How to Use TEX and $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

TEX is a program designed to do all the typesetting for you. Note that it is not an editor, you do not format your text directly. Rather, you type in the contents and specify using special TEX commands what should be done with this text (just like tags in HTML). Thus TEX is a language that describes how a text should look like when processed, and it is also a program that does the processing. It reads your source file and creates the appropriate document. This may sound complicated, but it is definitely worth it.

TEX does lots of things by itself and it is smart. It knows that the space after a period should be bigger than the one between words, it changes sizes of letters when they are used in fractions, it even does kerning (prints To instead of To) and ligatures (ff or fi instead of ff or fi). The output is exceptionally good, so good that no other program/editor can do better, which is amazing given that TEX was created in the 1970's. Moreover, TEX offers its user the possibility to further extend its capabilities by defining custom commands, many people use this to make routine tasks easier. Using TEX is easy (once you master it), so it is no surprise that it has found many applications even outside the scientific world.

While most users program some custom commands by themselves, many really useful improvements to TEX are offered as packages, files that contain definitions of new commands, a user can load them easily into their own text files. Some packages go so far that they even change the original TEX itself. Gradually two basic approaches appeared. Many people use the original TEX—called "plain TEX"—and augument its capabilities by packages that essentially preserve plain TEX. On the other hand, many people use LATEX, which is a package that totally changes the way TEX works. It is easier on the user, but it is harder to customize and it takes more work to write in it.

Here we will focus on the plain TEX and on one popular package called $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX (prepared by the American Mathematical Society). This package offers many improvements over the plain TEX when it comes to typesetting mathematical formulas. Unfortunately, it also makes some changes to the basic TEX, which is not exactly nice. Fortunately, these changes are small and a reader may not even encounter situations where things that work in the plain would not work in $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX.

$\mathcal{A}$ Thus most of the material below will apply both to plain and $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX, we will be careful to distinguish and warn where things are otherwise, paragraphs specific to $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX will be marked on the left like this.

This manual covers basic and intermediate TEXniques and even adds some bits for TEXperts. For a real mastery (for instance programming in TEX) we refer the reader to many thick books, for instance *The TEXbook* by Donald E. Knuth, the author of TEX.

Where do we get TEX? It has been conceived as a free product and various packages can be downloaded from the Internet. Many Linux distributions already contain TEX, for Windows there are several popular self-installing packages, for instance LiveTeX and MikTeX. All packages support foreign languages. While TEX itself allows the user to specify accented characters, this general way is rather complicated and it is better to support typed alphabets instead. For Czech there is a version of the basic `tex` program called `csplain` that can read and understand Czech characters. Coding depends on the system, Linux versions usually expect Iso-8859-2 (Iso Latin 2), while Windows versions naturally expect the Windows coding. Introduction is over, time to learn.

1. How to use TEX (general overview)
2. Just a text
3. And now math
4. Getting complicated in math (arranging objects)
5. Getting TEXnical 1 (sizes, global parameters, fonts)
6. Shaping text 1 (paragraphs, spaces, aligning)
7. Shaping text 2 (boxes, inserts, header/footer)
8. Getting lazy/efficient (macros, managing text)
9. More on macros
10. Registers
11. Running the Show
12. Getting TEXnical 2 (processing text)
Index
Bonus

1. How to use TEX (general overview)

You can write your text in any editor using the lower part of ASCII only (characters that you type directly on keyboard), for instance DOS-Edit or Notepad or SimpleText. You can even use programs like Word, just remember to save the file "as text". This flexibility is one of advantages of TEX, and it can work this way because you do not position elements precisely by hand as you would do in WYSIWYG editors (like Word), the way the appearance of your text in the source file is almost irrelevant to its final appearance.

It helps if the source file you type (we will call it "tex-file") has the extension `.tex`, but it is not completely necessary. Let the name be "mytext.tex". This file contains things that you want printed (the text) and specifications that describe special handling of this text (control sequences). For instance, to print the obligatory "Hello, world." it is enough to prepare a file with the line `Hello, world.\bye` in it. The resulting document is then constructed based on this input.

For this one uses a *TEX compiler*. Most likely it is something that you run from command line and usually it is called `tex`, so you would start it using `tex mytext` (Czech users will want to run `csplain mytext`). It expects the extension .tex, so if you called your file `mytext.txt`, you have to use `tex mytext.txt`. The TEX program goes through your tex-file and constructs the output: the shape, size and position of all characters. This information is stored in a so-called dvi-file, in this case it would be "mytext.dvi". Dvi stands for DeVice Independent and that is

exactly how the information looks like. The format is general but one cannot view it as it is. So the next step is visualizing it. By the way, the TEX compiler also stores information about its run in a log-file, called in this case `mytext.log`. It is extremely useful when fixing errors.

To see the output, you should choose a medium and then use an appropriate *device driver*, which is a program that takes the dvi-file and visualizes it the chosen way (your TEX package should supply these). Typically you would start by displaying the output on the screen, that is, you need a viewer. In DOS you could try calling `view mytext`, `xdvi mytext` should work in Linux, `windvi mytext` in Windows. You may also have an integrated TEX package that you start and then do your work (running TEX, viewing, printing) by clicking with your mouse.

Printing is also easy, in DOS you might try `print mytext`, under Linux you would use `dvips` and specify your printer (or translate dvi into ps for further processing), in WinDVI you click on a printer icon.

One usually repeats the cycle edit–tex–view many times before the text is ready for printing. First one repeats edit-tex to fix misprints and mistakes that actually bother TEX, that is, syntactic errors (wrong brackets, commands with typos etc). After TEX is satisfied and processes the file without complaining, it is time to look at it and polish it up visually.

It is rare to type a more substantial document without making some error (a typo or whatever). Thus it is very important to be able to decipher error messages. In principle it is very simple, you just have to ignore everything except the first and the last line. On the first line there is an exclamation mark plus a description of the error. The last line starts with the number of the line where the error occurred (so that you can find it easily in your tex-file), after this there is the good part of that line ending with the error. And then there is a question mark: TEX asks you what to do. Options are:

— Throw in the towel: type "x" and Enter, this stops TEX, you can correct the error and call TEX again, repeating the cycle.

— Leave it to TEX: just hit Enter. TEX tries the best it can, usually ignores the wrong part, sometimes it tries to fix the problem (in the processing procedure itself, not in the tex-file!), and you can proceed to the next error. Thus you can correct more errors within one edit–tex cycle, the error messages are all stored in the log-file.

— Ask for help: type "h" and Enter. Usually it does not help but quite frequently it is funny.

— help TEX: replace the wrong part with the correct one by typing "i" plus the new part and Enter. Again, this does not change the source tex-file, you still have to correct it with your editor afterwards, but it should help TEX to overcome this glitch and move on.

Actually, deciphering error calls can be quite hard, especially since some errors cause troubles later, so the line number shown by TEX is not necessarily the place where something is wrong. Experience helps a lot, we will discuss strategies for dealing with errors later, when we learn more about TEX.

That's all there is to it. Now how do we tell TEX what to do?

2. Just a text

In this chapter we will go through some general facts and learn how to typeset simple text without mathematical symbols.

File structure, characters.

The basic structure of a tex-file is like this:

text to typeset
`\bye`

$\mathcal{A}$ If we also want to use commands from $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX, we need to start the file with `\input amstex`.

The text part of our file consists of the text we want to typeset and of words that tell TEX how to format the supplied text. These are called *control sequences* and we will look at them shortly.

For writing the text itself we can use any key that we find on our keyboard, but some of these have a specific meaning. The *special characters* $ & % \ { } # ^ _ are not printed, they tell TEX to do things and we have to know when to use them. If we do want to print them, we will have to do it in a special way, see below.

$\mathcal{A}$ Note that if we use $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX, then also `@` becomes a special character (this is essentially the only text thing from the plain that $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX changes).

Thus when TEX obtains a text from us, it processes those special characters according to their nature (we will learn about it below), all remaining characters are simply read and arranged into lines, paragraphs, pages. There are two important rules:

— TEX counts spaces and Enter (perhaps followed by some spaces) as one space.

— TEX counts more blank lines as one blank line.

The first rule makes sense, some editors leave spaces at ends of lines and we do not want to see them in our document. Moreover, when we type a longer text over several lines, we want them joined as if words were connected by spaces. This however means that for making larger vertical spaces we will have to use some tricks.

So let's look at the first example. The following tex-files:

```
This is one line.\bye
```

and

```
This is one line.
\bye
```

and also

```
This is
                one line.
\bye
```

give the same output:

This is one line.

We did not start these examples with `\input amstex` because we did not really use any capabilities of the $\mathcal{AMS}$-TEX, but if we did, the outcome would have been the same. To save space, in all the following examples we will skip both the optional preamble `\input amstex` and the ending `\bye` and just show the relevant part of the source tex-file, printed using the different font above which looks like typewriter.

When TEX starts working on our file, it is in the so-called "text mode", which means that it is ready to typeset the document, presumably consisting of some text. This is the topic of this chapter, we will call this mode TM for short in this manual and it is the main mode, TEX starts with it and is supposed to end with it. If we want to input mathematics, then we have to switch (temporarily) TEX to the other mode, math mode, we will look into this in the next chapter.

Special characters.

If we type some of the special characters listed above, TEX will not print them, instead it will do something.

- `\` is the most important character in TEX. Backslash marks the beginning of "control sequences", which are commands that tell TEX what to do.
- `$` is used for typesetting mathematical text, it switches on and off the math mode.
- `%` causes that the rest of the line (including Enter) is ignored. If the source file reads

```
Thisisaverylo%nelyworld
ngword
```

the output is

Thisisaverylongword

since everything after `%` including Enter is ignored and no space is added into this word. This trick is useful when we need to type a long construction that does not contain spaces, but we want to break it into several lines in our tex-file to facilitate reading.

Typically we use `%` to start comments that explain more complicated parts of code, rows `%%%%%` also serve as a prominent section separator in the source file.

- { and } are used for the Creation of **groups**. These curly braces have to be paired and everything that is between them is considered to be one thing, one unit. We will see later what it is good for. These two characters do not produce anything on output, so something like this: ``{This {is{ a }very} {complicated} construction.}'' gives a simple output "This is a very complicated construction." As we just saw, we can open a group inside another group. Groups must be nested, a group opened inside another group must be also closed there. All groups must be closed when tex-file ends, that is, `\bye` cannot appear inside a group. There is also something special: an empty group {}.

Groups are very important, one uses them to restrict validity of some commands and also to supply information to control words.

- `^` and `_` can be used in math for putting things up and down.
- `&` and `#` are used when creating tables, we will get to it later, `#` also plays an important role in macro definitions.
- `@` is used in $\mathcal{AMS}$-TEX for special purposes.

Obviously there must be some way to get these characters printed, we just did it in this manual. For the answer see right below.

Control sequences.

Control sequences are commands that tell TEX what to do. They start with a backslash \. When TEX encounters it, it knows that it should not print anything, instead it reads on to recognize what command we put there and behaves accordingly. There are two kinds of control sequences:

Control symbols consist of a backslash and one character different from a letter. As an example we introduce a list of control symbols that allow us to print special characters:

Control symbol:	`\$`	`\&`	`\%`	`\#`	`\{`	`\}`	`\_`	`\ ` (\ and a space)
Output:	$	&	%	#	{	}	_	(a space)

So for instance "Walker & comp." can be obtained by typing `Walker \& comp.` One can also make larger spaces by repeated use of `\ `, `big \ \ \ space.` will print as big space. However, there are much better ways to make large spaces. Note that the curly braces only work within math mode (see below), so all braces { here were printed using `$\{$`. Note also that when we switch to the italic font (see below), the dollar sign becomes the pound sign, `{\it\$}` prints as £.

Talking of printing special characters, some authors like to show that a space should be typed using the mark ␣, for instance the control sequence above for space would be shown as `\␣`. It was printed using `{\tt\char'\ }`, obviously an advanced trick so we will not discuss it here.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX one also needs a command for typing @, not surprisingly it is `\@` (but naturally it would not work in plain). Another difference is that the control symbols for curly braces `\{` and `\}` also work in text mode in $\mathcal{AMS}$-TEX.

Control words consist of a backslash and an arbitrary number of letters, both lower- and upper-case (and the case matters).

After seeing the backslash, TEX checks on the next character. If it is not a letter, TEX knows that it must be a control symbol. Otherwise it starts reading characters until there is something different from a letter. That other character is not lost, TEX remembers that after processing the current command it should return to it. There is one important (and natural) exception to this rule, a space after a control word is ignored (hence more spaces plus probable Enter).

As an example we look at `\TeX`, which is a control word in place of which TEX prints the logo TEX. The input `\TeX, so what?` prints as TEX, so what? The program found the backslash and then read letters until it found that comma. TEX recognized "TeX" as a good control word and did what was needed. The "ending" character—comma—was remembered and printed. But if you write `\TeX is easy.`, the output will be TEXis easy. The space is "used up" by ending the control word. How can we add that space? Surely not by typing more spaces, even Enter would not help (and we cannot use two, see below). There are several possibilities:

— primitive but working: `\TeX \ is easy.`

— better: `\TeX\ is easy.` The second backslash is not ignored, together with the space after it prints as a space.

— another good one: `{\TeX} is easy.` Note that the closing brace is not ignored, hence the group is properly closed.

— neat trick: `\TeX{} is easy.`

There are many control sequences in TEX, in this manual we will go through the important ones and attempt to group them by their purpose. We started with commands for printing various symbols, now we add some more. We always list a symbol and after it comes the corresponding control sequence.

§ `\S`	† `\dag`	... `\dots`	© `\copyright`
¶ `\P`	‡ `\ddag`	/ `\slash`	

There are several kinds of control sequences. Some are commands that tell TEX to do something, but do not directly produce anything, for instance `\bye` tells TEX to finish the work. Many control sequences work as simple substitutions, they are replaced with some text (see our last two tables of symbols or `\TeX`). Finally, there are control words that work like switches, they modify the behaviour of TEX in a certain way but this modification can be also switched off, for instance by using them inside a group which then restricts their influence.

Some control sequences require an **argument** (or more arguments). In a typical case the argument is supplied as a group, but there are other possibilities. When TEX reads a control word that expects an argument, it then also reads the following characters (starting from the character that has just ended the control word), skipping spaces until a character that is not a space is found. If this character is {, TEX will read the whole group that has just been opened and use its contents as an argument. If this first character is \, then the control sequence that starts with it is used (that is, the *meaning* of this sequence). If none of this is the case, then the first non-space character itself is used as an argument. If there are more arguments needed, the whole game starts again.

As an example we introduce the control word `\underbar` which underlines things and expects one argument. The simplest case is `\underbar a` which produces a. We cannot write `\underbara`, since there is no control word "underbara". But we could have written `\underbar{a}` or `\underbar {a}`. We can write `\underbar3` to get 3, since 3 is not a letter and thus TEX correctly recognizes the end of the control word. It should not surprise us any more that `\underbar abc` will print abc, while `\underbar{abc}` or `\underbar {abc}` produce abc. Finally, `\underbar{\TeX}`, `\underbar\TeX` and `\underbar \TeX` give TEX.

The argument of `\underbar` cannot be longer than one line and the underlying line is always at the same level, for instance `\underbar x and \underbar y` gives x and y. We will see how to underline in a smarter way later.

If spaces after a control sequence are ignored, how can we have an argument that is a space? We use `\ ` or `{ }`. To get underlined space _ we can use `\underbar\ ` or `\underbar{ }`.

One important point: When an argument is a group then only its contents is taken by the control word. The full importance of this will be seen later on, now we just show a simple example. The control word `\nothing` takes one argument and simply places it in the text where used. For instance, `A \nothing{nice} trick` is interpreted

as `A nice trick`. If we use `A \nothing{\bf nice} trick`, it is interpreted as `A \bf nice trick` and printed as "A **nice trick**". To get just one word in boldface (see below) we have to make sure that even after the group delimiting the argument is stripped, there is still one left: `A \nothing{{\bf nice}} trick` is interpreted as `A {\bf nice} trick` and printed as "A **nice** trick". There are situations when this can make a difference.

However, many control words actually put groups of their own around the supplied argument because of the way they work, thus making it unnecessary to put double groups (for instance, `\centerline` below). Thus in many cases when we forget about this group stripping business we do not actually get anything wrong. On the other hand, it almost never hurts to put an extra group.

There is another thing called *keywords*. These are words that don't start with a backslash, so they can a normal part of a text, but when they appear at the right place at the right time they take on special meaning. For instance, the command for creating horizontal line `\hrule` expects dimensions, for that we use keywords. Example: `\hrule width .4 pt height 8 pt`. Here `width`, `height` and `pt` are not printed but used up by `\hrule`.

In general, the codes and cases of the letters forming keywords are irrelevant. Keywords are `at`, `bp`, `by`, `cc`, `cm`, `dd`, `depth`, `em`, `ex`, `fil`, `height`, `in`, `l`, `minus`, `mm`, `mu`, `pc`, `plus`, `pt`, `scaled`, `sp`, `spread`, `to`, `true`, `width`. Fortunately, in most cases a casual TEXer need not worry much about them.

Now we know all the basic things and we can go back to typesetting simple text.

Processing the text.

The text we write is read by TEX and split into lines, TEX does its best to fit the text into the given page width. First it tries to split a line between two words. It compares the length of the resulting line against the width of a page. If the difference is not too big, the spaces between words are shrunk or enlarged so that the line fits. If the difference is too big, TEX consults its huge dictionary of hyphenations and tries to split the word that sticks out, then does that comparing business again.

Some people feel the aligned right margin to be too much "computerized" or "impersonal". If you want to write a personal letter and you have the same feeling, put `\raggedright` before your text. TEX won't adjust sizes of spaces and you will get a nice ragged right margin. This control word works as a switch, so one can restrict its effect by using it within a group. For text typeset in `\tt` there is a special version `\ttraggedright`.

The created lines are arranged into paragraphs and these are broken into pages using similar tricks with spreading/shrinking of the spaces between paragraphs so that pages fit the prescribed size. Again, if we do not want alignment of bottoms of pages to the same level, there is a command for it, namely `\raggedbottom`. Another command, `\normalbottom`, tells TEX to align again.

The basic unit of text is a paragraph. We need to learn how to make them, we also look at simple ways of interfering with the automatic arranging process described above.

Simple formatting.

We can split the text into **paragraphs** using a blank line or the control word `\par`. These two are equivalent, TEX internally converts blank lines into `\par`'s. There is a rule that more blank lines (or `\par` and blank lines) are considered just one `\par`.

When this command is encountered, TEX leaves the current line unfinished and the text that comes afterwards starts a new paragraph, that is, the first line is printed with an indent as it was done in this very paragraph. Later we learn how to add some space between paragraphs or change the size of indentation (or even cancel indentation entirely).

Related commands `\smallskip`, `\medskip` and `\bigskip` end a paragraph like `\par`, moreover, they add some extra vertical space (small, medium and big) below this paragraph. This space is cancelled if it happens to appear at the bottom or top of a page. Similarly work `\smallbreak`, `\medbreak` and `\bigbreak`, but they also encourage TEX to choose this place for breaking a page if a pagebreak is near. Spaces created by these commands do not add; if we use several of these in a sequence, only the largest is taken.

We know that TEX has advanced mechanisms for breaking lines and pages, but they are not perfect. Fortunately, TEX also offers means for interfering with these automatic processes.

Line breaking: We can either force TEX to break a line, or suggest it, which we will come to later.

We may want to break text in unusual places for all kinds of reasons, often because it looks better or convey our thoughts better. Of course, we cannot use Enter for this purpose (it is understood as a space). When we want to break a line, we put the command `\break` and the line is broken immediately after this control word. The part of a line that is left behind is spread out to the right margin. Sometimes it is nice that way, but often it looks strange:

```
This is not nice.\break
```

gives

This is not nice.

If we want to avoid this spreading, we have to use `\hfil\break` (hfil is a stretchable space, see below). For instance, in the source-file for this manual, the previous paragraph ended `look strange:\hfil\break`; we could not use a blank line for starting a new paragraph there, since we did not want the next line to be indented.

$\mathcal{A}$ In $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX there is a convenient shortcut `\newline` for `\hfil\break`, many users of plain define macro `\newline` by themselves, we will see how to do it later. $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX also offers `\linebreak` as an equivalent of `\break`, it is not quite clear why.

One reason for manual breaking is when a long word (or another object like a table) appears at the end of a line and TEX does not know how to break it properly. Then TEX lets the long object stick out of a page, adds a black rectangle ("slug") after it, and also writes "Overfull \hbox ... at lines ... " on the terminal, but TEX does not stop its compilation there; it is not an error message, just a friendly warning. What can we do?

We can always break forcibly using `\break`, but this is a very bad idea, since we may later edit the text that comes before, thus moving this `\break` to places where it hurts. It is much better to just suggest a good place for breaking. When the offending object is a word, then usually it troubles TEX only because it does not know how to hyphenate it (for instance, the word "database" is not in the hyphenation database). We can tell TEX that this word can be hyphenated using the "discretionary hyphen" `\-`. This control symbol does not occur in the output, but if the word is at the end of a line, TEX knows that it can be broken at that place. The word `data\-base` would be normally printed as "database", but it could be split into "data" and "base". If some word is used frequently, we can fix its hyphenation once for the whole file using `\hyphenation{`*words with hyphens separated by spaces*`}` somewhere at the beginning of a tex-file. For instance, `\hyphenation{da-ta-ba-se da-ta-link}` would teach TEX how to hyphenate these words (but only within this tex-file).

$\mathcal{A}$ $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX had the control word `\allowlinebreak` that does the same as `\-`.

Note that TEX's hyphenation algorithm was developed to handle English, for other languages it often does not work well and localizations have to take this into account. Czech users that call `csplain` for processing their files can switch on Czech hyphenation by putting `\chyph` at the beginning of the file.

If the offending object is not a word but some larger structure, we can suggest that a break is possible somewhere in it. The command `\allowbreak` tells TEX that at this very place a line can be broken if needed. Note that then the object is simply broken, no extra mark is made, so this is not suitable for conditional breaking of words. `Data\allowbreak base` would be normally printed as "database" or broken at the end of a line, but without the hyphen.

Of course, we can always try to rearrange our text so that the object moves to someplace safe.

On a related topic, the slash `/` is often used to connect two words, but TEX does not break lines at that point, so expressions like "input/output" may cause trouble. `\slash` looks exactly like /, but a line can be broken at that point. We would therefore prefer `input\slash output`.

Conversely, sometimes we do not want a linebreak somewhere. If we have an object (say, a word) and we do not want it broken, we can enclose it into a horizontal box (see below), for instance a few paragraphs earlier we typed `\hbox{rectangle}` to prevent TEX breaking this word as usual and you saw the slug in action.

However, often we have a different problem, we do not want a linebreak at a break between two objects where TEX would normally do it. Then the easiest way is to put `\nobreak` there to forbid any linebreaking at a place immediately after this control word. Spaces after this are ignored (naturally), but not before.

Problem: We do not want "Mr. Longname" broken after "Mr.". We cannot use `Mr.\nobreak Longname`, since the space would disappear. If we use `Mr. \nobreak Longname`, TEX will break at the space before `\break`. The solution `\hbox{Mr. Longname}` would work, but it has another disadvantage, we are in this way preventing TEX from breaking the name. Obviously we need something new. The tilde `~` stands for an "unbreakable space", for instance we should type `a~priori`, or `Mr.~Longname`. All spaces around `~` are ignored.

$\mathcal{A}$ $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX offers a synonym `\tie` for people with keyboards without the `~` key. It also knows `\nolinebreak`, this tells TEX that a line should not be broken at the space preceding this command.

Page breaking: The command `\eject` tells TEX to finish a paragraph (it includes `\par`), close the page it has been working on and start a new one. If the text in the closed page is not long enough to fill a page, it is stretched by enlarging vertical spaces between paragraphs. If we want to prevent it, we should include a stretchable vertical space, that is, close the page using `\vfil\eject`.

Sidenote: We wrote that every tex-file must be ended with `\bye`. In fact it is a shortcut for `\vfill\eject\end`, where `\end` is really the command that tells TEX to stop processing.

Above we saw the command `\break` for breaking lines. If we use this command between paragraphs, then it breaks a page. It also spreads the text down, so it is almost like `\eject`, with the difference that we have to end a paragraph by ourselves.

Similarly, we can use `\nobreak` between paragraphs to prevent page breaking. We can also suggest TEX that some places might be good for page breaking. We already saw `\smallbreak`, `\medbreak` and `\bigbreak` that end a paragraph, insert vertical space and suggest an increasingly desirable place for page breaking. The command `\goodbreak` just ends a paragraph and indicates a good place for page breaking, no space is added.

$\mathcal{A}$ As usual, $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX offers some handy commands. `\newpage` is just a shortcut for `\vfill\eject`. Many users of plain define this control word for themselves to save typing. Note that is cannot be used within a paragraph. Conversely, `\nopagebreak` prevents TEX from breaking a page there. `\pagebreak` forces TEX to break a page and spread spaces between paragraphs so that the text spreads all the way down (similarly to `\eject`), but there is

one interesting difference. If this control word appears between paragraphs, the page is broken at that place. But if you use it within a paragraph, the line where it occurred is filled first and only then the page is broken.

Making a line.

The following four control words expect an argument and make it into a line—in different ways, of course. `\line` takes the text and makes it into a line by spreading spaces between words (and other objects). Here comes an example: `\line{This is not nice.}` prints exactly the same way as the example before. If the argument is longer than the width of a page, the text is shrunk as much as possible and the rest is left hanging out of the page.

`\leftline` takes its argument and puts it on a line, starting from the left. `\centerline` prints its argument centred, `\rightline` puts its argument on a line and moves it as far right as possible within the page. `\rightline{This is right.}` prints as

This is right.

All these control words create imaginary lines, objects that are as long as is the width of a page, but the position need not be an actual line, it depends on where we use those commands. If we use them within some other construction (for instance as a part of normal text), we are likely to have troubles with line breaking since the length of this construction is already the same as the width of the page and it cannot be broken. Here is an example: We now break a line with `\hfil\break` and then type

```
Not \centerline{very} nice.
```

We obtain

Not very
nice.

You can see that the word "very" was put in the middle of an imaginary line, which was then positioned after "Not". The black rectangle is at the position where this imaginary line ended and an Overfull message appeared on the terminal when running the TEX compiler on this tex-file. After this, TEX broke the line and "nice" appeared on the next one as a continuing text.

In a typical case we would use these `\line`-like constructions as individual paragraphs, or at least break a line manually before we use them (we used `\newline` before the `\rightline` example above). TEX is set up to help us in this way. If a paragraph starts with a `\line` construction, then this special line is not indented and the text that follows automatically starts a new paragraph.

By now we should be able to typeset simple text. The basic message probably is that the way your text looks like in your tex-file has nothing in common with the way it will look like on the output. TEX takes it and rearranges lines anyway.

One side remark for TEX fans: It is actually possible to tell TEX that it should keep the structure, that is, break the lines where we did. `\obeylines` is a switch (so we can restrict its validity by enclosing it, together with some part of our text, in a group) and it causes every line break in a tex-file (that is, every time we hit Enter) to be understood as `\par`. But then also every line becomes indented, one may want to fix this.

Punctuation, accents, spaces.

Correct punctuation is a complicated and important part of typesetting and TEX offers suitable tools. We start with a period. When TEX finds a period followed by a space, it checks the previous character. If it is a lower-case letter, TEX thinks that the period ends a sentence and puts a bigger space after this period. If the letter is upper-case, TEX thinks it is just an initial and uses the ordinary space. This algorithm sometimes does not yield the right thing and there are ways to interfere with it.

On a general level, `\frenchspacing` is a switch that puts ordinary spaces after periods, `\nofrenchspacing` returns back to the standard bigger after-sentence spaces. On a local level, we can always force the ordinary space using `\ `, for instance `Prof.\ Smith`, we can also use the unbreakable space `Prof.~Smith`. Conversely, `\space` is a space that is influenced by previous punctuation. If a sentence ends "my PC." and there is a space after it (and then some text), then TEX would leave just the ordinary space after the period. If we do not want this, we type `my PC.\space` and then the text that follows.

A $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX has the special "non-ending space" `\.`, so we could also use `Prof\.\ Smith` . Conversely, it allows us to force the larger "after-sentence" space using `@.`, `@!`, `@?`, `@;`, and `@:`, so we can write `my PC@.` Here we use the fact that in $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX, `@` is as a special character. This is not true in the plain TEX, so obviously these special punctuations will not work, there `@.` prints as @.

Some sentences end with "ellipsis" and it would be a mistake to put just three dots, in TEX we do it using the control word `\dots`, for instance `and then\dots` does and then ... TEX puts a larger space after an ellipsis. All spaces before `\dots` are ignored.

Now we look at **hyphens**, there are several types and each has its proper use.

Name (examples):	Shape:	How to get it:
hyphen (Hahn-Banach, so-called, n-dimensional)	-	`-`
dash or en-dash (67–78)	–	`--`
longer dash or em-dash (He was—as always—wrong)	—	`---`

However, note that this applies to English, typographic conventions differ and for instance in Czech we use en-dashes also in sentences as in the third example in the table above.

Our next topic is **quote marks**. If you want to have them right, do not use the key `"` which produces ". Proper quotation marks can be done using the "left quote" `‘` and the "right quote" (or apostrophe) `’` twice. For instance, `‘‘quote’’` gives "quote". Note that there should not be spaces between the left (or right) quotes.

The ,,Czech quote marks" were obtained using two commas `,,` and two left quotes `‘‘`. A better version would fix spaces between the symbols like this: `,\kern-0.7pt,quoted‘‘` does „quoted".

One of the TEX's strengths is its work with **accents**. The following table shows accented a:

á	`\’a`	a̋	`\H a`	ȧ	`\.a`	ā	`\B a`	ã	`\~ a`	ǎ	`\v a`	ă	`\u a`
à	`\‘a`	ą	`\d a`	ä	`\"a`	a̲	`\b a`	â	`\^a`	a̧	`\c a`	a͡a	`\t aa`

There is an alternative, `\=a` is the same as `\B a` and yields ā, but it only works in TEX, not in $\mathcal{AMS}$-TEX. Obviously, all the above works also with other letters, including capitals.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX there is one exception: Since `\.` is reserved for a period without an after-sentence space, it has to introduce another control sequence for upper dot, in $\mathcal{AMS}$-TEX one uses `\D a` to get ȧ.

Note that accents are control sequences with an argument. Hence we can even use `\^{ba}`, but the size of accents is fixed and applies to the first letter only anyway: We get b̂a, just as if we used `\^{b}a` or `\^ba`. Using this table one can get accents as characters, for instance ˜ using `\~{}` (that is, a tilde over nothing). Note the difference between control symbols like `\’` that can be written together with the corresponding character (say, `\’e` gives é), and control words like `\v` that have to be ended somehow (for instance ě can be obtained using `\v e` or `\v{e}`).

There are also special control symbols for some national characters. Czechs will appreciate `\accent23 u` which gives ů. Something for Swedes, Poles, and Germans:

å	`\aa`	æ	`\ae`	œ	`\oe`	ø	`\o`	ł	`\l`	ß	`\ss`
Å	`\AA`	Æ	`\AE`	Œ	`\OE`	Ø	`\O`	Ł	`\L`		

Spanish will like `?‘` giving ¿ and `!‘` for ¡, the same can be obtained using `>` and `<` in roman (see further), in `\tt` these make normal inequality signs. One last remark, if you use an accent over a "dot-letter" (i or j), you will get something ugly like í. To help this problem there are `\i` for "dotless" i and `\j` for "dotless" j. Thus `\’\i` gives í.

We now return to spaces. TEX offers spaces of several sizes that sometimes come handy.

Control word:	`\qquad`	`\quad`	`\enskip`	`\` (\ and a space)	`\thinspace`	`\negthinspace`
Name:	double quad	quad	en space	space	thin space	negative thin space
Size:	⊣ ⊢	⊣ ⊢	⊣⊢	⊣⊢	⊣⊢	⊣⊢

For instance, when a double quote and a single quote happen to meet ("It was an ‘accident’"), then `\thinspace` is exactly the right size of space between them. The size of `\quad` is the width of the letter m, while `\qquad` is twice this width. `\enskip` is approximately the width of n, precisely it is half of em, while `\thinspace` is one-sixth of em. In other words, the size of these spaces changes with the font used. A line can be broken at such spaces. There is also a command `\enspace`, this leaves the same space as `\enskip` but a line cannot be broken there, unless some glue follows.

Note that the last space is negative, that is, it moves text back. Say, `oo` yields oo, while `o\negthinspace o` prints like ꝏ.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX there are many more spaces and some equivalents.

Control word:	`\thickspace` or `\;`	`\medspace`	`\thinspace` or `\,`
Name:	thick space	medium space	thin space
Size:	⊣⊢	⊣⊢	⊣⊢

Control word:	`\negthickspace`	`\negmedspace`	`\negthinspace` or `\!`
Name:	negative thick space	negative medium space	negative thin space
Size:	⊣⊢	⊣⊢	⊣⊢

Fonts.

In the last part of this chapter we learn a little about *typeface*. In TEX one can easily specify what kind of letters should be used for printing the text. There are five basic "fonts":

Font:	roman	boldface	italic	slanted	typewriter
Control word:	\rm	\bf	\it	\sl	\tt
Example:	RoMan	**BoldFace**	*ITalic*	*SLanted*	TypewriTer

These control words are switches, that is, when you use one of them, text starts being written in the corresponding font and does so until another switch is used, or until a group within which it was used ends. Both

```
This is \it italic and this is \rm roman
```

and

```
This is {\it italic and this is} roman
```

print as This is *italic and this is* roman. Note that the space after \it was ignored. The second way is preferable. You surely noticed that the default font is roman. An example:

```
This is \underbar{\bf visualized }word
```

gives

This is **visualized** word.

Note that boldface was active for only one word, although we did not use \underbar{{\bf visualized }} and the single group we used was stripped when our argument was read. This shows that while doing the underlining, \underbar puts the argument into a group of its own. We included the space after "visualized" in the argument just to show off.

Also the line-making commands above have groups built-in, for instance, all chapter titles in this manual are done using \centerline{\bf ... }, no double braces needed.

There are two little problems with fonts. First, when using \tt font, TEX does not adjust spaces between words (just like a typewriter), so the right margin is not that nice and there is more "Overfull" messages.

The second problem appears when you switch from a slanted font (\it or \sl) to a font that is not slanted. Sometimes the space seems to be too small: {\it slanted} long gives *slanted* long. There is a control symbol \/ that puts a space of the right size there. {\it slanted}\/ long gives *slanted* long.

One interesting thing: \$ gives £ instead of $ when used in italic.

3. And now math

In order to typeset mathematics we have to switch from text mode to math mode (MM for short in this manual). In fact, there are two modes for mathematics, "text math mode" (TMM) and "displayed math mode" (DMM). They do not differ very much, almost everything we say about them works the same in TMM and DMM. We will usually just refer to MM (that is, to the math mode in general), and warn about differences where appropriate. However, the math mode differs quite a bit from TM, thus it is important to keep track of changes in modes.

We switch modes using special characters, the dollar $ serves as a switch between TM and TMM, "double dollar" $$ switches between TM and DMM. There is no switch between TMM and DMM. Once you decide to switch from text to, say, displayed math mode with $$, you keep writing displayed math until you switch to text with $$ again, only then you can start the other math mode with $. Note that $ $ is not displayed math, it is actually a space considered in the text math mode, which was then switched off again.

Here are some basic rules (compare to TM):

— it does not make much sense using fonts, in MM letters are typeset in a special version of italic and the rest is in roman

— one cannot use paragraphs (\par or blank lines) in MM

— all spaces (and hence single Enters which are treated as spaces) are ignored, the proper spacing is done by TEX. This is why punctuation should be done outside MM, in particular comma and period have a different meaning in mathematics. TEX does not leave a space after them in MM and does not break lines after them. Compare `$x<y, y<z$` printed as $x<y, y<z$ to the correct alternative `$x<y$, $y<z$` which gives $x<y$, $y<z$.

— most TM control sequences cannot be used in MM and likewise most MM control sequences do not work in TM.

From this one can see that it is really important not to miss some $. In the best case we don't switch off the math and the rest of the text will be printed without spaces in that weird math font. But TEX usually stumbles over some control sequence that can be only used in one of the modes and, because of the mistake, is now treated in the other one, so we get an error message. Also, \bye cannot be used in MM. There are some logical rules which one has to follow:

— DMM cannot be used within TMM and vice versa. So basically TEX expects ordinary text with pieces of TMM/DMM here and there.

— Every group opened in some MM part must be closed there as well, MM part opened in some group must be closed there again. This means that groups, TMM, and DMM parts must be "nested" and something like {${}}$

or `$${}$$` is not allowed. This is a common mistake, you will probably see the "Extra } or $ missing" error message quite often. It may help to imagine text as a sea with islands of TMM/DMM. Groups are circles on the water or circles drawn on the sand. Circles cannot intersect one another and they cannot be partially in water and partially on some island.

– To make things more interesting, math parts may contain textual parts, so we may imagine them as lakes on islands. The usual group rules are still true, we may have circles drawn on sand around a lake or entirely inside a lake, but not circles partially in a lake and partially out.

At least one thing from TM still works, though, the special characters from the beginning of the second chapter still have a special meaning in MM and the table describing how to print them is valid there as well.

TMM versus DMM.

The main difference between TMM and DMM is in text positioning. Text written in TMM (that is, between `$`) is used within normal text, it can be included into a line. For instance, `$f(x)=x$.` gives the output $f(x) = x$. On the other hand, DMM "displays" the text, it is printed centred on a separate line and there is some space added before and after. Here comes an example, `$$f(x)=x.$$` does

$$f(x) = x.$$

Please note the position of the period. This is one of the differences between DMM and TMM, punctuation is always put inside DMM. If we wrote `$$f(x)=x$$.`, TEX would print "$f(x) = x$" on a separate line as displayed math, then switch to text mode, make a space and start a new line with a period like this:

$$f(x) = x$$

. Similarly, we put `\eject` (or `\pagebreak` or `\newpage` in $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX) into DMM after which we want to break a page. If we put it after the DMM part, TEX would first add an empty space that belongs after DMM and then break the page. If we put it inside, TEX will know that no blank space should be added.

The part that we entered as DMM is always centered horizontally, but its vertical placement is influenced by how we typed it. If we type the DMM part as a part of a paragraph, then the normal text is interrupted, the DMM part is printed centered and then the paragraph continues, altogether forming one paragraph. This is what happened in the second example above, we actually typed the text `this: $$f(x)=x$$. Similarly,`.

On the other hand, if we put empty line (or `\par`) before or after a certain DMM part in the source file, then we break paragraphs at that point and paragraph rules come into play, in particular one can expect extra space around the DMM part (paragraph skip) and the text after it will be indented. The first example of displayed math actually had a blank line after the closing `$$`, so the text afterwards (Please note ...) started with an indentation.

Practical note: To improve readability of the source text it is customary to put DMM parts on their own lines in the source text, for instance in the first example we would type

```
does
$$f(x)=x.$$
```

or even

```
does
$$
f(x)=x.
$$
```

Other differences between TMM and DMM are connected with:

— the fact that DMM cannot exceed one line. Therefore none of the control sequences for breaking lines in TMM (see further) can be used in DMM.

— the fact that DMM offers more room. Some of the bigger mathematical objects (summation signs, for instance) look different in DMM, although the control word used is the same as for TMM. This is related to the fact that these two modes are typeset in different sizes, see the next chapter.

There is a control word `\displaystyle` that can be used in TMM and it works as a switch, the text that follows is printed as if it was in DMM. For examples and further information see below.

Writing math.

Now it is time to talk about mathematical text. We start with features analogous to TM. Paragraphs are easy to deal with, one cannot use them in MM. Also, one cannot **break lines** in DMM. TEX does break lines in TMM, but it is not really good at it. Many a time you will see TEX splitting a long equation in the middle of an algebraical expression instead of splitting it at a nearby "=". In general, TEX likes to split lines after relation symbols, in emergency after binary operators, but never inside groups or after punctuation.

The latter is especially important, for instance we prefer the chain $x \in A, y \in B$ split at the comma, not at the symbol $\in$. Thus one of the basic rules is to keep punctuation outside math whenever possible, the above should be

entered as `$x\in A$, $y\in B$` yielding $x \in A$, $y \in B$, another advantage of this form is better spacing. On the other hand, we would write `$x,y\in A$` to get $x, y \in A$, since we do not want any break at the comma and prefer the tighter spacing as well.

How do we manually prevent or force breaks? Preventing breaks is sometimes crucial, for instance if TEX decides to spread $S = \{x;\ x > 0 \wedge f(x) = 0\}$ over two lines. What can one do? If the formula is short and important, display it. One can also change the text before it so that the formula does not appear at the end of a line. Finally, one can enclose the whole expression (but inside TMM) into a group (TEX never breaks groups in MM) or close the whole math part into an h-box (see later). For instance, `\hbox{$x+1=3$}` or `${x+1=3}$` will not get broken. We can also forbid TEX to break a formula at a certain place using `\nobreak`.

Conversely, we can break a math part forcibly using `\break` or `\hfil\break` just like in TM.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX we also have `\mathbreak` which works just like `\newline` does in TM and `\nomathbreak` prevents breaking. The command `\allowmathbreak` allows a linebreak at a place where TEX normally does not break math.

Generally, it is a good idea to interrupt TMM parts as frequently as possible, especially to keep the punctuation in TM, TEX usually takes the advantage of breaking lines there. Another trick similar to TM: there is a "discretionary multiplication" `\*` which does not print but tells TEX that this may be a good place to break a line in TMM. If this happens, the character "×" appears at the end of the broken line.

Following the development of the second chapter, it is time now to introduce control sequences that allow us to print some strange characters. Very little survives from TM, namely the first table with special characters and also the commands for various spaces still work in MM. However, for math mode we also have special spaces.

The plain TEX has the following spaces in MM:

Control word:	`\thickspace`	`\medspace`	`\thinspace`	`\negthinspace`
Shortcut:	`\;`	`\>`	`\,`	`\!`
Size:	$\thickspace$	$\medspace$	$\thinspace$	$\negthinspace$

These various spaces are very useful in math, for instance, we can obtain the natural numbers symbol $\mathbb{N}$ using this neat trick: `$\mathbb{I\!N}$`. Some people like to pad set braces and type `$\{\,x\mid x>5\,\}$` to get $\{\,x \mid x > 5\,\}$.

When this chart is compared to the one in the previous section, we can see that $\mathcal{AMS}$-TEX adopted most of these spaces and made them available also in TM, but with exception of the shortcut for medium space: `\>` does not work in $\mathcal{AMS}$-TEX at all. Go figure.

$\mathcal{A}$ $\mathcal{AMS}$-TEX also offers some spaces that are available only in MM, namely `@,` is $\frac{1}{10}$ of `\,` and `@!` is $\frac{1}{10}$ of `\!`.

The command for ellipsis `\dots` also exists in MM, but behaves differently compared to TM. In MM, the height of these dots depends on surrounding characters, namely it adjusts their height for binary operators. Thus `$a,\dots,z$` prints as $a, \dots, z$ while `$1+\dots+n$` gives $1 + \cdots + n$. We can specify explicitly what level should be used, `\ldots` yields the low dots and `\cdots` gives those centred dots. If `\dots` is followed by a period or a parentheses or something else than the continuing formula, `\,` should be used after it.

The next table shows what **accents** we can use in MM:

$\acute a$	`\acute a`	$\hat a$	`\hat a`	$\tilde a$	`\tilde a`	$\bar a$	`\bar a`	$\dot a$	`\dot a`
$\grave a$	`\grave a`	$\check a$	`\check a`	$\breve a$	`\breve a`	$\vec a$	`\vec a`	$\ddot a$	`\ddot a`

These are control sequences with an argument, but unlike text mode accents, these apply not just to the first letter of the argument, but to the whole group. However, they do not get larger, so `$\tilde{aa}$` does $\tilde{aa}$. In such situations we appreciate control words `\widehat` and `\widetilde` that put a bigger accent which is good for up to three letters, for instance `$\widetilde{xyz}$` gives $\widetilde{xyz}$.

The control word `\skew` expects a number as its argument and an accent control word immediately after it, it shifts this accent to the right. Compare `$\vec A$` and `$\skew{10}\vec A$`: $\vec{A}$ and $\skew{10}\vec{A}$. It is useful when we want two accents combined, since TEX by default centers them. `$\hat{\hat A}$` looks like $\hat{\hat{A}}$, some people might prefer to shift the upper accent.

$\mathcal{A}$ $\mathcal{AMS}$-TEX has its own control words that differ by starting with capitals, e.g. `\Acute`, `\Grave`, ... that look just like `\acute` etc. when used singly, but they automatically skew when doubled: `$\Hat{\Hat A}$` looks like $\Hat{\Hat A}$.

$\mathcal{A}$ $\mathcal{AMS}$-TEX has also `\dddot a` yielding $\dddot a$ and `\ddddot a` yielding $\ddddot a$. Moreover, in $\mathcal{AMS}$-TEX these two and also other accents from the above table have analogues which start with "sp" and put the accents after and up. For example, `$(A+B)\sptilde$` prints as $(A+B)\sptilde$ and `$I\spdddot$` looks like this: $I\spdddot$. This is handy for printing just the marks, `$\spvec$` does $\spvec$.

$\mathcal{A}$ In order to save computation time, $\mathcal{AMS}$-TEX allows us to pre-define accented letters at the beginning of the file, say, `\accentedsymbol\HHA{\Hat{\Hat A}}`. Now when we type `\HHA`, it is replaced with `\Hat{\Hat A}`, but not as source text, the stored prepared picture is substituted. As such this symbol does not scale, so if we want to use it as a super/subscript, we need to define it extra: `\accentedsymbol\hha{{\ssize\Hat{\Hat A}}}`. Note the extra braces that allow us to use it easily as an index. For a second-order index we would have to define it again using `\sssize`.

There is another way to make arrows (and more) in TEX: `$\overarrow a$` does $\overrightarrow{a}$, compare to $\vec{a}$ obtained by `$\vec a$`. There are more commands of this sort, `\overarrow` and `\overrightarrow` are the same, but we also have `\overlefarrow`, `\overleftrightarrow`, `\overbrace`, and `\overline`.

Example: `$\overbrace{\overleftarrow{a}=\overarrow{b}}$` gives $\overbrace{\overleftarrow{a} = \overrightarrow{b}}$. The braces around "a" and "b" were not really necessary, if a group or a control sequence is not started, just the first character is taken as an argument anyway. We put it there so that you can note how the groups are nested and form arguments of the control words.

Again, note the difference between `$\overline a$` and `$\bar a$`: $\overline{a}$ and $\bar{a}$.

All these commands have their "under" counterparts, `\underarrow`, `\underrightarrow` (which are the same, $\underrightarrow{abc}$), `\underleftarrow`, `\underleftrightarrow` $\underleftrightarrow{a-z}$, `\underbrace` $\underbrace{A+B}$, and `\underline`.

The last one is an analogue of `\underbar` from TM, but unlike its text cousin, `\underline` first checks how much the argument goes bellow the baseline and then draws a line a bit lower. Thus it is possible to combine more under/over commands (as we already saw in the previous paragraph), for instance `$\underline{\underline{xyz}}$` gives $\underline{\underline{xyz}}$. However, this adjusting has one disadvantage, you may not like that `$\underline x+\underline y$` prints as $\underline{x}+\underline{y}$.

The character `\mathstrut` is nothing with the height and depth of the standard parenthesis (and it is very useful for aligning objects. In particular, `$\underline{\mathstrut x}+\underline{\mathstrut y}$` will look better: $\underline{x}+\underline{y}$. We will find `\mathstrut` useful also when constructing tables.

We commented that it does not make much sense using fonts in the math mode. In fact, we can use all the font switches (`\rm`, `\bf`, ...) in MM, they do change the shape of letters, but in mathematics we also have lots of other symbols and fonts do no change these, the resulting mix of styles is often far from acceptable. Therefore it is not assumed that a user would switch fonts in math (with a significant exception discussed below). If you do decide to use fonts, it may help to know that `\rm` does not affect accents and upper-case Greek (these are in `\rm` already), `\bf` affects letters, upper-case Greek, digits, and accents. `\mit` is math italic, affects upper-case Greek, has no accents.

$\mathcal{A}$ Note that in $\mathcal{AMS}$-TEX they for some reason decided to forbid the use of `\rm` and `\bf` in math mode. Changes to font must be done differently, see the discussion in the chapter on fonts.

Putting text in math.

Before we advance further, we look at the following question: What if we want some text within mathematics? In TMM one can simply switch TMM off and on again, but in DMM it is not possible, for instance
`$$0<n$$ for $$n$$ natural`
gives

$$0 < n$$

for

$$n$$

natural, which is far from what we want. Here is the answer: `\hbox` is a control word which expects one argument and this argument is then treated like an ordinary text. Actually, this control word has a much deeper meaning, but that will have to wait, right now it is nice to know what it does for us when used in MM: It prints the argument as a normal text using the font that was on before this particular MM part started. For instance,
`$$0<n\hbox{ for }n\hbox{ natural.}$$` prints as

$$0 < n \text{ for } n \text{ natural.}$$

I like `$$0<n\hbox{ for $n$ natural}$$` even better. This control word is very convenient, but it should only be used in DMM. First, the argument cannot be longer than one line. Second, TEX cannot break the argument of `\hbox` if it happens to appear at the end of a line, so it may cause a lot of troubles if used in TMM. Therefore the rule of thumb: if you need some text in TMM, switch it off with `$`, it will help TEX and save you some typing. I would surely prefer `$n$-dimensional` to `$n\hbox{-dimensional}$`.

This control word allows us to use capabilities from math mode in text mode and vice versa. For instance, we can raise a part of text like this: `A$^{\hbox{bc}}$` prints as $A^{\hbox{bc}}$. Another trick: $\overleftrightarrow{\text{both ways}}$ was printed using `$\overleftrightarrow{\hbox{both ways}}$`.

By the way, what happens if we write `$\overleftrightarrow\hbox{both ways}$`? When looking for argument of `\overleftrightarrow`, TEX encounters the backslash first so it knows that a control sequence is coming. After reading "hbox" there is the left brace, so TEX concludes that this control word—and hence the argument of the `\overleftrightarrow` command—is finished. So an arrow should be put over `\hbox`, but what is that? And TEX starts complaining.

$\mathcal{A}$ $\mathcal{AMS}$-TEX offers the control word `\text` that works similarly, but it is a bit smarter, for instance it respects position. Compare the previous example with `A$^{\text{bc}}$` that prints as A^{bc}. You can see the the exponent is smaller now. This can be very helpful, in plain TEX we would have to change the font manually.

Symbols.

Mathematics is big on symbols and TEX did not spare any expense to provide lots of them. We start with **Greek letters**:

α	\alpha	η	\eta	ν	\nu	σ	\sigma	ψ	\psi	Ξ	\Xi
β	\beta	θ	\theta	ξ	\xi	ς	\varsigma	ω	\omega	Π	\Pi
γ	\gamma	ϑ	\vartheta	o	o (omicron)	τ	\tau	Γ	\Gamma	Σ	\Sigma
δ	\delta	ι	\iota	π	\pi	υ	\upsilon	$\varGamma$	\varGamma	Υ	\Upsilon
ϵ	\epsilon	κ	\kappa	φ	\varphi	ϕ	\phi	Δ	\Delta	Φ	\Phi
ε	\varepsilon	λ	\lambda	ρ	\rho	φ	\varphi	Θ	\Theta	Ψ	\Psi
ζ	\zeta	μ	\mu	ϱ	\varrho	χ	\chi	Λ	\Lambda	Ω	\Omega

The prefix \var can be actually used with all capital Greek letters, for instance $\varXi$ gives Ξ.

$\mathcal{A}$ $\mathcal{AMS}$-TEX affers nice upper-case letters for the Latin alphabet, $\mathcal{A},\dots,\mathcal{Z}$ are obtained using \Cal, for instance $\Cal H$ gives $\mathcal{H}$ (note the math mode, \Cal does not work in text mode).

When presenting symbols here, we try to group them logically, but some do not possess any special properties, they are just useful pictures. Here they come.

$\forall$	\forall	∂	\partial	ℓ	\ell	$\surd$	\surd	$\S$	\S	$\spadesuit$	\spadesuit
$\exists$	\exists	$\prime$	\prime	$\aleph$	\aleph	$\angle$	\angle	$\dagger$	\dag	$\heartsuit$	\heartsuit
:	\colon	∇	\nabla	$\hbar$	\hbar	$\triangle$	\triangle	$\ddagger$	\ddag	$\diamondsuit$	\diamondsuit
$\neg$	\lnot	$\int$	\smallint	$\wp$	\wp	$\top$	\top	$\sharp$	\sharp	$\clubsuit$	\clubsuit
$\emptyset$	\emptyset	$\Re$	\Re	$\imath$	\imath	$\bot$	\bot	$\flat$	\flat		
∞	\infty	$\Im$	\Im	$\jmath$	\jmath	¶	\P	$\natural$	\natural		

\neg does the same as \lnot. \imath and \jmath are MM versions of dotless \i and \j. Colon \: is the ordinary colon moved slightly to the left, so it looks better in expressions like $\forall x>0\colon -x<0$ giving $\forall x>0\colon -x<0$ (in $\mathcal{AMS}$-TEX there is also \: that does this job). The symbol \angle is unusual in that it does not get smaller, so it cannot be used in indices and similar places.

Note that for derivative, people usually type just the apostrophe f' to get f' instead of using $f^\prime$.

Relations and operations, arrows.

We start with **binary operators**.

$+$	+	$\times$	\times	$\bullet$	\bullet	$\cap$	\cap	$\vee$	\lor or \vee		
$-$	-	$\cdot$	\cdot	$\circ$	\circ	$\cup$	\cup	$\wedge$	\land or \wedge		
$\pm$	\pm	$\odot$	\odot	$\bigcirc$	\bigcirc	$\uplus$	\uplus	$\amalg$	\amalg		
$\mp$	\mp	$\oslash$	\oslash	$\bigtriangleup$	\bigtriangleup	$\sqcap$	\sqcap	$\diamond$	\diamond		
$\div$	\div	$\otimes$	\otimes	$\bigtriangledown$	\bigtriangledown	$\sqcup$	\sqcup	$\dagger$	\dagger		
$*$	\ast or *	$\ominus$	\ominus	$\triangleleft$	\triangleleft	$\setminus$	\setminus	$\ddagger$	\ddagger		
$\star$	\star	$\oplus$	\oplus	$\triangleright$	\triangleright	$\wr$	\wr				

Binary operators have an interesting feature: TEX puts an extra space around them. For instance, if we take the control word \backslash (which can be used in MM only), it looks just like \setminus, but in combination with symbols they differ: $A\backslash B$ gives $A\backslash B$ whereas $A\setminus B$ looks like this: $A\setminus B$. If we do not want that extra space around, we put the operator into braces, compare $x+y$ and $x{+}y$ giving $x+y$ and $x+y$.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX there is also the binary operation \and, $A\and B$ prints as $A \And B$. Again, this has extra spacing compared to $A\&B$ that yields $A\&B$.

There are many **binary relations**, so we split them into two tables, we will save all arrow-like relations into the second table.

$<$	`<`	$\prec$	`\prec`	$\doteq$	`\doteq`	$\in$	`\in`	$\ni$	`\ni` or`\owns`
$>$	`>`	$\succ$	`\succ`	$\equiv$	`\equiv`	$\notin$	`\notin`	$\mid$	`\mid`
$\leq$	`\le` or `\leq`	$\preceq$	`\preceq`	$\approx$	`\approx`	$\subset$	`\subset`	$\parallel$	`\parallel`
$\geq$	`\ge` or `\geq`	$\succeq$	`\succeq`	$\asymp$	`\asymp`	$\subseteq$	`\subseteq`	$\perp$	`\perp`
$\ll$	`\ll`	$\sim$	`\sim`	$\smile$	`\smile`	$\sqsubseteq$	`\sqsubseteq`	$\vdash$	`\vdash`
$\gg$	`\gg`	$\simeq$	`\simeq`	$\frown$	`\frown`	$\supset$	`\supset`	$\dashv$	`\dashv`
$=$	`=`	$\cong$	`\cong`	$\propto$	`\propto`	$\supseteq$	`\supseteq`	$\bowtie$	`\bowtie`
$\neq$	`\ne` or `\neq`	$\models$	`\models`	$\iff$	`\iff`	$\sqsupseteq$	`\sqsupseteq`	mod	`\bmod`

$\mathcal{A}$ Note that $\mathcal{AMS}$-TEX also knows `\implies` $\implies$ and `\impliedby` $\impliedby$.

$\leftarrow$	`\leftarrow` or `\gets`	$\Leftarrow$	`\Leftarrow`	$\nwarrow$	`\nwarrow`
$\rightarrow$	`\rightarrow` or `\to`	$\Rightarrow$	`\Rightarrow`	$\nearrow$	`\nearrow`
$\leftrightarrow$	`\leftrightarrow`	$\Leftrightarrow$	`\Leftrightarrow`	$\swarrow$	`\swarrow`
$\longleftarrow$	`\longleftarrow`	$\Longleftarrow$	`\Longleftarrow`	$\searrow$	`\searrow`
$\longrightarrow$	`\longrightarrow`	$\Longrightarrow$	`\Longrightarrow`	$\uparrow$	`\uparrow`
$\longleftrightarrow$	`\longleftrightarrow`	$\Longleftrightarrow$	`\Longleftrightarrow`	$\downarrow$	`\downarrow`
$\mapsto$	`\mapsto`	$\Uparrow$	`\Uparrow`	$\updownarrow$	`\updownarrow`
$\longmapsto$	`\longmapsto`	$\Downarrow$	`\Downarrow`	$\leftharpoonup$	`\leftharpoonup`
$\hookleftarrow$	`\hookleftarrow`	$\Updownarrow$	`\Updownarrow`	$\rightharpoonup$	`\rightharpoonup`
$\hookrightarrow$	`\hookrightarrow`	$\rightleftharpoons$	`\rightleftharpoons`	$\leftharpoondown$	`\leftharpoondown`
				$\rightharpoondown$	`\rightharpoondown`

We can negate all relations using `\not`, for instance `$\not\subseteq$` gives $\not\subseteq$. However, for negations of $=$ and $\in$ the specials `\ne` and `\notin` look better.

Example: $f(x,y) \approx x + x(\alpha/\beta)\cdot(r!\,(n-r)!)$, that is, `$f(x,y)\approx x+x(\alpha/\beta)\cdot(r!\,(n-r)!)$`. That thin space `\,` was added for æsthetical reasons.

Note the operation `\bmod`. It is used like this: `$n\bmod m$` will print $n \bmod m$. There is a related control word (but not a binary operation) `\pmod` that expects one argument and is used as follows: `$n=m\pmod{2k+1}$` will look like $n = m \pmod{2k+1}$.

The rules about spacing we described for binary operations apply also to binary relations. We will now try `$A\iff B$`, `$A{\iff}B$`, `$A\Longleftrightarrow B$` and `$A{\Longleftrightarrow}B$` and you can compare the outcomes: $A \iff B$, $A {\iff} B$, $A \Longleftrightarrow B$ and $A {\Longleftrightarrow} B$.

TEX allows us to put something over a relation symbol, the resulting object is again considered a relation. The construction is `\buildrel` *something*`\over` *relation*. For instance, `$\buildrel\hbox{\rm def}\over=$` does $\buildrel\hbox{\rm def}\over=$. Note that we did not really have to use the `\hbox`, but then this construction would not work in $\mathcal{AMS}$-TEX as it does not allow us to use `\rm` in math mode. And if we did just `\hbox{def}` there, the the word would be printed in the font that was on just before this math piece started.

Actually, we would prefer to have a longer equality sign. One possibility is to put two in a row, but then there would have been a space between them, even if we tell TEX not to treat them as binary relations using braces: `$\buildrel\hbox{\rm def}\over{{=}{=}}$` does $\buildrel\hbox{\rm def}\over{=\!=}$. We see that we actually need three equalities and closer together. One possibility is to use negative space, say, this `{=\!=\!=}` should do the trick, if not, use `\!\!`. A better way is to let the proper spacing up to TEX, the command `\joinrel` puts two relations next to each other and treats the resulting symbol as a relation again. `$\buildrel\hbox{\rm def}\over{=\joinrel=\joinrel=}$` does $\buildrel\hbox{\rm def}\over{=\joinrel=\joinrel=}$.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX there is another interesting construction which allows us to create arrows. The simplest form is `@>>>`, which is the same as `\to`, and `@<<<` which is equivalent to `\gets`. But one can add something between characters `<` or `>`. Whatever is between the first and the second one is put above the arrow, the text between the second and the third `</>` is underset. The length of the arrow is set so that the stuff that is put over/under fits. Examples: `$@<\infty<This is a long text<$` makes $\xleftarrow[Thisisalongtext]{\infty}$, `$@>\text{\rm weakly}>>$` prints as $\xrightarrow{\text{weakly}}$. The switch `\rm` was not necessary in this case, but if we used this construction without it in a theorem, the text "weakly" would be written in the font we use for theorems, most likely in italic. Note also that if we used `\hbox` instead of `\text`, then the word would be in the usual text size.

Superscripts and subscripts.

The special character `^` (or equivalently, the control word `\sp`) is a control sequence that expects one argument and puts it smaller to the upper right corner. Lower index can be produced using `_` or `\sb`, so for instance `$x^i_j$`

gives x^i_j. This example shows that we can mix two different indices, the order does not make any difference. We cannot use two of the same kind in a row, we have to define their priority using groups that correspond to the mathematical meaning. `${x_m}_n$` is x_m with an added index n: x_{mn}, whereas `$x_{m_n}$` is x indexed by a subsequence: x_{m_n}. We can use more indices as long as the order is defined or does not matter. Just look at this horrible picture `${x_\infty^{23}}^{\odot^{13}}$` which looks like $x_\infty^{23^{\odot^{13}}}$. Again, look at the difference between x^{2^3} (`${x^2}^3$`) and x^{2^3} (`$x^{2^3}$`). In fact, it is customary to write the former as `$(x^2)^3$` to get $(x^2)^3$, a better option would be `${(x^2)}^3$` to get $(x^2)^3$.

The last example illustrates the following rule for index placement: The position of an index depends on the height/depth of the preceding character. If this character is the closing brace }, then TEX takes into account the height/depth of the whole group that has just ended.

Now we look at some tricks. Some people write indices before the letter. We can get it using the empty group, `${}^iF^j_k$` will give ${}^iF^j_k$, although here we might want to fix the spacing and also the height of the first index, for that we need to learn more about TEX. As an advertisement, ${}^iF^j_k$ was printed using `${\vphantom{F}}^{i\!}F^j_k$`.

Another use of an empty group: Instead of R^{kl}_i obtained by `$R_i^{kl}$` we may need ${R_i}^{kl}$, which was obtained by `$R_i{}^{kl}$`, but one can also use `${R_i}^{kl}$`.

Negative space is very handy with characters that include lots of white space somewhere, For instance, instead of Γ_3 we may prefer Γ_3 obtained by `$\Gamma_{\!3}$`; one can also try `$\Gamma\!_3$` to get Γ_3. Negative space can also help with a slash, instead of x^2/t one may prefer `$x^2\!/t$` to get the nicer $x^2\!/t$.

Finally, there is one handy shortcut for a **derivative**. Instead of `^\prime` one can use the right quote ´, for instance `$f´$` gives f'. But be careful, when compiling, TEX again substitutes back that prime thing, so for instance `$f´^2$` is interpreted as `$f^\prime^2$`, which is a double superscript error. However, we need not worry when using several derivative marks in a row. TEX knows that two or more derivatives have a special meaning, so it correctly recognizes `$f´´$` as f'' instead of complaining about two superscripts or doing something like $f'^{\prime}$ (superscript of a group, we printed this using `${f´}´$`).

Most characters are safe to be used in sub/superscripts, but there are some exceptions. We already met `\angle` that does not scale. Another trouble is with `\ne` and `\notin`, these represent constructions and therefore cannot be used directly as super/subscripts, one should close them into a group.

Roots.

In the symbols section we have already met `\surd` which can do $\surd 2$ using `$\surd2$`, but it is not too nice. `\sqrt` expects one argument and makes it into a root, $\sqrt{x^2+y^2}$ can be obtained using `$\sqrt{x^2+y^2}$`. For a general root there is the construction `\root` *root*`\of`, where `\of` expects one argument. Note that everything between the two control words is taken as an argument, so no group is necessary there. For instance, `$\root3+n\of{1+x^n}$` gives $\root 3+n \of{1+x^n}$. Note that it is always a good idea to add `\mathstrut` into the argument so that the root sign does not crowd the contents too much, here some might even prefer moving the 1 a bit to the right. It might be also a good idea to enlarge the exponent for better readibility, especially if the output is further converted for internet use, we will explain `\scriptstyle` below. Finally, a group, although not needed, may facilitate reading, `$\root{\scriptstyle 3+n}\of{\mathstrut\,1+x^n}$` prints $\root{\scriptstyle 3+n}\of{\mathstrut\,1+x^n}$. The space after `\scriptstyle` was not needed, `\scriptstyle3+n` would do the same, but again, it makes the source file easier to read.

$\mathcal{A}$ $\mathcal{AMS}$-TEX offers means to adjust position of the exponent. `\uproot` should be used immediately after `\root` and expects one argument, an integer that tells how far up should the root move. The distance is in points and negative numbers mean down. Similarly works `\leftroot`, their order does not matter. This $\root\leftroot{-2}\uproot23+n\of{\mathstrut\,1+x^n}$ was obtained by `$\root\leftroot{-2}\uproot23+n\of{\mathstrut\,1+x^n}$`. Now we again enlarge the exponent, but since we are in $\mathcal{AMS}$-TEX, we use `\ssize` (see below). For better reading we use groups to separate arguments visually. The construction `$\root\leftroot{-2}\uproot2{\ssize 3+n}\of{\mathstrut\,1+x^n}$` yields $\root\leftroot{-2}\uproot2{\ssize 3+n}\of{\mathstrut\,1+x^n}$.

Fractions.

The slash `/` can be used for division as we did in one of the examples above, but it is not a binary operator, it is not very nice and it is not suitable for more complicated formulas.

The basic fraction is obtained using {*numerator* `\over` *denominator*}, for instance, `${1\over 2}$` prints ${1\over 2}$. Fractions can be nested, `${1\over1+{b\over2}}$` prints ${1\over1+{b\over2}}$. Note two things. First, TEX recognized correctly different levels of fractions and changed the sizes of characters and lengths of fraction lines. Second, the resulting fraction was much bigger than the height of an ordinary line, so TEX adjusted the spacing between lines accordingly. This is one of the greatest advantages of TEX, one does not have to worry about these things.

An important feature of a fraction is that when we use it in displayed math, TEX takes advantage of the extra space and prints it larger. `$${1\over1+{b\over2}}$$` shows as

$$\frac{1}{1+\frac{b}{2}}.$$

This basic command can be modified in several ways. First, if we do not want to see any fraction line, we can use {*numerator* `\atop` *denominator*}, `${a\atop 23}$` does $\binom{a}{23}$.

The construction {*numerator* `\above`(*dim*)*denominator*} allows us to specify the width of the fraction line, for dimension (*dim*) see below. For instance, `${a\above2mm 23}$` does $\overset{a}{\blacksquare}{}^{23}$. Note that also the spacing was thrown into disarray, to get an acceptable result one has to learn more about TEX.

We can ask TEX to put delimiters around the resulting construction if we use `\overwithdelims` *ldelim rdelim* instead of `\over`, where *ldelim* and *rdelim* are some delimiters (see below), not necessarily left and right ones. Similarly we can use `\atopwithdelims` and `\abovewithdelims`. For instance, `${n\atopwithdelims()k}$` makes the binomial number $\binom{n}{k}$, while `${n\overwithdelims[<k}$` makes this: $\left[\frac{n}{k}\right\langle$.

TEX has a shortcut for the former, `${n\choose k}$` will do $\binom{n}{k}$. It also offers `${1+2\brace3}$` for $\left\{{1+2 \atop 3}\right\}$ and `${1+2\brack3}$` for $\left[{1+2 \atop 3}\right]$.

$\mathcal{A}$ $\mathcal{AMS}$-TEX offers an alternative control word `\frac` which expects two arguments and then puts one over another with a line between them, `$\frac{1}{2}$` gives $\frac{1}{2}$. In fact, since every control word always tries for just a character as an argument, we can also get this using `$\frac12$`. Similarly, `$\frac1a$` prints as $\frac{1}{a}$ and `$\frac\infty1$` gives $\frac{\infty}{1}$, while `$\frac123$` prints as $\frac{1}{2}3$.

$\mathcal{A}$ Many people find this way of entering fractions preferable to `\over`, especially when it comes to more convoluted fractions. For instance, using `$\frac{13-\frac{26}2}{\frac{\partial f}{\partial x}}$` we obtain $\frac{13-\frac{26}{2}}{\frac{\partial f}{\partial x}}$. We will see in chapter 8 how to define this macro, so one can have it without calling on $\mathcal{AMS}$-TEX.

$\mathcal{A}$ $\mathcal{AMS}$-TEX also allows to force the style of a fraction, `\dfrac` is a fraction that is always printed as if it was in displayed math, whereas fractions given by `\tfrac` have always the size of text-mode mathematics. Then it is up to us to make things nice, otherwise we get something like $\frac{1}{\frac{2}{3}}$ obtained using `$\frac{1}{\dfrac23}$`. By the way, instead of {1} we can put just `1`.

$\mathcal{A}$ There is also a control word `\thickfrac` which works in the same way but uses a thicker fraction line. Its thickness can be specified by putting (right after `\thickfrac`) the control word `\thickness`, which expects a number as an argument, 1 means the normal size. Example: `$\thickfrac\thickness{3}1{UGLY}$` will create this: $\binom{1}{UGLY}$. That group around 3 was not necessary but it is sometimes good to separate arguments visually for easier reading.

$\mathcal{A}$ For **combinatorial/binomial numbers** we can use `\binom`, in $\mathcal{AMS}$-TEX it again expects two arguments, so `$\binom nk$` gives $\binom{n}{k}$. The command `\tbinom` always prints this in small (text) size.

$\mathcal{A}$ $\mathcal{AMS}$-TEX also offers one control word for **continued fractions**: `\cfrac` and `\endcfrac` mark the beginning and the end of the construction and items (levels) are separated by double-backslashes `\\`. It is easier to show an example then to explain it:

$$a+\cfrac{1}{1+2\over b+\cdots+\cfrac{1+\cdots+27}{z}}$$

was obtained using `$\cfrac1\\a+\cfrac1+2\\b+\dots+\cfrac1+\dots+27\\z\endcfrac$`. The construction is somehow recursive. Note that numerators are centred. `\lcfrac` will do the same but with numerators moved to the left, `\rcfrac` has them on the right.

Operators.

Operators are objects that apart from appearance (symbol) also include possible modifications in behaviour. A good start are functions.

Functions.

In mathematics we write variables in italic and functions in roman. TEX therefore offers control words that allow us to do it easily.

`\exp`	`\sin`	`\sinh`	`\arcsin`	`\sec`	`\max`	`\lim`	`\det`
`\log`	`\cos`	`\cosh`	`\arccos`	`\csc`	`\min`	`\limsup`	`\dim`
`\ln`	`\tan`	`\tanh`	`\arctan`	`\deg`	`\sup`	`\liminf`	`\ker`
`\lg`	`\cot`	`\coth`	`\arg`	`\Pr`	`\inf`	`\gcd`	`\hom`

So for instance `$f(x)=\sin(x)+\log_2(x)+\sup(x^3)$` will print $f(x)=\sin(x)+\log_2(x)+\sup(x^3)$. These are called "operators with names" in TEX and note that some of them also belong to the group "sum-like operators" described below.

If we want another function not in this list, we can create it using the command `\mathop`. For instance, `$\alpha=\mathop{\rm arctg}(x)$` prints $\alpha=\operatorname{arctg}(x)$. For finer points of creating operators see the appropriate section below.

We know how to place limits next to symbols, for instance, `$\log_2(x)$` prints as $\log_2(x)$, nothing surprising here. However, note the following: If we type `$\max_{x\in M}(f)$`, we get $\max_{x\in M}(f)$ as expected, but in DMM (that is, typing `$$\max_{x\in M}(f)$$`) we get $\max\limits_{x\in M}(f)$. For some of the functions we actually do expect the limits put below (and above where appropriate) and displayed math offers the possibility to do so, however, in text it would spread the lines too much (as we saw above), therefore to save space limits are put in the standard subscript position.

Thus we have two different kinds of behaviour for operators, either the limits that follow are always in the subscript position, or they are in subscript position when used in TMM and below/above when used in DMM. Every operator belongs to one of these two categories. From the chart above, the following change position of limits.

`\max`	`\min`	`\sup`	`\inf`	`\lim`	`\limsup`	`\liminf`	`\det`	`\gcd`	`\Pr`

For instance, `$\sin_2$` always prints as $\sin_2$ no matter where used, but `$\lim_{n\to\infty}$` prints as $\lim_{n\to\infty}$ and `$$\lim_{n\to\infty}.$$` prints as

$$\lim_{n\to\infty}.$$

Note that when we create an operator using `\mathop`, then it behaves in the second way, it changes position. We can modify this behaviour for any operator by preceding super/subscript specification with a special control word. If we want the above/below position, we use `\limits`, for instance, `$\sin\limits_2$` prints as $\sin\limits_2$. Conversely, `\nolimits` forces the usual sub/superscript position, for instance `$$\lim\nolimits_{n\to\infty}.$$` prints as

$$\lim\nolimits_{n\to\infty}.$$

One can think that there is `\nolimits` in TMM and `\limits` in DMM implicitly for those position-changing operators.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX there are six more named operators, namely

$\overline{\lim}$	`\varlimsup`	$\operatorname{inj\,lim}$	`\injlim`	$\varinjlim$	`\varinjlim`
$\underline{\lim}$	`\varliminf`	$\operatorname{proj\,lim}$	`\projlim`	$\varprojlim$	`\varprojlim`

These all change positions of limits when in DMM.

$\mathcal{A}$ $\mathcal{AMS}$-TEX actually changes the way limits work. The modifiers `\limits` and `\nolimits` now work only for operators that change position of super/subscripts, for the others they have no effect at all. For instance, in $\mathcal{AMS}$-TEX the command `$\sin\limits_2$` prints as $\sin\limits_2$, as if we did not put `\limits` there.

$\mathcal{A}$ $\mathcal{AMS}$-TEX also has a different way of defining operators. If we do not want limits to change, we use `\operatorname`. For instance, `$\operatorname{codim}_X(Y)<\infty$` will give $\operatorname{codim}_X(Y)<\infty$ (and we cannot use `\limits` to change it). If we want the changing position, we use `\operatornamewithlimits`. We try this with the dimension example: `$\operatornamewithlimits{codim}_X(Y)<\infty$` looks exactly as before, but now we can use `$$\operatornamewithlimits{codim}\limits_X(Y)<\infty$$` to get $\operatorname{codim}\limits_X(Y)<\infty$.

$\mathcal{A}$ Note that while `\mathop` still works in $\mathcal{AMS}$-TEX, we cannot use `\rm` in math there, so the plain TEX solution is not universal for both.

Large Operators.

$\sum$	`\sum`	$\odot$	`\bigodot`	$\prod$	`\prod`	$\bigcap$	`\bigcap`	$\biguplus$	`\biguplus`	$\bigvee$	`\bigvee`
$\bigoplus$	`\bigoplus`	$\bigotimes$	`\bigotimes`	$\coprod$	`\coprod`	$\bigcup$	`\bigcup`	$\bigsqcup$	`\bigsqcup`	$\bigwedge$	`\bigwedge`

Some of these operators look similar to control words we have met before, for instance `\cup` and `\bigcup`, but there is a big difference, these are operators that change positions of limits. For instance `$\sum^\infty_{i=1}a_ix_i$` looks like this: $\sum^\infty_{i=1}a_ix_i$. but when we display it in DMM: `$$\sum^\infty_{i=1}a_ix_i.$$`, we get

$$\sum_{i=1}^{\infty}a_ix_i.$$

Note that we did not have to put `\infty` into a group, a control sequence is an allowable argument. As usual, we can modify position of limits using `\limits` and `\nolimits`, for instance `$\sum\limits^n_{k=1}$` will do $\sum\limits^n_{k=1}$.

Large operators (also called „sum-like operators“) have another distinct feature, the symbol gets larger in displayed math. We already saw it with the sum, similarly we have `\bigcup` like this $\bigcup$ in text mode and like this $$\bigcup$$ in DMM etc. There is a way to obtain the larger symbols also in text (we just did it), see Styles below.

Technical note: To make limits more readable, TEX leaves some space below and above sum-like operators (it is called "buffer"). If we don't want to have it there (for instance when TEX is fitting the parentheses of the right size, see further), we can do it by using `\shave` with the whole operator including its limits as an argument. `\topshave` kills the upper space and `\botshave` will get rid of the buffer under the operator.

Integrals.

There are two basic commands for integrals, `\int` prints $\int$ and `\oint` prints $\oint$. Both symbols get enlarged when displayed (this is similar to large operators), but unlike sums et al., integration limits are always in the proper super/subscript position. For instance, `$$\int_a^b f(x)\,dx.$$` prints as

$$\int_a^b f(x)\,dx.$$

Note the thin space separating the integrated function from the differential, it is highly recommended. The limit position can be changed using `\limit`, for instance, `$\int\limits_0^\infty$` looks like $\int\limits_0^\infty$.

When we write multiple integrals, it is a good idea to put a negative space between them. `$$\int\limits_0^1\!\int\limits_0^\pi frac1{1+x^2}\sin(y)\,dx\,dy.$$` gives

$$\int_0^1\!\int_0^\pi \frac{1}{1+x^2}\sin(y)\,dx\,dy.$$

$\mathcal{A}$ $\mathcal{AMS}$-TEX also has `\iint` $\iint$, `\iiint` $\iiint$, `\iiiint` $\iiiint$, and finally `\idotsint` $\int\cdots\int$.

Multiline limits.

How did we get $\sum\limits_{\substack{i,j=1\dots n\\ i\ne j}}$? Like this: `$\sum\limits_{\substack{i,j=1\ldots n\\ i\ne j}}$`. We will learn how to make the limit a bit larger below.

Note that a control word always takes the contents of the offered group as an argument, thus the outer braces are used up and the argument we are passing to superscript is in fact `{i,j=1\ldots n\atop i\ne j}`, exactly as needed. Actually, in this particular case it would work even without those double braces because of the way TEX processes this construction, but it is better to play it safe unless one knows very well what is happening behind the curtain.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX we can use the couple `\Sb`*text*`\endSb` that takes *text* and turns it into a so-called multi-level subscript, lines are separated by `\\`. Example: `$\sum\Sb1\le i\le n\\i\le j\le i\endSb a_{ij}x_ix_j$` prints as $\sum\Sb1\le i\le n\\i\le j\le i\endSb a_{ij}x_ix_j$. Similarly, `\Sp`*text*`\endSp` makes multi-level superscripts. In both cases we can use `\limits`.

Delimiters.

In the following table we put all kind of parentheses. They are part of the class called delimiters.

(	(	(	\lgroup	⟨	\langle	⌈	\lceil
)	)	)	\rgroup	⟩	\rangle	⌉	\rceil
[	[or \lbrack	{	\{ or \lbrace	|	| or \vert	⌊	\lfloor
]	] or \rbrack	}	\} or \rbrace	‖	\\| or \Vert	⌋	\rfloor

/	/
\	\backslash
⎰	\lmoustache
⎱	\rmoustache

Moreover, all the vertical arrows we had earlier, namely `\uparrow`, `\downarrow`, `\updownarrow`, `\Uparrow`, `\Downarrow`, and `\Updownarrow`, also behave like delimiters.

Delimiters are vertically-extensible, which can be achieved in two ways. First, we can specify the desired height by ourselves. This is done using four control words that expect a delimiter as an argument,

$$(\big(\Big(\bigg(\Bigg($$ was obtained using `( \big( \Big( \bigg( \Bigg(`.

Delimiters can also have orientation, that is, TEX has a special treatment for left and right delimiters where it makes sense. Almost all symbols in the above table are naturally paired (exceptions are vertical bars and slashes). However, when we use the above commands for enlarging, this orientation is lost. This does not matter in some situations, often with slashes. Say, $\frac{a+1}{b}\bigg/\frac{c+1}{d}$ was obtained by using `$\dfrac{a+1}b\bigg/\dfrac{c+1}d$`, and here is an example with a restriction: $\dfrac{\partial f}{\partial x}\Big|_{x=0}$ was obtained by `$\dfrac{\partial f}{\partial x}\bigg|_{x=0}$`.

However, often we do want TEX to know about orientation, then we use modifications of the above size-controlling commands. When we use `\bigl`, `\Bigl`, etc., the resulting symbol will be treated as a left delimiter, while `\bigr`, `\Bigr`, etc. is treated as a right delimiter. For instance, `\bigl]` will be considered a left delimiter. The difference lies in spacing when a delimiter meets a binary operator or a binary relation on the ,,inside". Compare spacing of $\bigl|+1$ and $\bigr|+1$ printed using `$\bigl|+1$` and `$\bigr|+1$` (or `$\big|+1$`). .

Finally, there is a modification \bigm, \Bigm, etc., where the resulting symbol is considered a binary relation, which means that we get extra spacing around it.

The size \big is just good enough to distinguish between different levels of bracketting, as in $\sin\big(2(x-\pi)\big)$ obtained using $\sin\bigl(2(x-\pi)\bigr)$ or in $\big||x|+|y|\big|$ obtained using $\bigl||x|+|y|\bigr|$. It is also the right size for simple roots as in $\big(1+\sqrt{2}\big)$ obtained using $\bigl(1+\sqrt2\,\bigr)$, note the tiny space that separates the root sign from the right parenthesis.

The size \Big is about right for a sum with limits (1.5 times bigger than big), and \Bigg is 2.5 times bigger than big.

If you do not feel like playing with the sizes, you can entrust TEX with it. There are control words \left and \right, each of them has to be followed by a delimiter whose size will be automatically adjusted. How does it work? \left includes the left brace {, that is, it opens a group. Similarly, \right contains }. These two control words must be paired and the groups they create cannot intersect with other groups, they can be nested with ordinary groups and all the usual rules apply. When TEX finds \right, the corresponding \left is determined and then the size for both delimiters is chosen so that it fits the thing in the group between them. The delimiters used may be of any kind (one can even use a left delimiter with \right and vice versa), since these two commands enforce orientation. For instance, $\left(\dfrac{\partial f}{\partial x}\right|_{x=0}=13$ gives $\left(\dfrac{\partial f}{\partial x}\right|_{x=0}=13$.

Sometimes one needs to fit only one delimiter, for instance in the restriction example. There is a trick for it: We can use a period instead of a delimiter after \left or \right, it is then considered as nothing which is a delimiter. Here it comes, $\left.\dfrac{\partial f}{\partial x}\right|_{x=0}=13$ prints $\left.\dfrac{\partial f}{\partial x}\right|_{x=0}=13$.

The commands \left and \right are also good when we do not want to change size, only define orientation. For instance, $x\in]0,1]$ was obtained by $x\in]0,1]$ and does not look too nice. We do not want to change size, just convince TEX that the first] is a left delimiter: $x\in\left]0,1\right]$ prints the nicer $x \in \left]0,1\right]$.

Unfortunately, this automatic adjustment does not always work well, sometimes it makes the size unnecessarily large. This happens often with sums, then we can shave off the buffer using \shave (see above) or we simply force the right size ourselves using \Bigl and similar control words. Another reason for manual adjustment is symmetry of delimiters. TEX always positions delimiters in such a way that their center is 2.5 points above the baseline. If we have a formula that is not vertically ballanced, the automatic size determination can be expected to overshoot.

Three delimiters deserve special remark. First, with \left and \right we can use < and > instead of \langle and \rangle. This also applies to constructions that use \left and \right, for instance \abovewithlimits.

Second, the difference between (and \lgroup is that the latter is thicker and after enlarging it is flatter: $\Bigl($ and $\Bigl\lgroup$ give $\Bigl($ and $\Bigl\lgroup$.

Third, those moustaches only work with enlarged sizes. In their ,,natural" size they look different, $\lmoustache1$ gives $\lmoustache1$. This $\rmoustache$ is $\lmoustache\rmoustache$.

One last remark: Note that \left and \right expect a delimiter, not an argument that is a delimiter. What is the difference? We can do \left[and \Bigl[, we can also do \Bigl{[}, but we cannot do \left{[}.

Sizes and styles.

When TEX typesets mathematics, it chooses shapes and sizes depending on where the resulting picture should be positioned. The basic size is the one for mathematics that will be used directly in text. This is called text style, we will refer to the size of characters and symbols as t-size. If the math part is to be displayed, then letters and most symbols are in t-size, but some symbols are enlarged and some elements (limits) are positioned differently. We will call it display style or d-size. Then there are two smaller sizes. Script style refers to a situation when something should be placed as a super/subscript, then letters and symbols get smaller (if possible), in s-size. However, even in a super/subscript we may ask for another super/subscript, then we get the scriptscriptstyle where characters are even smaller, in ss-size. For instance, in the power $1^{2^{3^4}}$ obtained using $1^{2^{3^4}}$ we see 1 in t-size, 2 is in s-size and 3 and 4 in ss-size. For comparison, in 1^{2^3} obtained using ${1^2}^3$, both 2 and 3 are in s-size, because they are both superscripts of a textstyle expression.

When we put a fraction in text style mathematics, its parts will be printed in s-size. But when a fraction is displayed, its numerator and denominator are in t-size. If these contain other fractions, the size goes down by one.

We can ask TEX to typeset in a specific style using commands \displaystyle, \textstyle, \scriptstyle, and \scriptscriptstyle. They work like fonts, that is, they are switches and we can resctrict their effect using groups.

For instance, the fraction $\frac{\frac{13}{123}}{\frac{1234}{13}}$ was printed in the t-size. If we want it larger but for some reason do not want it displayed, we put $\displaystyle\frac{\frac{13}{123}}{\frac{1234}{13}}$ and get $\displaystyle\frac{\frac{13}{123}}{\frac{1234}{13}}$, which is really much nicer. One has to be careful when using this command, because we force TEX to do something without regard to surrounding circumstances. For instance, if we try $\frac{\frac{13}{123}}{\displaystyle\frac{1234}{13}}$,

we asked for $\frac{\frac{13}{123}}{\frac{1234}{13}}$.

Some fun stuff: `$${\scriptstyle f(x)=x}\hbox{ is smaller than }f(x)=x.$$` gives

$$f(x)=x \text{ is smaller than } f(x)=x.$$

Try to find `$\scriptscriptstyle f(x)=x$`: $\scriptscriptstyle f(x)=x$.

We usually play with sizes when the size TEX chose is too small for our purposes (e.g. if we want to put our document on the Internet, small elements then often become unreadable). For instance, we saw a sum with a two-row limit $\sum_{\substack{i,j=1\dots n\\ i\ne j}}$ that was obtained using `$\sum\limits_{\substack{i,j=1\ldots n\\ i\ne j}}$`. It may be better to write `$\sum\limits_{\textstyle{i,j=1\ldots n\atop i\ne j}}$`. and get $\sum\limits_{\textstyle{i,j=1\ldots n\atop i\ne j}}$.

$\mathcal{A}$ $\mathcal{AMS}$-TEX has alternative control words `\dsize`, `\tsize`, `\ssize`, and `\sssize`. It also has control word `\dfrac` which is a combination of `\dsize` and `\frac`, and `\tfrac` standing for `\textstyle\frac`, say $\frac12$ will look like this even in DMM if we use `$\tfrac12$`. Similarly we have `\dbinom` and `\tbinom`.

Sometimes we want to write a symbol and we do not know beforehand in which style it will appear (this applies mostly to situations when we want to define a new macro, see the corresponding chapter below). Then we can use the construction `\mathchoice{`*f1*`}{`*f2*`}{`*f3*`}{`*f4*`}`, TEX will print *f1* if it is in the display math, *f2* in math, *f3* in script and *f4* in scriptscript.

Assigning classes.

We learned some ways to build new objects and we will learn more later. Often we want a new object to behave in a certain way. We already know that we can change a delimiter into a relation using `\bigm` etc., and we can assign orientation to delimiters. We also saw a way to create new operators using `\mathop`. This last example is actually just a special case of a general approach.

Symbols are divided into eight categories in math mode: ordinary, binary operation, relation, operator, opening (delimiter), closing (delimiter), punctuation, and inner (inside delimiters). This affects spacing between adjacent symbols (an 8×8 table defines what space is made when this symbol goes after that symbol) and other things. There are seven control words that take the argument and treat it as if it was of the corresponding class: `\mathord`, `\mathop`, `\mathbin`, `\mathrel`, `\mathopen`, `\mathclose`, `\mathpunct`. There is no control word for inner. For instance, `$\mathopen]0,1\mathclose[$` does the correct spacing for $]0,1[$ and `$\mathopen|-x\mathclose|$` does $|-x|$ instead of $|-x|$ that one would get by writing `$|-x|$`.

Note that `\mathopen` and `\mathclose` only assign orientation for spacing, but they do not turn the object into a delimiter. Thus we cannot do `\Big{\mathopen=}` and things like that.

These class assignments are hidden behind many standard macros, for instance `\lim` is in fact just a shortcut for `\mathop{\rm lim}`, while `\log` is interpreted as `\mathop{\rm log}\nolimits`. We also saw a control word for building new relations by putting things on top of existing ones, in fact `\buildrel` *arg1*`\over` *arg2*, is defined as `\mathrel{\mathop{\kern 0 pt` *arg2*`}\limits^{`*arg1*`}}`. That `\kern` is there to make sure that the main symbol does no get shifted, see Stacking symbols below.

It is actually useful to see things like that, since they can serve as inspiration for our own tricks. If we want to print this: $\overset{\infty}{\circ\!\!\rightarrow}$**Wow!**, we can do it like this: `$\mathop{\hbox{\bf Wow!}}\limits^\infty_{\circ\!\!\rightarrow}$`. We will return to this topic below (Stacking symbols).

Note that `\overbrace` and `\underbrace` discussed above include `\mathop`, which in particular means that the resulting construction treats any limits that follow as if there was `\limits` included. Thus `$\overbrace{A+B}^C$` prints as $\overbrace{A+B}^{C}$.

4. Getting complicated in math (arranging objects)

In the previous chapter we were mostly concerned with getting symbols on paper. Here we will look at different ways of arranging them. We will also encounter the biggest differences between plain and $\mathcal{AMS}$-TEX, almost nothing is done in the same way.

Matrices.

Basic matrices—that is, arranging objects into rows and columns—are done using the control word `\matrix`. Objects are supplied as one argument, first one row, then `\cr`, then second row etc., items in a row are separated by `&`. For instance, `$\matrix{a & b \cr c & def\cr}$` prints as $\matrix{a & b\cr c & def}$. Note that the resulting matrix was considered one object again that can be made a part of a line, individual entries are centred in columns and a `\quad` was inserted between columns. As expected, entries in a particular row are aligned by their baselines.

Since spaces are ignored in MM, we preferred to put them in the matrix specification, it is easier to read then things like `a&b&11&-3`. We can actually skip the last `\cr` that end the bottom line, the `\matrix` command (and all its relatives) will then supply it automatically.

Each entry in a matrix becomes a group of its own, so if we want to change the font in one entry, we can put just `\rm x` there instead of `{\rm x}` and it will work. But that also means that if we want to change font in all entries, we have to do it for each one separately. We cannot write `{\rm x & y}`, since this would cross groups.

We can put delimiters around, $\begin{vmatrix} a & b \\ c & d \end{vmatrix}$ was obtained using `$\left|\matrix{a & b \cr c & d}\right|$`. For the most popular type there is a special control word, `\pmatrix` typesets the corresponding matrix and then adds parentheses around it, `$\pmatrix{a & b \cr c & d}$` does $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

We need not always fill in all the entries, and we can even skip specification for some, then the longest row is taken as a pattern maker and if some row contains less entries, the remaining ones are automatically filled as blanks. `$\pmatrix{a & b \cr c & d & e\cr f & & g}$` does $\begin{pmatrix} a & b & \\ c & d & e \\ f & & g \end{pmatrix}$.

We can put suitable dots in a matrix easily usind `\dots`, `\vdots` and `\ddots`, for instance `$\matrix{11 & \dots & 1n\cr \vdots & \ddots & \vdots\cr N1 & \dots & Nn}$` does $\begin{pmatrix} 11 & \dots & 1n \\ \vdots & \ddots & \vdots \\ N1 & \dots & Nn \end{pmatrix}$.

How can we modify this basic behaviour?

If we want a smaller matrix, we have to do it ourselves, `${a\,b\choose c\,d}$` does $\binom{a\,b}{c\,d}$, another idea is `$\bigl({a\atop c}{b\atop d}\bigr)$` that does $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)$.

After every `\cr` we can put `\noalign` with an argument that is printed between the corresponding lines, with no regard to the matrix, even with no regard to math, it is considered a normal text. One possible use is to spread rows a bit, `$\pmatrix{a & b \cr\noalign{\medskip} c & d}$` does $\begin{pmatrix} a & b \\ \\ c & d \end{pmatrix}$.

We can also try `$\pmatrix{a & b \cr\noalign{$\longrightarrow$} c & d}$` to get

$$\begin{pmatrix} a & b \\ \longrightarrow \\ c & d \end{pmatrix}.$$

Why is the right delimiter so far to the right? Because that extra text is put in as a whole line, that is, it has the width of the line. If we want just the small part to be put in, we have to use `\noalign{\hbox{$\longrightarrow$}}`

Another example: If we put into that matrix `\noalign{\hbox{experiment}}`, it does $\begin{pmatrix} a & b \\ \text{experiment} \\ c & d \end{pmatrix}$.

One popular application of matrices is for **functions defined by cases**. The picture $|x| = \begin{cases} x, & \text{for } x \ge 0; \\ -x, & \text{for } x < 0. \end{cases}$ was typeset using `$|x|=\left\{\matrix{x, & \hbox{for }x\ge0;\cr -x, & \hbox{for }x<0.}\right.$`

Since this construction is used often, TEX offers the command `\cases` that is a modification of the above construction. It expects an argument consisting of two lines of text ended with `\cr`, each line with one `&` in it. The text before `&` is treated in math mode, the text after it is treated in text mode. All entries are aligned to the left and a space is left between the columns, there is a brace on the left side.

Thus `$|x|=\cases{x, & for $x\ge0$;\cr -x, & for $x<0$.}$` prints as $|x| = \begin{cases} x, & \text{for } x \ge 0; \\ -x, & \text{for } x < 0. \end{cases}$

What if we want a different alignment in a matrix? We can move items around by including spaces in entries. If we want them flush left or right, we can use the space `\hfill` that is as large as possible (see the following chapters). For instance, `$|x|=\cases{\hfill x, & for $x\ge0$;\cr \hfill-x, & for $x<0$.}$` prints as $|x| = \begin{cases} \quad x, & \text{for } x \ge 0; \\ -x, & \text{for } x < 0. \end{cases}$ Similarly, `$\matrix{1\hfill \cr 12\hfill \cr 123\hfill}$` creates $\begin{matrix} 1 \\ 12 \\ 123 \end{matrix}$.

TEX also has an interesting command `\bordermatrix` that leaves the first column and the first row out of the matrix. `$\bordermatrix{0 & 1 & 2\cr 0 & a & b\cr 0 & A & B}$` does $\begin{matrix} & 0 & 1 & 2 \\ 0 & & & \\ 0 & & \begin{pmatrix} a & b \\ A & B \end{pmatrix} & \end{matrix}$. Note that the resulting object is not centred by its middle, but by the middle of the matrix.

$\mathcal{A}$ In $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX things are essentially the same, we just supply them in a different package. A matrix is defined by putting `\matrix`, then we specify its contents with rows separated by `\\` and entries in a row separated by `&`, and then we put `\endmatrix`. Example: $\begin{matrix} a & b \\ c & d \end{matrix}$ was obtained using `$\matrix a&b\\c&d\endmatrix$`. Note that we did

not have to put \\ at the end of the last row.

$\mathcal{A}$ All the basic facts are the same, since this new \matrix just calls on the plain TEX version. So we can leave out entries, put delimiters around, or insert extra rows with \noalign. $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX has more matrices with delimiters prepared: \pmatrix is exactly the same as \left(\matrix and \endpmatrix is exactly the same as \endmatrix\right); there are also pairs \bmatrix—\endbmatrix (with [ ]), \vmatrix—\endvmatrix (with | or \vert), and \Vmatrix—\endVmatrix with \| or \Vert. They can mix, $\pmatrix 0 & 1\endbmatrix$ gives $\begin{pmatrix} 0 & 1 \end{bmatrix}$, but one cannot use one of these control words containing delimiters together with \matrix or \endmatrix. The reason is simple, for instance \pmatrix—\endmatrix would have one \right missing.

$\mathcal{A}$ Note that once you load $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX, the command \matrix gets redefined, so it is no longer possible to use it in the plain TEX way. This remark applies to all control words that are given by the same word (\cases and \cases) but their behaviour in $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX is different. There is a way to recover the old meaning, see below.

$\mathcal{A}$ When it comes to dots, $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX has not only the usual \vdots and \ddots, but it has also \hdots (which is like \dots, but the name nicely fits with the others). Moreover, it also offers an easy way to fill more slots with dots. The control word \hdotsfor expects one argument, a natural number that specifies how many columns started from the left should be filled with dots. Naturally we use this as the first entry in the corresponding row. Since these dots fill the specified number of columns, there's no need to put any entries for them, not even &'s.

$\mathcal{A}$ If the dots are to start later in a row, we put there the construction \innerhdotsfor *number*\after, where \after expects one argument which is put between the previous entry and dots. In a normal matrix there is a \quad between columns, so that is what we usually put there. *number* tells how many columns are to be filled. Note that in the case of \hdotsfor this number was an argument, so for instance 13 would have to be put into a group. In the latter construction, everything that is between the two control words is taken as *number*. Again, one does not put anything for columns filled with dots.

$\mathcal{A}$ If you don't like the spacing of dots, you can use \spacehdots *number1*\for, which works just like \hdotsfor. The argument of \for says how many columns have to be filled, but the dots are spread approximately *number1* times. Finally, for dots further in a row there is the construction \spaceinnerdots *number1*\for *number2*\after, where *number1* is the spacing again, *number2* tells how many columns is to be occupied and \after is as before. Here comes one all-embracing example:

```
$\vmatrix 12 & 345 & 678 & 90\\
 \hdotsfor2 & abc & \innerhdotsfor1\after{}\\
 xyz & \spaceinnerhdots0.25\for2\after\quad & ?\endVmatrix$
```

looks like $\left\|\begin{matrix} 12 & 345 & 678 & 90 \\ \dots\dots\dots & abc & \dots. & \\ xyz & \dots\dots\dots\dots & ? & \end{matrix}\right\|$.

$\mathcal{A}$ $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX offers an easy way to customize matrices using \format. This control word is put right after the opening mutation of \matrix and then there is a line which looks like an ordinary row—it ends with \\ and entries are separated by &—but instead of matrix entries there are things that control the way the matrix looks. Once we start doing it ourselves, no spaces between columns are put automatically, so that is one of the things we put into this format line. For every entry we also put one of the following control words: \c means that entries in this column will be centred, \l puts them flush left and \r puts entries to the right. Since the matrix size is now fixed, we cannot use more entries in a row then specified in the format line (but we can put less). Example:

```
$\pmatrix\format\quad\c\quad & \l & \r\\
 x & 0.1 & 1\\
 x+y & 0.01 & 10\\
 x+y+z & 0.001 & 100\endpmatrix$
```

will print as $\begin{pmatrix} x & 0.1 & 1 \\ x+y & 0.01 & 10 \\ x+y+z & 0.001 & 100 \end{pmatrix}$. The position of spaces is flexible, for instance the second \quad could have been put after & into the second column.

$\mathcal{A}$ The formatting line has one interesting feature that is helpful for matrices with many columns of the same format. You start by a specification that is empty (that is, you put in & &) and then you put one item with a certain specification. This defines a matrix that has as many columns as necessary and they all have that specific format. You can put different specifications before the empty one and the first few columns will be set accordingly. For instance, if the format row is \format \c & & \qquad \l\\ then the matrix will have the first column centred and after that there will be as many columns as needed, each will be set flush left and preceded by \qquad.

$\mathcal{A}$ **Split functions** are done using \cases—\endcases which works like a 2×2 matrix with columns aligned to the left and the left brace added. Therefore writing `$f(x)=\cases x, & x\ge0;\\ -x, & x\le0.\endcases$` does exactly the same as if we wrote

```
$ f(x)=\left\{\matrix\format\l & \quad\l\\
 x, & x\ge0;\\-x, & x\le0.\endmatrix\right.$
```

namely this: $f(x) = \begin{cases} x; & x \ge 0 \\ -x; & x \le 0. \end{cases}$ Note that unlike plain TEX's \cases, this construction does not switch to text

mode in the second column, so we have to use `\text` if we want to put some text there.

Note that if we put the period after `\endcases`, then it will be put after the whole construction on the level with the baseline.

$\mathcal{A}$ The last matrix-like object is a **commutative diagram**. In $\mathcal{AMS}$-TEX it is done using `\CD`—`\endCD`, rows are separated by `\\`, entries in a row are separated by spaces (!) and there is no entry for the central item. Horizontal arrows are done using `@` and `<` or `>` as we saw it before, for a vertical arrow up there is `@AAA`, and `@VVV` does an arrow down. The text between the first two `A` resp. `V` is put to the left of the arrow, everything between the second and third goes to the right. Hence this commutative diagram

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\alpha'} \\ X' & \xrightarrow[f']{} & Y' \end{array}$$

can be obtained as follows:

```
$\CD X @>f>> Y \\ @V\alpha VV @VV\alpha'V \\ X' @>>f'> Y'\endCD$.
```

$\mathcal{A}$ If you want to have an empty corner, just leave it out, but one has to use `@.` for no arrow. To get a long horizontal equality sign instead of an arrow, use `@=`, while `@|` or `@\vert` will do the vertical equality sign.

Stacking symbols.

We already saw some tricks for stacking symbols vertically. If they should be of the same importance, we can use `\atop`, say `${w\atop\to}$` does $\overset{w}{\to}$. Note that both symbols appear in s-size, but if we displayed this, they would be in t-size, `\atop` works just like a fraction. We can easily fix the size, for instance, if we always want the components in t-size, we can use `${\textstyle w\atop\textstyle\to}$` and get $\overset{w}{\to}$.

Often we prefer a tighter fit, for which we can borrow `\oalign` from text mode. This works like a matrix with one column, but beware that the entries are in text mode. For instance, `$\oalign{w\cr$\to$\cr}$` does $\underset{\to}{w}$ (note that "w" is now in roman, we need to put `$w$` into that construction). The resulting construction is aligned by its first row, there are ways to change it, but for that we need to wait.

Often we have one symbol that is the main one and we want to put others above or below in a subservient role, which means that we expect it to get smaller. If the main symbol is a relation, then we can use `\buildrel`, for instance `$\buildrel w\over\to$` does $\overset{w}{\to}$. For general symbols we can use the idea that is built into `\builder`. We take the main symbol, change it into an operator and then we can put symbols in smaller size on top and below using `\limits`. For instance, to get $\mathop{S}\limits_a^b$ we use `$\mathop{S}\limits_a^b$`.

Technical note: Sometimes the main symbol gets shifted down with respect to the baseline. If you want to prevent it, there is a trick: Just include some space in this symbol, and since the size of the space does not matter, it is best to use space of size 0. `$\mathop{\kern 0 pt S}\limits_a^b$` now does $\mathop{S}\limits_a^b$.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX all the above tricks work, but there is more. The construction `\overset` *something*`\to` takes that *something* and puts it (in smaller size) above the argument of `\to`. Hence `$\overset w\to\to$` looks like $\overset{w}{\to}$. Note that the first `\to` is a part of the construction, the second one does the arrow, no confusion can arise. Similarly `\underset`—`\to` puts *something* under the argument of `\to`. There are also `\oversetbrace`—`\to` and `\undersetbrace`—`\to` which work in the same way, but the argument is enveloped with a horizontal brace. So `$\overset\implies\to{\undersetbrace strange\to{A+B}}$` looks like $\underbrace{A+B}_{strange}\overset{\Longrightarrow}{}$. One more picture: the embedding $\ell \overset{C}{\hookrightarrow} X$ was obtained with `$\ell\overset C\to\hookrightarrow X$`.

$\mathcal{A}$ If something is under/overset to a large sum-like operator, the resulting thing is still considered a large sum-like operator. The same applies to binary relations and operators. $\mathcal{AMS}$-TEX has also a special provision for putting things next to operators. `\sideset` *thing1*`\and` *thing2*`\to` is a command which allows us to put things to the left (*thing1*) and to the right (*thing2*) of a large sum-like operator, which is specified as an argument of `\to`. The new symbol is again treated as a large sum-like operator. The position of the symbol can be controlled using `_` and `^`. For instance, `\sideset\star\and^*\to\sum` will print (when used in MM) as $\star \sum^*$.

Tagging formulas in DMM.

When we display a formula, we may want to add a tag next to it. This is done by putting `\eqno` *text* just before the closing `$$`. The *text* between `\eqno` and the closing pair `$$` is then put to the right of the displayed formula that follows, and is treated in math mode. `\leqno` puts the tag to the left.

For instance, `$$\sin^2(x)+\cos^2(x)=1.\eqno(1a)$$` prints

$$\sin^2(x)+\cos^2(x)=1. \tag{1a}$$

If the displayed formula is taller than one line (it may contain a matrix or several aligned lines, see below), then this tag is vertically centred.

$\mathcal{A}$ $\mathcal{AMS}$-TEX offers something more (although \eqno still works the same there). The format is $$*text*\tag *tag*$$, there are two differences. First, this construction adds parentheses automatically, so we could have written \tag1a in the example above, and the text of tags is printed in text mode. Second, tags created using \tag are put to the left. One can change it, control words \TagsOnRight and \TagsOnLeft work like switches and do the obvious thing. There is one special feature. If the tag is closed in double quotes ", it will be printed as it is, without parentheses. Here comes an example: $$\TagsOnRight1<2\tag"---deep truth"$$ does

$$1 < 2 \qquad \text{---deep truth}$$

From now on, all tags will be on the right.

Aligning formulas in MM.

Now we are getting to the most interesting part: what if we want to display more equations? The natural solution $$f(x)=x^2+1$$$$g(x)=\sqrt x$$ looks like

$$f(x) = x^2 + 1$$

$$g(x) = 2\sqrt{x}.$$

Not bad, but there is a big space between them, it looks too loose. Also, what if we want them aligned? Then we would need to place those formulas in one DMM part and we already have the basic tool for it, the \matrix construction. It is quite flexible, it even allows us to move items left and right using \hfill, so even quite complicated things can be done this way. However, for the most usual constructions TEX provides special control words.

If we just want to centre formulas, we use \displaylines, formulas ended with \cr are supplied as an argument. For example, $$\displaylines{f(x)=x^2+1\cr g(x)=2\sqrt x}$$ does

$$\begin{aligned} f(x) &= x^2 + 1 \\ g(x) &= 2\sqrt{x} \end{aligned}$$

As usual, in the last row we may skip the \cr, so we did. One should keep in mind that aligned lines cannot be broken at the end of a page.

The centering business is done using \hfil, so we can override it in various ways using \hfill as explained in further chapters.

If the formulas include tall objects, they may seem too crowded, then it is nice to know that lines in aligned displays can be spread using \openup, this has to be placed before \displaylines (or other similar math aligning command). We will see an example below.

Note that \displaylines cannot be made a part of another construction, it simply centers several lines and nothing more is expected to happen. That is, there should be $$ right after the closing brace } of \displaylines and the only thing before it that is truly safe is \openup.

Very often we not only want to put formulas together, but we also want them aligned, typically by equality sign. Then we use \eqalign, where we again enter individual formulas separated by \cr as argument, but this time each formula should include one &. These symbols are not printed, but used as follows: All lines are put in such a way that &'s are vertically aligned, the whole thing is then centred.

Example: $$\eqalign{f(x)&=x^2+1\cr g(x)&=2\sqrt x}$$ does

$$\begin{aligned} f(x) &= x^2 + 1 \\ g(x) &= 2\sqrt{x} \end{aligned}$$

Note that now we cannot adjust positioning using \hfill. On the other hand, this construction is more flexible than \displaylines. It becomes one object, so we can use it as a part of further constructions, we can even place a tag next to it (one for the whole construction). In the following example we also show how to spread lines and add braces.

$$\left.\openup 2 pt\eqalign{f(x)&=x^2+1\cr g(x)&=2\sqrt x}\right\}\eqno(\star)$$ does

$$\left.\begin{aligned} f(x) &= x^2 + 1 \\ g(x) &= 2\sqrt{x} \end{aligned}\right\} \qquad (\star)$$

Another trick: \noalign can be used after \cr, its argument is put just before the next row to the left of the box created by \eqalign. This can be used to put extra skip between certain lines.

Note that \eqalign is in fact just a two-column matrix with no space between columns, the left one is set flush right and the right one is flush left.

What if we want to put a tag next to individual formulas in an aligned display? The way \eqno works it is obvious that we cannot have more of them in one displayed math unit. For multi-tagging we have a version of the previous command that also allows for specification of one tag per line. \eqalignno again expects as an argument lines with one & as aligning position, but now there can also be a second & and everything between this second ampersand and the ending \cr is treated as a tag.

`$$\openup 2 pt\eqalignno{f(x)&=x^2+1&(1)\cr g(x)&=2\sqrt x\cr h(x)&=13&(1a)}$$` does

$$\begin{aligned} f(x) &= x^2+1 && (1)\\ g(x) &= 2\sqrt{x} \\ h(x) &= 13 && (1a) \end{aligned}$$

If we want those tags on the left, we use \leqalignno.

$\mathcal{A}$ In $\mathcal{AMS}$-TEX all the above work, but it may be better to avoid mixing two different approaches and focus solely on $\mathcal{AMS}$-TEX's special aligning commands.

$\mathcal{A}$ For simple aligning of several formulas that are then centred it has the construction \gather *formulas*\endgather, where we supply lines separated by \\. Unlike \displaylines, every line can be tagged, in that case everything between \tag and the following \\ or \endgather is considered a tag. Hence
`$$\gather f(x)=x^2+1\tag1\\g(x)=2\sqrt x.\tag"[2]"\endgather$$` gives

$$\begin{aligned} f(x) &= x^2+x^2 && (1)\\ g(x) &= 2\sqrt{x}. && [2] \end{aligned}$$

In this example we also put in the period, note its position.

$\mathcal{A}$ How do we align? The picture

$$\begin{aligned} f(x) &= x^2+1 && (1)\\ g(x) &= 2\sqrt{x}. && [2] \end{aligned}$$

was obtained using
`$$\align f(x)&=x^2+1\tag1\\`
`g(x)&=2\sqrt x\tag"[2]"\endalign$$`.

$\mathcal{A}$ So the construction \align—\endalign is similar to \gather but we also put the aligning sign & at every line. Formulas are moved left/right so that all these signs are vertically aligned, the resulting construction is then centred. Note that if we put & *after* the equality sign, then TEX will no longer put the binary relation spaces there.

$\mathcal{A}$ If there is no & in some line, TEX assumes that it is at its end. To show how it works we type this:
`$$\align a+b=c\qquad&\\ d+e=f\tag"no \&"\\&\qquad g+h=i\endalign$$`, here it comes:

$$\begin{aligned} a+b&=c \\ d+e&=f && \text{no \&} \\ g+h&=i. \end{aligned}$$

$\mathcal{A}$ The commands we just saw simply put together several individual lines without wrapping the whole thing up, so the resulting construction cannot be further used as a whole. This is a good thing (we can tag every line), but also a bad thing (we cannot tag it as a whole, we may want to use it as a part of another formula).

$\mathcal{A}$ If we want a new object, we use modifications \gathered—\endgathered and \aligned—\endaligned. Now we cannot tag individual lines, but the resulting construction can be further used. Example:
`$$\left.\aligned a&=b\\a&<a+1\endaligned\right\}\implies b<b+1\tag13$$` looks like

$$\left.\begin{aligned} a &= b \\ a &< a+1 \end{aligned}\right\} \implies b < b+1. \qquad (13)$$

$\mathcal{A}$ Note that the baseline of the newly constructed unit runs through the centre. The construction created using \topaligned—\endtopaligned will have the baseline going through the top line and to have the baseline determined by the last line we use \botaligned—\endbotaligned.

$\mathcal{A}$ There may be a problem when one of these "multi-line displays" occurs at the end of a page. Normally, TEX does not split \gather, \align and other similar constructions, but we can allow it or force it. The later is easy, just put \displaybreak at the right place, that is, right after \\ ending the line that is to be the last one on

a page. \allowdisplaybreak at the same place allows TEX to break a page there if needed. One can also put \allowdisplaybreaks right after the opening $$ and the multi-line display then can be broken after any line. However, none of this applies to the -ed constructions, since they create one unit.

$\mathcal{A}$ Now we mention several less frequent constructions.

\alignat—\endalignat allows us to align more columns of formulas. \alignat expects an argument—how many columns is to be made. It is a bit messy, because in a line there will be many &'s playing two roles: odd ones are for aligning and even ones separate columns. That is, if there is to be n columns, then there must be $2n+1$ of &. We'd better look at an example.

```
$$\alignat2 f(x)&=x^2 & f´(x)&=2x\\
    g(x)&=\sin(x) & g´(x)&=\cos(x).\endalignat$$
```

gives

$$\begin{aligned} f(x) &= x^2 & f'(x) &= 2x \\ g(x) &= \sin(x) & g'(x) &= \cos(x). \end{aligned}$$

Again, each line can be tagged, and there is the \alignedat—\endalignedat version as well. On a similar note, \xalignat—\endxalignat will put the columns more apart and \xxalignat—\endxxalignat will spread them out to the margins, which prevents one from using tags.

$\mathcal{A}$ The construction \split—\endsplit works like \align, but only one tag is allowed (it must be put after \endsplit). If tags are on the left, it is put level with the first line, if they are on the right then it appears at the bottom line. It is intended for formulas like

$$\begin{aligned} f(x) &= \sqrt{(x+y)(x-y)+y^2} \\ &= \sqrt{x^2} \\ &= x \end{aligned} \tag{1}$$

which was done using `$$\split f(x)&=\sqrt{(x+y)(x-y)+y^2}\\&=\sqrt{x^2}\\&=x\endsplit\tag1$$`.

\multline—\endmultline has the same tagging rules but puts the first formula flush left, the last one is set flush right, and those between are centred.

$\mathcal{A}$ If we want to spread lines in the above constructions, we use \spreadlines{⟨*dim*⟩}, it spreads lines in aligned constructions and also in matrices in the DMM group within which it was used, but its effect can be further restricted with a group. \spreadmatrixlines works similarly, but only applies to matrices.

$\mathcal{A}$ We can insert an unrelated material to aligned constructions and matrices. The control word \intertext should be used right after some \\ and expects one argument, it is then printed in text mode as a new paragraph. One can start a new paragraph using \endgraf there. \foldedtext can be used right after some \\, too, and its argument is printed to the right next to the formula that has just ended. Again, one can use \endgraf to start a new paragraph and the width of the text can be specified with \foldedwidth{⟨*dim*⟩} used right after \foldedtext.

That's about all concerning basic TEXniques. Now we should be able to write an average mathematical paper. There is no other chapter dedicated to mathematics, but many topics from the following chapters apply (making spaces, custom definitions of new macros etc.) and some sections of the next chapter focus on the math mode.

5. Getting TEXnical 1 (sizes, global parameters, fonts)

Here we will learn to influence the shape and size of the document.

Lengths

For size specifications we need **dimension**. Such a specification is denoted (*dim*) in this manual; it consists of a real number (unless specified otherwise) and a unit. TEX recognizes the following units: cm, mm, in, em (the width of "m" in italic), ex (the height of "x"), pt (point, the traditional measurement for type sizes, there are 72.27 pt in 1 in, this parenthesis is 10 pt high), and pc (pica, there are 12 points to pica). Less used are sp (scaled point, which is really a small unit, there are 65536 sp in 1 pt) and bp (big point, there are 72 big points per inch).

When specifying a dimension, there may be a space between the number and the unit, but commas and spaces cannot be used within the number. Some examples: +3 pc, .3in, -0.01 cm, 0 pt. Note that one cannot just write 0, it must be zero of something. The font we are using now is sometimes called 10 points typeface, actually the letters are 8 points high and go 2 points deep. The distance between baselines is 12 points or 1 pica, so this typeface is also called "10 to 12".

It should be remarked that \quad and \qquad are not dimensions but instructions for TEX to leave a horizontal space of dimensions 1em and 2em.

Lengths in TEX are internally all converted to integer multiples of sp. Since visible light has wavelength approximately 100 sp, it is fine enough. The advantage is that all arithmetics is in integers, so it should not be platform dependent. Maximal dimension is 2^{30} sp, which comes to about 5.76 m.

Another kind of length is **glue**. It is a dimension which also has some stretchability and shrinkability, so that it can be adjusted if needed. The format for (*glue*) is (*dim*) `plus` (*dim*) `minus` (*dim*), for instance, we can specify the glue `12pt plus 1pt minus 2 pt`. The dimension after `plus` defines how much can be added, the dimension after `minus` defines how much can be taken away. All three dimensions can be negative, we can also skip one of the two glue parts (e.g. `1cm minus -2mm`). A glue specification will be denoted (*glue*) here.

Note that TEX need to recognize where a description of a glue ends. If we plan to finish earlier, say, `2cm plus 1mm`, then if the following word is not "minus" we need not worry. However, what if the next word is "minus". Whenever we want to break such a logical chain, it is safer if we put the command `\relax` there, it interrups the train of thought for TEX. So it would be safer to put dimensions as `13pt\relax` and the above glue as `2cm plus 1mm\relax`. Usually we do not worry and we get lucky, but if it happens that TEX starts acting up, this is a possible problem to check on.

Glues are used in most places when typesetting documents. A typical example is the inter-word space, which is adjusted to make nice margins. When TEX typesets a certain object (for instance a line), all glues available are stretched/shrinked proportionally with respect to their allowed stretchability/shrinkability, and once they are determined, they are set. Similarly, distances between paragraphs are glues.

There is a special unit one can use for stretch/shrinkability: `fil`, `fill`, and `filll` are different "degrees of infinite". For instance, `1 pt plus 1 fil` defines an infinitely stretchable glue, that is, the object with this length will be as large as possible under circumstances. When several such glues are encountered, it is first determined what is the highest degree used there (say, there might be five glues, two with finite stretchability, one with `fil` and two that have `fill`, then the highest level is `fill`). Then the "finite" glues and all but the highest order glues do not get any stretching/shrinking at all, while the highest fills share available space proportionally (e.g., `2 fill` vs. `5 fill`).

Dimensions that are important (page width etc.) are kept in special registers and we can change them, which is actually the topic of the following sections. We assign a new value to a register like this: *register*`=`*value*.

Size of the document

TEX allows us to enlarge the whole document, this is done using `\magnification=`*size*. This control word must be used before the text itself starts and the whole document is typeset in the required *size*. The default *size* we have here is 1000 and we can use any number that is not too different, depending on the fonts we have in our computer. For instance, `\magnification = 2000` will make everything twice as big. But this example is not a good one since TEX has troubles when the ratio is too large and characters are not enlarged properly, 2000 is already too large. There are several sizes which are recommended to use since TEX knows how to treat them. These sizes are denoted by `\magstep` and a natural number between 0 and 5. The size `\magstep0` corresponds to the basic size 1000, so if this document started with `\magnification=\magstep0`, nothing would change. `\magstep1` enlarges 1.2 times, that is, it is almost like `\magnification=1200`, the word "almost" is used because when we use `\magstep1`, it looks better. `\magstep2` enlarges $1.2^2 = 1.44$ times and so on, `\magstep5` makes the document $1.2^5 \approx 2.488$ times bigger. Here is how they look like:

magstep0aA,1aA,**2aA,**3aA,**4aA,**5aA.

There is also the size `\magstephalf` which enlarges the document by the factor of $\sqrt{1.2}$. This size is not as tiny as the one we use here in order to save space but not as big as `\magstep1`, which is probably the one people use most. Why is it better to use these predefined sizes? If we use numbers, TEX has to scale all objects into the size we specify. However, for the six `\magstep`'s above it already has fonts ready that look much better.

Now we address the problem of what does it mean "to enlarge". Almost all dimesions, both implicit (font, baseline skip, paragraph indentation etc.) and explicit (those that we specify in our text), get enlarged, which makes sense, this guarantees that shapes of various constructions (tables we created etc.) stay the same. However, with some dimensions it makes sense not to enlarge, namely the basic outline (height, width) of the document. Consequently, after changing magnification, we can still print on the same paper size as before, but lines and pages now most likely break at different places since the contents got larger.

If we want to specify a dimension and make it immune to magnification changes, we put `true` before unit, for instance `5 true cm` means 5 cm no matter how we enlarge the document.

Since `\magnification` specifies the size of the whole document, it can be used only once. There is no easy way to enlarge just a part of the document in TEX, such things have to be done by hand (see below).

Size of a page

The printed page size can be changed rather easily using `\hsize=`(*dim*) for the width and `\vsize=`(*dim*) for the height. The default values are `\hsize=6.5 true in` and `\vsize=9 true in`. The change of the width starts from the paragraph within which it was done, it can be actually used more times withing a text and we can restrict its influence using groups. On the other hand, we strongly recommend to use `\vsize` only once, at the beginning of the file, unless you really know what you are doing.

The typeset page can be moved within the real one. The upper left corner of the text is implicitly 1 inch horizontally and vertically from the corner of the paper. We can change it using `\hoffset=`(*dim*) for horizontal and `\voffset=`(*dim*) for vertical change, the positive direction is to the right and down. For instance, `\hoffset=0pt` does not change anything. This offset dimension does not get bigger when the size of the document is changed.

We can also specify width for all displayed math parts using `\displaywidth=`(*dim*). This can be done more than once.

Sizes related to paragraphs

A typical paragraph consists of lines, each line sits on a baseline and the distance between baselines is stored in `\baselineskip`. The default value is `12 pt`, we can change it using `\baselineskip=`(*dim*). `\baselineskip=1pc` does not change anything here.

If we want double spacing, we can use `\baselineskip=2\baselineskip` (we can call on dimensional registers when supplying dimensions, see below). However, the value of the original baseline skip would get lost this way, fortunately it is stored in a register `\normalbaselineskip`, so we can call on it. To simplify this, TEX offers the command `\normalbaselines` that reverts all parameters relating to baselines to their default values.

We can also change baseline skipping using the command `\openup`(*dim*) that increases `\baselineskip` by the specified amount, it is useful for local changes. Note that only the value of `\baselineskip` valid at the end of a paragraph is relevant. For instance, if we want to have a paragraph with 1.5 linespacing, we could enclose it in a group and finish it with `\openup 0.5\baselineskip\par}`.

There is also a vertical skip between paragraphs, but we do not see it since by default `\parskip=0pt`.For instance, if we want paragraph skip to be the same as the spacing between lines, we could use `\parskip=\baselineskip`.

Every paragraph starts with an indentation whose size is stored in `\parindent`, by default it is 20 pt long. One can change it, at the beginning of this manual I did `\parindent=5pt`. When `\parindent=0pt` is used, there is no indentation at all.

Both control words' influence starts from the end of the current paragraph. Now I will issue `\parindent=23pt` and `\parskip=.5cm` and we will see that the next paragraph will start with a larger indent after a small space. Within this next paragraph I again switch back using `\parindent=5pt` and `\parskip=0pt`. The validity of these control words can be restricted by using them within a group, but there one has to be a bit careful.

We can also use `\displayindent` to set default indentation for displayed math groups.

An interesting register is `\everypar`. We set `\everypar=`{*something*} and that *something* will be put at the beginning of every paragraph. Similarly we have `\everycr`, where the contents is included after every `\cr` (see tables below) and two analogous math registers, see the next paragraph.

Parameters for math

`\mathsurround=`(*dim*) defines the space that is put before and after all in-line math things (i.e. around `$`). For displayed math we have `\abovedisplayskip` (the space between a displayed math and the text before it) and `\belowdisplayskip`, these are used under normal conditions and their default value is the glue `12pt plus 3pt minus 9pt`. However, if the displayed math is short and also the adjacent lines of text are short, then there would be too much white space and TEX uses `\abovedisplayshortskip` and `\belowdisplayshortskip`, whose default value is much smaller, `0pt plus 3pt` for the former and `0pt plus 3pt minus 4 pt` for the latter.

The argument of `\everymath`, resp. `\everydisplay` is put before every in-line math part, resp. displayed math part.

We use registers covered in this chapter to set up the global look of the document. For instance, if we want to make a document look tighter, we may decrease the spacing around displayed math. But we also saw that sometimes it makes sense to change some register locally. Local formatting is the topic of the next chapter below.

Fonts

We introduced several fonts already at the beginning. These are the basic fonts that people usually find quite sufficient. However, when TEX starts, it actually pre-loads a few more fonts that we can use right away. Namely, for the roman family it knows fonts `\tenrm` (which is the same as `\rm`), `\sevenrm` (subscript size) and `\fiverm`, this looks small was printed using `this {\sevenrm looks} {\fiverm small}`. For boldface we have `\tenbf`, `\sevenbf` and `\fivebf`. The command `\it` is actually just an equivalent for `\tenit`, simiarly we have `\tensl` and `\tentt`.

However, sometimes we are not quite happy with these and want a different font. The standard distribution of TEX contains the Computer Modern font family, created alongside TEX, but before we can use some of these fonts, we have to load it. For that we need to know how fonts are treated by TEX.

Information about characters in a particular font is stored in two kinds of files. Files with an extension `.tfm` contain info about a rough shape and size and these are used when TEX compiles the tex-file. When we use some driver to visualize/print the file, TEX gets precise information from files with an extension `.pxl`; since these files are rather huge, they are usually stored in a compressed form with an extension `.pk`. For instance, the default

roman font is stored in `cmr10.tfm` and `cmr10.pk`. When we want to load a font, we call it by the name of the corresponding file (without extension) and we assign it a name, that is, we turn it into a control sequence. The format is `\font\`*controlseq*`=`*fontname*.

Names in TEX follow a common pattern. They start with a family name shortcut (`cm` standing for Computer Modern), then there is a shortcut for the font type and finally its size in points. Thus if we define `\font\title=cmr12`, we will get a control word `\title` that switches to somewhat larger roman.

The most common Computer Modern fonts come in the basic size 10, bigger sizes 12 and 17 and in small sizes 5 (for second order indices), 6,7 (for first order indices), and 8,9. Less popular fonts have less sizes, some have 5,7,10,12, some have 10 only. At the end of this chapter we list some common Computer Modern fonts with their file names. Generally, larger fonts are useful for titles, while smaller fonts are handy for footnotes, running heads (see the top of the page) and similar purposes.

We can ask for the fonts to be scaled when loaded. There are two possibilities:

`\font\`*controlseq*`=`*fontname* `at` *size*

`\font\`*controlseq*`=`*fontname* `scaled` *size*

The difference is that with the first specification we supply *size* as dimension, while in the second specification as a scaling number that gets divided by 1000, similarly to `\magnification`, we can also use `\magstep`.

For instance, if we want a boldface font with size 12 (which is 1.2 times larger than the standard one), we can try any of the following:

```
\font\mybold=cmbx12
\font\mybold=cmbx10 at 12 pt
\font\mybold=cmbx10 scaled 1200
\font\mybold=cmbx10 scaled \magstep1
```

The first option is preferable, since the font `cmbx12` is not just `cmbx10` scaled, but it is also subtly modified for better viewing. However, the letters are therefore a bit leaner, which one may not like (which is why in this manual we use the fourth version for all titles). Note that the second and the third version are equivalent and the least preferable of the four.

Things work well with respect to global scaling. If we make the whole document bigger using the command `\magnification=\magstep2`, our font `\mybold` would be scaled again, getting to the size $1.2 \times (1.2)^2$ (which is `\magstep3`), and the title would be proportionally bigger relative to text again, we need not worry.

Note that when we switch to a smaller font, the distance between lines does not get automatically adjusted, that is up to us if we want it. Also, font changes affect only the text parts, math formulas are still printed in the original default 10pt size. Changing font size for both math and text is rather involved and we will not go into it here.

Getting creative with scaling fonts is no problem for TEX, but we may run into trouble later, for instance a printer may object to some sizes. However, given that modern TEX installations usually work through postscript, this is rarely a problem now.

Here is the promised table of basic fonts that should be available in a typical TeX distribution. Most likely you have hundreds more, just check out the relevant directories. Note that the standard boldface `\bf` is the `cmbx` font.

cmr	roman	ABCabc123	cmsl	slanted	*ABCabc123*
cmb	boldface	**ABCabc123**	cmbxsl	boldface slanted	***ABCabc123***
cmbx	boldface extended	**ABCabc123**	cmss	"sans serif"	ABCabc123
cmcsc	caps and small caps	ABCabc123	cmssbx	sans-serif boldface	**ABCabc123**
cmti	italic	*ABCabc123*	cmssi	sans-serif italic	*ABCabc123*
cmbxti	italic boldfaced	***ABCabc123***	cmssdc	sans-serif boldface	**ABCabc123**
cmu	unslanted italic	ABCabc123	cmtex	tex	`ABCabc123`
cmtt	typewriter	`ABCabc123`	cmdunh	Dunhill	ABCabc123
cmitt	italic typewriter	*`ABCabc123`*	cmmi	mathematical italic	*ABCabc*123
cmsltt	slanted typewriter	*`ABCabc123`*	cmmib	math italic boldfaced	***ABCabc*123**
cmtcsc	csc typewriter	`ABCabc123`	cmex	mathematical symbols	
cmvtt		ABCabc123	cmsy	mathematical symbols	

Computer Modern also includes the font `cminch`, which contains upper-case letters and digits only and they are one inch high.

There are many other fonts coming with various packages. One popular family is Euler fonts. One of these fonts, `euex`, contains mathematical symbols. Several other fonts from this family are similar to the Old English font. Their names start `eu` and then there are two more letters. The third letter in the name denotes type, `r` is a wide font only slightly changed, `f` is a narrow font which really looks Gothic, `s` contains mathematical symbols. The fourth letter can be `m` (normal) or `b` (boldface). **𝔗𝔥𝔦𝔰 𝔴𝔞𝔰 𝔴𝔯𝔦𝔱𝔱𝔢𝔫 𝔲𝔰𝔦𝔫𝔤** `eufb10`. The last font we mention offers the Cyrillic, "**абцде фгхик лмноп чрсту вщшыз**" is actually "abcde fghik lmnop qrstu vwxyz" written using `wncyb`.

A $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX has the control word `\oldnos`, it expects one argument which should be just numbers which are then printed like this: 183.7629.

Modern TEX installations offer various ways to tap on other font families, most notably TrueType fonts and Postscript fonts. However, since the TEX way relies on calling appropriate files, these methods are platform dependent, for instance to get the classical Times, on one computer it might be `\font\mytimes=PS-Times-Roman`, while on another `\font\mytimes=Times at 10pt` (normally 10pt need not be specified, but here the default size was 1pt for some reason). Of course, this platform dependency goes against the basic philosophy of TEX, so we will not go into it here. Check out documentation coming with the your particular installation.

Fonts in math

Letters in mathematics are written using a font that looks like italic, but it is not quite the same, see `cmmi` in the above table. We can change this as well. We can use the usual fonts `\rm` etc. in MM, but this is usually not a good idea since the text spacing rules are not suitable for mathematical formulas, we use it only in special circumstances (e.g. function names). In fact it is not assumed that a typical user would change font used with mathematical symbols, so TEX does not offer a direct way to load a special math font and switch to it like we do in text mode.

$\mathcal{A}$ Warning! In $\mathcal{AMS}$-TEX it is not possible to use `\rm` etc. in MM. It does offer some consolation, the control words `\bold`, `\roman`, `\italic`, and `\slanted` that expect one letter and write it in the appropriate font. It really wants just one letter, the only exception is when we want an accented letter, this can be put into a group like `\bold{\vec x}`. There is a similar control word `\boldsymbol` which works with Greek letters and some delimiters, and `\boldkey` which can handle binary relations/operators and delimiters, punctuation, letters (in this case it produces boldfaced italic), and numerals. As a last resort there is a control word `\pmb`, which expects one argument and it is printed in boldface. Well, sort of, "pmb" stands for "poor man's boldface" and it is achieved by printing the thing four times in slightly different positions. Apparently this is to be used in an emergency only.

$\mathcal{A}$ $\mathcal{AMS}$-TEX also offers `ma` and `mb` lines of extra mathematical symbols. There are two sets of extra characters which can be loaded and then used. The latter set is loaded using `\loadmsbm` and offers nice letters $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, ... , $\mathbb{Z}$ obtained by `\Bbb A`, `\Bbb B`, ... The former set is loaded using `\loadmsam` and characters for MM it offers immediately after loading are listed here:

⌜ \ulcorner	⌝ \urcorner	⇢ \dasharrow	✓ \checkmark	® \circledR
⌞ \llcorner	⌟ \lrcorner	⇠ \dashleftarrow	✠ \maltese	¥ \yen

$\mathcal{A}$ Other characters from **msam** and **msbm** fonts have to be assigned individually before use. For each character we create a control sequence that calls on it by its number using `\newsymbol` *name number*. The character numbers consist of four digits (or letters), those from **msam** start with 1 and those from **msbm** start with 2. For instance, a black square has number 1004, so we can do `\newsymbol\blcksqr 1004` and then use it (in MM): `\blcksqr` gives ■. If we do not insist on our own names, we can (after loading the fonts) use the command `\UseAMSsymbols` that assigns all symbols to implicit names. Some more popular choices follow, symbols in the right column are binary operators.

□	\square	1003	⧫	\blacklozenge	1007	⊡	\boxdot	1200
■	\blacksquare	1004	★	\bigstar	1046	⊞	\boxplus	1201
△	\vartriangle	134D	∡	\measuredangle	105D	⊟	\boxminus	120C
▲	\blacktriangle	104E	◁	\sphericalangle	105E	⊠	\boxtimes	1202
▽	\triangledown	104F	∄	\nexists	2040	∖	\smallsetminus	2272
▼	\blacktriangledown	1048	⁄	\diagup	213E	⋉	\ltimes	226E
◊	\lozenge	1006	∖	\diagdown	231F	⋊	\rtimes	226F

6. Shaping text 1 (paragraphs, spaces, aligning)

The topic of this chapter is special formatting that changes default behaviour for just a part of a text.

Shaping a paragraph

If you want just one paragraph to start without an indent, there is a control word `\noindent` which does it when used at the beginning of a paragraph. If we use `\noindent` within a paragraph (that is, after it already started), then it is ignored. Note that it already starts that paragraph, that is, switches to horizontal mode. The meaning of this remark will be made clear later when discussing innards of TEX.

Another possibility is to end a paragraph with `\flushpar` instead of `\par`, the paragraph that follows will start without indentation.

Conversely, we can always use `\indent` to force indentation. When used after a paragraph ended (in vertical mode), this command starts a new paragraph with indentation. When used within a paragraph, it produces a horizontal space of size `\parindent`, these spaces add.

If we start a paragraph with `\hang`, the second and all successive lines are indented by `\parindent`. Since the first line already is indented by default, the whole left edge of this paragraph will be vertically aligned and moved right.

We can move left and right edges of a paragraph. \leftskip=(*dim*) tells TEX how much should the left margin be moved to the right, negative value moves to the left. \rightskip=(*dim*) moves the right margin to the left for positive dimensions, both registers have default value 0. We used \rightskip=1cm at the beginning of this paragraph. Note that the size of a paragraph is determined at the moment when it ends using the last defined value of all relevant parameters. In order to restrict the effect of \rightskip to just this paragraph, we enclosed it into a group. The ending looks like this: "look normal.\par}". If we ended the source text of this paragraph this way: "look normal.}\par", then the closing brace would cancel the effect of \rightskip and the paragraph would end while the default value 0 is valid, that is, this paragraph would look normal.

There is also a control word \narrower which "squeezes" a paragraph from both sides by the size of \parindent, in fact this control word increases both \leftskip and \rightskip by the amount of \parindent. Consequently, effects of multiple use add and also here only the last defined value matters for a particular paragraph.

If one wants to change only a part of a paragraph, \hangindent=(*dim*) must be used. This control word moves the left margin to the right if (*dim*) is positive and the right one to the left if (*dim*) is negative. It affects the whole paragraph within which it was used and if there is more of these commands, only the last one counts. And now the important part: \hangafter=*number* has to be used right after \hangindent and restricts its action. If *number* is positive, it tells how many lines won't be changed by \hangindent, counted from the first. Negative *number* tells how many lines will be affected starting from the first one again. An important note: all dimensions with control words in the last three paragraphs are changed by \magnification and we can use true to prevent it.

For a complete control over the shape of a paragraph there is the construction \parshape=$n\ i_1\ l_1\ \ldots\ i_n\ l_n$. For $k < n$, the k^{th} line will have indentation i_k and length l_k (both being (*dim*)), and lines from n^{th} on have indentation i_n and length l_n. This only affects the paragraph within which it was used. The effect can be cancelled using \parshape=0.

At the end of every paragraph, \parshape, \hangindent, \hangafter, and \looseness are reset to 0. The latter expects an integer n as an argument, and the paragraph which is affected will be made longer/shorter by n lines than it would be otherwise.

Often we want to offset a paragraph in a special way:

(1) This is just a demonstration.
   (1a) Once upon a midnight dreary, while I pondered, weak and weary Over many a quaint and curious volume of forgotten lore—While I nodded, nearly napping, suddenly there came a tapping.
   (1b) I needed something longer than one line, so enters the Raven.
(2) Shall be lifted—nevermore!
   I again need something long, but the poem has already ended so I will try to think of something else, well, it seems to be long enough already.

We can get this thing using the control word \item, which expects one argument and then some text afterwards. The first paragraph after \item is all intended by the current value of \parindent (as if the left margin shifted to the right) and the argument of this \item is printed on the first line into the reserved space. If this indent is too small for that argument, it sticks out to the left. There is also a control word \itemitem which works similarly but moves the left margin by "two parindents". The spaces between the argument of an \item and the text after it are ignored. Note that the control word \item includes \par and therefore ends the previous paragraph. So here is how it was done:

```
\item{(1)}This is just a demonstration.
\itemitem{(1)}Once ... tapping.
\itemitem{(1)} I needed ... Raven.
\item{(2)}Shall be lifted---nevermore!\hfil\break
 I again ... already.\par
```

Note two things. It was necessary to use \hfil\break (in $\mathcal{AMS}$-TEX we can use \newline) in (2). If we used \par there, it would end the item (2) and the text after it would be printed normally. What if we want several paragraphs in one indented item? We do the first as usual with the item mark as an argument and then we put the remaining ones with \item{}, that is, with no mark. Note that the \par after the last item was necessary. The the previous items were closed by \par's included in \item's that came later, but the last one has to be closed by us. Note also that in this manual we have a very small indent, so we changed to \parindent=20pt for the duration of that itemized thing.

If we start a paragraph with \textindent, then the following happens. The paragraph is typeset as usual, that is, the first line is indented and the rest set flush left. However, this control word expects an argument. To this argument the enspace is attached on the right and the resulting object is positioned flush right into the empty space left by indentation in the first line. In other words, it is works like \item, just the lines that follow after the first one are not indented.

Spaces.

We start with **vertical spaces**. The control word \vskip ends the previous paragraph (i.e. it includes \par) and

then leaves a space of the specified size, the text after this starts a new paragraph. The format is \vskip(*glue*), often we use just \vskip(*dim*). This control word is quite smart. First, if this space falls to the top or bottom of a page, then it is not done. This makes perfect sense, for instance there is no use of having a blank space before a chapter title that starts a new page. Moreover, if the space you require is equal or bigger then what is left of a page, no space is made and paragraphs are spread down.

The control word \vglue(*glue*) does almost the same, difference being that it is always done exactly as specified, no adjustments are made. This is useful for instance in case you want to add a picture later. If the space happens to be at the end of a page and there is not room enough for it, paragraphs are spread down and the space is made on the top of the next page. \vglue includes \par as well.

Both of these accept negative values, the text than backs up, that is, it overlaps what was written before it.

We already saw commands \smallskip, \medskip and \bigskip that leave a space between paragraphs. These are composed macros, \medskip is \vskip\medskipamount, where \medskipamount is `6 pt plus 2 pt minus 2 pt`. Similarly, \bigskipamount is `12 pt plus 4 pt minus 4 pt` (the same as \baselineskip), \smallskipamount is `3 pt plus 1 pt minus 1 pt`. If we decide to change these defaults, it is wise to maintain the original ratio (`med` should be half of `big` and `small` half of `med`).

\topskip(*glue*) leaves the specified white space at the top of the current page.

A $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX offers \smallpagebreak, \medpagebreak, and \bigpagebreak that can be used between paragraphs and leave some empty space. The sizes are the height of "x", the height of "k", and the big one leaves a space which is as big as the distance between baselines. Also, if a page is to be broken in a paragraph following any of the above control words, TEX rather breaks it between the paragraphs (i.e. at the place where the control word is) and spreads paragraphs down.

A $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX also has \midspace. This one expects (*dim*) as an argument and works similarly to \vglue, the difference being that the space that is left out is a little bit bigger than what is specified. \topspace leaves a space a bit bigger than that specified by an argument again, but this time it is done at the top of a current page. If there is already some empty space made by another \topspace, the new one is added below the previous one. Another difference is that none of them includes \par. Both can be used with another one, \caption. \caption must be used right after \midspace or \topspace and expects some text as an argument, this text will be printed below the space created by \midspace or \topspace. It can be longer then one line and it can contain some TMM parts as well. The width of the caption can be specified using \captionwidth. The dimension is put as an argument and the caption of the proper size is then centred. As an example, after ending a paragraph we will try `\midspace{1cm}\caption\captionwidth{5 cm}{There will be some description of a picture.}` to get

There will be some description of a picture.

The text in a caption cannot be broken into more paragraphs.

Now some **horizontal spaces**. \hskip(*glue*) leaves a space within a line. It is similar to \vskip: If the space falls at the beginning or the end of a line, it is ignored. The same is true for \quad and \qquad. There is one exception to this rule, the space is really made if it is at the beginning of a paragraph or some line construction (like \line).

The command \hglue(*glue*) is similar to \vglue, it leaves a space within a line that is always done. It then forms an unbreakable unit with the word (object) that comes before and after this space, so if this whole unit happens to span the end of a line and this line cannot be comfortably broken before, TEX will leave this thing (glue and the word after it) sticking out of the line.

We can also add a space of size \parindent using \indent in a horizontal mode (within a line).

Universal space: \kern(*dim*) makes horizontal or vertical spaces, depending on which mode it is used in (roughly: within a line or not). This space is ignored if it falls to the end of a page or a line. Note that \kern only accepts dimensions, not glues. Quite a few other space-making commands use \kern to actually do it.

Stretchable spaces are based on the dimension `fil`. \hfil is a horizontal space that is as long as possible (up to a whole line-width). When we write `Short text\hfil\linebreak`, the words "Short line" are printed with their natural spacing, since \hfill will make a space occupying the rest of a line, and then the line break occurs. Another nice trick, `\line{Left margin\hfil Right margin}` does

Left margin Right margin.

In the case there are more \hfil's in one line, they divide the available space between themselves evenly as illustrated here, `\line{One\hfil Two\hfil Three}` will do

One Two Three.

Now look at `\line{\hskip 3 cm One\hfil Two\hfil Three}` printing

One Two Three

or this `\line{One\hfil\hskip 3 cm Two\hfil Three}` doing

One Two Three

or finally `\line{One\hfil Two\hfil\hfil Three}` which looks like

One Two Three.

If you feel playful, you can use `\line{\hfil Title\hfil}` instead of `\centerline{Title}`. There is also `\hfill` which has a higher priority when allocating the available space, in `\line{One\hfil Two\hfill Three}` there will be no space left for `\hfil`:

OneTwo Three.

This is very handy. In some constructions there is an implicit `\hfil` involved, for instance to make things printed to the left. Using `\hfil` on the opposite side we can neutralize it (then we have centred things) or we can override it using `\hfill`. In order to work properly, there must be something definite to the left and right of this stretchable space.

`\vfil` and `\vfill` are vertical equivalents, they make vertical spaces within one page. For instance, we already saw that to break a page without spreading the material all the way down, we use `\vfil\pagebreak`.

Both `\hfil` and `\vfil` are ignored if they appear at the beginning of a line, resp. a page. This does not apply to `\hfil` within `\line`-like constructions. There are two interesting commands, `\hfilneg` and `\vfilneg`, that represent "infinite negative stretchability". They are often used to cancel out `\hfil`'s and `\vfil`'s.

Finally, `\hss` and `\vss` have both infinite stretchability and shrinkability.

It is possible to use other objects than white space to fill the specified dimension. For four objects there are special commands `\dotfill`, `\hrulefill`, `\rightarrowfill` and `\leftarrowfill` behave just like `\hfill`, bit with a difference in appearance. `\line{Dots\dotfill Rule\hrulefill Arrow\rightarowfill Period.}` looks like

Dots.....Rule———————Arrow———————→Period.

In general we can use any kind of a rule or even any object to fill the available space. The specification has the form `\leaders` *object*`\hskip`(*glue*) or `\leaders` *object*`\hfill`. If we do not use a rule, we need to supply the object as a horizontal box (see Boxes). For instance, `\dotfill` is defined as `\leaders\hbox to 1em{\hss.\hss}\hfill`. Example: `\line{A \leaders\hrule\hskip3cm B \leaders\hbox{here }\hfill C}` does

A ———————B here here here here here here here here here here here here here here here here here here C

We can add a space before B by using `\leaders\hrule\hskip3cm\ B` there.

If two `\leaders`' follow, the boxes are vertically aligned. One can also use `\cleaders` (these are centred and adjacent, space is left at the sides of the whole thing) and `\xleaders` (centred and spaced equally).

Spaces in math

We can use `\hskip` or `\kern` in math, they work as expected, but it is better to use special tools available. There are analogous commands `\mskip` and `\mkern` that can be used only in math and we also use a special unit with them, namely `mu`. We have 18`mu`=`em`, but in fact this only refers to the usual math font in its basic size. The interesting thing about `mu` is that its size changes depending on the font. For instance, if we use it in a subscript, it gets smaller, which means that a mathematical symbol that we compose using this unit keeps its shape regardless of where in a formula we use it. Dimensions specified with `mu` will be denoted (*mudim*) or (*muglue*) and they cannot be used with `\hskip` and similar commands described above.

In MM, we can precede a glue or `\kern` specification with `\nonscript` and this glue/kern is then ignored when it happens to be in a sub/superscript.

$\mathcal{A}$ $\mathcal{AMS}$-TEX also has the control word `\vspace` that adds some space between lines in super/subscripts, multi-line displays and matrices. It should be used right after `\\` ending the line after which we want the space and expects (*dim*) as an argument. Example: `$\sum\limits\Sb i\in M\\\vspace{-3pt}j\in N\endSb$` prints $\sum_{\substack{i\in M\\ j\in N}}$.

Rules (lines).

Horizontal lines can be done using `\hrule width`(*dim*) `height`(*dim*) `depth`(*dim*). Any of the parameters can be omitted, the default values are depth = 0 pt, height = 0.4 pt, and width is equal to the width of a page (or, in restricted situations, all available space). This control word first ends a paragraph, then draws a rule just below the previous line. `This is\hrule a line.` will do

This is

a line. The words "a line" start a new paragraph. Note that the rule is too close to lines (it is drawn exactly next to the text before and after), consequently even the spacing between lines was spoilt. One can use `\vskip` before and after `\hrule`, another possibility is to use `\strut` in the text, it is an imaginary character that is as tall and deep as the largest letters plus some extra space, which is just right for keeping things apart. Indeed, the outcome of `This is\strut\hrule\strut a line.` looks much better:

This is

a line.

For **vertical lines** we use \vrule height(*dim*) depth(*dim*) width(*dim*). This construction does a vertical rule which is considered a part of a line, it does not include \par. This is very handy, for instance one can do the QED mark

by \rightline{\vrule width 7 pt height 7 pt depth 0 pt\qquad}.

Implicit values of \vrule are width .4 pt, depth 2 pt, height 8 pt, so typing \vrule here will produce here. If the vertical rule is longer, the distances between lines are enlarged so that it fits. Again, in some specific situations the height of a \vrule is determined by room available (see Boxes below). There are no requirements on dimensions we supply, so we can do for instance \vrule width 20 pt depth 0 pt height 3 pt and obtain , a horizontal rule, this time without interrupting a paragraph.

As with lengths, we can use \relax to avoid trouble when we leave some specification out and the text that follows starts with "width" or some other word that could have been considered a part of the rule description.

Aligning text (tables)

Text can be aligned into columns of constant (predetermined) width, this is started using the control word \settabs *number* \columns. This divides the width of a page into *number* columns of equal size, but only formally, it has no influence on the text we enter in the usual way. TEX just remembers where the tabs are placed (they stay there until we redefine them with another \settabs) and waits for us to call on them.

This is done by starting a line with \+, then there should be entries separated by &, and \cr at the end. What happens is that \+ ends a paragraph (it includes \par), the entries will be put into proper places (each in one column) on a new line, and then the text after \cr starts a new paragraph. If an entry is longer than the column width, then the text is printed into the next one, perhaps out of the page, TEX no longer checks these things once you start positioning text by yourself.

Here comes an example. First, we do \settabs 10 \columns right now and then we continue writing this as if nothing happened. But now we do \+ Algebra&& ODE &PDE.\cr and we will get

Algebra ODE PDE.

Note the position of the period. As you can see, we can skip some entries and end the line before using up all columns. Now we do \+&& Complex analysis\cr:

Complex analysis

and we can see that this longer text spread into the fourth column. The sign & has the meaning "go to the next tab" even if it had to be backward, since the "next" refers to order, not position. Here comes another example, \+This is quite long & This is after it\cr will print as

This is quite This is after it.

Now some technical things. The spaces *after* \+ and & (that is, before entries) are ignored. Entries are printed flush left and it is because there is \hfil involved in the construction after every item. So if we want some entry centred, it is enough to put \hfil before it. If we put \hfill there, it will override that implicit \hfil and the entry will be printed to the right. One can also use \hskip, \dotfill and other horizontal spaces within entries.

Let's look at some tricks. We will start with \settabs 4 columns to make columns bigger, then do \+1&2&3&4\cr to show the position of tabs and then

```
\+\hfil Centre& \hfill Right& \hskip 13 pt Moved & \hskip -8pt Back\cr
\+ \hfil\qquad$\star$& \hfill Almost right\qquad& \hfill End\cr
\+ \hrulefill Center \dotfill& \hfil\bf Bold& \hfill Real End& Wall\cr
```

and we obtain

1 2 3 4

Centre Right Moved Back

⋆ Almost right End

_____Center ..... **Bold** Real EndWall.

In the first line we confirmed that centering and moving items flush right worked as advertised, we also used a negative space to move an item out of its column.

In the second line we show two things. First, the effect of a space added at an appropriate place. In the first column, the whole unit \quad⋆ was centred, therefore the star appears a bit to the right. We also asked for something that ends with a white space to be pushed to the right. Most curious is the third entry. One would expect "End" to be printed to the right just like "Real End" below it, but it is not. The reason is that all \hfil's work properly in tables only if the next entry is not skipped, if there is something they can lean against. That's what "Wall" is doing in the third line. In fact it would be enough to end the second line with "\hfill End&\cr" and it would work fine.

In the third line we again see some interesting features. In the second column we need not use a group to restrict the effect of boldface font, since every item in a table is automatically taken as a separate group.

There is a second way to set tabs, we can explicitly supply widths of columns. To do that we follow \settabs by \+ and then describe columns in the usual way by putting some entries (typically spaces and \hskips) separated by & and end it with \cr. This specification is not printed, but used to set up tabs based on the supplied texts.

For instance, two columns, the first one 5 cm wide and the second one two \qquad's wide, can be set up using \settabs\+ \hskip 5 cm&\qquad\qquad\cr. The following example will illustrate several tricks:

```
\settabs\+\hskip 4 cm & Notebook\hskip 3 cm & \qquad Pri&ce\cr
\+& SX-25, 4MB RAM, 120 MB HD& \hfill 1750.&00\cr
\+& keyboard \& mouse, bag etc.& \hfill \underbar{ 100.}&\underbar{00}\cr
\+& Total& \hfill 1850.&00 CAD\cr
```

will print as

SX-25, 4MB RAM, 120 MB HD	1750.00
keyboard & mouse, bag etc.	100.00
Total	1850.00 CAD.

The first unused column shifted the table to the right. We could have done it by fixing three columns only and starting entries in the first one with \hskip 3 cm, or by closing the lines into a vertical box and then move it using \moveright. The third and fourth columns combine into something that looks just like one column of numbers. This is one reasonable way of getting numbers aligned by decimal points. One can also see how to underline things. We cannot use another \+ line with \hrulefill in the third and fourth columns as there would be an empty line inserted.

There is a third possibility for setting tabs, we can set them as we go. Whenever we are in a tabbed environment (in a line started by \+) and we use & when all the defined tabs were already used, a new one is defined at that place. Example: We now do \settabs \+ oneone & \cr, we get two tabs defined (note that we could have put something in the second column, but in fact this information would not be used, the second tab is already set with the & and nothing more needs to be done). Now we write \+ one & two & three\cr:

one two three

and two things happened. The text "two three" was placed at the second tab and a new tab was created where "three" starts. We try it with \+ 1 & 2 & 3\cr:

1 2 3

If there were no tabs set previously, we can even use this trick to set all tabs from scratch. We will now clear all tabs using \cleartabs and then do the following:

```
\+ {\tt if} $i<j$ &{\tt then}&\cr
\+ && $a:=b$\cr
\+ & {\tt else}\cr
\+ && $b:=a$\cr
\+ & {\tt endif}\cr
```

The outcome is

```
if i < j then
            a := b
        else
            b := a
        endif
```

Which brings us to the last topic. Tabs are in effect until they are rewritten with another \settabs command or cleared using \cleartabs. We can also make tab setting local by doing it within a group. Finally, note that in fact \cleartabs only clears tabs to the right of the current column, which gets important if we use it within some item in a \+ line.

Flexible aligning

TEX can also create tables where widths of columns adjust automatically according to their contents. The control word \halign expects one argument—the description of the table. Its rows are described similarly to tabbed tables, in every row entries are separated by & and the row is ended by \cr. But first one has to specify a pattern row where we specify how many columns we need and what objects should be put in each row by default. This description has the same form, it is a row of entries separated by & ended with \cr. In every entry there must be one character # that serves as a variable, where the real contents in rows will be put, other text that we place in the pattern row is replicated in all rows.

Example: \halign{X #&-#- here\cr short: & A \cr so long:&AAAA\cr} will do

X short: -A - here
X so long:-AAAA- here

Note that there is nothing that would mark the beginning of a row, everything that follows after \cr is considered the first entry of the next row. It worked as advertised, for instance in the second column, the lower entry is obtained by substituting "AAAA" into the pattern "-#- here". Important rule: all spaces that come *before* an entry are ignored, but not spaces *after* it. Indeed, the upper entry in the right column substitutes the text "A ", since we left a space between A and \cr, but the space before this A was ignored.

TEX first creates the contents of all rows by combining the pattern row with values that we specified for # in rows that follow. The width of every column is then set using the longest entry in it, in the second column the width is

given by the length of "-AAAA- here". The columns are put next to each other.

Placement of text in individual entries goes as follows: An entry starts at the left edge of the available space and if there are any spaces in this entry, then TEX spreads them so that the entry covers the whole width of the column space. The first item in the second column is "-A - here" and TEX did spread the two spaces available.

We will now tinker with this example a bit. First, we take away spaces around `A`, thus also the first entry in the right column will have no spaces and will be set flush left. Second, we add some spaces into the pattern row to spread the columns. From `\halign{ X # & -#- here\cr short:&A\cr long:&AAAA\cr}` we get

X short: -A- here
X long: -AAAA- here

Note that `\halign` includes `\par`, that is, it ends a paragraph, then it positions the table at the beginning of the next line and the text after the table starts a new paragraph (the table is an insert, extra piece of text inserted between two paragraphs, so the table itself is not indented).

This way of making tables has two advantages: First, the size can be made to fit the contents, second, everything that is common to all rows can be collected in the pattern row.

We can easily move entries within a column. If we put `\hfil` at the end of an entry, TEX will not spread it and puts it flush left. We move an *entry* to the right if we specify it as "`\hfil` *entry*", and "`\hfil` *entry* `\hfil`" will center it. Using `\hfil` in the pattern row handles all entries in this way. We now show an example that leaves the first column aligned left (but adds `\hfil` to prevent spreading) and centers the second column:

```
\halign{\vrule\ $\backslash${\tt #}\ \hfil\vrule & \vrule\ \hfil$#$\hfil\ \vrule\cr
     ast & \ast \cr
     times & \times \cr}      will do
```

\ast	∗
\times	×

Note that we even included `$`'s in the pattern row, this works well since for instance the top-right corner of the table is interpreted (after substituting) as `\vrule\ \hfil$\ast$\hfil\ \vrule`. However, it was not possible to include the backslash directly there. If we tried using `$\#$` in the pattern row, upon reading this TEX would understand `\#` as the control sequence for the symbol # and send an error message "`#` missing". Note that there is really no space left between columns, the rules are next to one another.

We can include some regular text into a table without referring to the pattern row mechanism. We supply it as an argument of `\noalign` that should be put right after `\cr` ending some row. This argument can include paragraphs, even DMM parts, and is normally printed as if TEX forgot for a moment that there was a table going on. If `\hrule` is used, it's length is equal to the width of the table in question, this is the typical use. Another popular use is to add space between rows in a table.

We now return to the above example, add `\noalign{\hrule}` around all rows and also add `\strut` into the pattern row to make sure that all rows will have a certain minimal height and depth. We can also add a little space between columns.

We will learn that when TEX breaks pages, the distances between lines are adjusted. But if rows get spread more than usual, it would mess up our table. Thus it is recommended to tell TEX not to spread lines while doing the table, for which we have the command `\offinterlineskip`. We use it right before `\halign` and close it all into a group to restrict the effect of that control word. If we want to center the resulting table, we need to first close it into a vertical box.

```
\centerline{\vbox{\offinterlineskip\halign{\vrule\strut\ $\backslash${\tt #}\ \hfil
\vrule\,& \vrule\ \hfil$#$\hfil\ \vrule\cr
\noalign{\hrule} ast & \ast \cr
\noalign{\hrule} times & \times \cr\noalign{\hrule}}}}      looks like
```

\ast	∗
\times	×

Now we will learn some ways to tweak this basic setup. First, sometimes we want to supply an entry without actually using the pattern predetermined for that column in the pattern row. This is done by preceding the entry with `\omit`. Such an entry is set up as if there was just `#` in the pattern row for this column. For instance, if we add `\omit N N & \cr\noalign{\hrule}` to the end of the previous table, then the outcome will be

\ast	∗
\times	×
N N	

Note that by using `\omit` in the first entry we did quite a lot. Along with `\vrule`'s we also disposed of `\hfil` designed to keep entries together, so as described above, TEX used the two spaces in the argument "N N " to spread the argument to the whole width. We also accidentally removed the `\strut` that guaranteed a certain height and depth of lines, so the height of the last row was determined by the size of "N". In the second column we see that it is possible to suply nothing as an entry.

We can put one entry across several columns. When we use `\span` instead of `&` when specifying a row, then the

entries before and after this \span are combined into one and it is placed into the space obtained by joining the corresponding columns. We can combine more entries into more columns.

Example: Assume that we have \halign with four columns and no patterns and we specify the row AAA & \span\span\hfil BBB\hfil\cr. This puts AAA in the first column and then centers BBB in the middle of the space determined by the other three columns.

If we have something predefined in the pattern row, it would get included in the contents, in such case we would have to use AAA & \omit\split\omit\split\omit\hfil BBB\hfil\cr. To make things easier, TEX offers the construction \multicolumn{*number*}*text*. The *text* is put in place of *number* following columns. In our example we would do AAA & \multicolumn{3}\hfil BBB\hfil\cr.

We can also make periodic patterns. If we use & when TEX expects to see #, then everything between this extra & and \cr is repeated as many times as needed to create enough columns for row specifications that follow. For instance, the pattern row =# & now & -#- & =#= \cr creates the following pattern: Every row starts with "=*argument* now" and after that, because there is no # in the cell # now #, the pair of columns "-*argument*-=*argument*=" is repeated as many times as needed.

In a typical case we would use && instead of & in a pattern row, then everything between this && and \cr is repeated.

We can also make global adjustments to such tables. The distance between two columns is defined in \tabskip. We can change this globally or locally, even in the pattern row, all successive &'s then use the new value.

We can also adjust the size of the resulting table. Instead of \halign we can use \halign to(*glue*) and the resulting table is spread/shrinked to have the predetermined width. Similarly, if we use \halign spread(*dim*), then the size of the resulting table is corrected by the amount (*dim*).

Last little thing: We can make horizontal lines of size roughly \quad by putting the character | into entries or in the pattern row.

Technical notes: 1) \halign takes into account the contents of \everycr. The mutation \ialign is just like \halign, but it ignores \everycr and resets \tabskip to its default value 0pt.

2) When TEX processes a table, it first reads the patern row, but does not interpret commands found there (see macro expansion below). It first reads characters, then substitutes for #, and only then replaces commands with their meanings. If we put \span into the pattern row, it does something completely different than we saw above: It tells TEX to immediately expand the macro that comes after it.

Vertical aligning

The control word \valign is very similar to \halign, but with one crucial difference. The material that we put in the form *entry* & *entry* & ... & *entry*\cr actually specifies not a row, but a column. Other than than, the behavour is very similar to \halign including various tweaks with \omit, \span etc.

7. Shaping text 2 (boxes, inserts, header/footer)

The basic unit for composing text is a box. For instance, every character is a box. Every box has a reference point on the left edge and three parameters: height, depth, and width (measured to the right from the reference point). A single letter is a box with the reference point going through the baseline.

When two boxes meet, they are aligned. Typical boxes are "horizontal boxes", when they meet, they are put side by side so that their reference points are on the same level (go through the baseline) and together form a new (horizontal) box that has its reference point on its left edge and on the original level. For instance, by joining individual characters we get a word.

There are also vertical boxes, some get created naturally (whole paragraphs), but we may also tell TEX to treat some boxes as such. When two vertical boxes meet, the second is put below the first so that their reference points align vertically. Together they form a new vertical box whose reference point is on the left edge and its vertical position is determined by the reference point of its last component (often last line).

TEX composes pages by manipulating boxes, but we leave this topic for the next chapter. Here we focus on boxes that we can manipulate by ourself.

A **horizontal box** is created by supplying some text as an argument of \hbox. For instance, \hbox{oh oh my} created a box "oh oh my" that is normally incorporated into the line here (you do not see any difference in its appearance). TEX sees it as one unit, in particular h-boxes are never broken at the end of a line. This makes h-boxes a handy tool for keeping things together. For instance, to make sure that $\ell \hookrightarrow X$ does not get split at line break, we write \hbox{$\ell\hookrightarrow X$}. Obviously this makes linebreaking more difficult for TEX and sometimes we need to help it a bit.

When creating a horizontal box, we have to take into account that it makes TEX think of lines, the purpose is to arrange objects in rows. Thus we should not do any vertical aligning directly in it (paragraph breaks, line breaks, displayed math etc.), it is ignored. For instance, if we type \hbox{Hey $$f(x)=x$$ here.}, we get Hey $f(x) = x$ here. (Note how spaces in the preceding line were spread, since we prevented TEX from breaking the line at its natural place.) There are ways to put more complicated structures into an h-box, but this must be done by hiding them in other boxes, we cannot do it directly.

Another thing to keep in mind is that the argument of \hbox is always taken in text mode. We already used this to include text in MM parts.

Since an h-box is a unit, it can be easily moved around by placing \lower(*dim*) or \raise(*dim*) right in front of \hbox. For instance, this was done using `... instance, \raise -2 pt\hbox{this} was ...`. As usual, a negative argument moves in the opposite way.

The width of the new h-box is by default determined by the width of its argument. We can also prescribe its width using \hbox to (*dim*). Then we will get a horizontal box that is (*dim*) wide no matter how long is the text inside. If the text is shorter, it will be spread according the rules valid for the \line construction. For instance, `\hbox to 2 cm{\vrule\ A B C \vrule}` will do | A B C | (note that \vrule's are as high as the box). This works great in combination with fills, one can create white spaces of various sizes, or this nice picture

Chapter Two 13

that was obtained by `\centerline{\hbox to 5 cm{Chapter Two\dotfill 13}}`. In fact, \line is actually defined as \hbox to \hsize. Instead of asking for a specific size directly, we may also ask for a box's natural width to be enlarged by a specified amount using \hbox spread (*dim*).

Vertical boxes are created using \vbox, where the argument should be some object of vertical nature, for instance a paragraph, we can also include MM parts, tables (see \halign above) and many other things. Naturally, we cannot use it in MM. One important aspect of v-boxes is that they cannot be broken at the end of a page, so again they are a good tool for keeping things together. For instance, if a theorem consists of just two lines, it would not be nice having them on different pages. We can write, say, `\vbox{\bf Theorem.\newline\it ... }`. Note that the argument is a group so only the text of the theorem will be written in italic. One piece of advice: never use \par right after } closing a v-box. If the end of the vertical box coincides with the end of a paragraph, use \par}. (V-boxes should contain vertical units of texts, so it wants to have a finished paragraph in it.)

Note that arguments of \hbox and \vbox are kept as groups, so we need not do things like `\hbox{{\it word}}` to restrict the effect of font change. This also means that all the usual rules about nesting apply.

The height of a v-box is determined by its contents, but we can force it to take another dimension by using \vbox to(*dim*), the contents is then spread/shrinked to predetermined value using rules for page breaking (that is, by enlarging vertical white spaces between individual components (paragraphs, displayed math parts etc.) if possible). Here's an interesting trick. `\vbox to 20 cm{\vfil\centerline{\bf TITLE}\vfil}` will put TITLE in the middle of a white space.

We can move vertical boxes horizontally by preceding \vbox with \moveleft(*dim*) or \moveright(*dim*), but only if it forms a paragraph by itself. Such v-boxes are treated as "inserts", something that was included between regular paragraphs. If there is \par before \vbox, this v-box is printed to the left (unless we use \moveleft or \moveright), and after this v-box, TEX returns to its usual paragraph composing.

This can have some interesting consequences. If we write `Text1\flushpar \vbox{Text2\par} Text3` then the following happens. Text1 ends and since it ended a paragraph, the following v-box forms a new paragraph. But, since it is just inserted between the two paragraphs (Text1 and Text3), it will be indented! Then Text2 ends and Text3 starts a new paragraph without an indent because of the \flushpar. In other words, the control word that ended a paragraph before some v-box affects the paragraph after this v-box. But we usually put \par everywhere, so normally we need not worry about such details.

We can also include v-boxes in text or other constructions, then it behaves differently, see below.

Things start getting real interesting when we **combine boxes**. When h-boxes are put into a v-box, they are arranged below one another and flush left. For instance, we now put `\vbox{\hbox{Short}\hbox{Long}}` as a separate paragraph (see discussion above), obtaining

Short
Long.

Note that it was not indented indeed, as described above.

The reference point of the resulting v-box is on its left edge, the level is determined by the level of the reference point of the lowest h-box. Every vertical box also has height and width assigned to it. If the v-box consists of an ordinary text, its width is equal to the page width. If it consists of h-boxes only, then its width is given by the length of the longest h-box. In the example above, the v-box was "Short" wide. Why is this important? If \hrule is used in a v-box, its width is equal to the width of the v-box. Just look at the following example:

`\vbox{\hbox{Short}\vskip 2 pt\hrule\vskip 2 pt\hbox{Longer}\hbox{The longest}}` will do

Short
Longer
The longest.

Those \vskip's made sure that lines don't get too close, we could have put \strut into the first two h-boxes instead. One can also use \vskip and \vglue in a v-box to spread the h-boxes.

We can also use \vbox within a line. Then it becomes a normal part of that text and it is printed so that its reference point lies on the baseline. We use the first example and not end a paragraph before we do it, here it comes: Short
Long . Of course, we cannot move such a vertical box horizontally, but we can use \lower and \raise

exactly as we did with \hbox. We can also use \vtop. This is a version of \vbox which positions the reference point according to the first, not last h-box. An example: this is \vtop{\hbox{an}\hbox{example}} looks like this is an example.

If we put some v-boxes into an h-box, the are arranged one next to another (without any space between them) and aligned by their reference points. The resulting horizontal box has a height that is given by the highest vertical box. If we use \vrule inside a h-box, its length is given by the height the h-box.

If we want the v-boxes to be aligned by the top line (top component), we use \vtop's instead. Note that the alignment does not use the first printed line; rather, it alignes using the baseline of the first building block of a given \vtop. If this \vtop consists of h-boxes and the first in turn consists of several lines, the alignment is done by the last line of this first h-box. Example: We start with a simple situation, two plain v-tops.

```
\hbox{\vtop{\hbox{One}\hbox{Two}\hrule\hbox{Three}}\vtop{\hbox{base}}}
```

does One base
Two
Three

We forgot to put \strut with "Two" and "Three" and thus it is too crowded. Note that the left v-box has \hbox{One} as its first component, therefore its baseline became reference level for it.

If we use \vbox, the left v-box will be aligned by its last line. What if we want alignment by the middle one? One possibility is to turn the first two entries into one component of the \vtop. Thus we first enclose the two h-boxes into a v-box, they will get aligned vertically and the new reference point will be by the last h-box, that is, by the baseline of "Two". However, in order to align remaining parts under it, we have to make TEX think that it is a horizontal box, so we enclose it into an h-box. This then becomes the first component of the whole \vtop, which therefore adopts the reference point of our construction—the baseline of "Two"—as its new reference point. We will also show how to spread boxes by adding some space between them and a \vrule.

```
\hbox{\vrule\,\vtop{\hbox{\vbox{\hbox{One}\hbox{\strut Two}}}\hrule\hbox{\strut Three}}\qquad
 \vtop{\hbox{base}}}
```

does One
Two base.
Three

The trick with using \vbox and \hbox to change status of some object can be very useful. All tables in this manual are produced using \halign and we center them by first making v-boxes out of them. Thus every table here starts with \centerline{\vbox{\offinterlineskip\halign{...}}}.

The mutual interplay of horizontal and vertical boxes can be used to produce **tables** of sort. An example shows it best.

We start with the \vtop from the example above, the first one that gets aligned by "One". We add struts and also a space before and after every entry, which makes the box wider and thus the horizontal line extends beyond those words. We similarly prepare another \vtop with Four etc., this time with line after the first entry. We align these two columns by enclosing them into an h-box, we also add \hrule's around all columns (before, between and after \vtop's), their heights will be given by the height of the resulting h-box.

Finally, we close this h-box into a v-box together with two horizontal rules.

```
 \vbox{\hrule\hbox{\vrule
 \vtop{\hbox{\strut\ One\ }\hbox{\strut\ Two\ }\hrule\hbox{\strut\ Three\ }}%
\vrule
\vtop{\hbox{\strut\ Four\ }\hrule\hbox{\strut\ Five\ }\hbox{\strut\ Six\ }}%
 \vrule}\hrule}
```

will look like

One	Four
Two	Five
Three	Six

Quite a nice thing, isn't it? Why did we put the percentage marks in? For instance, if we left out the second one, then the Enter after }} would be counted as a space and this space would be added between the second v-box and the right vertical line. As a consequence, there would be a gap between this vertical line and the horizontal lines in the box. As it is, everything after % is ignored up to the \vrule, exactly as we need. Now why didn't we put % at the end of the first line? Because there the Enter (=space) is used up to end the control word \vrule.

In a similar manner we can obtain boxed using

```
\hbox{\vrule\vbox{\hrule\hbox{ \strut boxed }\hrule}\vrule}
```

Note that the baseline goes through the basis of the boxed picture, \lower 2 pt or so would make it nicer. We could also use \vbox{\hrule\hbox{\vrule\ \strut boxed \vrule}\hrule} which could be further enclosed into an h-box if necessary.

We can use boxes to create **subpages** of a given size which can be combined into the normal page. For that we use `\vbox to` to determine vertical size of such a page, inside the group that defines the page we restrict its horizontal size. For instance,

```
\hbox{\vbox to 17 mm{\hsize = 5 cm This is a longer text that will be broken into more lines,
perhaps as many as three.\par
And a new paragraph.\par}\ \vrule\quad
\vbox to 2 cm{\hsize = 5 cm Some text hopefully longer than one line $$f(x)=\sin(x).$$
And math.\par}\vrule}
```

will do

This is a longer text that will be broken into more lines, perhaps as many as three.

And a new paragraph.

Some text hopefully longer than one line

$$f(x) = \sin(x).$$

And math.

We added vertical rules to show how the math part is centred, and spaces in the middle to spread the pages apart a bit. Boxes like this can be further manipulated the usual way, so for instance one can put several mini-pages into one printed page.

$\mathcal{A}$ $\mathcal{AMS}$-TEX offers a shortcut, the control word `\boxed` expects one argument and makes a nice box around it. It is actually made a little bit off, so we don't have to use `\strut` and we can even do `$\boxed{\boxed{1+1=2}}$` to get $\boxed{\boxed{1+1=2}}$. This control word can be used in TM as well, but then one cannot use MM in its argument.

Inserts.

By an insert we mean an object that is included in a page, but not through TEX's regular page-making routine. We already saw a example when a v-box is given as a separate paragraph, another popular insert is displayed math when not used as a separate paragraph. TEX offers three basic constructions to create inserts directly.

`\topinsert` *text*`\endinsert` takes the *text* and puts it at the top of the current page with `\bigskip` under it. If there is not room enough (e.g. another `\topinsert`), it is put into buffer and used at the first opportunity, most likely on the next page. The *text* is a page in miniature, it can contain `\vskip`'s, `\raggedright`, can have its parameters changed and all this is just local, it does not influence the rest fo the page.

`\midinsert` *text*`\endinsert` includes the *text* right away with `\bigskip` before and `\bigbreak` after. If there is not room enough left for it, it is put into buffer and used at the first opportunity on the top of some page, just like `\topinsert`.

`\pageinsert` *text*`\endinsert` takes the *text*, formats it into a full-size page and prints it at the next page.

All three pairs form groups so the usual rules apply and they can only be used between paragraphs. The buffer can become quite big sometimes. `\supereject` not only ends a page but also flushes out the buffer, everything is printed and only then the regular text follows.

`\vadjust` expects text of vertical nature (paragraphs, tables, etc.) and creates an insert as follows. After TEX encounters this command, it first finishes the line where it was, then puts the insert and then continues with the usual text. For instance, we put `\vadjust{\kern 2 pt\hrule}` just after "`finishes`" above.

Header and footer.

We can put something above the page using `\headline={`*something*`}`, and `\footline={`*something*`}` puts text below the page itself. The text is formatted as if it was an argument of `\line`, that is, it is spread unless we do something about it using `\hfil`'s. These two control words are in fact names for environmental registers and we supply their contents. When TEX finishes a page, the current text of `\headline` and `\footline` is put up and down. It follows that when we use this command, the given text is printed starting from the page where we used it and on all successive ones until we change it (for instance using `\footline={}` to have nothing there). Note that the text is written in the font that is used when the relevant page ends. Since we not always have a control over this, it is better to always include font specification in `\headline` and `\footline`.

The implicit value for `\footline` is `{\hfil\folio\hfil}`, where `\folio` is a name for `\the\pageno`, where `\pageno` is a variable storing the **page number**. In other words, the page number is centred at the bottom. The implicit value for `\headline` is `{}`. In this manual, after writing the title of a chapter we put another command, here it was `\headline={\hfil\sevenrm 7. Shaping text... \hfil}`.

Some tricks with page numbers: `\headline={Page \folio\hfil}` would print page numbers to the top-left corner preceded by "Page ". Interesting idea: `\footline={\ifodd\pageno\hfil\folio\else\folio\hfil\fi}` This prints odd page numbers to the bottom right corner and the even ones to the left (conditional commands are covered in a special chapter below).

We hinted that the page number is counted in `\pageno`. We can do something about it. The easiest thing is to use `\nopagenumbers`, page numbers stop shown. We can confuse the counting with `\pageno=`*number*, which persuades

TEX that *number* is actually the number of the current page and it will start counting from this one. If the number is negative, it will be actually decreased at every page and printed using Roman numbers. For instance, we can use `\pageno=13` and the pages will be counted 13, 14, ... A few pages later we can write `\pageno=-13` and pages will then be counted -13,-14, -15, ... but `xiii`, `xiv`, ... will be written. And then we can start from the beginning again, `\pageno=1`.

If we want to advance pages manually, the command `\advancepageno` comes handy. It increases `\pageno` by 1 if its contents is positive or zero, and decreases by 1 if it is negative.

Another object that people like to put down is footnotes. However, these are not put into the `\footline` area, instead, the regular page is ended a bit earlier to make room for the footnote. The specification is done using `\footnote`, which expects two arguments. The first one is printed right away, and both are printed under the page. If we want to make the symbol printed smaller and up (as is customary), we have to do it manually, usually drawing on math mode. Example: typing `some\footnote{$^\dagger$}{short} text` we get some[†] text.

The text of footnote can include paragraphs and displayed math, but no insert, in particular we cannot put footnote in a footnote. In general we should not use `\footnote` in nested constructions (a box within a box, subformula of a formula), the root of the problem is the mark. To fix this there is the command `\vfootnote` that works almost the same as `\footnote`, but it only places the text below, it does not place the mark in text itself, this is up to us. The above example could have been also done this way: `$^\dagger$\vfootnote{$^\dagger$}{short}`.

A AMS-TEX cannot do footnotes! For some reason, when loaded, AMS-TEX makes TEX forget all about `\footnote`. If they are needed, three possibilities are available. Probably the best is to save the meaning of `\footnote` before loading macros, see the chapter on macros below. It is also possible to load some style that includes footnote making or simply do it by hand (check how pages are broken and then put it in, add horizontal rule and some skips), but that is really the last desperate resort.

8. Getting lazy/efficient (macros, managing text)

Undoubtedly, this rather short chapter is one of the most important parts, because it allows you to customize the set of TEX's control sequences and thus work effectively and with a relative ease. Indeed, TEX allows us to define our own control sequences, which is great if there is some longer structure that you keep writing again and again (longer may even mean "longer than four letters", it is surely easier to write `\ep` instead of `\varepsilon`). It is done by `\def` *new-control-sequence*{*definition*}. For instance, `\def\ep{\varepsilon}` will define a new control word `\ep` which can be used as advertised above. Another handy definition is `\def\codim{\mathop{\rm codim}}` or `\def\codim{\operatorname{codim}}` for AMS-TEX users.

The definition of a new control sequence must come before we use it first. The name of a control sequence has to be either a backslash followed by several letters (both lower- and upper-case possible and they do make a difference) or a backslash and one non-letter. There can be a space right after `\def` and there can also be spaces between the new control sequence and the opening brace {. There is actually an exception to this which will be discussed later in this chapter, but that thing is more technical and is for real TEXperts.

TEX does not process definitions when reading a tex-file, so if we make a mistake in a definition, it is discovered only when we actually use it. When some unknown control sequence is used, TEX looks it up in the text that was read so far and substitutes the defined meaning instead. This replacement process is called "expanding a control sequence".

It follows that if we first define `\def\bs{$\backslash$}` and later type `use \bs and !`, then TEX understands it as `use $\backslash$and !` (here you see why we included dollars in this definition—we planned on using it in the text mode). Note that the space after `\bs` was spent to recognize the end of this control word. It would be better to write for instance `use \bs\ and !`.

Actually, TEX only scans all groups within which it is at the moment when an unknown control sequence was used, that is, a definition of a control word is valid only in the group where it was defined. Thus we can have one control sequence with more local meanings. Some neat tricks can be made using this.

If you define MM construction to be used in MM, it is vital not to put dollars into the definition, they would switch off MM. For instance, if we defined `\def\half{${1\over2}$}`, then we can use it only in text mode. If we type `$\half$`, TEX will interpret it as `$${1\over2}$$` and display the half, which is still better than `$f(x)=\half$` that is interpreted as `$f(x)=${1\over2}$$`. TEX will complain that we used `\over` in text mode.

It is possible to define a universal symbol. For instance, the definition `\def\2{\hbox{$\sqrt2$}}` can be used in both modes. In TM, `\hbox` is ignored and TEX properly does `$\sqrt2$`, that is, $\sqrt{2}$. If used in MM, then `\hbox` switches to TM and `$` back to math, so it works again.

When defining a new control sequence, one can also use something already defined in another definition, but beware of things like `\def\dumb{\dumb}` or the more inventive `\def\vicious{\circle} \def\circle{\vicious}`. TEX would keep substituting until you run ouf of patience and stop it forcibly (or until the next blackout). Only slightly better is `\def\dumb{less \dumb}`. In this case TEX would store "less" every time it goes around, after some time it would run out of memory, stop and let you know.

[†] short

Before we go to something more complicated, it should be emphasized that only the contents of the group in definition is used when doing that substitution. This is very important. For instance, we used {\sl dim} to get *dim* in this manual, and since it appeared frequently, it was made into a new control word. If the definition was like this \def\dim{\sl dim}, then when we use \dim, only \sl dim would be substituted and the rest of this manual would be printed in \sl. Therefore, \def\dim{{\sl dim}} is the correct definition in this case. Now when we type \dim, TEX substitutes {\sl dim} and all is well. By the way, note that using definitions is good not only for speeding up the typing, but also to ensure that one symbol (simple or complicated) is done uniformly throughout the text. If we later on decide to make it look a bit different, we simply tweak the one definition and need not worry about hunting it all over the text. This is what happened here, midway through writing this manual we decided to add parentheses and changed the definition to \def\dim{({\sl dim})}. This is very useful, we strongly advise to make as much formatting as possible (chapter titles etc.) through custom-defined macros.

In fact, packages that we can load to teach TEX something new are usually just big files full of macro definitions.

Redefining.

One can actually define a control sequence that already exists, but it is not advisable. First, the original meaning meaning is not available anymore, second, the control sequence may be used in some other definition and if you change the meaning of it, the outcome is unpredictable. However, if have a good reason for it, there is the right way to go about it. As an example we try to simplify my life, I want to have indices down with my limits and I do not feel like adding \limits all the time. If we put \def\lim{\lim\limits} at the beginning, we just created a vicious circle. How should we do it properly?

First, we need to "rename" the original copntrol sequence using format is \let *new-name*=*used-name*. In our case we would do \let\LIM=\lim, the control word \lim has been renamed into \LIM and we can still access the original meaning. Now we are free to use \lim any way we wish, for instance \def\lim{\LIM\limits}.

It comes very handy if we want to protect some control sequence from a package we plan to load. For instance, we mentioned that $\mathcal{AMS}$-TEX kills \footnote. The simplest solution is to put the line \let\myfoot=\footnote before we put \input amstex and now we can use \myfoot for making footnotes freely. In fact, we did it here, in the example in the previous chapter we just pretend that we typed \footnote, in the source file there is \myfoot instead.

$\mathcal{A}$ $\mathcal{AMS}$-TEX has its own ways, \predefine *new-name*{*used–name*} and \redefine *used-name*{*new-definition*}, but the thing described above still works there.

Control sequences with arguments.

The definition of a control sequence with arguments is done in almost the same way as we saw above, but now after the name of the new control sequence (and possibly some spaces) we add "variables" denoted by numbered symbols #. They are numbered by single digits starting at 1 and going up by 1, obviously there cannot be more than 9 arguments. Unless one knows what (s)he is doing (see the last part for TEXperts), these variables have to go one after another, #1#2#3 ... , and { has to follow immediately after the last one. In the definition of the new control sequence we then use these variables wherever we want. Note that when using such a control sequence, argument(s) can contain MM parts but no paragraph breaks.

For example, there are lots of control sequences written in \tt in this manual. To simplify typing, the following definition was made: \def\cs#1{$\backslash${\tt #1}}. Now when we type \cs{infty} in this file, the argument "infty" is put instead of #1 in the definition, so TEX replaces \cs{infty} with $\backslash${\tt infty} and \infty is printed. Note how a group was used to limit the influence of \tt (now I typed \cs{tt}!).

When TEX reads a control sequence, at its end it checks whether an argument is expected. If it is so, it reads the next character. If it is an opening brace {, then the contents of the group this opens is taken as the argument. If this next character is \, then the control sequence it defines is read and expanded, its *contents* is taken as the argument. Otherwise just this next character is taken for the argument.

With more arguments it is just the same repeated. For instance, the popular macro for doing fractions can be defined like this: \def\frac#1#2{{#1\over #2}}, now we can use \frac{13}{abc} to obtain (in MM) $\frac{12}{abc}$. We will look deeper into working of this macro in chapter More on macros.

The defined control sequence can be quite long. For instance, if one is to write the same letter to several people, it is possible to define \def\letter#1{Dear #1, ... Yours truly} and then put \letter{John}\letter{Jane} there, two letters will be printed.

Beware! There is a catch caused by the fact that only the contents of the arguments are taken (that is, the groups that were used to supply them are stripped). Imagine that we use $\mathcal{H}$ (that is, \Cal H in MM) often and with a changing subscript, so we define \def\H#1{\Cal H_#1}. Then \H k will print (in MM, of course) $\mathcal{H}_k$, \H\alpha gives $\mathcal{H}_\alpha$. It seems that everything is fine, but then one day there is a double subscript. We try \H{n_k}, but we get a "double subscript" error message. Why? Only the contents of {n-k} were used and therefore we actually asked TEX to print \Cal H_n_k. We can correct it by writing \H{{n_k}} yielding \Cal H_{n_k}, but it is preferable to use a better definition: \def\H#1{\Cal H_{#1}}. Similarly, if we define a common index \def\ind#1{#1_{i,j}},

we can use it for `\ind y` giving $y_{i,j}$, but it fails when we want to index y_0, the result of `\ind{y_0}` is an error: `y_0_{i,j}`. A better definition is `\def\ind#1{{#1}_{i,j}}`.

This also plays role when calling on another macro in a definition, the correct way of passing an argument inside is `\def\OneMacro#1{\AnotherMacro{#1}}`.

Advanced control sequence definitions.

If `##` appears in the replacement text for a macro, it is expanded into `#`. This allows for some neat tricks, for instance definitions inside definitions: `\def\trick#1{\def\here##1{write##1}\here{#1}}`.

It is also possible to include some text between the new control sequence declaration and the opening { of the description. We know that all spaces after the name are ignored, but if after the name and perhaps some spaces there are characters other than { or # followed by a digit, then all of them up to the first { (which starts the definition) are considered an indispensable part of the control word, with the exception of variables. It is best illustrated on an example, say `\def\a 1{}`. With this definition, `\a` itself does not exist. We have to use `\a 1` or `\a1`. When TEX reads a control word, it stops when some non-letter is met, in this case a space in the definition and the first example, or "1" in the second one. Then all further spaces are ignored. But then TEX finds the "1" in the definition of `\a` and considers it a necessary part, that is, it is required for `\a` to be recognized. So the first non-space after `\a` in our examples is checked on and if it is "1", everything is fine, otherwise an error is called. Note that since spaces after a control sequence are ignored, the above definition could have looked like this `\def\a1{}` as well and it could be accessed using `\a1` or `\a 1` again.

The same idea applies when arguments are declared, too. When `#1` in the name declaration part is not followed by `#2` or { but by some *text*, then the first argument is determined as the shortest possible part of the source text that is followed by *text* and obeys group rules. The same applies to the second argument if there is some text between `#2` and `#3` or { etc.

Imagine that there is `;` after `#1`. In that case, TEX knows that this control sequence needs an argument followed by `;`. Therefore everything after the control word is considered the first argument until the first group rules-obeying `;` is found, but this `;` is not considered a part of this argument, it is just something that has to be there.

What is it good for? For instance, when using `\frac`, we may find it tiresome to keep enclosing arguments in braces. We can help it by defining `\def\f#1,#2;{{#1\over #2}}`. Now we can write (in MM) `\f x+y,x-y;` and we will get $\frac{x+y}{x-y}$ as expected. Really, everything up to the comma was taken as the first argument, and then everything up to the semicolon was taken as the second one, `,` and `;` were not included in arguments.

Opening and closing groups.

Many custom structures in TEX packages have the form of "environments". For instance, when we want to start a theorem, we would type `\theorem`, then the text and conclude it with `\endtheorem`. The text is then typically typeset in a different font, probably italic. Obviously, the macro `\theorem` (among other things, like writing **Theorem**) opens a group and switches to `\it`, whereas the macro `\endtheorem` has to close this group. How would we do it if we wanted to create such an environment by ourselves?

An attempt to define it like this `\def\theorem{{\it}` will not work. Why? When we use `\theorem`, TEX will look up the definition and as it reads it, it considers the closing brace } after `\it` to be paired with the second opening {. Consequently, the main brace that started the definition would remain unclosed and TEX will keep reading until it hits another unpaired } (if there is any available), then consider everything up to that point to be the meaning of `\theorem`. Most likely there would be no such orphaned }, TEX would reach `\bye` at the end of the file and complain that } is missing.

Similarly, if we define `\def\endtheorem{}}` and use `\endtheorem`, TEX replaces it with the contents of the group that comes in the definition, which is actually just the contents of {}, that is, nothing. Then the second closing brace would be read and TEX would start scanning backwards for some brace that was left open to complete a pair, most likely ending up with an error message.

But the need for opening and closing groups is obvious and TEX offers `\begingroup` and `\endgroup` that can be used in definitions. When the corresponding control sequence is substituted for, TEX replaces these with proper braces. Note that we cannot use `\begingroup` in a situation where it does not really delimit a group, but just supplies an argument. For instance, if we want our control word to open a vertical box, we cannot define it as `\vbox\begingroup`, similarly we cannot use `\begingroup` with `\hbox`, `\line` and in other similar situations.

For such a purpose there is the pair `\bgroup` and `\egroup`. These two cannot be mixed up with the previous two (if we open something with `\begingroup`, we cannot end it with `\endgroup`). If we want our control sequence to open a v-box, we can put `\vbox\bgroup` in the definition. When used, it will be turned into `\vbox{`.

We will return to advanced macro making later.

Organizing source files.

Experience taught us to organize TEX source files in a standard pattern, it helps not only other users of it (if any) but also us as authors when we return to some text after several years and want to make some changes.

Usually we start with a general setup of the document (height, width, page numbering, etc., we are not always

happy with default values). We also load new fonts at this point if needed. If we decide to change the size of the document using \magnification, then it is usually the very first thing in this general settings section.

Then we put all custom definitions, we definitely do not want them mixed up with the text itself. It is customary to put every definition on its own line. We can comment, every % and the rest of line after it are ignored by TEX. This makes a row of %%% ... a handy section separator in our source file.

Note that if a definition is very long, we may prefer to break it into more lines so that one can see it whole on screen. However, every linebreak is interpreted as a space, which is not always desirable. The solution is simple, use percentage signs to fix this. If we type

```
\def\name{Pe%
tr}
```

then TEX ignores the % and everything after it including the line break, and thus reads it as \def\name{Petr}.

After definitions we put the contents of the document and end it with \bye.

Sometimes we want to include packages that allow us to get more out of TEX. This is then typically done at the very beginning of the source file, we use the control word \input for it.

In fact, \input is a general command that reads the file whose name we specify (the form is \input *filename*, we should not put the name as a group) and pastes it into the source file as if it was typed there, right where this \input is. If the file extension is .tex, then we need not specify it. We can also supply the path to the loaded file, say, \input C:\letters\book.me.

The source file for this manual starts with the line \input amstex that loads amstex.tex, just like the source file for my recent book, there the next line reads \input tikz, this loads a package that contains lots of definitions of macros that allow us to draw pictures in TEX directly using control words.

We can use \input to better organize our work. Example: When typing a substantial document (a long book), the resulting file would be very unwieldy. Then it is a good idea to put every chapter into a separate file, chapter1.tex, chapter2.tex, etc. Then we create a master file called book.tex which would contain definitions and formatting commands, the title, perhaps a short introduction, then there would be \input chapter1, \input chapter2, etc., and finally \bye (note that we definitely do not want to put \bye to files with individual chapters). In this way we can organize our typing and we can even make debugging simpler, for instance by % commenting out \input lines for chapters that we do not need at the moment.

Another example: Most people after a while develop a set of macros that they like to use. It is possible to put them all into one file, say, defs.tex and then include them in every file using \input defs. But then one would have to make sure that the file defs.tex is available every time we want to run such files, so perhaps it is not the best idea in case we want to share our files with other people. I do have such a file, but I just paste the definitions into my source files so that they can stand on their own.

Of course, inputs can be chained (file1 calling for file2, this one calling for file3), but it is not recommended to have \input file2 in file1 and \input file1 in file2 unless you like loops (this one would probably stop when your computer's memory is full).

Including pictures.

This is actually fairly simple. The picture must be saved in the Encapsulated PostScript format. Assume that the file is pic.eps. To include it in the text, first we must load the picture-handling package with \input epsf, typically at the beginning of the file. At the appropriate place we then put the picture in using \epsfbox{pic.eps}. If used within a paragraph, the picture is considered a part of the corresponding line and is aligned by its bottom.

We can prescribe the width of the picture by preceding \epsfbox with \epsfxsize=(*dim*). The picture is then scaled (both height and width, preserving the original ratio) to fit.

Debugging your file

When running a source file for the first (and second, ...) time, TEX usually complains about syntax, that is, there are structural problems (typos in commands, mismatched groups, etc.). It is not always easy to understand what the problem is, and the best strategy is to avoid problems in the first place. The most common mistakes include a forgotten $ or {,}. A little trick works for me: When I start writing math (or a group), I first do both dollar signs $$ and then fill in the text between them, similarly I start with $$$$ for displayed formulas or {} for groups. This way I rarely forget to close a unit.

When the beginning of the file is already fine, we have to wait for TEX to go through it before it comes to parts where things are happening. It is possible to skip them. The text between \comment and \endcomment is ignored; more such pairs can be used, but they cannot be nested and \endcomment has to be put on a separate line all by itself. Also, \bye can be moved forward in a file so that one can focus on the "battle line". Another possibility of making TEX run faster is to put \syntax at the beginning of the file. TEX will then only check on syntax of the tex-file without constructing any output.

Another helpful suggestion is to start with \magstep0. There is less "Overfull" messages then and less pages to compose, so things go faster. After TEX is satisfied with the syntax, we can adjust the size to what we want and start working on appearance. That is usually much easier compared to deciphering TEX's error messages.

Actually, error messages often explain the problem well enough, but sometimes it is not that easy. One possible catch may be an error message different from the actual mistake. For instance, if we use `\pmatrix` and `\endmatrix` together, we get the "`\right` missing" error message. Also, if we try `\boxed{$1+1$}`, we get "} missing", although it is not apparent why. A real TEXpert would know, but sometimes it is easier to just experiment instead of spending several days learning advanced concepts and deciphering what is happening behind the curtain. For instance, the latter problem can be fixed by typing `$\boxed{1+1}$`.

Another tricky problem is when an error appears long after the real mistake was made. For instance, if a brace is left out, it may cause an error several pages later. Of course, that is the place listed in the error message, so it is not easy to find the mistake then. If one does not see it right away, there is a little trick which helps localizing mistakes: take some parts out using `\comment`–`\endcomment` and move `\bye` forward in the file, waiting for the error to disappear. Then you know where it happened.

The fact that error messages are stored in the log-file can be very helpful. When TEX puts out an error message and asks what to do, we can type `q`, which will cause it to go through the rest of the file without bothering us, error messages still being stored in the log-file. Then we can open our tex-file together with the log-file and correct many mistakes at a time. But this can backfire—some errors can be corrected by TEX in a wrong way, which causes another error, e.t.c., this may go for a long time (I have seen as many as 20 error messages caused by one `&` written instead of the correct `\&`).

With this friendly advice we conclude the "reasonably versed user of TEX" part of this manual. For writing a typical paper one does not really need more, even a decent book can be made. In the chapters that follow we will try to look deeper into the bowels of TEX.

9. More on macros

TEX has two levels of commands. *Primitive* commands are directly implemented, whereas higher level commands are defined in terms of primitive commands, that is, these are in fact macros. Among the most important primitives are `\begingroup` and `\endgroup`, more popular under the names { and }. We can learn about a command using `\show` *control-sequence* in the tex-file or on TEX's command line, the output always appears on the terminal and in the **.log**-file. Example for a primitive command:

```
> \show\input
> \input = \input
```

Example for a higher level command:

```
> \show\thinspace
> \thinspace = macro:
->\kern .16667 em.
```

We can also ask for explanation to appear in the document. `\meaning` *token* prints an explanation of the *token*. For instance, if we define `\def\A#1B{C\a}`, then `\tt\meaning\A` is substituted by `macro:#1B->C\a`.

On the other hand, `\string` *token* reads the *token* without expanding it, just prints its name. Thus if we write a manual and want to show what `\qquad` does, we can type `\string\qquad: \meaning\qquad`
and if we forget to switch to `\tt`, we will get
"qquad: macro:-¿"hskip 2em"relax
If we precede our commands with `\tt` we get what we want:

```
\qquad:  macro:->\hskip 2em\relax
```

One related command: `\fontname` *font* prints the real name of the *font*.

Sometimes (especially in macro definictions) we may be given a word we asked to create a new macro with this name. There is a construction for it: `\csname` *text* `\endcsname` takes *text*, expands it until just characters remain, the resulting thing is then considered a control sequence.

For instance, if we define `\def\newmacro#1#2{\def\csname#1\endcsname{#2}}`, then typing
`\newmacro{oops}{drop here}` is the same as typing `\def\oops{drop here}`

When defining more complicated macros, one should pay attention to the inner workings of TEX. One possible problem is that some procedures do not have a uniquely defined end. For instance, the specification of a horizontal line may be `\hrule width 12 pt` but also `\hrule width 12 pt height 3 pt`. If we put the former into a macro, it would be unpleasant if we use it in text where the next word is ,,height". To break such potential chains we use the command `\relax`, this tells TEX to finish whatever process it is in. Thus a safe definition would be `\def\myrule{\hrule width 12 pt\relax}`.

This interference between contents of a macro and surrounding (source) text can sometimes be unpleasant. Another command that comes sometimes handy at this situation is `\ignorespaces`, it makes TEX ignore whatever spaces come next.

An important consideration when designing macros is the order of evaluation (action first, substitution next). For instance, imagine that we use uppercasing a lot and the command is long, so we define `\up#1{\uppercase{#1}}`. Then `\up{abcd}` does ABCD. However, what if we have `\def\string{abcd}` and call `\up\string`? We get ,,abcd".

When TEX read our input, it first sees the command \uppercase and remembers what should be done. Then comes the action, TEX therefore applies lowercasing to \string, which is one token and nothing is done, and still then the expansion of \string comes. Sometimes this behaviour may be exactly how we want it, but in this particular case we do not. How do we fix it?

The command \expandafter expects two tokens coming after it, note that it does not expect arguments, it simply reads the next two tokens. The first token is left for a moment, the second is expanded, and only then both tokens are processed. We use this in our definition.

New version: \up#1{\uppercase\expandafter{#1}}. Now TEX processes \up{\string} as follows. After \uppercase it wants an argument. It encounters \expandafter, so the next token { is left as it is and the one after it should be expanded. So substitution comes, TEX reads \string and expands it. The resulting text {abcd is then submitted to the reading mechanism of TEX, so in fact TEX reads \uppercase{abcd}.

Note, however, that \expandafter only expands the second token, not more, so \up{\string\string} produces "ABCDabcd". Expansion of macros can easily become a matter for real TEXperts.

Here is an example where the order action-substitution helps. When we define \def\frac#1#2{{#1\over#2}}, somebody may call it with \frac1a. If the substitution came first, TEX would have faced {1\overa} and complain. But it goes the other way, TEX first realizes that it should put something over another something, and only then it substitutes for that something.

Of course, a prudent way is to simply define \def\frac#1#2{{#1\over #2}} and not worry about it at all.

All definitions (macros, variables) are local, that is, they affect the defined thing only within the group where they were done. When the relevant group ends, the original value is restored. We can make a definition global by preceding the relevant command with \global. For definitions there is a handy shortcut, \gdef is equivalent to \global\def.

There are several others modifications of \def. If we precede it with \long, then its arguments can accept \par breaks. However, such macros need a lot of memory when running, so it is preferable to create environments. \outer means that the following macro cannot be used in an argument or replacement text of another macro. The control words \long, \global, and \outer may be in any order before the relevant \def.

The command \edef causes the definition to be read immediately when TEX encounters it in the source-file (normally it ignores definitions and comes back to them when they are used). Moreover, while reading it, TEX expands all parts of definition that do not produce boxes. For instance, if this definition uses some parameters, then their values at the time this definition is encountered will be used in it. Similarly, if conditions are to be evaluated, then they are done with values valid at the time TEX encountered this defnition. \xdef is a shortcut for \global\edef.

One more advanced trick: Note that we cannot use { in the *text* that we want to play with in the name of a control sequence, since the first { that is encountered automatically starts the macro contents description. There is a way to get around this limitation. If the last character before the first { in the definition is #, then it is as if instead of #{ there was {{, but only the second one opens the description, the first one determines the end of the last argument. This opening brace is then also attached at the end of this argument. Example: If we define \def\a#1#{\hbox to #1} and type \a3pt{hey boy}, then the argument #1 will be 3pt{ and hey boy} is a normal text that follows; \a3pt{ will be interpreted as \hbox to 3pt{ and we get \hbox to 3 pt{hey boy}.

Another example: \def\a#{\b}, now \a{*text* is expanded into \b{*text*. For instance, \a{hi} yields \b{hi}.

10. Registers

TEX has the following registers:
\count0–\count255: integers with magnitude at most 2,147,483,647,
\dimen0–\dimen255: dimensions,
\skip0–\skip255: glues,
\muskip0–\muskip255: muglues,
\box0–\box255: boxes, work differently (see below).

Some registers are reserved, namely all registers with numbers 0 through 9 and also \box255 that contains the current page. Some registers get to be used by various contructions, for instance, the text of footnotes is in \box254, magnification for page breaking in \count254, max insertion size per page in \dimen254, and extra space to allocate in \skip254.

Registers other than boxes:

We assign values by *register*=*value*, for instance \count13=-21. The equality sign can be left out, but it is better not to do it. In some cases we can mismatch types. If (*glue*) is assigned to \dimen, the fixed part is taken. If (*glue*) or (*dim*) are assigned to \count, the fixed part is taken as a number (after being converted to sp). For instance, if \skip1=1pt plus 2 pt, then \count10=\skip1 defines \count10 as 65536.

Changing registers: \advance*register* by *something* adds to a register. When advancing glues, the fixed (*dim*)'s are added, for stretch/shrink parts we take bigger of both. We can also use \multiply*register* by *integer* and \divide*register* by *integer*, with glues it affects all components. Results of \divide are rounded to integer, counted in sp for dimensions.

Counts and also standard parameters (which are also registers) can be used like any other number, so we can do \multiply\dimen20 by \count5 or \dimen20=\count5 pt or \glue5=\parskip. When the register/parameter we are using is preceded by a number, then it is multiplied. For instance, \dimen31=-0.5\baselineskip will assing minus half of the baseline skip. Since this is a glue but the register expects dimension, only the fixed part of baseline skip will be used.

We can access the contents of a register using \the*register*. For instance, if the source file reads indentation is \the\parindent, we would see its value on the output. The command \showthe prints the value on the terminal.

The value of a number register can be printed using \number*register*. It is also possible to modify the appearance, \romannumeral*register* prints positive integers in lower-case roman numerals.

Counts 0 to 9 are used for numbering in the document, page number is stored in \count0 and the other counts are used for chapters, sections, subsections etc. and are 0 by default. Page numbers are printed as \count0.\count1. ... .\count9, trailing zeroes are omitted. For \count0 TEX has the synonym \pageno, so we can for instance advance the page count by \global\advance\pageno by 1 and then print it using \number\pageno. This is used often, so TEX also has \folio as a shortcut for page number.

TEX also offers \time, \day, \month, and \year. Example: If we want our document to show copyright with a year in upper-case roman numerals, we use the following construction:
\copyright\uppercase\expandafter{\romannumeral\year}. Note that since an argument is always processed before macros it includes are expanded, we had to force TEX to expand \romannumeral and \year before they were passed to \uppercase. If we tried \uppercase{\romannumeral\year}, then the action would be first, all character tokens in the argument \romannumeral\year would be made uppercase (there are none) and then the macros in it would be expanded, yielding the year in lower-case letters.

Box registers:

We define a box register using \setbox *number*=*box*. We can use it via \box*number* (the box is used and erased) or \copy*number*, where the box is inserted but not erased. We use it just like any other box, for instance after defining \setbox13=\hbox{word} we can do \raise 2pt\copy13. The dimensions of a defined box are available using \wd*number* for width, \ht*number* for height and \dp*number* for depth. These are dimensions that can be used like other registers, e.g. \hbox to {4\wd3} makes a box whose width is 4 times that of \box3. Sometimes we need a box that does not contain anything, but it is assigned. Then we use \null, a box with nothing in it.

The \box and \copy commands have special versions that give you the box required but strip it off its outer \hbox or \vbox, i.e. they give the contents only. Commands are \unhbox (\unhcopy) or \unvbox (\unvcopy). We can ask for the contents of a box to be show on the terminal using \showbox *number*.

Note that all register changes (boxes and non-boxes) are local in groups, unless the change/setting is preceded by \global.

Named boxes

We can assign names to registers, \countdef*controlseq*=*number* does it for counts. After \countdef\mycount=13 we can use \mycount in the same way we would use \count13. Commands \dimdef, \skipdef, \muskipdef, \boxdef work analogously.

However, we rarely know what registers are already being used, therefore it is better to leave choices of concrete numbers to TEX, we just tell it that we want some register and leave the rest to it. For counts we use \newcount*controlseq*, this command finds a count that is not used yet and assigns its number to *controlseq*. Analogously we use \newdimen, \newskip, \newmuskip, and \newbox. For instance, after \newbox\mybox we can fill it: \setbox\mybox=\hbox{13}, and then use it without erasing: \copy\mybox.

11. Running the Show

We start with flow directing commands. General form of the conditional statement is
\if*condition truetext*\else *falsetext*\fi.

The part \else *falsetext* is optional. The possibilities for \if*condition* are:

- \ifnum *number relation number* where *relation* can be =, <, or >.

The numbers in this condition can be numerical constants (13), counters (\count13) or numerical parameters (\baddness). All tokens following \ifnum are expanded until TEX obtains a sequence of tokens of the right form. For instance, \folio is defined as \ifnum\pageno<0 \romannumeral-\pageno \else \number\pageno \fi. It is wise to put spaces between parts of the conditional command. For instance, if we write \ifnum\pageno=1\first\fi and the very first character of \first happens to be 2 and the next is not a digit, the condition becomes \ifnum\pageno=12. So it's better to write \ifnum\pageno=1 \firstpage. Another option is to use \relax.

- \ifodd *number*

Again, the text after \ifodd is expanded until TEX obtains a number/register/parameter followed by a token that cannot be a part of a number. In the headline section we saw an example using \ifodd\pageno.

- \ifcase *counter textforcase0* \or *textforcase1* ... *textforcaseN* \else *othercases* \fi

For an example see below.

- \ifdim(*dim*)*relation*(*dim*)

Again, text after \ifdim is expanded until it has the form of a dimension (given directly, as a dimensional register or a parameter like \hsize), relation =, <, or >, and another dimension.

• \if *token1 token2*. All tokens following \if are expanded until there are two that cannot be expanded further. Their character numbers are then compared and if they agree (regardless of their cat codes), the condition is satisfied. A control sequence that cannot be further expanded is considered to have character code 256.

• \ifx *token1 token2*. Tokens are not expanded but compared directly. If one is a macro and the other is not, then they are not considered equal. If both are macros then their first level expansions are compared, for equality their expansions (replacement texts) must be identical and status must agree (\long, \outer).

If none is a macro, then they are considered equal if both are characters (or control sequences denoting characters) and their character codes and cat codes agree, or if both refer to the same TEX command, font etc.

Example: \def\d{A}, \def\c{A}, \def\a{\c}, \def\b{\d}, then \ifx\c\d is true, \ifx\a\b is false.

Note that if the two tokens are control sequences that were not defined yet, then they agree.

As an example we will create a named environment that prints text in italic. We open it with \begenv{*name*} and close it with \endenv{*name*}, where the name must agree.

```
\def\begenv#1{\begingroup\it\def\blockname{#1}}
\def\endenv#1{\def\test{#1}%
    \ifx\test\blockname\endgroup%
        \else\errmessage{You shouda said \string\endenv\{\blockname\}}\fi}
```

We store the name of the environment as a macro \blockname. Note that just one is needed even though somebody may try to nest these environments (open them again and again before closing), because the definition is always local in the currently open group.

• \ifcat *token1 token2*. All tokens following \if are expanded until there are two that cannot be expanded further. Then the command checks whether they have the same category code.

• \ifvoid *number*, \ifhbox *number*, \ifvbox *number*: tests a box register whether it is void, an h-box or a v-box. Note that {} is not a void box but an empty box! We obtain a void box for instance if we assign some box a value, \setbox13=\hbox{}, and then use it up, \setbox5=\box13.

• \ifvmode: true if TEX is currently in (internal) vertical mode.

• \ifhmode: true if TEX is currently in (restricted) horizontal mode.

• \ifmmode: true if TEX is currently in math mode.

• \ifinner: true if TEX is currently in internal vertical or restricted horizontal mode or in-line math mode.

• \iftrue, \iffalse: always true/false

• \ifeof *number*: tests for EOF of file *number*

We can make our own Boolean variable. The command \newif\if*text* creates a new Boolean control sequence \if*text* and sets up commands *text*true, resp. *text*false for setting \if*text* true, resp. false. One can imagine that *text* is a local variable that is set true or false by the two commands and then we ask about its status using \if*text*.

For instance, we may prepare a text in such a way that the user decides at the beginning whether it should be printed in font1 or in font2. So we would first define \newif\ifFirstFont.

Then we would read user input (see below) and if the preference is for font1, we would call \FirstFonttrue, otherwise we call \FirstFontfalse. In this way the user preference is stored and we can switch on the correct font using \ifFirstFont\font1\else\font2\fi.

Another example is below.

Loops are done using the construction \loop *text1* \if*condition text2*\repeat. This prints *text1*. Then the condition is tested, we can use any \if described above or one that we set up ourselves using \newif. If the condition is not true then the procedure ends; if the condition is true, then *text2* is printed, TEX goes back to *text1* and the cycle starts again.

This construction is actually rather simple:

• \def\loop#1\repeat{\def\body{#1}\iterate}

• \def\iterate{\body\let\next=\iterate\else\let\next=\relax\fi\next}.

Since the expansion of \iterate ends with expansion of \next, TEX is able to remove \iterate from its memory before invoking \next. Thus even a long loop does not fill up the memory.

Input/Output

\jobname prints the file name that TEX is just working on. This brings us to more interesting things:

\openin *number*=*filename* opens a file for reading and assigns it the *number*, possible values are between 0–15. Recall that we can test the file for EOF. We can read from this file line by line using \read *number* to*controlseq*. This defines *controlseq* as a parameterless macro whose replacement is the contents of the next line from the file. This definition is local unless we use \global\read. When a file is open for reading, it is locked and we cannot do anything else with it. We can close it using \closein *number*, then we can use this file for other things, for instance write in it.

If we do not want to manage files by numbers, we can ask TEX to do it. First, \newread *controlseq* creates the

control sequence *controlseq* as a handle for a file to be read from and assigns it a number that is not used yet for files. Then `\openin` *controlseq*`=`*filename* opens a file for reading and we refer to it using our control sequence. For instance, `\newread\toc` and `\openin\toc=mybooktoc.tex` makes a file where we stored Table of contents available for reading and we can read a line from it using `\read\toc to\tocline`. We close it `\closein\toc`.

If we use `\read` *number* `to\`*controlseq* and the *number* is more than 15 or the file not open, then TEX prompts for input from the terminal. For instance, when reading `\read16to\textin`, TEX interrupts the file processing, types `\textin=` on the terminal and waits for the user to type in some text ended with Enter.

If *number* is negative, the value for the macro is taken from the terminal, but this time TEX only waits for the input without any prompting. This is often used when we want to ask the user in a different way than the default prompt.

We print things on the terminal (and the log-file) using `\message`, the text is supplied as argument. For instance, we could do `\message{Type a number: }\read-1to\number`. Note that with this, the number is just a text stored in a macro. In order to transform it into a genuine number (and perhaps store it in a numerical register) some TEXpert procedures must be used.

Example: We check who is at the terminal. Note that we have no conditional command that would allow us to compare with a piece of text. We get around this by putting the texts we want to compare into macros and then compare their expansions.

```
\def\B{Beavis}\newif\ifB
\message{Who's there?}
\read-1 to\answer
\ifx\answer\B\Btrue\else\Bfalse\fi
\ifB\else Go away, huh, huh\fi
```

Actually, we could have just done, say, `\ifx\answer\B Hi, bunghole.\else Go away, huh, huh\fi`, but we wanted to show a custom conditional command in action.

We can also write text into a file. First we open a file for writing, `\openout` *number*`=`*filename*. We insert text into an opened file using `\write` *number*`{`*text*`}`, the *text* is written into this file as one separate line, macros are automatically expanded. If this file is not open (in particular if *number* is more than 15), then the text goes to terminal and the **.log**-file. If *number* is negative, the text goes just to the **.log**-file. At the end we close this file. As with reading, we can use nicknames, first declare it `\newwrite` *controlseq*, then open `\openout` *controlseq*`=`*filename* and after writing `\write` *controlseq*`{`*text*`}` we close `\closeout` *controlseq*.

Note that opening/closing a file and writing is delayed until the current paragraph/construction is finished, so that the page numbers etc. are right. However, this may mean that some registers that we wanted to store change by the time the write is actually done. If we precede any of these commands by `\immediate`, the action is taken immediately.

Typically we open a file, write some lines into it, and then close it, which makes sure that also the delayed writings were done. The we can for instance open it again and read it, or add more lines. The command `\bye` closes all files that were open.

Debugging.

If we set `\tracingmacros=1`, macros will be explained in the **.log**-file. If we set `\tracingmacros=0`, it will not happen.

If we put `\tracingcommands=1`, TEX explains all steps in the **.log**-file (commands, modes etc.). If we also want explanations of conditional commands and their outcomes, we use `\tracingcommands=2`. Finally, `\tracingonline=1` sends all comments not only to the **.log**-file, but also to the terminal.

12. Getting TEXnical 2 (processing text)

When TEX transforms our source text into finished pages, it actually does three main processes. It does them simultaneously (it reads our text only once), but it will be easier to look at them individually.

The first stage is conversion of our source text into **tokens**, these are basic units for further processing. There are two kinds of tokens. One is characters that have meaning by themselves, for instance all letters in this sentence are individual tokens. The other kind are control sequences, for instance, characters in `\TeX` do not represent themselves but are a part of a control word description. The source text `\Tex reads` is interpreted as tokens TeX␣r e a d s.

This conversion is fairly simple, TEX reads input characters and after each decides what to do. In order to simplify things for it, characters are divided into groups called categories according to what effect they may have. Each character thus has its unique category code (catcode). There are 16 of them, the following table shows how characters are divided in plain TEX:

0	escape character	\	8	subscript	_
1	beginning of group	{	9	ignored char.	<null>
2	end of group	}	10	space	

3	math shift	`$`	11	letter	`A`–`Z`, `a`–`z`
4	alignment tab	`&`	12	other	none of 0–11, 13–15
5	end of line	`<return>`	13	active char.	`~`
6	parameter	`#`	14	comment char.	`%`
7	superscript	`^`	15	invalid char.	`<delete>`

Note that characters that form individual tokens will carry their catcodes also to the following stages of text processing.

We can change catcodes for characters using `\catcode`*ASCII-code*=*number* or `\catcode`‘`\`*character*=*number* (see below, this is more popular). For instance, after `\catcode`‘`\<=1` the character `<` will work in precisely the same way as {, they will be the same from the TEX's point of view: Every time we type `<`, no character will be produced and TEX will open a group. This change only works inside the group where it was made unless we precede `\catcode` by `\global`. Obviously, changing catcodes can significantly influence how text is processed.

The behaviour at each step of the translation procedure depends not only on the catcode of the character that was just read, but also on what **state** TEX currently is in. When it reads a text, it may be in one of the following three states: N: beginning of a line, M: middle of a line, S: skipping blanks. Here are some of the rules that determine what happens when the next character is read.

— spaces at the end of a line are deleted, the `<return>` character is then attached (so playing with its catcode can have very spectacular consequences),

— if the character is of category 0 (e.g. the backslash), the control sequence reading begins. If it is a control word and ends with a space then TEX goes to state S, otherwise goes to M,

— if the character is of category 7 (e.g. `^`), the next character is checked. If this has code 7 again then a special procedure for character substitution starts, see below.

— if the category is not 0, 5, 7, 9, 10, 14, 15, or the character is not of category 7 followed by another category 7 character, then this character is made into a token with its corresponding category,

— if it is of category 5 (e.g. `<return>`), then it is converted to a control sequence token $\boxed{par}$ if TEX is in state N; it is converted to character 32 (a space) of category 10 and TEX switches to state S if in state M; and it is ignored in state S,

— if it is of category 10 (e.g. a space), then it is ignored if in state N or S; otherwise it is converted into a token of category 10 with character code 32, and TEX enters state S.

Note that the character code of the space token (code 10) is always 32.

In this way the source text is transformed into a stream of tokens, either tokens representing control sequences or tokens as individual characters that also have catcodes assigned. Note that in this stream of tokens we never see tokens of character 0, 5, 9, 14, and 15.

Sidenote: Entering characters. If we do not want (or cannot) enter a character directly, we can input it in several ways. We can call it by its ASCII code using `\char` *number*, note that spaces after *number* are ignored. If we just write *number* (between 0 and 255), it is considered to be in decimal. We can also precede *number* with `’`, then it is considered octal, or `"`, then it is considered hexadecimal. Thus we can write `\char98` or `\char’142` or `\char"62` instead of “b”. Note that TEX has internally only 128 typeable characters, the upper half is for math etc.

Another way of obtaining characters is `^^`*character*, which stands for the character whose ASCII number is determined by a formula. Let A be the ASCII code of *character*. Then the new character's ASCII code is $A+64$ if $A<64$ or $A-64$ if $A\geq 64$. Thus `^^?` is the same as `\char127`. The advantage is that `^^?` is one token, so it can be used in places where we could not use the `\char` construction. Also, we can use it to get “invisible” characters, for instance `^^M` is the same as `<return>`.

We can get the ASCII-code of a character using `‘`*character*, say, `‘b` would yield 142 at places where TEX expects to get an ASCII-number (in normal text we would get ‘b of course). One such occasion was in the paragraph above on catcodes. If we cannot type the character directly (for instance because it would be changed into tokens of some other category), we may precede it by a backslash. Thus `‘b` yields the same number as `‘\b`, or more importantly, `‘\%` gives the ASCII code of %. In fact, `\char‘\%` is how the character % is obtained when we type `\%`.

The next stage is to change the stream of tokens we just obtained into another stream of tokens, but this time all tokens are definite/concrete. In other words, if some token is in fact a macro referring to other objects or containing conditions that need to be evaluated etc., then this macro is “expanded”, that is, replaced with its proper meaning, which may include other macros that need to be expanded and so on. If not told otherwise, TEX does expansion as it reads, that is, left to right and into the deep, as long as any expansion is to be done. The basic rule here is “first action, then substitution” (unless we force it otherwise).

For instance, if we type `\centerline{Left\hfil Right}`, then TEX first creates the token $\boxed{\texttt{centerline}}$, therefore it starts the centering procedure. This calls for reading some argument, so the group is read and interpreted as `Left` $\boxed{\texttt{hfil}}$ `Right`. This is passed to `\centerline` and only when the centering itself is done, `\hfil` is expanded.

This can explain why `\uppercase{a\lowercase{bC}}` yields Abc. TEX first does the action of $\boxed{\texttt{uppercase}}$, which replaces all letter tokens in its argument with the corresponding letter in upper case, yielding `A\lowercase{BC}` (`\lowercase` is not a letter token, so it is not changed). Then `\lowercase` is expanded and acted upon.

Finally, the third stage is to construct the output using the simple tokens obtained in the second stage. Before we focus on this stage we reiterate that these three steps happen simultaneously. TEX creates a token and if this token requires an action, TEX does it. This action may require additional information, then TEX reads a bit more of the source text, always only as much as it needs to finish the task at hand. For instance, when it encounters { opening some group, it first reads on until it hits the appropriate closing brace } and only then it starts working on this.

Since action is done right away, we may include commands (e.g. catcode settings) that change the way the text that follows is processed.

Constructing the output.

This stage is nothing more that puting together boxes. The simplest boxes are letter tokens. Boxes are further composed together, forming bigger ones. Note that TEX only works with boxes (shapes with dimensions and reference points), it does not care about their contents when typesetting. Thus TEX goes from the basic boxes to progressively bigger ones, adding glues where appropriate, and the largest possible box is one resulting page. On the output (in the **.log** file or on the terminal) we can see how deep TEX currently is in boxes, it is indicated by the number of periods. The treatment of boxes differs depending on the current **mode**.

There are the following modes:
— vertical: when building the main list of boxes from which pages are made.
— internal vertical: building a vertical list of h-boxes for a v-box.
— horizontal: building a horizontal list of v-boxes for a paragraph.
— restricted horizontal mode: building a horizontal list of v-boxes for an h-box.
— math mode
— display math mode: it is considered a part of the paragraph surrounding it. When a paragraph is formed, its number of lines is counted and each embedded display math piece counts as three lines.

TEX starts in a vertical mode and returns to it at every opportunity (whenever a paragraph ends). It switches from vertical to horizontal mode when it encounters a letter or some other inherently horizontal material. In such a case it starts a paragraph, leaves a space, calls `\indent` and goes to horizontal mode. We can also force the switch to horizontal mode by calling `\indent` or `\noindent`. We can also change the current mode using `\hbox` and `\vbox`.

Since TEX does not care about contents of boxes, we may do some interesting things by changing parameters of boxes. We already learned how to change the position of the reference point for a v-box. We can also change the apparent size.

The control word `\smash` will print its argument but TEX will think that it has no width, height, or depth. `\topsmash` smashes only the height, `\botsmash` takes care of the depth of its argument.

`\rlap` is a "right overlap" box, it takes its argument and turns it into a box with zero width and reference point on its left edge. Therefore the text that comes after will go over this argument. We now type `and \rlap{see} this` and see this.

The definition is `\def\rlap#1{\hbox to 0pt{#1\hss}}`. It works best when used in a horizontal mode. `\llap` is a "left overlap" box, it turns its argument into a box of zero width with reference point on its right edge. It is defined as above with `\hss #1`. The zero width means that TEX does not "see" it when arranging lines. We will now do `\centerline{\llap{left}\hfill\rlap{right}}`:

left right

We can also make virtual boxes. `\phantom` is a control word with an argument that produces a blank symbol exactly of the size of the argument. `\hphantom` is a blank symbol that is as wide as the argument but has no height and depth, `\vphantom` represents a blank symbol as high and deep as the argument but having no width.

The crucial unit for typesetting is a paragraph. TEX first reads all text that forms a paragraph and then tries to arrange this paragraph in the best possible way. This explains why any changes to values of crucial parameters (`\parindent`, `\leftskip` etc.) inside this paragraph are irrelevant, only the value valid when it ended counts.

In a typical case we have a stream of words that needs to be broken into lines. On every line TEX finds a breaking point (a space between words) and determines `\badness`, a parameter describing the change of spacing between words needed to make a nice right edge. If none of the lines in the paragraph has badness exceeding the value stored in `\pretolerance`, this line breaking is accepted and the paragraph is set. TEX then goes to the next paragraph.

Otherwise TEX again goes through the text, but this time also considers hyphenation. As before, badness is calculated and if no line exceeds `\tolerance`, the paragraph is set.

If this did not work, then TEX tries the best it can. For each line it then compares badness with a constant called `\hbadness` and if it is bigger, the "Overfull" or "Underfull" error message is sent. For instance, the example `\line{One\hfil Two}` would cause the "Underfull hbox" message as the space is spread a lot. Every overfull line gets a small black rectangle called "slug" attached as a warning. The width of this rectangle can be changed using `\overfullrule=`(*dim*). When `\overfullrule=0pt` is used, no black boxes are made.

The implicit value for \hbadness is 10000, but it can be changed using \hbadness=*(number)*. The implicit value for \pretolerance is 100 and for \tolerance is 200. If we put \tolerance=10000, TEX becomes infinitely tolerant and accepts any spacing.

TEX is trying to make the right margin straight, but some small ragging is allowed. It cannot be bigger than \hfuzz, which has an implicit value 1 pt and can be changed by \hfuzz=(*dim*).

$\mathcal{A}$ $\mathcal{AMS}$-TEX has a control word \NoBlackBoxes that cancels black boxes as well, this should be used at the beginning of the tex-file before any text starts.

In fact, TEX tries many scenarios, at every line it considers all possible breaking points that have reasonable badness and calculates an overall badness for the paragraph given a certain linebreaking situation. Then it chooses linebreaks so that the paragraph as a whole looks best.

The resulting lines are then arranged vertically and "interline glue" is put between them. The amount of this glue depends on the situation and three parameters. When two lines are put together, TEX first tries to add as much glue as needed so that their baselines are \baselineskip apart. However, when the depth of the top line and height of the lower line are too large, this would require very small, perhaps even negative space. If the space that would appear between lines is smaller than \lineskiplimit, then TEX simply puts glue \lineskip between such lines. In other words, TEX always adds at least \lineskip, but sometimes more, so that the distance between baselines is at least \baselineskip.

This whole thing can be forbidden using \offinterlineskip, then no interline glue is used at all, lines are put one on the top of another and it is up to us to make sure that these lines also include some white vertical space where needed.

Once paragraphs are set, they are broken into pages. The method is similar, TEX uses glue between vertical objects to get the best page breaking, \badness stores how much glues must be changed from their natural sizes and it is compared to the value of \vbadness. There is another factor in this game, quality of a page break at a certain point is described using the parameter \penalty. The maximal penalty is 1000 (never breaks), minimal is -1000 (always breaks). In fact, \smallbreak, \midbreak and \bigbreak assign penalties -50, -100, -200 to that particular place, thus encouraging a page break. Another nice command is \goodbreak, which is \par\penalty-500. We also have \filbreak, which is something like a suggested new page, it is defined by \vfil\penalty-200\vfilneg (if page doesn't break, the \vfil is cancelled).

This concludes the manual. If the reader wants to dwelve deeper into the mysteries of TEX, the obvious next step is *The TEXbook* by D. Knuth. If the reader want to get some inspiration concerning what can be done with the level of knowledge attained here, the Bonus comes next.

Bonus: Some useful constructions

Here we will show some useful constructions. While most of them can be easily found on the Internet as parts of various packages, we have two good reasons to show them here. First, most packages try to offer a complete solution (lots of options) and allow for a user who may not do things properly. As a result, these packages are very sophisticated, which translates to ,,unnecessarily complicated“ in case you need just some basic functionality and trust yourself to use the new macros in the right way. Moreover, complications come with a price, for instance many powerful packages play with token definitions (most notably, authors like to play with the token @), thus increasing the chance that various packages will clash. A simpler solution may avoid this problem.

Finally, the presented low-tech solutions show how commands covered in our TEX manual work and that they can be used for some powerful applications. This may encourage some people to play with macros even though they do not have time (or desire) to become real TEXperts.

Characters

1. A handy macro for **Czech quotation marks**:

```
\def\uv#1{,\kern-0.7pt,#1‘‘}
```

2. Changing numbers into letters

Assume that, say, a section number is stored in `\sect`. We want to see it as a letter. We will show two ways, one elegant but requiring some insight and one less elegant but simple. We start with the latter, we will show a special version that expects only numbers 1 through 13, for larger it shows a warning, for 0 it does nothing:

```
\def\lletter#1{\ifcase#1 {}\or a\or b\or c\or d\or e\or f\or g\or h\or i\or j \or k\or l%
\or m\or $\langle$range error$\rangle$\fi}
```

Now if `\sect` is 7, then `\lletter{\sect}` or `\lletter\sect` or `\lletter7` or `\lletter{7}` all yield “g”. Similarly, `\lletter0` produces nothing and `\lletter{14}` prints “⟨range error⟩”.

The elegant way:

```
\newcount\xxx
\def\lletter#1{\xxx=#1\advance\xxx by 97\char\xxx}
```

If we want to add some checks on range, it would become less elegant but still workable.

```
\def\lletter#1{\xxx=#1\relax\ifnum\xxx<1 $\langle$range error$\rangle$\else\ifnum\xxx>13%
$\langle$range error$\rangle$\else\advance\xxx by 96\char\xxx\fi\fi
```

Note that now we had to end the assignment of `\xxx` using `\relax`. If we did not do it, the definition would work fine with `\lletter\sect`, but not with `\lletter{7}`. Remarkably, we did not have to use `\relax` in the first macro (without checks). To see why would require going even deeper into TEX than we are willing to here in this manual, it is related to expansion of tokens described above. This is another example that unless one wants to become a real TEXpert, it is sometimes necessary to do some experimentation to make macros work properly.

Environments

A typical environment is a piece of text that is supposed to be formatted in a special way. As an example we develop an environment for theorems.

To start with, we want the name of the theorem boldface and its contents in italic. We also include a space before it so that we need not remember putting it there manually, and we actually want TEX to prefer page breaking before this statement, so we use `\medbreak`.

```
\def\begThm{\medbreak{\bf Theorem.}\hfil\break\begingroup\it}
\def\endThm{\endgroup}
```

We use it as follows:

```
\begThm $13$ is a nice number.\endThm
```

Now we will tweak it a bit. First, we not always call statements “Theorem”, so we use an argument to supply the proper title (which may then include a name as well). Second, we highlight our theorems by indenting them from sides. Third, we want to keep the statement together, no page breaking in the middle of a theorem. This can be done by enclosing the whole structure into a v-box, which uses another command for opening and closing groups (see above). This will also nicely restrict the narrowing effect just to our statements. We also recall that paragraphs should be closed within v-boxes where they appear. Here we go.

```
\def\begThm#1{\medbreak\vbox\bgroup\narrower{\bf #1.}\hfil\break\begingroup\it}
\def\endThm{\endgroup\par\egroup}
```

Recall that the effect of `\narrower` depends on the value of `\parindent`, we may want to preserve independence. Thus it is a good idea to actually specify the size of narowing using a custom register and define parindent locally to this value. Or, we can use this register to manually squeeze the statement, leaving `\parindent` independent. We will now show the last version, where we actually set `\parindent` to 0, we feel it looks better. Again, this change will be local thanks to the v-box.

To make things more interesting we will enclose statements in a box. We definitely do not want to use `\boxed`, since this command is quite picky, for instance it does not like math mode in its argument. Thus we create the

box using rules and boxes, just like in the appropriate chapter above. Now we want three registers that specify the distance between the bounding box and the contents on the sides, top and bottom. Thus we can easily finetune the appearance. Note that these spaces and bars take up some space, so we have to make the text itself narrower by the right amount, there will be also some moving involved for various parts to line up properly.

```
\newdimen\sidebx   % size of skip between box and statement on the side
\newdimen\topbx   % size of skip between box and statement on top
\newdimen\botbx   % size of skip between box and statement on bottom
\sidebx=5pt
\topbx=5pt
\botbx=3pt
\def\begThm#1{\medbreak\vbox\bgroup\parindent=0pt\hrule\hbox\bgroup\vrule\kern\sidebx%
        \vtop\bgroup\rightskip2\sidebx\vglue\topbx{\bf #1}\hfil\break\it}
\def\endThm{\par\vglue\botbx\egroup\kern-\sidebx\vrule\egroup\hrule\egroup}
```

Now `\begThm{Proposition (Euler).} $13$ is a nice number.\endThm` prints

Proposition (Euler).
13 *is a nice number.*

We can also incorporate automatic numbering into `\begThm`, see below.

Putting things up and down

We already saw how to put things into headline and footline. For instance, what we see here on the top was obtained using `\headline={\sevenrm Bonus: Some Useful Constructions \hfil Petr Habala}`

If we wanted "Book" on the right page and "pH" on the left, with the page number on the outside, we could try

```
\headline={\ifodd\pageno{\rm\folio}\hfil{\rm pH}\hfil\else\hfil{\rm Book}\hfil{\rm folio}\fi}
```

Now what if we want to put into footline information about what section we are at? We could prepare a macro that does it like this.

```
\def\putDown#1{\footline={\hfil{\rm #1}\hfil}}
```

When we start a section, we would supply the information using `\putDown{1.7}` or `\putDown{Macros}`. However, this could mean that we define footline several times on the same page. Is there a way to avoid it? Here is an interesting idea. We can create a box where we will store the information and define footline only once at the beginning of our file in terms of this box. Here we go:

```
\setbox13=\hbox{}
\footline={\hfil\copy13\hfil}
\def\putDown#1{\setbox13=\hbox{\rm #1}}
```

We can supply more kinds of information, each in its own box, and footline would show current status.

Putting things left and right

$\mathcal{A}$ The $\mathcal{A}$ mark to the left of some paragraphs is done as follows. First we prepare the object to be shown. If this object is taller than the height of a line, the second line would be pushed lower. We do not want it, so we have to make TEX think that our object has no depth and height, in this case we have `{\smash{$\Cal A$}\,\,}`. We added some spaces since we want this mark to be a bit to the left of the text.

Then we use `\llap` to convince TEX that this object has no width and place the pointer on its right edge. However, such a macro cannot be used at the beginning of a paragraph, since `\llap` works properly only in horizontal mode. Indeed, if we used just `\llap{\smash{$\Cal A$}\,\,} Now a paragraph`, TEX would start the paragraph on a new line:

$\mathcal{A}$

Now a paragraph. In order to avoid this we need to start a paragraph before we use `\llap`, the easiest way is to use `\noindent`. Thus far we came up with `\noindent\llap{\smash{$\Cal A$}\,\,}`.

If we put this and then some text, we get the mark on the left and the text starts immediately. However, we want this to look like a normal paragraph, so we add a space of the right size, it is better to use the actual value of paragraph indentation rather than some constant dimension. Finally, the user may want to type some spaces after our mark, we need to cancel them. The resulting macro reads

```
\def\ams{\noindent\llap{\smash{$\Cal A$}\,\,}\kern\parindent\ignorespaces}
```

We can make a general macro for putting a mark on the left, but there we encounter another problem, what if we want to position a longer text? We can create a mini-page there using `\vtop` (we want alignment by the first line). We now define the macro

```
\def\markleft#1{\noindent\smash{\llap{\vtop{\hsize=35pt\parindent=0pt #1}\quad}}%
            \kern\parindent\ignorespaces}
```

and we try it: `\markleft{Hi there people} So this seems to work ...` will print as

Hi there people So this seems to work well. We are quite happy with this macro. However, it is another story if we want to put a mark to the right. We will want to use `\rlap`, but it is not clear where to position such a mark in the source text.

We could try to run the text, see where the first line ended and then place the mark there, but that would be a truly desperate solution. Is there a simple way (perhaps not elegant, but without need of expert knowledge)?

Here's an idea. First we create an empty line that ends with our mark, then we tell TEX to back up by the appropriate amount and after that we can start our paragraph. We did it here like this: XX

```
\line{\hfill\rlap{\quad XX}}\kern-\baselineskip
```

A general definition for shorter marks (without \vtop):

```
\def\markright#1{\line{\hfill\smash{\rlap{\quad#1}}}\kern-\baselineskip\ignorespaces}
```

So far we explored the situation when we want to have a mark next to the first line of a paragraph. But what if we want to create some mark when we are in the middle of a paragraph, without influencing its shape? A low-tech solution uses the command \vadjust that finishes the line it appeared on, inserts its argument (some vertical matter) and then continues with the main text. Let's say that we want a mark on the right that can be used anywhere. We define

```
\def\floatmarkright#1{\vadjust{\line{\hfill\smash{\rlap{\quad#1}}}}}\ignorespaces}
```

and now we just write something so that it is longer than one line which could be any time now probably we are there so now we put `\floatmarkright{hey!}` and we will see what happens as we continue with this paragraph, yes, it seems that it did the job. hey! The mark is not quite positioned right, since the new line consisting of our mark was put below the current line, and this mark line has its level determined by the baseline of our text. Consequently, the baseline of our mark is lowered compared to the current line's baseline by the amount equal to the depth of the current line (how far it reaches below baseline) and the interline skip. We can shift the mark up by this amount and everything will be fine. If we do not know the precise amount, we can simply experiment, a good starting place is that the current font's depth is `2 pt`. We will show it all in a macro for marks on the left.

```
\def\fmarkleft#1{\vadjust{\line{\smash{\raise2.5pt\hbox{\llap{#1\quad}}\hfill}}}\ignorespaces}
```

yup And now we use it `\markleft{yup}` and this seems to work, we are obviously happy about it and we can close this paragraph now, it already is more than two lines long.

What happens if we first smash and then raise, that is, if we use the order `\raise\hbox{\smash{ ... }}`? Then we will be actually pushing lines apart, try it. Experimenting is the best way to get to understand TEX.

Putting things side by side

Situation: We have an object whose size we do not know beforehand (say, a picture). We want to typeset normal text and the picture to the right of it (with a \qquad between) so that the whole thing has the proper width of the page. For simplicity, we will assume that the text is not significantly taller than the picture (we will not try make the text flow around the picture).

To this end we take this picture (second argument), make it into a box and use its width to locally change the horizontal size of the page.

```
\newdimen\whatisleft
\long\def\rightset#1#2{\setbox37=\hbox{#2}\whatisleft=\hsize%
\advance\whatisleft by -\wd37\advance\whatisleft by -2em%
\noindent\hbox{\vtop{{\hsize=\whatisleft #1\par}}\qquad\vtop{\box37}}}
```

Note that we had to end #1 by \par so that its last paragraph ends *before* the group around `\hsize=` is closed, a paragraph is typeset at the width that was valid when it ended.

This definition has a flaw. The v-box with text has its pointer on the level of the first baseline. This is aligned with the pointer in the other argument (picture or whatever), but its pointer may well have a different location, for instance it may be at the bottom!

Fixing this is relatively straightforward for a TEXpert, a bit tougher if we want to use just some simpler tools. One possibility is to convince TEX that the box on the right has its pointer on the top. This is quite simple, we just add an empty horizontal box on the top of it and align it by the top. In this way we will always have the same level in the right component, but it is not the right level, as on the left we are aligning along the first baseline. We can decide to apply similar trick also to the left component to move the pointer to its top, or we may shift the right component up. If we decide for the latter, the macro becomes

```
\newdimen\whatisleft
\long\def\rightset#1#2{\setbox37=\hbox{#2}\whatisleft=\hsize%
\advance\whatisleft by -\wd37\advance\whatisleft by -2em%
\noindent\hbox{\vtop{{\hsize=\whatisleft #1\par}}\qquad\vtop{\hbox{}\kern-7pt\hbox{\box37}}}}
```

This has the advantage that by modifying the `\kern` we may make the picture start a bit lower that the previous text ended, which is a bit nicer. This definition is still not perfect, for instance it is sensitive to cases where the first line of the left component is taller than usual, but to make a really foolproof definition one would have to work harder on it.

If we want a picture first, text next, we can use (we show the simple version)

```
\newdimen\whatisleft
\long\def\leftset#1#2{\setbox37=\hbox{#1}\whatisleft=\hsize%
\advance\whatisleft by -\wd37\advance\whatisleft by -2em%
\noindent\hbox{\vtop{\box37}\qquad\vtop{{\hsize=\whatisleft #2\par}}}}
```

Automatic Numbering

We will show how to number two objects, chapters and statements. First we need counters.

```
\newcount\chapno
\newcount\statno
```

Then we need to make sure that they are properly reset. The chapter counter should be reset just once at the beginning, so after defining it we also put
`\global\chapno=0`.
We used global just to make sure that it applies to the whole document, in case we would later for instance enclose this part into a group.

We will want to refer to these numbers, it is convenient to prepare shortcuts for that as well. In this example we will want to refer to statements also by their chapters (since they will be reset for every chapter).

```
\def\refchap{\the\chapno}
\def\refstat{\the\chapno.\the\statno}
```

Next we prepare macros for chapters and statements. The formatting is irrelevant now, so we just use a very simple one, the important part is the work with counters. Note that we reset the statement counter for every chapter.

```
\def\chap#1{\global\advance\chapno by 1\relax\statno=0\centerline{\bf\refchap.  #1 ... }}
\def\stat{\global\advance\statno by 1{\bf Theorem \refstat.}}
```

Now all chapters and statements will be numbered automatically.

References

Our statements are numbered automatically, but we now also need some way of referring to them. The simplest way is to store the right numbers into special control sequences. For instance, we just wrote an important theorem about derivatives. Its number (whatever it is) is stored in the appropriate counter, so we can store it as a control sequence.

```
\xdef\Sder{\refstat}
```

Later on we can type `see Theorem~\Sder` and the right number will be substituted.

Since we are creating control words, the usual rules apply, this means that we can safely only use chains of letters as labels. It is possible to fix this so that we can label our references as we want (things like `derivatives-theorem2`), but that requires a better TEXpertise. I am happy with it as it is (I value simplicity), I would use the label `DerivativesTheoremTwo`.

Note: We had to use `\xdef` to force TEX to actually evaluate the contents. If we used just `\def\Sder{\refstat}`, then when we use `\Sdef`, TEX would substitute the contents, that is, the macro `\refstat`, which in turn would call `\statno` and thus put in the last typed statement's number.

The method we have shown is very simple, but it has a serious disadvantage, it only allows us to call those references after we make them. In some cases this is not a problem, then it can be a good idea to do it so easily.

If we want to automate this, we can define a macro,

```
\def\makeref#1{\xdef#1{\refstat}}
```

then we just put `\makeref{\Sder}` or `\makeref\Sder` and it works just like when we did this definition directly.

If there are more numbered objects (statements, chapters, examples) and we want to create references, it may be a good idea to create a more general reference making macro:

```
\def\label#1#2{\xdef\csname #1\endcsname{#2}}
```

To make `\Sdef` as above we would call `\label{Sdef}{\refno}`

We can also create a macro for calling such references, at this moment it does not make anything easier, but it makes our file compatible with various referencing packages, this can come handy if we want to use our file in some larger environment.

```
\def\ref#1{\csname#1\endcsname}
```

Now we just type `\ref{Sder}` and it is as if we called `\Sder`.

General References

If we want to refer to all our numbered parts, even those that are still to come, then we have to use an entirely different approach. In order to achieve this we have to run our source text through `tex` twice. First we let it automatically number things that we want, and important numbers must be stored in a separate file. When we run the file the second time, proper references are taken from this other file. How can TEX read info from another file? We have the command `\input`, but this simply pastes the contents into our source file. This means that we have to store the information is such a way that TEX understands it after pasting it into the source file. There is little choice, we have to store into this auxiliary file commands for creating references as we have them above.

We will show several versions, starting from the simplest. By the way, the name of this file is `mtex.tex`, it is customary to keep the name itself also for the auxiliary file and use the extension `.aux`.

```
\input mtex.aux
\def\ref#1{\csname XX#1\endcsname}
```

```
\newwrite\auxwrite
\immediate\openout\auxwrite=mtex.aux
\def\label#1#2{\immediate\write\auxwrite{\noexpand\def%
            \expandafter\noexpand\csname XX#1\endcsname{#2}}\ignorespaces}
```

What is happening here? The beginning is clear, we input definitions stored in `mtex.aux` and define a macro that allows us to refer easily to references there. When we call `\ref{`*text*`}`, it is as if we put there the macro (a stored reference) `\XX`*text*. What is the XX doing there? People tend to call their references in various ways and if they happen to use a name that is already a control word, it would redefine the original meaning with potentialy disastrous consequences. However, there is no control word beginning with `XX` in TEX, so we include it automatically into the created references (see the definition of `\label`) and then also include it into the `\ref` macro, now the user need not worry.

Then we open this auxiliary file for writing, but we need to give it a name, so we create a write-file register called `\auxwrite` and assign it as a handle for our file. We ask TEX to do it right away, because to speed up things it often delays writing and then does it in a batch, but this opens the possibility that references change meanwhile. We want it now.

Then we create a macro for writing definitions of references into this auxiliary file. The command for writing requires two arguments, a file and what should be written there. This brings some complications, since `\write` actually expands what we supply into it, which is sometimes what we want, but sometimes not. We want to write `\def\`*name*`{#2}` into the file, but if we allowed TEX to expand while writing it, TEX would simply read `\def` as a definition in the current file. Thus we need to tell TEX not to expand this particular command. On the other hand, we want `\csname` to be expanded so that it creates a control word in the aux-file. We can force it using `\expandafter`, but note that this command just copies the next token and expands the second one. We can put just anything as the first one, for instance a command that does nothing, `\expandafter\relax\csname #1\endcsname` is interpreted by TEX when it reads it as `\relax\`*name*.

However, then `\write` tries to write it into the file and as we already commented, it expands, so it would actually try to expand the new command `\`*name*. This would be no problem when done the first time, but if we already assigned this macro some meaning when reading old references from aux-file, then those old meanings would be substituted for `\`*name*. We need to prevent it. The sequence `\expandafter\noexpand\csname #1\endcsname` is read by TEX as `\noexpand\`*name* and `\write` writes it into the file without trying to replace `\`*name*.

We also added `\ignorespaces` so that the user can place `\label` on a separate line or otherwise shape the source file without influencing the outcome.

Now we add some sophistication. The above version works, but what happens when we run the source file the first time? Since there is nothing in the aux-file, our references will call on non-existing macros and TEX will complain. If we do not like it, there are two possible options. A simple one is to create the first aux-file by hand when typing the source file. When I put `\label{mytheorem}{\refstat}` in my tex-file, I also put a line
`\def\XXmytheorem{13} % Theorem on number writing`
in a separate txt-file. While this does not look too sophisticated, it actually serves another purpose: I have my references nicely organized, which makes it easier to call on them. Before I run the tex-file the first time, I simply copy this txt-file into an aux-file.

A modern solution is to check whether a file actually exists, and if not, then it is not included and the definition of `\ref` is changed in such a way that it does not call on those macros. We check a file for existing by opening it and asking whether it actually contains something. As usual, in order to manipulate it we need to create a handle for it, a register for reading files. A new version therefore looks like this.

```
\newread\auxread
\newwrite\auxwrite
\openin\auxread=mtex.aux\relax
\ifeof\auxread
        \closein\auxread \def\ref#1{\hbox{$\langle${\sl label}$\rangle$}}
    \else
        \closein\auxread \input mtex.aux \def\ref#1{\csname XX#1\endcsname}
    \fi
\immediate\openout\auxwrite=mtex.aux
\def\label#1#2{\immediate\write\auxwrite{\noexpand\def%
            \expandafter\noexpand\csname XX#1\endcsname{#2}}\ignorespaces}
```

What does it do? We open the aux-file. If it does not exist, then we see its end right away, so TEX closes it again and we define `\ref` so that ⟨*label*⟩ show us in our text at places where we try to refer.

Otherwise we close the file, which releases the lock placed on it when we opened it for reading, now we can include it and later open for writing.

We used `\relax` since filenames can interfere with what comes next. This is just a reminder for those who want to experiment, `mtex.aux` is safe also without it.

This is a pretty good version.

As a bonus we show an interesting trick. If we have a master file where we store our favourite macros, we may want to include also the file opening procedure as a macro that we would just call with the appropriate filename as argument. Moreover, we can make even the name automatic, since TEX stores the current filename in `\jobname`.

However, this presents a problem. If we enclose the `\ifeof` block above as the contents of some macro, then all the `#1` parts will be replaced by the argument of this macro. We need to protect it and there is a tool for it, we use `##1`. When this new macro is called, TEX substitutes its contents and in this process, `##1` is replaced by `#1`, exactly as needed.

```
\def\fileCheckRead#1{\openin\auxread=#1\relax
    \ifeof\auxread
        \closein\auxread \def\ref##1{\hbox{$\langle${\sl label}$\rangle$}}
    \else
        \closein\auxread \input #1 \def\ref##1{\csname XX##1\endcsname}
    \fi}
```

Now if we call `\fileCheckRead{\jobname.aux}`, this command will be expanded into exactly the same text as we see above.

Special positioning of even/odd pages

If we want to bind our document, we need to leave extra room for the binding. However, if we also want to print it both-sided, then this extra space should be positioned alternatively left and right. This is not easy to achieve for a casual TEX user, one needs to modify the standard output routine of TEX, obviously something that we did not cover here. The following procedure is due to Petr Olšák.

Somewhere at the beginning of our tex-file (before the text starts) we put

```
\output={\ifodd\pageno \advance\hoffset by 1cm \else
                       \advance\hoffset by-1cm \fi \plainoutput}
```

The equality sign after `\output` is not necessary, see the next example.

Printing in two columns

To have text printed in two columns per page is sometimes very desirable, Again, to achieve this we have to mess with the standard output routine, this time it is even more advanced.

Somewhere before the beginning of file put

```
\output={\ifvoid100 \global\setbox100=\box255 \else
\shipout \vbox{\hbox{\box100\kern15pt\box255}\bigskip
\hbox{\hskip8.2cm\the\pageno}}\advancepageno \fi}
```

However, first we have to fix the width of columns, that is, of virtual pages. We will show a better version that uses parameter `\colgap` specifying what gap between columns we want. We also show how a running head can be created.

```
\newdimen\colgap \newdimen\xxx
\colgap=15pt
\xxx=\hsize \advance\xxx by -\colgap \hsize=0.5\xxx
\output={\ifvoid100 \global\setbox100=\box255 \else
\shipout \vbox{\hbox{Running head}\bigskip
\hbox{\box100\kern\colgap\box255}\bigskip
\hbox{\hskip8.2cm\the\pageno}}\advancepageno \fi}
```

This is simple but it has a problem, the last page is not balanced. For that some more advanced tricks are needed.

Color in TEX

Getting color in TEX is not trivial because of the `pdftex` dichotomy.

If you plan on running `tex` to produce a dvi file and then proceed to or through postscript, then color can be obtained easily through the package `colordvi`. After `\input colordvi`, switches of the form `\textBlue`, `\textBrown` etc. become available. For each color there is also a block command like `\Blue` that expect text as an argument. There are almost a hundred color commands available. If you still feel the need for more colours, you can define custom color commands as follows:

```
\def\Blue#1{{\special{color push rgb 0.0 0.0 1.0} #1 \special{color pop}}
```

The command `\special` is ignored by TEX and is used to supply information to the driver that interprets the dvi file. In this case the `\special` is used to pass color information. Of course, the driver must understand the language, and most classical drivers do.

If we prefer to switch on and switch off color, we can use these custom commands:

```
\def\Color#1{\special{color push rgb #1}}}
\def\removeColor{\special{color pop}}
```

There is a CMYK version of the switch or the block command. To get a mix of 25% of cyan, 35% of magenta, 40% of yellow and 10% of black we would type `\textColor{.25 .35 .4 .1}` or `\Color{.25 .35 .4 .1}{text}`.

We can also change the color of background using, say, `\background{Yellow}`. Alternatives are `\special{background rgb .8 1 .8}` or `\special{background cmyk .183 .054 0 0}`.
Beware: If a colored text is broken accross pages, footline and headline will be colored as well, so use `\footline{\Black{\hss\tenrm\folio\hss}}` etc.

If we prefer to run the tex-file using `pdftex` to produce a pdf file directly (which is generally advisable as it often produces a better output, depending on the platform), then the above approach will not work. The `pdftex` program has certain quirks, in particular it does not recognize the color language used by `colordvi`. Then one has to use a different approach, but that one does not work with plain TEX for a change.
First one needs some auxiliary commands.

```
\newcount\Color
\Color=\pdfcolorstackinit page {0 g 0 G}
\def\resetColor{\pdfcolorstack\Color pop{}}
\def\setColor#1{\pdfcolorstack\Color push {#1 rg #1 RG}}
```

The last one is also sometimes written as

```
\def\setColor#1{\pdfcolorstack\Color push {#1 rg #1 RG}\aftergroup\resetColor}
```

The one can define coloring commands as follows:

```
\def\Blue#1{{\setColor{0.0 0.0 1.0} #1\resetColor}}
```

Or the alternative

```
\def\mycolor{0.0 0.0 1.0}
\def\Blue#1{{\setColor\mycolor #1\resetColor}}
```

Setting the background color is possible, but requires an arcane piece of code

```
{\catcode`\p=12 \catcode`\t=12 \gdef\noPT#1pt{#1 }}
\def\setBackground{\pdfliteral page {q {\cmykLightGray}
  0 0 \expandafter\noPT\the\dimexpr(.996264\pdfpagewidth)\relax % recalculating dimensions
  \expandafter\noPT\the\dimexpr(.996264\pdfpageheight)\relax re f Q}} % from TeX to PS points
```

Sometimes we need to use both approaches with one file. This can be solved by checking on the way the file is being processed. The simple version:

```
\ifx\pdfoutput\relax
    not running PDFTeX, put in colordvi
  \else
    running pdftex, define colors the other way
  \fi
\fi
```

A more sophisticated version could read

```
\ifx\pdfoutput\undefined
      not running pdftex
  \else
    \ifx\pdfoutput\relax
      not running pdftex
    \else
      running pdftex
    \fi
\fi
```

Some people actually use `pdftex` to produce a dvi output, so another alternative is

```
\ifx\pdfoutput\undefined
      not running pdftex
  \else
    \ifnum\pdfoutput>0
      running pdftex with pdf output
    \else
      running pdftex with dvi output
    \fi
\fi
```

Telling apart `tex` and `pdftex` is useful for other reasons as well. For instance, if we want to set the dimensions of the resulting pdf document, we have to use

```
\special{papersize=200mm,150mm}
```

if we plan to go through a postscript stage, but

```
\pdfpagewidth=200 true mm
\pdfpageheight=150 true mm
```

if we plan on using `pdftex`.

Pictures in document

If possible, avoid importing pictures into TEX documents. The PGF/Tikz package alows for drawing pictures directly in the `tex`-file.

However, this is not always possible. Then one has to take into account whether `pdftex` will be used or no.

For a long time, the standard for including pictures was the `epsf` package. After `\input epsf` we are ready to include pictures using

`\epsfbox`{*picture.eps*}

The picture must be in an encapsulated postscript format and the extension is obligatory when supplying the file name of the picture. By default, it is inserted in its original size, and we can place it using appropriate commands, for instance `\centerline`.

One can scale the picture using

`\epsfxsize=`(*dim*)`\epsfbox`{*picture.eps*}

or

`\epsfysize=`(*dim*)`\epsfbox`{*picture.eps*}

Note that when a dvi is produced, the picture is not actually inserted in this file. Rather, a space is reserved, and a pointer made to the picture file. It gets inserted only at the next stage by the driver. This causes trouble if the pdftex engine is used, and pictures are not inserted. The same happens when `pdftex` is used directly to compile the tex-file.

If we want to run `pdftex`, then the tools for including pictures are actually already built in, but these pictures must be in pdf form (preferably) or png or jpeg. Installations of TEX should also include the command for converting postscript images to pdf,

`\epstopdf` *picture.eps*

We then include pictures using the code

`\pdfximage width` (*dim*) `height` (*dim*) `depth` (*dim*){*picture.pdf*}

`\pdfrefximage\pdflastximage`

The first line reserves room, and any of the dimension specifications may be left out. If nothing is given, the picture is shown in its original size. When just one dimension is given, the picture is scaled. The second line inserts the picture

This pair of commands can be centered.

In fact, the command `\pdfrefximage` expects a number as the argument, which should be the number assigned to this particular picture at shipout. In order to save the author the trouble of finding out numbers of pictures, the `\pdflastximage` supplies the latest used number.

Hyperlinks in pdf

Clickable pdfs can be produced using TEX. The classical package for that is `hyperbasics`. After loading, the following commands are available.

`\href`{*url*}{*text*} changes *text* into a clickable reference to an outside document or another place in the current document if the # convention is used. For that, one can use these commands:

`\hname`{*anchor*}{*text*} drops a named anchor when *text* is typeset, and `\href`{`#`*anchor*}{*text*} makes a corresponding clickable link out of *text*.