the":
\the\mynumber, \the\mydimension, \the\myskip, \the\mytoks.
\bye
```

This example prints: To print these values use the primitive command "the": 42, -369.88582pt, 28.45274pt plus 34.1433pt minus 1.0fil, abCd ef. Note that the human readable dimensions are converted to typographical points (pt).

The general syntactic rule for storing values to registers is ⟨*register*⟩ = ⟨*value*⟩ where the equal sign is optional and it can be surrounded by optional spaces. Syntactic rules for each type of ⟨*value*⟩ depending on type of the register (i. e. ⟨*number*⟩, ⟨*dimen*⟩, ⟨*skip*⟩ and ⟨*toks*⟩) follows.

- The ⟨*number*⟩ could be
  - a register of counter type;
  - a character constant declared by \chardef or \mathchardef primitive command.
  - an integer decimal number (with optional + or − prefixed)
  - " ⟨*hexa number*⟩ where ⟨*hexa number*⟩ can include digits 0123456789ABCDEF;
  - ' ⟨*octal number*⟩ where ⟨*octal number*⟩ can include digits 01234567;
  - ` ⟨*character*⟩ (the prefix is the reverse single quote `). It returns the code of the ⟨*character*⟩. Examples: `A or one-character control sequence `\A). Both examples represent the number 65. Unicode of the character are taken here if luaTEX or XeTEX are used;

[12] The declarators \newcount, \newdimen, \newskip and \newtoks are plain TEX macros used in all known TEX formats. They provides ⟨*address*⟩ allocation and uses the \count ⟨*address*⟩, \dimen ⟨*address*⟩, \skip ⟨*address*⟩ and \toks ⟨*address*⟩ TEX registers. The \countdef, \dimendef, \skipdef and \toksdef primitive commands are used internally.

[13] Very similar muglue type for math glues exists too but it is not described in this text.

- `\numexpr` ⟨*num. expression*⟩.[14] The ⟨*num. expression*⟩ uses operators `+`, `-`, `*` and `/` and brackets `(`, `)` in normal sense. The operands are ⟨*number*⟩s. It is terminated by something incompatible with the syntactic rule of ⟨*num. expression*⟩ or by `\relax`. If the result is non-integer, then it is rounded (no truncated).

- The ⟨*dimen*⟩ could be
  - a register of dimen type or counter type;
  - a decimal number with an optional decimal point (and optional `+` or `-` prefixed) followed by ⟨*dimen unit*⟩. The ⟨*dimen unit*⟩ is `pt` (point)[15] or `mm` or `cm` or `in` or `bp` (big point) or `dd` (Didot point) or `pc` (pica) or `cc` (cicero) or `sp` or `em` (quad of current font) or `ex` (ex height of current font) or a register of dimen type;
  - `\dimexpr` ⟨*dimen expression*⟩. The ⟨*dimen expression*⟩ uses operators `+`, `-`, `*` and `/` and brackets `(`, `)` in normal sense. The operands of `+` and `-` are ⟨*dimen*⟩s, the operators of `*` or `/` are the pair ⟨*dimen*⟩ and ⟨*number*⟩ (in this order). The ⟨*dimen expression*⟩ is terminated by something what is incompatible with syntactic rule of ⟨*dimen expression*⟩ or by `\relax`.
- The ⟨*skip*⟩ could be:
  - a register of glue type or dimen type or counter type;
  - ⟨*dimen*⟩ `plus` ⟨*generalized dimen*⟩ `minus` ⟨*generalized dimen*⟩. The ⟨*generalized dimen*⟩ is the same as ⟨*dimen*⟩, but normal ⟨*dimen unit*⟩ or pseudo-unit `fil` or `fill` or `filll` can be used.
- The ⟨*toks*⟩ could be
  - ⟨*expandafters*⟩`{`⟨*text*⟩`}`. The ⟨*expandafters*⟩ is typically a sequence of `\expandafter` primitive commands (zero or more). The ⟨*text*⟩ is scanned without expansion but the exception can be given by ⟨*expandafters*⟩.

The main processor reads input tokens (from the output of activated or deactivated expand processor) in two contexts: *do something* or *read parameters*. By default it is in the context *do something*. When a primitive which allows a parameters is read, main processor reads the parameters in the context *read parameters*.

Whenever the main processor reads a register in the context *do something* it assumes that an assignment of a value to the register is declared here. The following text (equal sign and ⟨*value*⟩) is read in the context *read parameters*. If the following text isn't compliant to the appropriate syntactic rule, TEX reports an error.

Examples of register manipulations:

```
\newcount\mynumber \newdimen\mydimension \newdimen\myskip
\hsize = .7\hsize  % see the rule for <dimen>, unit could be a register
\hoffset = \dimexpr 10mm - (\parindent + 1in) \relax % usage of \dimexpr
\myskip = 10pt plus15pt minus 3pt
\mydimen = \myskip    % the information "plus15pt minus 3pt" is lost
\mynumber = \mydimen  % \mynumber = 10*2^16  because \mydimen = 10*2^16 sp
```

Each dimension is saved internally as an integer multiple of `sp` unit in TEX. When we need a conversion ⟨*dimen*⟩ → ⟨*number*⟩, then simply the internal unit `sp` is omitted.

The summary of most commonly used primitive registers including their default value given by plain TEX follows.

- `\hsize=6.5in`, `\vsize=8.9in` are paragraph width and page height.
- `\hoffset=0pt`, `\voffset=0pt` give left margin and top margin of the page. They are calculated from *page origin* which is defined by coordinates `\pdfvorigin=1in` and `\pdfhorigin=1in` measured from left upper corner of the page.
- `\parindent=20pt` is the indentation of the first line of each paragraph.
- `\parfillskip=0pt plus 1fil` is horizontal glue added to the last line of the paragraph.

[14] This is a feature of eTEX extension. It is implemented in pdfTEX, XeTEX and luaTEX.
[15] 1 pt = 1/72.27 in ≐ 0.35 mm ; 1 pc = 12 pt ; 1 bp = 1/72 in ; 1 dd ≐ 1.07 pt ; 1 cc = 12 dd ; 1 sp = 2^{-16} pt = TEX accuracy.

- `\leftskip=0pt`, `\rightskip=0pt`. Glues added to each line in the paragraph from the left and the right side. If the stretchability is decared here, then the paragraph is ragged left/right.
- `\parskip=0pt plus 1pt` is vertical space between paragraphs.
- `\baselineskip=12pt`, `\lineskiplimit=0pt`, `\lineskip=1pt`. The `\baselineskip` *rule* says: Two consecutive lines in vertical list have baseline distance given by `\baselineskip` by default. The appropriate real glue is inserted between the lines. But if this real glue (between boxes) is less than `\lineskiplimit` then only `\lineskip` between boxes is inserted.
- `\topskip=10pt` is the distance between top of page box and the baseline of the first line.
- `\linepenalty=10`, `\hyphenpenalty=50`, `\exhyphenpenalty=50`, `\binoppenalty=700`, `\relpenalty=500`, `\clubpenalty=150`, `\widowpenalty=150`, `\displaywidowpenalty=50`, `\brokenpenalty=100`, `\predisplaypenalty=10000`, `\postdisplaypenalty=0`, `\interlinepenalty=0`, `\floatingpenalty=0`, `\outputpenalty=0`. These penalties are put to various places in the vertical or horizontal list. Most important are `\clubpenalty` (below the first line of a paragraph) and `\widowpenalty` (before last line of paragraph). Typographical rules give us to set these register to 10000 (no page break is allowed here).
- `\looseness=0` allows to create a "suboptimal" paragraph. The page-building algorithm tries to builds the paragraph with `\loosenes` lines more than the optimal solution. If the `\tolerance` has not sufficiently large value then this setting is simply ignored. It is reset to zero after each paragraph is completed.
- `\spaceskip=0pt`, `\xspaceskip=0pt`. If non-negative they are used as glues between words. Default values are read from the font metric data of the current font.
- `\pretolerance=100`, `\tolerance=200`, `\emergencystretch=0pt` `\doublehyphendemerits=10000`, `\finalhyphendemerits=5000`, `\adjdemerits=10000`, `\hfuzz=0.1pt`, `\vfuzz=0.1pt` are parameters for paragraph building algorithm not described here in detail.
- `\hbadness=1000`, `\vbadness=1000`. TEX reports a warning about badness on the terminal and to the log file if it is greater than these values. The warning has the form `underfull \hbox` or `underfull \vbox`. The value `100` means that the `plus` limit for glues is reached.
- `\tracingonline=0`, `\tracingmacros=0`, `\tracingstats=0`, `\tracingparagraphs=0`, `\tracingpages=0`, `\tracingoutput=0`, `\tracinglostchars=1`, `\tracingcommands=0`, `\tracingrestores=0`, `\tracingscantokens=0`, `\tracingifs=0`, `\tracinggroups=0`, `\tracingassigns=0`. If these registers have positive values then TEX reports details about the processing of build-in algorithms to the log file. If `\tracingonline>0` then the same output is on the terminal.
- `\showboxbreadth=5`, `\showboxdepth=3`, `\errorcontextlines=5`. The amount of information when boxes are traced to the log file or an error is reported.
- `\language=0`. TEX is able to load more hyphenation patters for more languages. This register points to the index of currently used hyphenation patterns. Zero means English.
- `\lefthyphenmin=2`, `\righthyphenmin=3`. Maximal letters left or right of hyphenated words.
- `\defaulthyphenchar=`\-`. This character is used when words are hyphenated.
- `\globaldefs=0`. If it is positive then all settings are global.
- `\hangafter=1`, `\hangindent=0pt`. If `\hangindent` is positive, then after `\hangafter` lines all following lines are indented. Negative/positive values of `\hangindent` or `\hangafter` does indentation from left or right and from the top or bottom of the paragraph. The `\hangindet` is set to 0 after each paragraph.
- `\mag=1000`. Magnification factor of all used dimensions. The value 1000 means 1:1.
- `\escapechar=`\\` use this character in the `\string` primitive.
- `\newlinechar=-1`. If positive, this character is interpreted as the end of the line when printing to the log or by `\write` primitive command.
- `\endlinechar=`^^M`. This character is appended to the end of each input line. Tokenizer converts it (Ctrl-M character) to the space token.

- `\time=now`, `\day=now`, `\month=now`, `\year=now`. The values about current time/date are set here when TEX starts to process the document. The `\time` counts minutes after midnight.
- `\overfullrule=5pt`. A rectangle to this width is appended after each overfull `\hbox`.
- `\mathsurround=0pt` is space inserted around formula from internal math mode.
- `\abovedisplayskip=12pt plus3pt minus9pt`, `\abovedisplayshortskip=0pt plus3pt`, `\belowdisplayskip=12pt plus3pt minus9pt`, `\belowdisplayshortskip=7pt plus3pt minus 4pt`. These spaces are inserted above and below the formula generated in math display mode.
- `\tabskip=0pt` is used by `\halign` primitive command for creating tables.
- `\output={\plainoutput}`, `\everypar={}`, `\everymath={}` `\everydisplay={}`, `\everyhbox={}` `\everyvbox={}` `\everycr={}`, `\everyjob={}`. These tokens lists are processed when algorithms of TEX reach a corresponding situation: opens output routine, paragraph, internal math mode, display math mode, `\vbox`, `\hbox`. Or it is at the end of the line in the table or it starts the job.

12 Expandable primitive commands

Notes about notation in this and the following sections. If the documented command is from eTEX extension (i. e. implemented in pdfTEX, XeTEX and luaTEX) then one * is prefixed. If it is from pdfTEX extension (implemented in XeTEX and luaTEX too) then two ** are prefixed. If it is a luaTEX only command then three *** are prefixed.

- `\string`⟨*control sequence*⟩ expand to an `\escapechar` (if positive) followed by the name of the control sequence. All characters of the output are "other characters$_{12}$", only spaces (if exist) are kept as space tokens $␣_{10}$.
- ***`\csstring`⟨*control sequence*⟩ works like`\string` but without `\escapechar`.
- *`\detokenize`⟨*expandafters*⟩`{`⟨*text*⟩`}` re-tokenizes all tokens in the text. Control sequences used in ⟨*text*⟩ are re-tokenized like `\string` primitive, spaces are tokens $␣_{10}$ and all other tokens are set as "other characters$_{12}$".
- `\the`⟨*register*⟩ expands to the value of the register. Examples were in previous section. The output is tokenized like the output of `\detokenize`. The exception is `\the`⟨*tokens register*⟩: the output is the value of the ⟨*tokens register*⟩ without re-tokenizing and expand processor does not expand this output in `\edef` `\write`, `\message` etc. arguments.
- `\scantokens`⟨*expandafters*⟩`{`⟨*text*⟩`}` re-tokenizes ⟨*text*⟩ using actual tokenizer setting. The behavior is the same as to write ⟨*text*⟩ to a virtual file and reading this file immediately.
- ***`\scantextokens`⟨*expandafters*⟩`{`⟨*text*⟩`}` is the same as `\scantokens` but removes problems with end-of-virtual-file.
- `\meaning`⟨*token*⟩ expands to the meaning of the ⟨*token*⟩. The text is tokenized like in the `\detokenze` output.
- `\csname`⟨*text*⟩`\endcsname` creates a control sequence with name ⟨*text*⟩. If it is not defined, then it gets the `\relax` meaning. For example `\csname TeX\endcsname` is the same as `\TeX`. The ⟨*text*⟩ must be expandable to characters only. Non-expandable control sequences (a primitive command at the main processor level, a register, a character constant, a font selector) are disallowed here. TEX reports the error `missing \encsname` when this rule isn't compliant.

  Example: `\csname foo:\the\mynumber\endcsname` expands to control sequence `\foo:42` if the `\mynumber` is a register with the value 42. Another example: macro programmer should implement key/value dictionary using this primitive:

```
\def\keyval #1 #2 {\expandafter\def\csname dict:#1\endcsnme{#2}}
\def\value #1 {\csname dict:#1\endcsname}
\kyval Peter 21 % key=Peter, value=21, saved to the dictionary
                % it does \def\dict:Peter{21}
\value Peter    % expands to \dict:Peter and it expands to 21.
```

- `\expandafer` ⟨*token 1*⟩ ⟨*token 2*⟩ does transformation ⟨*token 1*⟩ ⟨*expanded token2*⟩. The token processor will expand ⟨*token 1*⟩ after such transformation. The ⟨*expanded token2*⟩ is only the first level of expansion. For example, a macro is transformed to its ⟨*replacement text*⟩ but without expansion of ⟨*replacement text*⟩ at this time. Or `\csname...\endcsname` pair creates a control sequence but does not expand it at this time.

  If ⟨*token 2*⟩ is not expandable then `\expandafter` silently does nothing.

  The example above (the `\keyval` macro) shows usage of `\expandafter`. We need not define `\csname` by `\def`, we want to define a `\dict:key`. The `\expandafter` helps here.

  The ⟨*token 2*⟩ should be another `\expandafter`. We can see `\expandafter` chains in many macro files. For example `\expandafter A\expandafter B\expandafter CD` is processed as `ABC`⟨*expanded D*⟩.

  The ⟨*expandafters*⟩`{`⟨*text*⟩`}` syntax rule enables to prepare ⟨*text*⟩ using `\expandafter`(s). For example `\detokenize{\macro}` expands to $\backslash_{12}\mathtt{m}_{12}\mathtt{a}_{12}\mathtt{c}_{12}\mathtt{r}_{12}\mathtt{o}_{12}$. But if you need to detokenize the ⟨*replacement text*⟩ of the `\macro` then use `\detokenize\expandafter{\macro}`. Not only `\expandafter`s should be here. Expand processor does full expansion here until the opening brace $\mathtt{\{}_1$ is found.
- The general rule for all `\if*` commands is ⟨*if condition*⟩ ⟨*true text*⟩ `\else` ⟨*false text*⟩ `\fi`. The ⟨*if condition*⟩ is evaluated and ⟨*true text*⟩ or ⟨*false text*⟩ is skipped or processed depending on the result of ⟨*if condition*⟩. When the expand processor is skipping the text due to `\if*` command then it expands nothing in the skipped text. But it is noticing all control sequences with meaning `\if*`, `\else` and `\fi` during skipping in order to skip correctly all nested `\if*...\else...\fi` constructions.

  The following ⟨*if condition*⟩s are possible:

  ◦ `\if` ⟨*token 1*⟩ ⟨*token 2*⟩ is true if
  a) both tokens are characters with the same Unicode (or ASCII code in classical TEX) or
  b) both tokens are control sequences (with arbitrary meaning but not "the character") or
  c) one token is a character, second is a control sequence equal to the character (by `\let`) or
  d) both tokens are control sequences, their meaning (set by `\let`) is the same character code.
  Example: you can say `\let\test=a` then `\if\test a` returns true.
  ◦ `\ifx` ⟨*token 1*⟩ ⟨*token 2*⟩ is true if the meanings of ⟨*token 1*⟩ and ⟨*token 2*⟩ are the same.
  ◦ `\ifnum` ⟨*number 1*⟩ ⟨*relation*⟩ ⟨*number 2*⟩. The ⟨*relation*⟩ could be `<` or `=` or `>`. It returns true if the comparison of two numbers is true.
  ◦ `\ifodd` ⟨*number*⟩ returns true if the ⟨*number*⟩ is odd.
  ◦ `\ifdim` ⟨*dimen*⟩ ⟨*relation*⟩ ⟨*dimen*⟩ The ⟨*relation*⟩ could be `<` or `=` or `>`. It returns true if the comparison of two dimensions is true.
  ◦ `\iftrue` returns constantly true, `\iffalse` returns constanty false.
  ◦ `\ifhmode`, `\ifvmode`, `\ifmmode` – true if the current mode is horizontal, vertical, math.
  ◦ `\ifinner` returns true if the current mode is internal vertical, internal horizontal or internal math mode.
  ◦ `\ifhbox` ⟨*box number*⟩, `\ifvbox` ⟨*box number*⟩, `\ifvoid` ⟨*box number*⟩ returns true if the specified ⟨*box number*⟩ represents `\hbox`, `\vbox`, void box respectively.
  ◦ `\ifcat` ⟨*token 1*⟩ ⟨*token 2*⟩ is true if catogory codes of ⟨*token 1*⟩ and ⟨*token 2*⟩ equals.
  ◦ `\ifeof` ⟨*file number*⟩ is true, of the file attached to the ⟨*file number*⟩ by `\openin` primitive does not exist or the end of file was reached by the `\read` primitive.
- *`\unless` ⟨*if condition*⟩ negates the result of ⟨*if condition*⟩ before skipping or processing the following text.
- `\ifcase` ⟨*number*⟩ ⟨*case 0*⟩ `\or` ⟨*case 1*⟩ `\or` ⟨*case 2*⟩ `...` `\or` ⟨*case n*⟩ `\else` ⟨*else text*⟩ `\fi`. It processes the branch given by ⟨*number*⟩. It processes ⟨*else text*⟩ (or nothing if no ⟨*else text*⟩ is declared) when a branch with given ⟨*number*⟩ does not exist.
- `\noexpand` ⟨*token*⟩. The expand processor does not expand the ⟨*token*⟩ if it is expanding the text in `\edef`, `\write`, `\message` or similar lists.
- *`\unexpanded` ⟨*expandafters*⟩`{`⟨*text*⟩`}` returns ⟨*text*⟩ and applies `\noexpand` to all tokens in the ⟨*text*⟩.

- *\numexpr ⟨*num. expression*⟩, *\dimexpr ⟨*dimen expression*⟩. Documented in the ⟨*dimen*⟩ and ⟨*number*⟩ syntax rules in the section 11.
- \number ⟨*number*⟩, \romannumeral ⟨*number*⟩ prints ⟨*number*⟩ in decimal digits or as a roman numeral (with lowercase letters).
- \topmark (last from previous page), \firstmark (first on current page), \botmark (last on current page). They expand to the corresponding \mark included in the current or previous page-box. Usable for implementing running headers in the output routine.
- \fontname ⟨*font selector*⟩ expands to the file name ***(or font name) of the font given by its ⟨*font selector*⟩. The \fontname\font expands to the file name of the current font.
- \jobname expands to the name of the main file of this document (without extension .tex).
- \input ⟨*file name*⟩ ⟨*space*⟩ (classical TEX) or \input" ⟨*file name*⟩" or \input{ ⟨*file name*⟩} opens the given ⟨*file name*⟩ and starts to read input from it. If the ⟨*file name*⟩ doesn't exist then TEX tries to open again ⟨*file name*⟩.tex. If it doesn't exist even this, TEX reports an error. The alternative syntax with "..." or {...} allows having spaces in the file names.
- \endinput. The current line is the last line of the inputted file. File is closed and reading continues from the place where \input of this file was started. \endinput done in the main file causes future reading from the terminal and headache of users.
- ***\directlua {⟨*text*⟩} runs a Lua script given in ⟨*text*⟩.

13 Primitive commands at the main processor level

Commands used for declaration control sequences

- \def \edef \gdef \xdef were documented in the section 9.
- \long is prefix, it can be used before \def \edef \gdef \xdef. The declared macro accepts the control sequence \par in its parameters.
- *\private is prefix, it can be used before \def \edef \gdef, \xdef. The declared macro is not expanded by the expand processor in \write, \message, \edef etc. parameters.
- \outer is prefix, can be used before \def \edef \gdef, \xdef. The declared macro must be used only when the main processor is in the context *do something* or TEX reports an error.
- \global is a prefix, it can be used before each assignment (commands from this subsection and ⟨*register*⟩=⟨*value*⟩ settings). The assignment is global regardless of the current group.
- \chardef ⟨*control sequence*⟩=⟨*number*⟩, \mathchardef ⟨*control sequence*⟩=⟨*number*⟩ declares a constant ⟨*number*⟩. When the main processor is in the context "do something" and it gets a \chardef-ed control sequence, it prints the character with Unicode (ASCII code) ⟨*number*⟩ to the typesetting output. If it gets a \mathchardef-ed control sequence, it prints a math object (it works only in math mode, not documented here).
- \countdef ⟨*control sequence*⟩=⟨*number*⟩ declares ⟨*control sequence*⟩ as an equivalent to the \count ⟨*number*⟩ which is a register of counter type. The ⟨*number*⟩ here means an address in the array of registers of counter type. The \count0 is reserved for the page number. The macro programmer uses rarely direct addresses (1 to 9), more common is using the allocation macro \newcount ⟨*control sequence*⟩.
- \dimendef, \skipdef, \muskipdef, \toksdef followed by ⟨*control sequence*⟩=⟨*number*⟩ declare analogically equivalents to \dimen ⟨*number*⟩, \skip ⟨*number*⟩, \muskip ⟨*number*⟩ and \toks ⟨*number*⟩. Usage of allocation macros \newdimen, \newskip, \newmuskip, \newtoks are preferred.
- \font ⟨*font selector*⟩=⟨*file name*⟩ ⟨*space*⟩ ⟨*size specification*⟩ declares ⟨*font selector*⟩ of a font implemented in the ⟨*file name*⟩.tfm. The ⟨*size specifiaction*⟩ can be at ⟨*dimen*⟩ or scaled ⟨*factor*⟩. The ⟨*factor*⟩ equal to 1000 means 1:1. New syntax (supported by Unicode engines) is

```
\font ⟨font selector⟩="⟨font name⟩:⟨font features⟩" ⟨size specification⟩   % or
\font ⟨font selector⟩="[⟨font file⟩]:⟨font features⟩" ⟨size specification⟩
```

The ⟨*font file*⟩ is file name without `.otf` nor `.ttf` extension. The ⟨*font features*⟩ are font features prefixed by `+` or `-` and separated by semicolon. The `otfinfo -f` ⟨*file name*⟩`.otf` command (on command line) can list them. LuaTEX supports alternative syntax: `{...}` instead of `"..."`. Example: `\font\test={[texgyretermes-regular]:+onum;-liga} at12pt`.

- `\let`⟨*control sequence*⟩`=`⟨*token*⟩ sets to the ⟨*control sequence*⟩ the same meaning as ⟨*token*⟩ has. The ⟨*token*⟩ can be whatever, a character or a control sequence.
- `\futurelet`⟨*control sequence*⟩⟨*token 1*⟩⟨*token 2*⟩ works in two steps. In the first step it does `\let`⟨*control sequence*⟩`=`⟨*token 2*⟩ and in the second step ⟨*token 1*⟩⟨*token 2*⟩ is processed with activated token processor. Typically ⟨*token 1*⟩ is a macro that needs to know the next token.

Commands for box manipulation

- `\hbox{`⟨*cmds*⟩`}` or `\hbox to`⟨*dimen*⟩`{`⟨*cmds*⟩`}` or `\hbox spread`⟨*dimen*⟩`{`⟨*cmds*⟩`}` creates a box. The material inside this box is a ⟨*horizontal list*⟩ generated by ⟨*cmds*⟩ in the horizontal mode in a group. The width of the box is natural width of the ⟨*horizontal list*⟩ or ⟨*dimen*⟩ given by `to`⟨*dimen*⟩ parameter or it is spread by ⟨*dimen*⟩ given by `spread`⟨*dimen*⟩ parameter. The height of the box is the maximum of heights of all elements in the ⟨*horizontal list*⟩. The depth of the box is maximum of depths of all such elements. These elements are put to the common baseline (exceptions can be given by `\lower` or `\raise` commands).
- `\vbox{`⟨*cmds*⟩`}` or `\vbox to`⟨*dimen*⟩`{`⟨*cmds*⟩`}` or `\vbox spread`⟨*dimen*⟩`{`⟨*cmds*⟩`}` creates a box. The material inside this box is a ⟨*vertical list*⟩ generated by ⟨*cmds*⟩ in the vertical mode in a group. The height of the box is the natural height of the ⟨*vertical list*⟩ (eventually modified by values from `to` or `spread` parameters) without the depth of the last element. The depth of the last element is set as the depth of the box. The width of the box is the maximum of widths of elemens in the ⟨*vertical list*⟩. All elements are put to the common left margin of the box (exceptions can be given by `\moveleft` or `\moveright` commands).
- `\vtop{`⟨*cmds*⟩`}` (with optional `to` or `spread` parameters) is equal as `\vbox`, but the baseline of the resulting box goes through the baseline of the first element in the ⟨*vertical list*⟩ (note that `\vbox` has its baseline equal to the the baseline of the last element inside).
- `\vcenter{`⟨*cmds*⟩`}` (with optional `to` or `spread` parameters) is equal to `\vbox`, but its *math axis*[16] is exactly in the middle of the box. So its baseline is appropriately shifted. The `\vcenter` can be used only in math modes but given ⟨*cmds*⟩ are processed in vertical mode.
- `\lower`⟨*dimen*⟩⟨*box*⟩, `\raise`⟨*dimen*⟩⟨*box*⟩ move the ⟨*box*⟩ up or down by the ⟨*dimen*⟩ in horizontal mode. `\moveleft`⟨*dimen*⟩⟨*box*⟩, `\moveright`⟨*dimen*⟩⟨*box*⟩ move the ⟨*box*⟩ by the ⟨*dimen*⟩ in vertical mode.
- `\setbox`⟨*box number*⟩`=`⟨*box*⟩. TEX has a set of *box registers* addressed by ⟨*box number*⟩ and accessed via `\box`⟨*box number*⟩ or alternatives described below. The `\setbox` command saves given ⟨*box*⟩ to the register addressed by ⟨*box-number*⟩.

  Macro programmers use only 0 to 9 ⟨*box numbers*⟩ directly. Other addresses to box registers should be allocated by the `\newbox`⟨*control sequence*⟩ macro. The ⟨*control sequence*⟩ is equivalent to a ⟨*box number*⟩, not to the box register itself.

  The `\setbox` command does an assignment, so `\global` prefix is needed if you want to use the saved box outside the current group.
- `\box`⟨*box number*⟩ returns the box from ⟨*box number*⟩ box register. Example: you can do `\setbox0=\hbox{abc}`. This `\hbox` is not printed but saved to the register 0. At different place you can use `\box0` which prints `\hbox{abc}` or you can do `\setbox0=\hbox{cde\box0}` which saves the `\hbox{cde\hbox{abc}}` to the register 0.
- `\copy`⟨*box number*⟩ returns the box from ⟨*box number*⟩ box register and keeps the same box in this box register. Note that the `\box`⟨*box number*⟩ returns the box and empties the register ⟨*box number*⟩ immediately. If you don't want to empty the register, use `\copy`.
- `\wd`⟨*box number*⟩, `\ht`⟨*box number*⟩, `\dp`⟨*box number*⟩. You can measure or use the width, height and depth of a box saved in a register addressed by ⟨*box nuber*⟩. Examples

[16] Math axis is a horizontal line which goes through centers of + and − symbols. Its distance from baseline is declared in the math font metrics.

`\mydimen=\ht0`, `\hbox to\wd0{...}`. You can re-set the dimensions of a box saved in a register addressed by ⟨*box number*⟩. For example `\setbox0=\hbox{abc} \wd0=0pt \box0` gives the same result as `\hbox to0pt{abc}` but without the warning about overfull `\hbox`.

- `\unhbox`⟨*box number*⟩, `\unvbox`⟨*box number*⟩, `\unhcopy`⟨*box number*⟩, `\unvcopy`⟨*box number*⟩ do the same work as `\box` or `\copy` but they don't return the whole box but only it contents, i. e. a horizontal or vertical material. Example: try to do `\setbox0=\hbox{abc}` and later `\setbox0=\hbox{cde\unhbox0}` saves the `\hbox{cdeabc}` to the box register 0.

  The `\unhbox` and `\unhcopy` commands return the `\hbox` contents and `\unvbox`, `\unvcopy` commands return the `\vbox` contents. If incompatible contents is saved then TEX reports an error. You can test the type of saved contents by `\ifhbox` or `\ifvbox`.
- `\vsplit`⟨*box number*⟩ `to`⟨*dimen*⟩ does a column break. The ⟨*vertical material*⟩ saved in the box ⟨*box number*⟩ is broken to a first part of ⟨*dimen*⟩ height and the rest remains in the box ⟨*box number*⟩. The broken part is completed as `\vbox` which is the result of this operation. For example, you can say `\newbox\column \setbox\column=\vbox{...}` and later `\setbox0=\vsplit\column to5cm`. The `\box0` is `\vbox` with first 5cm of saved material.
- `\lastbox` returns the last box in the current vertical or horizontal material and removes it.

Commands for rules (lines in the typesetting output) and patterns

- `\hrule` creates a horizontal line in current vertical list. If it is used in a horizontal mode, it finishes the paragraph by `\par` first. `\hrule` `width`⟨*dimen*⟩ `height`⟨*dimen*⟩ `depth`⟨*dimen*⟩ creates (in general, with given parameters) a full rectangle (something like a box, but it isn't treated as the box) with given dimensions. Default values are: "width"=width of outer `\vbox`, "height"=0.4 pt, "depth"=0 pt.
- `\vrule` creates a vertical line in current horizontal list. If it is used in a vertical mode, it opens the horizontal mode first. `\vrule` `width`⟨*dimen*⟩ `height`⟨*dimen*⟩ `depth`⟨*dimen*⟩ creates (in general, with given parameters) a full rectangle with given dimensions. Default values are: "width"=0.4 pt, "height"=height of outer `\hbox`, "depth"=depth of outer `\hbox`.

  The optional parameters of `\hrule` and `\vrule` could be specified in arbitrary order and they can be specified more than once. In such a case, the rule "last wins" is applied.
- `\leaders`⟨*rule*⟩ ⟨*glue*⟩ crates a glue (maybe shrinkable or stretchable) filled by a full rectangle. The ⟨*rule*⟩ is `\vrule` or `\hrule` (maybe with its optional parameters). If the ⟨*glue*⟩ is specified by `\hskip` command (maybe with its optional parameters) or by its alternatives `\hss`, `\hfil`, `\hfill`, then the resulting glue is horizontal (can be used only in horizontal mode) and its dimensions are: width derived from ⟨*glue*⟩, height plus depth derived from ⟨*rule*⟩. If the ⟨*glue*⟩ is specified by `\vskip` command (maybe with its optional parameters) or by its alternatives `\vss`, `\vfil`, `\vfill`, then the resulting glue is vertical (can be used only in vertical mode) and its dimensions are: height derived from ⟨*glue*⟩, width derived from ⟨*rule*⟩, depth is zero.
- `\leaders`⟨*box*⟩ ⟨*glue*⟩ creates a vertical or horizontal glue filled by a pattern of repeated ⟨*box*⟩. The positions of boxes are calculated from the boundaries of the outer box. It is used for dots patterns in the table of contents. `\cleaders`⟨*box*⟩ ⟨*glue*⟩ does the same, but the pattern of boxes is centered in the space derived by the ⟨*glue*⟩. Spaces between boxes are not inserted. `\xleadres`⟨*box*⟩ ⟨*glue*⟩ does the same, but the spaces between boxes are inserted equally.

More commands for creating something in typesetting output

- `\par` closes horizontal mode and finalizes a paragraph.
- `\indent`, `\noindent`. They leave the vertical mode and open paragraph with/without paragraph indentation. If horizontal mode is current then `\indent` inserts an empty box of `\parindent` width, `\noindent` does nothing.
- `\hskip`, `\vskip`. They insert a horizontal/vertical glue. Documented in the section 7.
- `\hfil`, `\hfill`, `\hss`, `\vfil`, `\vfill`, `\vss` are alternatives of `\hskip`, `\vskip`.
- `\kern`⟨*dimen*⟩ puts nonbreakable horizontal/vertical space depending on the current mode.
- `\penalty`⟨*number*⟩ puts the penalty ⟨*number*⟩ to the current horizontal/vertical list.

- `\char` ⟨*number*⟩ prints the character with code ⟨*number*⟩. The "character itself" does the same.
- `\accent` ⟨*number*⟩ ⟨*character*⟩ places an accent with code ⟨*number*⟩ above the ⟨*character*⟩.
- `\␣` is control space. In horizontal mode, it inserts the space glue (like normal space but without modification by the `\spacefactor`). In vertical mode, it opens horizontal mode and puts the space. Note that normal space does nothing in vertical mode.
- `\discretionary{`⟨*pre break*⟩`}{`⟨*post break*⟩`}{`⟨*no break*⟩`}` works in horizontal mode. It prints ⟨*no break*⟩ in normal cases but if there is a line break then ⟨*pre break*⟩ is used before and ⟨*post break*⟩ after the breaking point. German Zucker/Zuk-ker (sugar) can be implemented by `Zu\discretionary{k-}{k}{ck}er`.
- `\-` is equal to `\discretionary{\char\hyphenchar`⟨*font*⟩`}{}{}`. The `\hyphenchar`⟨*font*⟩ is used as a hyphenation character. It is set to `\defaulthyphenchar` value when the font is loaded, but it can be changed.
- `\/` does an italics correction. It puts a little space if the last character is slanted.
- `\unpenalty`, `\unskip` removes last penalty / last glue from the current horizontal/vertical list.
- `\vadjust{`⟨*cmds*⟩`}`. It works in horizontal mode. The ⟨*cmds*⟩ must create a ⟨*vertcal list*⟩ and `\vadjust` saves a pointer to this list into the current horizontal list. When `\par` creates lines of the paragraph and distributes them to a vertical list, each line with the pointer from `\vadjust` has the corresponding ⟨*vertical list*⟩ immediately appended after this line.
- `\insert`⟨*number*⟩`{`⟨*cmds*⟩`}`. The ⟨*cmds*⟩ create a ⟨*verical list*⟩ and `\insert` saves a pointer to such ⟨*vertical list*⟩ into the current list. The output routine can work with such ⟨*vertical list*⟩s. The footnotes or *floating objects* (tables, figures) are implemented by the `\insert` primitive.
- `\halign{`⟨*declaration*⟩`\cr`⟨*row 1*⟩`\cr`⟨*row 2*⟩`\cr...\cr`⟨*row n*⟩`\cr}` creates a table of boxes in vertical mode. The ⟨*declaration*⟩ declares one or more column patterns separated by `&`$_4$. The rows use the same character to separate the items of the table in each row. The `\halign` works in two passes. First it saves all items to boxes and the second pass performs `\hbox to`*w* for each saved items, where *w* is the maximal width of items in each actual column.

  Detailed documentation of `\halign` is out of scope of this manual. Only one example follows: the macro `\putabove` puts `#1` above `#2` centered. The width of resulting box is equal to max of widths of these two parameters. The ⟨*declaration*⟩ `\hfil##\hfil` means that the items will be centered: `\def\putabove#1#2{\vbox{\halign{\hfil##\hfil\cr#1\cr#2\cr}}}`.
- `\valign` does the same as `\halign` but rows ↔ columns. It is not used commonly.
- `\cr`, `\crcr`, `\span`, `\omit`, `\noalign{`⟨*cmds*⟩`}` are primitives used by `\halign` and `\valign`.

Commands for registers calculation

- `\advance`⟨*register*⟩ by ⟨*value*⟩ does (formally) ⟨*register*⟩ = ⟨*register*⟩ + ⟨*value*⟩. The ⟨*register*⟩ is counter type or dimen type. The ⟨*value*⟩ is ⟨*number*⟩ or ⟨*dimen*⟩ (depending on type of the ⟨*register*⟩).
- `\multiply`⟨*register*⟩ by ⟨*number*⟩ does ⟨*register*⟩ = ⟨*register*⟩ * ⟨*number*⟩.
- `\divide`⟨*register*⟩ by ⟨*number*⟩ does ⟨*register*⟩ = ⟨*register*⟩ / ⟨*number*⟩. If the ⟨*register*⟩ is number type then the result is truncated.
- See *`\numexpr` and *`\dimexpr`, expandable primitives documented in the sections 11 and 12.

Internal codes

- `\catcode`⟨*number*⟩ is category code of the character with ⟨*number*⟩ code. Used by tokenizer.
- `\lccode`⟨*number*⟩ is lower case alternative to the `\char`⟨*number*⟩. If it is zero then lower case alternative doesn't exist (for example for punctuation). Used by `\lowercase` primitive and when breaking points are calculated from hyphenation patterns.
- `\uccode`⟨*number*⟩ is upper case alternative to the `\char`⟨*number*⟩. If it is zero, then the upper case alternative doesn't exist. Used by `\uppercase` primitive.
- `\lowercase`⟨*expandafters*⟩`{`⟨*text*⟩`}`, `\uppercase`⟨*expandafters*⟩`{`⟨*text*⟩`}` transform ⟨*text*⟩ to lowercase / uppercase using current `\lccode` or `\uccode` values. Returns transformed ⟨*text*⟩ where catcodes of tokens and tokens of type ⟨*control sequence*⟩ are unchanged.

- `\sfcode`⟨*number*⟩ is spacefactor code of the `\char`⟨*number*⟩. The `\spacefactor` register keeps (roughly speking) the `\sfcode` of the last printed character. The glue between words is modified (roughly speaking) by this `\spacefactor`. The value 1000 means factor 1:1 (no modification is done). It is used for enlarging spaces after periods and other punctuation in English texts.[17]

Commands for reading or writing text files

- Note, that main input stream is controlled by `\input` and `\endinput` expandable primitive commands documented in the section 12.
- `\openin`⟨*file num*⟩`=`⟨*file name*⟩ ⟨*space*⟩ (or `\openin`⟨*file num*⟩`={`⟨*file name*⟩`}`) opens the file ⟨*file name*⟩ for reading and creates a file descriptor connected to the ⟨*file num*⟩[18]. If the file doesn't exist nothing happens but macro programmer can test this case by `\ifeof`⟨*file num*⟩.
- `\read`⟨*file num*⟩`to`⟨*control sequence*⟩ does `\def`⟨*control sequence*⟩`{`⟨*replacement text*⟩`}` where the ⟨*replacement text*⟩ is tokenized next line from the file declared by `\openin` as ⟨*file num*⟩.
- `\openout`⟨*file num*⟩`=`⟨*file name*⟩ ⟨*space*⟩ (or `\openout`⟨*file num*⟩`="`⟨*file name*⟩`"`) opens the ⟨*file name*⟩ for writing and creates a file descriptor connected to ⟨*file num*⟩. If the file does exist, then its contents are removed.
- `\write`⟨*file num*⟩`{`⟨*text*⟩`}` writes a line of ⟨*text*⟩ to the file declared by `\openout` as ⟨*file num*⟩. But this isn't done immediately. TEX does not know the value of the current page when the `\write` command is processed because the paragraph building and page building algorithms are processed asynchronously. But a macro programmer typically needs to save current page to the file in order to read it again and to create a Table of contents or an Index.

  `\write`⟨*file num*⟩`{`⟨*text*⟩`}` saves ⟨*text*⟩ into memory and puts a pointer to this memory into the typesetting output. When the page is shipped out (by output routine), then all such pointers from this page are processed: the ⟨*text*⟩ is expanded at this time and its expansion is saved to the file. If (for example) the ⟨*text*⟩ includes `\the\pageno` then it is expanded to the correct page number of this page.
- `\closein`⟨*file num*⟩, `\closeout`⟨*file num*⟩ closes the open file. It is done automatically when TEX terminates its job.
- `\immediate` is the prefix. It can be used before `\openout`, `\write` and `\closeout` in order to do the desired action immediately (without waiting to the output routine).

Others primitive commands

- `\relax` does nothing. Used for terminating uncomplete optional parameters, for example.
- `\begingroup` opens group, `\endgroup` closes group. The $\{_1$ and $\}_2$ does the same but moreover, they are syntactic constructors for primitive commands and math lists (in math mode). These two types of groups (declared by mentioned commands or by mentioned characters) cannot be mixed, i. e. `\begingroup...}` gives an error. Plain TEX declares `\bgroup` and `\egroup` control sequences as an equivalent to $\{_1$ and $\}_2$. They can be used instead $\{_1$ and $\}_2$ when we need to open / close a group, to create a math list or when a box is constructed. For example `\hbox\bgroup`⟨*text*⟩`\egroup` is syntactically correct.
- `\aftergroup`⟨*token*⟩ saves the ⟨*token*⟩ and puts it back in the input queue immediately after the current group is closed. Then expand processor expands it (if it is expandable). More `\aftergroup`s in one group creates a queue of ⟨*token*⟩s used after the group is closed.
- `\afterassignment`⟨*token*⟩ saves the ⟨*token*⟩ and puts it back immediately after a following assignment (⟨*register*⟩`=`⟨*vaue*⟩, `\def` etc.) is done.
- `\lastskip`, `\lastpenalty` returns the value of last element in the current horizontal or vertical list if it is glue / penalty. It returns zero value if the element is not found as the last.
- `\ignorespaces` ignores spaces in horizontal mode until next primitive command occurs.

[17] This feature is not compliant with another typographical traditions, so `\frenchspacing` macro which sets all `\sfcodes` to 1000 is used very often.

[18] Note that ⟨*file num*⟩ is an addresses to the file descriptor. Macro programmers don't use these addresses directly but by `\newread`⟨*control sequence*⟩ and `\newwrite`⟨*control sequence*⟩ allocation macros.

- `\mark{`⟨*text*⟩`}` saves ⟨*text*⟩ to memory and puts a pointer to it in the typesetting output. The ⟨*text*⟩ is used as expansion output of `\firstmark`, `\topmark` and `\botmark` expansion primitives in output the routine.
- `\parshape`⟨*number*⟩ ⟨*I*1⟩ ⟨*W*1⟩ ⟨*I*2⟩ ⟨*W*2⟩ . . . ⟨*In*⟩ ⟨*Wn*⟩ enables to set arbitrary shape of the paragraph. The ⟨*number*⟩ declares the amount of data: the ⟨*number*⟩ pairs of ⟨*dimen*⟩s follow. The *i*-th line of the paragraph is shifted by ⟨*Ii*⟩ to right and its width is ⟨*Wi*⟩. The `\parshape` data are re-set after each paragraph to zero values (normal paragraph).
- `\special{`⟨*text*⟩`}` puts the message ⟨*text*⟩ to the typesetting output. It behaves as a zero-dimension pointer to ⟨*text*⟩ and it can be read by printer drivers. It is recommended to not use this old technology when PDF output is created directly.
- `\shipout`⟨*box*⟩ outputs the ⟨*box*⟩ as one page. Used in the output routine.
- `\end` completes the last page and terminates the job.
- `\dump` dumps the memory image to the `\jobname.fmt` and terminates the job.
- `\patterns{`⟨*data*⟩`}` reads hyphenation patterns for current `\language`.
- `\hyphenation{`⟨*data*⟩`}` reads hyphenation exceptions for current `\language`.
- `\message{`⟨*text*⟩`}` prints ⟨*text*⟩ on the terminal and to the log file.
- `\errmessage{`⟨*text*⟩`}` behaves like `\message{`⟨*text*⟩`}` but TEX treats it as an error.
- Job processing modes can be set by `\scrollmode` (don't pause at the errors), `\nonstopmode` (don't pause at the errors and missing files), `\batchmode` (`\nonstopmode` plus no output on the terminal). Default is `\errorstopmode` (stop at errors).
- `\inputlineno` includes the number of current line from current inputted file.
- `\show`⟨*control sequence*⟩, `\showbox`⟨*box number*⟩, `\showlists` and `\showthe`⟨*register*⟩ are tracing commands. TEX prints desired result on the terminal and to the log file and pauses.

Commands specific for PDF output (available in pdfTEX, XETEX and luaTEX)

- `\pdfliteral{`⟨*text*⟩`}` puts the ⟨*text*⟩ interpreted in a low level PDF language to the typesetting output. All PDF constructs defined in the PDF specification are allowed. The dimensions of the `\pdfliteral` object in the output are considered zero. So, if ⟨*text*⟩ moves the current typesetting point then the notion about its position from TEX point of view differs from the real position. A good practice is to close ⟨*text*⟩ to `q...Q` PDF commands. The command `\pdfliteral` is typically used for generating graphics and for linear transformation.
- `\pdfcolorstack`⟨*number*⟩ ⟨*op*⟩`{`⟨*text*⟩`}` (where ⟨*op*⟩ is `push` or `pop` or `set`) behaves like `\pdfliteral{`⟨*text*⟩`}` and it is used for color switchers. For example when ⟨*text*⟩ is `1 0 0 rg` then red color is selected. TEX sets the colors stack at the top of each page to the colors stack opened at the bottom of the previous page.
- `\pdfximage` `height`⟨*dimen*⟩ `depth`⟨*dimen*⟩ `width`⟨*dimen*⟩ `page`⟨*number*⟩`{`⟨*file name*⟩`}` loads the image from ⟨*file name*⟩ to the PDF output and returns the number of such data object in the `\pdflastximage` register. Allowed formats are PDF, JPG, PNG. The image is not drawn at this moment. Macro programmer can save `\mypic=\pdflastximage` and draw the image by `\pdfrefximage\mypic` (maybe repeatedly). Data of the image are loaded to PDF output only once. The `\pdfximage` allows more parameters, see pdfTEX documentation.
- `\pdfsetmatrix {`⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*b*⟩`}` multiplies the current transformation matrix (used for linear transformations) by `\matrix{`⟨*a*⟩`&`⟨*c*⟩`\cr` ⟨*b*⟩`&`⟨*d*⟩`}`.
- `\pdfdest name{`⟨*label*⟩`}`⟨*type*⟩`\relax` declares a destination of a hyperlink. The ⟨*label*⟩ must match with the ⟨*label*⟩ used in `\pdfoutline` or `\pdfstartlink`. The ⟨*type*⟩ declares the behavior of pdf viewer when the hyperlink is used. For example `xyz` means without changes of the current zoom (if not specified). Other types should be `fit`, `fith`, `fitv`, `fitb`.
- `\pdfstartlink` `height`⟨*dimen*⟩ `depth`⟨*dimen*⟩ ⟨*attributes*⟩ `goto name{`⟨*label*⟩`}` declares a begining of a hyperlink. A text (will be sensitive on mouse clik) immediately follows and it is terminated by `\pdfendlink`. The height and depth of sensitive area and the ⟨*label*⟩ used in `\pdfdest` are declared here. More parameters are allowed, see the pdfTEX documentation.

- `\pdfoutline goto name{`⟨*label*⟩`} count`⟨*number*⟩ `{`⟨*text*⟩`}` creates one item with ⟨*text*⟩ in PDF outlines. ⟨*label*⟩ must be used somewhere by `\pdfdest name{`⟨*label*⟩`}`. The ⟨*number*⟩ is the number of direct descentants in the outlines tree.
- `\pdfinfo {`⟨*key*⟩`(`⟨*text*⟩`)}` saves to PDF the information which can be listed by the command `pdfinfo` ⟨*file*⟩`.pdf` on the command line for example. More ⟨*key*⟩`(`⟨*text*⟩`)` should be here. The ⟨*key*⟩ can be `/Author`, `/Title`, `/Subject`, `/Kyewords`, `/Creator`, `/Producer`, `/CreationDate`, `/ModDate`. The last two keywords need a special format of the ⟨*text*⟩ value. All ⟨*text*⟩ values (including ⟨*text*⟩ used in the `\pdfoutline`) must be ASCII encoded or they can use a very special PDFunicode encoding.
- `\pdfcatalog` enables to set a default behavior of PDF viewer when it starts.
- `\pdfsavepos` saves an internal invisible point to the typesetting output. These points are processed when the page is shipped out: the numeric registers `\pdflastxps` and `\pdflastypos` get values about absolute position of this invisible point (measured from the left upper corner of the page in `sp` units). The macro programmer can follow `\pdfsavepos` by the `\write` command and save these absolute positions to a text file which can be read in the next run of TEX in order to get these absolute positions by macros.

Microtypographical extensions (available in pdfTEX, luaTEX and not all of them in XETEX)

- `\pdffontexpand` ⟨*font selector*⟩ ⟨*stretching*⟩ ⟨*shrinking*⟩ ⟨*step*⟩ declares a possibility to deform the characters from the font given by ⟨*font selector*⟩. This deformation is used when stretching or shrinking paragraph lines or doing `\hbox to{...}` in general. I. e. not only glues are stretchable and shrinkable. The numeric parameters are given in 1/1000 of the font size. ⟨*stretching*⟩ and ⟨*shrinking*⟩ are maximal allowed values. The stretching nor shrinking are not applied continuously but by given ⟨*step*⟩. To activate this feature you must to set the `\pdfadjustspacing` numeric register to a positive value.
- `\rpcode` ⟨*font selector*⟩ ⟨*char. code*⟩`=`⟨*number*⟩, `\lpcode` ⟨*font selector*⟩ ⟨*char. code*⟩`=`⟨*number*⟩ allows to declare hanging punctuation. Such punctuation is slightly moved to the right margin (if `\rpcode` is declared and the character is at the right margin) or to left margin (for `\lpcode` analogically). The ⟨*number*⟩ gives the amount of such moving in 1/1000 of the font size. To activate this feature you must to set `\pdfprotrudechars` to a positive value (2 or more means better algorithm).
- `\letterspacefont` ⟨*control sequence*⟩ ⟨*font selector*⟩ ⟨*number*⟩ declares a new font selector ⟨*control sequence*⟩ as a font given by the ⟨*font selector*⟩. Additional space declared by ⟨*number*⟩ is added between each two characters when the font is used. Tne ⟨*number*⟩ is 1/1000 of font size. Unicoded fonts support analogical `letterspace=`⟨*number*⟩ font feature.
- The same syntax like `\rpcode` have the following commands: `\knbscode` (added space after the character) `\stbscode` (added stretchability of the glue after the character) `\shbscode` (added shrinkability after the character) `\knbccode` (added kenr before the character) `\knaccode` (added kern after the character). To activate this feature you must to set `\pdfadjustinterwordglue` to a positive value. This feature is supported by pdfTEX only.

Commands used in math mode

- `\displaystyle`, `\textstyle`, `\scriptstyle`, `\scriptscriptstye` switches to specified style.
- `\mathord`, `\mathop`, `\mathbin`, `\mathrel`, `\mathopen`, `\mathclose`, `\mathpunct` followed by `{`⟨*math list*⟩`}` create an math object of given type.
- `{`⟨*nominator*⟩`\over`⟨*denominator*⟩`}` creates a fraction. The primitive commands `\atop` (without fraction rule), `\above`⟨*dimen*⟩ (fraction rule with given thickness) should be used in the same manner. The commands `\atopwithdelims`, `\overwithdelims`, `\abovewithdelims` allow to specify brackets around the generalized fraction.
- `\left`⟨*delimiter*⟩ ⟨*formula*⟩`\right`⟨*delimiter*⟩ creates a math ⟨*formula*⟩ and gives ⟨*delimiter*⟩s around it with an appropriate size (comparable with the size of the formula). The ⟨*delimiter*⟩s are brackets typically.

- The exponents and scripts are typically at the right side of the previous math object. But if this object is a "big operator" (summation, integral) then exponents and scripts are printed above and below this operator. The commands `\limits`, `\nolimits`, `\displaylimits` used before exponents and scripts constructors (`^`$_7$ and `_`$_8$) declares an exception from this rule.
- `$$` ⟨*formula*⟩ `\eqno` ⟨*mark*⟩ `$$` puts the ⟨*mark*⟩ to the right margin as `\llap{$` ⟨*mark*⟩ `$}`. Analogically `$$` ⟨*formula*⟩ `\leqno` ⟨*mark*⟩ `$$` puts it to the left margin.

14 Summary of plain TEX macros

Allocators

- `\newcount`, `\newdimen`, `\newskip`, `\newmuskip`, `\newtoks` folowed by a ⟨*control sequence*⟩ allocate a new register of given type and set it as the ⟨*control sequence*⟩. `\newbox`, `\newread`, `\newwrite` folowed by a ⟨*control sequence*⟩ allocate a new address to given data (to a box register or to a file descriptor) and set is as the ⟨*control sequence*⟩. All these allocation macros are declared as `\outer` in plain TEX, unfortunately. This brings problems when you need to use them in skipped text or in macros (in ⟨*replacement text*⟩ for example). Use `\csname newdimen\endcsname \yoursequence` in such cases.
- `\newif` ⟨*control sequence*⟩ sets the ⟨*control sequence*⟩ as a boolean variable. It must begin with `if`; for example `\newif\ifsomething`. Then you can set values by `\somethingtrue` or `\somethingfalse` and you can use this variable by `\ifsoemthing` which behaves like others `\if*` primitive commands.

Vertical skips

- `\bigskip` does `\vskip` by one line, `\medskip` does `\vskip` by one half of line and `\smallskip` does the vertical skip by one quarter of line. The registers `\bigskipamount`, `\medskipamount` and `\smallskipamount` are allocated for this purpose.
- `\nointerlineskip` ignores the `\baselineskip` rule for the following box in the current vertical list. This box is appended immediately after the previous box. `\offitnerlineskip` ignores the `\baselineskip` rule for all following boxes until the current group is closed.
- All vertical glues at the top of the page inserted by `\vskip` are ignored. Macro `\vglue` behaves like `\vskip` primitive command but its glue is not ignored at the top of the page.
- Sometimes we must switch off the `\baselineskip` rule (by `\offinterlineskip` macro for example). This is common in the tables. But we need to keep the baseline distances equal. Then the `\strut` can be inserted to each line. It is an invisible box with zero width and with height+depth=`\baselineskip`.
- `\normalbaselines` sets the registers for vertical placement `\baselineskip`, `\lineskip` and `\lineskiplimit` to default values given by the format. User can set other values for a while and then he/she can use `\normalbaselines`.

Penalties

- `\break` puts penalty -10000, so line/page break is forced here. `\nobreak` puts penalty 10000, so line/page break is disabled here. It should be preceded before a glue, which is "protected" by this penalty. `\allowbreak` puts penalty 0, it allows breaking similar as in normal space.
- `\goodbreak` puts penalty -500 in vertical mode, this is "recommended" point for a page break.
- `\filbreak` breaks the page only if it is "almost full" or if a big object (that doesn't fit the current page) follows. The bottom of such page is filled by a vertical glue, i. e. the default typographical rule about equal positions of all bottoms of common pages is broken here.
- `\eject` puts penalty -10000 in vertical list, i. e. it breaks the page.

Miscellaneous macros

- `\magstep` ⟨*number*⟩ expands to a magnification factor 1.2^x where x is given ⟨*number*⟩. This follows old typographical traditions that all sizes (of fonts) are distinguished by factors 1,

1.2, 1.44, etc. For example `\magstep2` expands to 1440, because $1.2^2 = 1.44$ and 1000 is factor 1:1 in TEX. The `\magstephalf` macro expands to 1095 which corresponds to $1.2^{(1/2)}$.

- `\nonfrenchspacing` sets special space factor codes (bigger spaces after periods, commas, semicolons etc.). This follows English typographical traditions. `\frenchspacing` sets all space factors as 1:1 (usable for non English texts).
- `\endgraf` is equavalent to `\par`, `\bgroup` and `\egroup` are equivalents to `{`$_1$ and `}`$_2$.
- `\space` expands to space, `\empty` is empty macro and `\null` is empty `\hbox{}`.
- `\quad` is horizontal space 1 em (size of the font), `\qquad` is double `\quad`, `\enspace` is kern 0.5 em `\thinspace` is kern 1/6 em and `\negthinspace` makes kern −1/6 em.
- `\loop` ⟨*body 1*⟩ ⟨*if condition*⟩ ⟨*body 2*⟩`\repeat` repeats ⟨*body 1*⟩ and ⟨*body 2*⟩ in a loop until ⟨*if condition*⟩ returns false. Then ⟨*body 2*⟩ is not processed and the loop is finished.
- `\leavevmode` opens a paragraph like `\indent` but it does nothing if the horizontal mode is opened already.
- `\line{`⟨*text*⟩`}` creates a box of line width (which is `\hsize`). `\leftline`, `\rightline`, `\centerline` do the same as `\line` but ⟨*text*⟩ is shifted left / right / is centered.
- `\rlap{`⟨*text*⟩`}` makes a box of zero size, the ⟨*text*⟩ is stuck outright. `\llap{`⟨*text*⟩`}` does the same and the ⟨*text*⟩ is pushed left.
- `\ialign` is equal to `\halign` but the values of registers used by `\halign` are set to default.
- `\hang` starts the paragraph where all lines (exception first) are indented by `\parindent`.
- `\texindent{`⟨*mark*⟩`}` starts a paragraph with `\llap{`⟨*mark*⟩`}`.
- `\item{`⟨*mark*⟩`}` starts the paragraph with `\hang` and with `\llap{`⟨*mark*⟩`}`. Usable for item lists. `\itemitem{`⟨*mark*⟩`}` can be used for the second level of items.
- `\narrower` sets wider margins for paragraphs (`\parindent` is appended to both sides), i. e. the paragraphs are narrower.
- `\raggedright` sets the paragraph shape with the ragged right margin. `\raggedbottom` sets the page-setting shape with the ragged bottoms.

Floating objects

- `\footnote{`⟨*mark*⟩`}{`⟨*text*⟩`}` creates a footnote with given ⟨*mark*⟩ and ⟨*text*⟩.
- `\topinsert` ⟨*object*⟩ `\endinsert` create the ⟨*object*⟩ as a *floating object*. It is printed on the top of the current page or on the next page. `\midinsert` ⟨*object*⟩ `\endinsert` does the same as `\topinsert` but it tries if the ⟨*object*⟩ fits on the current page. If it is true then it is printed to its current position, no floating object is created.

Controlling of input, output

- `\obeyspaces` sets the space as normal, i. e. it deactivates special treatment of spaces by tokenizer: more spaces will be more spaces and spaces from the left line are not ignored.
- `\obeylines` sets end of lines as `\par`. Each line in the input is one paragraph in the output.
- `\bye` finalizes the last page (or last pages if more floating objects must be printed) and terminates the TEX job. The `\end` primitive command does the same but without worrying about floating objects.

Macros used in math modes

- Spaces in math mode are `\,` (thin space), `\>` (medium space) `\;` (thick space, but still small), `\!` (negative thin space).
- `{`⟨*above*⟩ `\choose` ⟨*below*⟩`}` creates a combination number with brackets around it.
- `\sqrt{`⟨*math list*⟩`}` creates the square root symbol. `\root` ⟨*n*⟩ `\of{`⟨*math list*⟩`}` creates a general root symbol.
- `\cases{`⟨*case 1*⟩`&`⟨*condition 1*⟩`\cr...\cr`⟨*case n*⟩`&`⟨*condition n*⟩`}` creates a list of variants (preceded by the brace {) in the math mode.
- `\matrix{`⟨*a*⟩`&`⟨*b*⟩`...&`⟨*e*⟩`\cr...\cr`⟨*u*⟩`&`⟨*v*⟩`...&`⟨*z*⟩`}` creates a matrix of given values in math mode (without brackets around it). `\pmatrix{`⟨*data*⟩`}` does the same but with ().

- $$\displaylines{ ⟨formula 1⟩ \cr ... \cr ⟨formula n⟩ }$$ prints more (centered) formulae in the display mode.
- $$\eqalign{ ⟨form.1 left⟩ & ⟨form.1 right⟩ \cr ... \cr ⟨form.n left⟩ & ⟨form.n right⟩ }$$ prints more formulae aligned by & character in the display mode.
- \eqalignno behaves like \eqalign but second & followed by a ⟨mark⟩ can be in some lines. These lines have ⟨mark⟩ in the right margin. \leqalignno does the same as \eqalignno but ⟨mark⟩ is put to the left margin.

Index

Petr Olšák petr@olsak.net
Czech Technical University in Prague
Version of the text: 0.2 (2020/3/31)