# An introduction to TEX

## --- part I ---

## NTG course June 1992

David Salomon

## — 1. Introduction —

TEX is not written TEX, it is not spelled 'T' 'E' 'X', and is not pronounced tex. It is written TEX (a trademark of the AMS), it is spelled Tau Epsilon Chi, and is pronounced tech or rather teck.

TEX is not a word processor. It is a program that sets text in lines and paragraphs—but its design philosophy, its methods, and its approach—are all different from those of a word processor. A modern, state of the art word processor is a WYSIWYG type program; TEX is not, it uses one-dimensional input to generate its two-dimensional output. With a word processor it is easy to generate special printing effects, easy to underline text, to italicize and emphasize; TEX can do those things—and with a lot more attention to detail—but the user has to work harder. The same is true for applications that mix text with graphics. Good, modern word processors can also create diagrams, whereas TEX does not support graphics and some work is necessary to insert diagrams into its output. Other features that are standard and easy to use in a word processor are either non-existent or are hard to use with TEX.

If TEX is not a word processor, then what is it?

From the point of view of the user, TEX is a typesetting program which can be extended to a complete computerized typesetting system by adding a printer driver, the METAFONT program and some utilities.

From the point of view of the computer, however, TEX is a *compiler* whose main task is to *compile* a source program. The source language has variables (with types), block structure, two executable statements (assignment & if), and a powerful macro facility that makes it possible to extend the language. The main output of the compiler is a file containing, not machine instructions, but detailed instructions on how to place characters on a page.

When a compiler sees something in the source file which does not belong in the source language, it issues an error message. TEX, on the other hand, simply typesets any material in the source file that's not part of the language. The source file for TEX contains the text to be typeset, embedded commands, and comments. It's a text file and can thus be prepared with any editor or word processor. TEX generates two outputs, a log file—with run-time information and error messages—and a DVI file (for Device Independent), containing the page coordinates of each character to be typeset. A separate printer-driver program later reads the DVI file and generates printer commands (which are different for different printers) to print the text.

Two font files must exist for each font used in the text. A `.tfm` file and a bitmap file (which is usually called `.gf` or `.pk` file). The `.tfm` file (for TEX Font Metric) contains the dimensions of each character in the font (width, height, & depth) and some other information. The bitmap file contains the actual shape of the characters. TEX only uses the `.tfm` file. When the position of a character on the page is determined, TEX uses the character's width to move the reference point to where the next character should appear. As a result, TEX is ignorant of the actual shape of the characters, and its final output is a list of commands that specify what should be placed on the page and where. A typical command is a triplet of the form:

<char. # in current font, x-coord., y-coord.>

The bitmap file is only used by the printer driver, which sends the individual pixels of each character to the printer.

The diagram above summarizes the relationships between TEX, METAFONT, the printer driver, and the files involved.

To come back to the point of view of the user, the main aim of TEX is to produce *beautifully typeset documents*, especially documents containing mathematics. As a result, TEX is ideal for book publishing, for documents that should look beautiful—such as concert programs and invitations—and for technical publishing. Since typesetting is not a simple process (it is in fact very complex, especially the typesetting of tables and mathematics), TEX is *not easy to use.* It has several features that are complex and hard to master. As a result, new users tend to use TEX only if they need high quality output. When such a user needs a quick letter, a simple memo, or a short note, they usually go back to their familiar word processor. However, advanced users usually have macros that help produce all kinds of documents easily, thus using TEX's power all the time. Either way, if you like beautiful text, or if you write mathematics, TEX is by far the best typesetting tool available today.

To understand the power of TEX, it is necessary to understand the difference between traditional typing and book printing. When upgrading from a typewriter to a computer keyboard, some adjustment is necessary. The keys for '1' and 'l' are identical on a typewriter but different on a computer. Most typewriters only have one, non-oriented single quote ('), but computer keyboards typically have two oriented single quotes, a left (‘) and a right (’). When taking the next step of upgrading from word processing to typesetting (or book printing), a few more adjustments become necessary. The most important ones are:

■ Computer keyboards have just one kind of double-quote (") but books have two kinds, a left double-quote (“) and a right one (”). To produce them in TEX, you simply type two single quotes of the apropriate kind in a row. Thus to typeset “with regret.” you need to type `‘‘with␣regret.’’`

■ Another important difference is the dash (or hyphen). Computer keyboards have only one dash, namely (-); in a carefully typeset book, however, there are four different symbols:
a hyphen (-);
an en-dash (–);
an em-dash (—);
a minus sign ($-$);

Hyphens are used for compound words like ‘right-quote’ or ‘X-Register’. En-dashes are used for number ranges, like ‘Exercises 12–34’. They are obtained by typing two consecutive dashes ‘--’. Em-dashes are used for punctuation in sentences—like the ones around this section—they are what we usually call dashes. To get one, just type three consecutive dashes. Minus signs are, of course, used in typesetting mathematics.

■ The main task of TEX can be described as converting a one-dimensional stream of text into a two-dimensional page where all lines have the same width. This is done by stretching or shrinking the spaces between words on each line. In fine typesetting, however, those interword spaces have limited flexibility, so sometimes words have to be hyphenated. TEX uses a sophisticated hyphenation algorithm, so hyphenation is automatic. In cases where the algorithm does not work, the user can specify the correct hyphenation of words.

■ Well-printed books use ligatures and kerning. Certain combinations of letters, like ‘ff’, ‘fl’, ‘fi’, ‘ffl’, look better in the traditional roman type when the letters are combined. Such a combination is called a ligature. Foreign languages may have other ligatures, such as ‘ij’ in Dutch. Compare, for instance, the word ‘fluffier’ set by TEX to the same word ‘fluffier’ generated by a word processor. When a font is designed for TEX, the designer should specify all the special combinations that should be replaced by ligatures, and **design** the ligatures. That information goes into the font's `.tfm` file and is used by TEX to substitute the ligatures automatically. TEX can even remove a ligature later, for example, if it decides to hyphenate the word at that point.

Kerning refers to certain letter combinations that should be moved closer (negative kerning) or apart (positive kerning) for better appearance. An 'A' adjacent to a 'V' is a good example (compare 'AVAV' to 'AVAV'). Other examples are 'away', 'by', 'ox', 'ov', 'xe', and 'OO' (the last one has positive kerning). Again, the font designer decides what the kerning should be, and that information also goes into the font's `.tfm` file. TEX, of course, allows the user to easily override ligatures and to change the kerning to any desired value.

Ligatures and kerning improve the relationship between adjacent letters. They, together with hyphens and flexible interword spaces, are the main participants in the delicate balancing act required for the line break decisions.

Considering the power and flexibility of TEX, it is surprising that its basic algorithms are based mostly on three concepts—*boxes, glue* and *penalties*.

■ A box in TEX is an indivisible unit of material to be typeset. TEX will not break the contents of a box across lines or between pages.

To begin with, each character in a font is enclosed in a box whose dimensions (width, height, and depth) become the dimensions of the character. When TEX typesets a word, it pastes the individual character boxes side by side, with no spaces in between (except when kerning demands shifting boxes horizontally, or when the user wants boxes moved—which is how the TEX logo is produced). The result is a box containing the typeset word. The width of such a box is the total widths of the boxes inside, plus the kernings which, of course, may be negative. The height of the word-box is the maximum height of the component boxes, and the same is true for the depth.

Similarly, when TEX decides to break a line, it pastes the individual word boxes side by side, with appropriate spaces between them, and generates a new horizontal box, a line-box. The width of the line-box is the sum of widths of the word boxes inside it plus the sum of the spaces (glue) between the individual word boxes. The height and depth of the line box are the maximum heights and depths of the component boxes.

To set an entire page, TEX accumulates enough horizontal line-boxes; it then pastes them vertically, one below the other, with appropriate interline spaces, to generate a vertical page-box. The next step is to call an *output routine* which is either written by the user or supplied by TEX. The output routine adds finishing touches such as heading, footing, and page numbers. It may even decide to trim the page, to return part of the bottom of the page to TEX (to become the top of the next page), or do anything else to the page. The output routine should normally execute `\shipout`, which translates the contents of the page box to DVI file specs. TEX then continues to read the source to build the next page.

■ The term 'glue' refers to the spaces between boxes. In order to justify the text, TEX adjusts the spaces between words (but not the spaces between characters in a word). Those spaces must, therefore, be flexible. The same applies to interline spaces on the page. A glob of glue in TEX is a triplet $< w, y, z >$ where $w$ specifies the natural size of the glue, $y$ is the amount of stretch in the glue and $z$, the amount by which it can shrink. For interword glue, these values depend on the font and are specified by the font designer. For the standard 10 point roman font used by TEX, the values are <3.3333pt, 1.66666pt, 1.1111pt>. The line breaking algorithm considers various alternatives of combining words into lines, and for each alternative it calculates the amount by which the individual globs of glue have to be stretched or shrunk. After considering all the possible line breaks for an entire paragraph, the best line breaks are chosen (best in a sense that will be explained elsewhere), the entire paragraph is set, and TEX reads the next item from the source file (usually the start of the next paragraph).

■ The third mathematical construct used by TEX is the penalty. A penalty can be inserted into the text at any point either explicitly, by the user, or automatically, by TEX. The penalty specifies the desirability of breaking a line or a page at the point, and it is used to discourage bad breaks, to encourage good ones, to force certain breaks, and to avoid others. Penalty values are in the range $[-10000, 10000]$. Any value $\leq -10000$ is treated as $-\infty$, and any value $\geq 10000$ is considered $+\infty$. The user may, for example, insert '`\penalty100`' at a certain point in a paragraph, which has the effect that, if TEX decides to break the line at that point, a penalty of 100 will be added to the line, making that breakpoint less desirable. This tends to discourage line breaking at that point. A negative penalty is actually a merit, and it encourages a break. Infinite penalty prohibits a line break where it is specified, and $-\infty$ forces one. The commands (control sequences) `\break` `\nobreak` can be used to insert those infinite penalties at any point in the text. There are two common examples of penalties. The first occurs at any hyphenation. When TEX decides to hyphenate a word, it inserts a penalty of 50 at any potential hyphenation point. The second has to do with psychologically bad breaks. In a text containing '...Appendix G' it is psychologically bad to break between the two words because it interrupts the smooth flow of reading. To prevent such a break, a TEX user should type '`Appendix~G`'.

The tilde (~) acts as a 'tie'. It ties the two words such that in the final document there will be a space between them but no line break.

Similar examples are:

| | | | | | |
|---|---|---|---|---|---|
| Table~G-5 | Figure~18 | dimension~$d$ | Louis~XVI | 1,~2, or~3 | from 0 to~1 |
| Y~Register | modulo~$e^x$ | Rev.~Henry | HRH~prince Abdul | | |

Each tie is converted by TEX into a penalty of $-\infty$ to prevent a break.

This mechanism of boxes, glue, and penalties to arrange text in lines and pages, has proved extremely flexible and powerful. It makes it possible to set text in non-standard ways to achieve special effects. Features such as narrow paragraphs, newspaper formats, paragraphs with variable line widths, punctuations in the margins, ragged right margins, centered text, and complex indentations, can all be achieved with TEX. This mechanism is one of the main innovations introduced by TEX.

## — 2. Line breaking and page layout —

To achieve a straight right margin, TEX adjusts the spaces between words but not the spaces between characters in a word. The boxes defining each character are juxtaposed with no intervening spaces, for the following reasons:
- It makes for a more uniform appearance of the page. Because of the nature of human vision, spaces between words disturb the eye less than spaces between characters in a word.
- In an underlined font, spacing the characters would break the underline into separate segments.

This is perhaps a good point to discuss underlining in TEX. Underlining is one of the features that distinguish TEX from word processors, and can give an insight into the design philosophy of TEX. In a word processor it is usually easy to underline text; in TEX, underlining is discouraged. The reason is that TEX is not a word processor but a typesetter designed to produce beautiful books. In a book, underlining is rare, and emphasizing is done using either **boldface**, *italics*, or *slanted* fonts. If a certain text requires a lot of underlining, the best way to do it in TEX is to design a special font in which all the letters are underlined. Such a font requires no spaces between the characters in a word.

To achieve a uniform page layout, TEX uses two principles: 1. The vertical distance between lines is kept as constant as possible, using the rules below. 2. When a good page break requires squeezing another line on a page, or removing a line from a page, TEX changes the vertical distance between **paragraphs** (or between math formulas), not between lines. This again has to do with the way our eyes see, and guarantees that all the pages would appear to have the same proportion of black to white areas.

To determine the vertical distance between consecutive lines, three parameters $a, b, c$ are used. Their values depend on the font, and for the common 10 point roman font they are: $a = 12$pt, $b = 0$, $c = 1$pt. Typically, consecutive lines are not juxtaposed vertically but are spaced such as to make the distance between consecutive baselines equal to $a$. However, if the distance [top of lower line]−[bottom of upper line] is less than $b$, then the lines are spaced such that that distance is set equal to $c$. This may happen if the line contains a large (say, `18pt`) character. Parameters $a, c$ are of type `<glue>` so they may have flexibility. Typically this flexibility is zero, but the user may want to define, e.g. $a =< 12, 2, 1 >$. Such value makes sense for a short, one page, document. The flexibilty of the glue would make it easier for TEX to fit the text on one page.

Any discussion of TEX's line breaking algorithm should start with an outline of a typical line breaking algorithm used by a modern, commercial word processor. The method uses three values—for the natural, minimum and maximum spacings between words—and proceeds by appending words to the current line, assuming natural spacing. If, at the end of a certain word, the line is too long, the algorithm tries to shrink the line. If that is successful, the next word will start the next line, and the current line is printed. Otherwise, the word processor discards the last word and tries to stretch the line. If that is successful, the discarded word becomes the first one of the next line. If neither shrinking nor stretching works (both exceed the preset parameters), a good word processor tries to hyphenate the offending word, placing as much of it on the current line as would fit. The user may be asked to confirm the hyphenation, and the rest of the hyphenated word is placed at the start of the next line. A word processor that does not hyphenate has to resort to overstretching and may generate very loose lines.

The important feature of all such methods is that once a break point is determined, the algorithm does not memorize it but starts afresh with the next line. We can call such an algorithm 'first fit' and its main feature is that it does not look at the paragraph as a whole. Such an algorithm produces reasonably good results. For a high quality typesetting job, however, it is not fully satisfactory. For such a job, an algorithm is needed which considers the paragraph as a whole. Such an algorithm makes only tentative decisions for line breaks and may, if something goes bad toward the end of the paragraph, go back to the first lines and change their original, tentative, breakpoints. TEX's algorithm determines several feasible breakpoints for each line and calculates quantities called the *badness* and *demerits* of the line for each such breakpoint. After doing this for the entire paragraph, the final breakpoints are determined in a way that minimizes the demerits of the entire paragraph. Mathematically the problem is to find the shortest path in an acyclic graph.

## — 3. Fonts —

Traditionally, the word *font* refers to a set of characters of type that are all of the same size and style, e.g., Times Roman 12 point. A *typeface* is a set of fonts of different sizes but in the same style, e.g., Times Roman. A *typeface family* is a set of typefaces in the same style, e.g., Times.

The size of a font is normally measured in points (more accurately *printer's points*), where 72.27 points equal 1 inch. The reader should refer to [Ch. 10] for a description of all the valid dimensions in TEX. The *style* of a font describes its appearance. Traditional styles are roman, **boldface**, *italic*, *slanted*, `typewriter` and ßsans serif. In TEX, a font can have up to 256 characters, although most fonts only have 128 characters.

## — 4. The CM fonts —

Computer Modern (CM) is a metafont, developed in METAFONT, from which many different fonts have been derived, by different settings of parameters. The fonts all look different, but they blend together. They are called the CM fonts, and their names start with 'cm'. The standard CM fonts are:
▪ `cmr` or Roman. These are used for plain text. The standard sizes are (the '*' indicates fonts that are automatically loaded by the `plain` format) `cmr17 cmr12 cmr10* cmr9 cmr8 cmr7* cmr6 cmr5*`.
▪ `cmsl` or Slanted. They are slanted versions of the `cmr` characters. The standard sizes are `cmsl12 cmsl10* cmsl9 cmsl8`.
▪ `cmdunh` or Dunhill. Same as `cmr` but with higher ascenders. Only `cmdunh10` is standard.
▪ `cmbx` or Bold Extended. These are used for Boldface characters. `cmbx12 cmbx10* cmbx9 cmbx8 cmbx7* cmbx6 cmbx5*`.
▪ `cmb` or Bold. This is bold but too narrow for normal use. `cmb10`.
▪ `cmbxsl` or Bold Extended Slanted. `cmbxsl10`.
▪ `cmtt` or Typewriter. Fixed-space, resembling old typewriter style. `cmtt12 cmtt10* cmtt9 cmtt8`.
▪ `cmvtt` or Variable Typewriter. Same as `cmtt` but with proportional spacing. `cmvtt10`.
▪ `cmsltt` or Slanted Typewriter. `cmsltt10`.
▪ `cmss` or Sans Serif. Used for headings and for formal texts. `cmss17 cmss12 cmss10 cmss9 cmss8`.
▪ `cmssi` or Sans Serif Italics. `cmssi17 cmssi12 cmssi10 cmssi9 cmssi8`.
▪ `cmssbx` or SS Bold Extended. `cmssbx10`.
▪ `cmssdc` or SS Demibold Condensed. Normally used for chapter headings. `cmssdc10`.
▪ `cmssq` or SS Quote. Special SS font for quotations. `cmssq8`.
▪ `cmssqi` or SS Quote Italics. Again used for quotations. `cmssqi8`.
▪ `cminch` or Roman Inch. These are used for titles. `cminch`.
▪ `cmfib` or Fibonacci. In this version the parameters have relative sizes determined by the Fibonacci sequence. `cmfib8`.
▪ `cmff` or Funny Font. Different from the other cm fonts. Rarely used. `cmff10`.
▪ `cmti` or Text Italics. This is the normal italics font. `cmti12 cmti10* cmti9 cmti8 cmti7`.
▪ `cmmi` or Math Italics. Slightly different from text italics, and without spaces (spaces are automatically supplied in math mode). `cmmi12 cmmi10* cmmi9 cmmi8 cmmi7* cmmi6 cmmi5*`.
▪ `cmbxti` or Bold Extended Text Italics. `cmbxti10`.
▪ `cmmib` or Math Italics Bold. For math mode. `cmmib10`.
▪ `cmitt` or (text) Italics typewriter. `cmitt10`.
▪ `cmu` or Unslanted (text) Italics. Unslanted version of cmti. Used for Editor's notes in TUGboat. `cmu10`.
▪ `cmfi` or Funny Italics. An Italics version of cmff. `cmfi10`.
▪ `cmsy` or Math Symbols. Contains the math symbols normally used by TEX. `cmsy10* cmsy9 cmsy8 cmsy7* cmsy6 cmsy5*`.
▪ `cmbsy`. A Bold version of the math symbols. `cmbsy10`.
▪ `cmex` or Extension. More math symbols. `cmex10*`.

- `cmtex` or TEX Extended. An extended ASCII font. `cmtex10 cmtex9 cmtex8`.
- `cmcsc` or Caps & Small Caps. Contains small caps instead of lower case letters. `cmcsc10`.
- `cmtcsc`. A Typewriter version of `cmcsc`. `cmtcsc10`.
- `cmc` or Concrete. These were specially developed for the book *Concrete Mathematics*, to blend with the Euler math fonts.

Some of these fonts are rarely used, but were easy to obtain, by trying various settings of parameters. Any special sizes not mentioned above should also be easy to derive.

In `plain` TEX, the default font is cmr10. Also, macros `\bf`, `\it`, `\sl` and `\tt` are set [351] to select the different styles in 10 point size. If large parts of the document should be typeset in, say, 12 point, the definitions of `\bf` and its relatives should be changed accordingly. To do this, the different twelve point fonts should be loaded, at the start of the document, and assigned names by

```
\font\twerm=cmr12
\font\twebf=cmbx12
\font\tweit=cmit12
\font\twesl=cmsl12
\font\twett=cmtt12
```

Macro `\twelve` should then be defined as

```
\def\twelve{\def\rm{\twerm}\def\bf{\twebf}\def\it{\tweit}%
\def\sl{\twesl}\def\tt{\twett}%
\rm}
```

Text areas that should be set in 12 points should be bounded by `\begingroup\twelve` and `\endgroup`.

The CM family (Ref. 1) represents the most ambitious attempt so far to develop a general metafont. It is based on an earlier version called AM (almost modern), which is now obsolete. Two interesting adaptations of CM are outline fonts (Ref. 2) and the Pica fonts. The reader should refer to [Chs. 2, 4, 9 & App. F] for more information on the CM fonts.

Many special fonts have been developed in METAFONT. Examples are exotic languages, music notes, chess figures, astronomical symbols & logic gates. Ref. 3 is a detailed listing. The MetaFoundry (Ref. 4) developed many fonts in the early 1980s. However, very few other metafonts exist, the most well known of which are:

- Pandora, developed by N. Billawala (Ref. 5).
- The Euler family, designed by Herman Zapf, and developed at Stanford (Ref. 6). It is not a true metafont, as the characters were digitized.
- The Gothic family, including Fractur and Schwabacher, developed by Y. Haralambus (Ref. 7).

## — 5. Font examples —

This is an example of font cmr10 (roman)

**This is an example of font cmbx10 (bold extended)**

Some math symbols $\bigcap\Big[\Big]\sqrt{\Big]}\sqrt{\phantom{x}}\coprod\Big\{\,\widetilde{}\;\uparrow\coprod\Big|\surd\Big\lceil\widetilde{}$

Notice the absence of spacing in the next two lines. Math italics is automatically used and spaced in the math mode.

$Thisisanexampleoffontcmmi10\leftarrow mathitalics\leftarrow$

$\scriptscriptstyle Thisisanexampleoffontcmmi5\leftarrow mathit5pt\leftarrow$

*This is an example of font cmti10 (text italics)*

`This is an example of font cmtt10 (typewriter)`

This is an example of font helvetica (sans serif)

`This is an example of font Courier (fixed spacing)`

This is an example of font Palatino

This is an example of font times (roman)

THIS IS AN EXAMPLE OF FONT CMCSC10 (CAPS AND SMALL CAPS)

This is an example of font cmdunh10 (ascenders).

**This is an example of font cmssdc10 (sans serif demibold condensed)**

## — 6. Magnification —

The `\magnification` command scales the entire document. Everything—except the width & height of the text, and the margins—is made bigger or smaller. The magnification gactor in an integer, thus '`\magnification=2000`' scales everything by a factor of 2 (the document will spread over more pages), and '`\magnification=500`' makes everything half its original size. Note that this command can only be used once in a document, and should be placed at the beginning.

It is also possible to magnify individual fonts using the `scaled` parameter. Thus if font cmr12 is not available, it is possible to say '`\font\twelve=cmr10 scaled 1200`'. However, font `\twelve` will not look as good as the real thing.

Experience shows that certain font sizes blend together better than others. To encourage users to use those sizes, `plain` TEX includes [349] the quantities `\magstep0` (= 1000), `\magstep1` (= 1440), `\magstep2` (= 1728), `\magstep3` (= 2074), `\magstep4` (= 2488), and `\magstephalf` (= 1095). Again, it should be emphasized that magnification of fonts reduces their quality. For best results, all the fonts needed in a document should be available without having to magnify anything.

## — 7. Advanced Introduction —

The rest of this chapter is an advanced introduction to TEX, stressing the main parts, main operations, and certain, advanced, concepts. As with most other presentations of this type, some terms have to be mentioned before they are fully introduced, so the best way to benefit from this material is to read it twice.

## — 8. Registers —

A register is temporary storage, a place where data can be saved for later use. Using a register thus involves two steps. In the first step something is stored in the register; in the second step, the contents of the register is used. With the exception of box registers, the contents of a register can be used and reused indefinitely. There are six classes of registers, summarized in the following table.

| Class | Contents | Default value |
|---|---|---|
| `\count` | integer | 0 |
| `\dimen` | dimension | 0pt |
| `\skip` | glue | 0pt plus0pt minus0pt |
| `\muskip` | muglue | 0mu plus0mu minus0mu |
| `\toks` | token string | empty |
| `\box` | a box | void |

Note that each class of registers can only be used for data of a certain type. The registers are thus similar to strongly typed variables used in many programming languages. Each class contain 256 registers, so there are, e.g., the 256 count registers `\count0` through `\count255`.

Reserved Registers. Certain registers are reserved by TEX for special purposes. `\box255` is used by the OTR. `\count0` through `\count9` are used by the `plain` format for the page number. Those registers should not be used unless you know what you are doing.

Declaring registers. Since it is not a good idea to refer to registers explicitly, the plain macro `\new` should be used to declare registers and assign them names. Thus `\newcount\temp` is a typical use. It allocates the next available count register, and assigns it the name `\temp`.

Five of the six register classes are similar, but box registers are different. Boxes have dimensions and internal structure, they can be nested by other boxes, and tend to consume memory space. As a result, there are differences between box registers and other registers.

- The command `\newcount\temp` creates a quantity `\temp` whose value is, e.g., `\count18`. In contrast, `\newbox\Temp` creates `\Temp` as the number 18. Saying 'A\Temp B' is thus identical to 'A`B'. (This can be verified by `\show\temp`.)
- Assigning a new value to a register is done by an assignment statement of the form `\temp=1234`. Assigning a new value to a box register is done by means of '`\setbox\Temp=...`'.

▪ Box register 255 is reserved for the use of the OTR. Registers `\count255`, `\skip255` etc. are available for general use.

▪ A box register is emptied when it is used. The register thus becomes void. In contrast, other registers cannot be void; they must always contain a value.

▪ The contents of any register, except a box register, can be written on a file. Thus we can say, e.g., `\write\abc{\the\skip0}`, but we cannot say `\write\abc{\the\box0}`. The reason is that boxes are complex structures.

▪ The primitive `\the` can be used to produce the value of any register except a box register.

▪ At the start of a job boxes are normally void, but there is a subtle point, involving `\box0` and the `plain` format, that users should keep in mind. The `plain` format says (on [361]) `\setbox0=\hbox{\tenex B}`. This is used in the definition of macro `\bordermatrix`. As a result, `\box0` is not void at the start of a job. This point is mentioned again, in connection with `\lastbox`.

## — 9. `\the` —

The primitive `\the` can be used to produce the values of certain internal quantities. TEX novices are always confused by it. A typical complaint is “Why do I say ‘`\box0`’ to typeset `\box0` but ‘`\the\count0`’ to typeset `\count0`?” The answer is—to make it easier for TEX to compile the document and to detect errors.

The command `\the` must be followed by an internal quantity, such as a register. It produces tokens that represent the value of the quantity. Thus after saying ‘`\newcount\temp \temp=1234`’, the command `\the\temp` creates the tokens ‘1234’. Thus ‘`\hskip\the\temp pt`’ will skip 1234 points, and ‘`ABC\the\temp GHK`’ will typeset ‘1234’ between the `C` and the `G`.

The command `\temp`, in contrast, does not create tokens. TEX considers it the start of an assignment, unless it expects an integer at this point. Consider, e.g., the command ‘`\hskip\temp pt`’. After TEX has read the `\hskip`, it expects a dimension (a number followed by a valid unit of dimension). This command will thus skip 1234 points.

The two examples above suggest that `\temp` and `\the\temp` produce the same results, but this is not generally true. Consider the assignment ‘`\skip0=3pt plus 2pt`’. Saying ‘`\hskip\the\skip0 minus 1pt`’ will skip by ‘`3pt plus 2pt minus 1pt`’, but saying ‘`\hskip\skip0 minus 1pt`’ will skip by ‘`3pt plus 2pt`’ and will consider the ‘`minus1pt`’ text to be typeset.

In the former case, the tokens produced by `\the\skip0` blend with the rest of the document and become part of the `\hskip` command. In the latter case, TEX expects the `\hskip` to be followed by a valid dimension and, on finding `\skip0`, is satisfied and executes the command, not looking for any more arguments.

## — 10. Modes —

Of all the advanced concepts, the idea of modes [Ch. 13] is, perhaps, the most important. At any given time, TEX is in one of six modes, and its behaviour depends on the current mode. The mode can frequently change and can also be nested. The six modes are:

▪ Horizontal, or H, mode. TEX is in this mode when it reads the text of a paragraph.

▪ Vertical, or V, mode. This is where TEX usually spends its time between paragraphs, executing commands.

▪ Restricted horizontal (or RH) mode. TEX switches to this mode when it builds an `\hbox`. This mode is very similar, but not identical to, H mode.

▪ Internal vertical (or IV) mode. Commands such as `\vbox` or `\vtop` force TEX to go into this mode, which is very similar, but not identical to, V mode.

▪ Inline math mode, where a math inline formula is built.

▪ Display math mode, where a display formula is constructed.

When TEX starts, it is in V mode (between paragraphs). It reads the input file and, when it sees the first character to be typeset (or anything that's horizontal in nature, such as `\noindent`, `\vrule`), it switches to H mode. In this mode, it first executes the tokens in `\everypar`, then reads the entire paragraph into memory. The paragraph is terminated by `\par`, by a blank line, or by anything that doesn't belong in H mode (such as `\vskip` or `\hrule`). TEX then switches to V mode, where it sets the paragraph and takes care of page breaking.

A paragraph is set by breaking it into lines which are appended—each as an `\hbox`—to the *main vertical list* (MVL). After appending the lines of a paragraph to the MVL, TEX determines whether there is enough material in the MVL for a full page. If there is, the page breaking algorithm is invoked to decide where to break the page. It moves the page material from the main vertical list to `\box255` and invokes the output routine. Some material is usually left in the MVL, to eventually appear at the top of the next page. The routine can further modify `\box255`, can add material to it or return some material from it to the main vertical list. Eventually, the output routine should invoke `\shipout` to prepare the DVI file. TEX stays in V mode and continues reading the input file.

Modes can be nested inside one another. When in V mode, between paragraphs, TEX may be asked to build a `\vbox`, so it enters IV mode. While in this mode, it may find characters of text, which send it temporarily to H mode. In that mode, it may read a '`$`', which causes it to switch to inline math mode. The curious example on [88] manages to nest all the modes at once.

## — 11. Anatomy of TEX —

A quote, from [38], is in order. "It is convenient to learn the concept (of tokens) by thinking of TEX as if it were a living organism." We will develop this concept further, and try to get a better understanding of the overall organization of TEX by considering the functions performed by its main "organs", namely eyes, mouth, gullet, stomach, and intestines (see anatomical diagram on [456]).

▪ TEX uses its "eyes" to read characters from the current input file. The `\input` command causes TEX to 'shift its gaze' and start reading another input file.

▪ In its "mouth", TEX translates the input characters into tokens, which are then passed to the "gullet." Spaces and end-of-line characters are also converted into tokens and sent to the gullet. A token is either a character of text or a control sequence. Thus the name of a control sequence, which may be long, becomes a single token. The process of creating tokens involves attaching a category code (see below) to each character token, but not to a control sequence token.

▪ The "gullet" is the place where tokens are expanded and certain commands executed. Expandable tokens [373] are macros, `\if...\fi` tests, and some special operations such as `\the` and `\input`.

A token arriving at the gullet is sent to the stomach, unless it is expandable. Expanding a token results in other tokens that are, in turn, either expanded, if they are expandable, or sent to the stomach. This process continues until no more expandable tokens remain in the gullet, at which point the next token is moved from the mouth to the gullet, starting the same process (the word "regurgitation", on [267], nicely describes this process). Certain commands, such as `\expandafter` & `\noexpand`, affect the expansion of tokens, so they are executed in the gullet.

Expandable tokens are macros, active characters, conditionals, and some primitives, e.g. `\romannumeral`, listed on [215]. They are expanded in the gullet. For macros with no parameteres the expansion is a simple replacement (also for some primitives, such as `\jobname`). Normally, however, the token expanded depends on arguments, which have to be read before the expansion can take place.

Exceptions: The construct `\expandafter`$\langle$token$_1\rangle\langle$token$_2\rangle$ is treated by the gullet in a special way. It is replaced by $\langle$token$_1\rangle\langle$expansion of token$_2\rangle$, which is then scanned again by the gullet.

A `\noexpand`$\langle$token$\rangle$ prevents the gullet from expanding the token.

▪ As a result, there is a constant stream of tokens arriving at the "stomach", where they get executed. Most tokens are characters of text, and are simply typeset (appended to the current list). Tokens which are TEX primitive commands are executed, except that the "stomach" may have to wait for the arguments of the command to arrive from the gullet. Recall that non-primitive commands are macros and are expanded in the gullet.

Another way of explaining stomach operations is: The stomach executes tokens coming from the gullet. It classifies all tokens into two groups, tokens used to construct lists, and tokens that are mode independent. The former group includes characters of text, boxes and glue. They are mode sensitive, can change the mode, and are appended to various lists. The latter can be assignments, such as `\def`, or other tokens, such as `\message` or `\relax`. The `\relax` control sequence deserves special mention. It is a primitive and thus unexpandable. The gullet passes it to the stomach, where its execution is trivial. It is an important control sequence, however, because it serves as a delimiter in both the gullet and the stomach.

The result of executing tokens in the stomach is larger and larger units of text. Individual characters are combined to make words, which are combined to form lines, which are combined to make pages of text. When the next page of text is ready, it is sent to the "intestines."

■ The output routine corresponds to the "intestines" of TeX. It receives a page of text and, after some processing, translates it into DVI commands that are appended to the DVI file. The processing may consist of adding something to the page (such as a header, a footer, footnotes or margin notes) or deleting something from it (certain lines or even a big chunk). The deleted material is either discarded or is returned to the stomach.

The DVI file is the final output of TeX, and it comes out of the intestines in pages. It consists of commands that tell where each character is to be placed on the page. Ref. 8 is a detailed description of the DVI file format.

The entire anatomy is summarized in the diagram below.

$$\begin{tikzcd}
\boxed{\text{Source file}} \arrow[r, "\text{eyes}"] & \boxed{\text{ASCII chars}} \arrow[r, "\text{mouth}"] & \boxed{\text{Tokens}} \arrow[r, "\text{gullet}"] & \boxed{\text{Tokens}} \arrow[r, "\text{stomach}"] & \boxed{\text{Lists}} \arrow[r, "\text{OTR}"] & \boxed{\text{DVI file}}
\end{tikzcd}$$

The advantage of this anatomical description is that we can think of the process as a pipeline. Material (mostly tokens) advances from organ to organ and is processed in stages. However, the individual organs sometimes affect each other. The best example is the `\catcode` primitive. It is executed in the stomach, and immediately starts affecting the way the mouth assigns catcodes to tokens. Another example is the `\def` primitive. When a `\def\abc` is executed, the definition becomes available for the gullet to use whenever `\abc` has to be expanded. As soon as another `\def\abc` is executed in the stomach, the gullet is affected, and will use the new definition in future expansions of `\abc`.

## — 12. Characters —

TeX inputs characters from the input file and outputs characters to the DVI file, so characters are important for an overall understanding of TeX. Characters usually come from the input file (through the eyes), but may also come from the expansion of macros (in the gullet). These are the only character sources of TeX. Most characters are simply typeset, but some, such as '\' and '`$`', have special meanings and start TeX on special tasks.

A character is input as an ASCII code, which becomes the *character code*. The character then gets a *category code* attached to it (in the mouth), which determines how TeX will process the character. When a macro is defined (in the stomach), catcodes are attached to each character (actually, to each character token) in the definition, and are used when the macro is later expanded (in the gullet). Advanced users would like to know that a certain amount of processing goes on even in the mouth. Among other things (see complete description on [46–49]) consecutive spaces are compressed into one space, carriage returns are inserted at the end of each line, consecutive spaces at the beginning of a line are ignored, and spaces following a control word are ignored (they never become space tokens).

There is a simple way to find the ASCII code of a character. Just write a left quote followed by the character [44]. Thus `` `b `` has the value 98, and `` \number`b `` will actually typeset a 98.

**Exercise 1:** What will be typeset by `` \number`1 `` and what, by `` \number`12 ``?

**Answer.** The result of `` \number`1 `` is 49, the character code of '1', being typeset. The result of `` \number`12 `` is the same 49, which is typeset, and is immediately followed by a 2; thus 492.

**Exercise 2:** What is the result of `` \number`{0} `` and what, of `` \number`% ``?

**Answer.** The character code of '`{`' (123) will be typeset, The right brace will cause an error (too many '`}`'). The `%` in `` \number`% `` will be considered a comment, so the result will be the code of whatever character happens to follow (see later on how to get the character code of `%`).

The ASCII table provides 128 codes, but keyboards normally don't have that many keys. The `\char` command or the '`^^`' notation [45–46] can be used to refer to a keyless character. Thus the notation '`^^`$x$', where $x$ is any character, refers to the character whose ASCII code is either 64 greater than, or 64 less than, the ASCII code of $x$. Examples are '`^^M`' (return), '`^^J`' (line feed), '`^^@`' (null), '`^^I`' (horizontal tab), and the other ASCII control characters. The `\char` command is easy to use; if the current font is cmr10, then `\char98` is the code of 'b'. In general, `\char127` is the code of the character in position 127 of the current font.

The difference between the two notations is that `\char` is easier to use but is executed in the stomach (i.e., late); in contrast, a '`^^`*x*' is preprocessed into a single token in the mouth (i.e., as soon as it is input). As a result, the concoction `\def\a^^"c{...}` defines a macro `\abc`, but `\def\a\char98c{...}` defines a macro `\a` with the string `\char98c` as a delimiter.

## — 13. End of line —

We intuitively think of a line of text as ending with a carriage return. TEX, as usual, offers a more general treatment of this feature. At the end if every line of text, a special character is inserted, that is the value of parameter `\endlinechar` [48]. This parameter is set by `INITEX` to a ⟨return⟩ (`^^M`), but can be changed by the user. If `\endlinechar` is negative or is greater than 255, no end-of-line character is inserted. In this case, the input is considered one long line. The same effect can be obtained by ending every line with a comment character.

## — 14. Numbers —

Not everything can be typeset by `\the`. The definition of a macro (its replacement text), e.g., can only be typset by `\meaning`. To print a numeric quantity, the primitive `\number` can always be used. Even if the number is the value of a macro. Defining the macro `\def\ctst{10}`, we cannot say `\the\ctst`, but we can say `\number\ctst`. Also `\dimen` registers are printed differently by `\the` and by `\number`.

The left-quote character usually acts as a normal character of text. Sometimes, however, it signifies the start of a number. Similarly, the right-quote and double-quote characters sometimes have special meanings; they signify the start of an octal or hexadecimal number. When these characters arrive at the stomach, they are tokens with catcode 12 (other). If the stomach is expecting a number, it will assign them their special meanings and expect them to be followed by digits (decimal, octal, or hex). Otherwise, they will simply be typeset.

The following are all valid representations of the decimal number 98: `98 +98 098 '142 "62 `b `\b`. They may be used with any command requiring an argument of type ⟨number⟩ (see definition of ⟨number⟩ on [269]). Thus ``\number`b`` wil typeset 98, and `\catcode+98="D` will convert the letter 'b' into an active character. (Incidentally, once you do that, 'b' isn't a letter any longer, so something like `\bye` will be interpreted as the control symbol `\b` [52], followed by the letters 'ye'.)

**Exercise 3:** What is the result of ``\catcode101=14 \number`e``?

**Answer.** The character code of 'e' is 101, so the assignment makes 'e' a comment character. The `\number` is now considered the control sequence `\numb` (which is normally undefined), followed by a comment. The result is the error message `! Undefined control sequence \numb`.

Both notations `` `b `` and `` `\b `` produce the character code of 'b'. The difference between them is that the latter form can be used with any character. Thus ``\number`\%`` produces 37 but ``\number`%`` will treat the '%' as a comment, and will look for an argument on the next line. Similarly, ``\number`\^^M`` produces 13 (ASCII ⟨return⟩), but ``\number`^^M`` will consider the `^^M` an end-of-line, will replace it with a space, and will produce 32 (ASCII ⟨space⟩).

Non-integer numbers can only be used with dimensions, and are considered multipliers. Thus `1pt` multiplies the basic unit of a `pt` by one, and `2.5\baselineskip` mutiplies the current natural size of `\baselineskip` (not its stretch and shrink components) by 2.5.

## — 15. Category Codes —

One of the main considerations behind the design of TEX was to make it as general and flexible as possible, so it could be adapted to many different tasks. Thus the character '\' is normally used to start the name of a control sequence (it is the *escape character*), but if the '\' is needed for other purposes, any character can be defined as the escape character. The same thing is true for the other special characters, namely { } $ & # ^ _ and %.

Each of those characters is special only because it is assigned a special *category code* when TEX starts. It is easy to change the category codes, thereby changing the meanings of characters. There are 16 category codes [37] numbered 0 to 15:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | Escape character | 4 | Alignment tab | 8 | Subscript | 12 | Other character |
| 1 | Begin. of group | 5 | End of line | 9 | Ignored character | 13 | Active character |
| 2 | End of group | 6 | Parameter | 10 | Space | 14 | Comment character |
| 3 | Math shift | 7 | Superscript | 11 | Letter | 15 | Invalid character |

The category code of the character 'A' can be typeset by the command `\the\catcode'A`. It can be displayed in the log file by `\showthe\catcode'A`. The category code is assigned to a character in the mouth, and that assignment is permanent. The pair ⟨character code, category code⟩ becomes a *token*. The mouth also converts control sequences into tokens, but they do not get a catcode.

When a macro is defined (in the stomach) each token of the replacement code gets a catcode assigned. When the macro is later expanded (in the gullet), the replacement code is copied, with arguments replacing the parameters, and the resulting tokens are sent to the stomach.

The catcode of a character can be changed by the `\catcode` primitive. The most common example is `\catcode'\@=11`, which makes '@' a letter. It can now be used in control words, as any other letter, and the `plain` format [App. B] makes heavy use of this in order to define 'private' macros, inaccessible to the user. Another example is `\catcode'\<=1`, `\catcode'\>=2`, which define the characters '<' and '>' as group delimiters. This does not change the definition of the braces, so now a group may be specified by `{...>`.

Here are some notes about catcodes:

▪ A character created by '`^^`' is assigned a catcode in the mouth, but a character created by `\char` is not assigned a catcode at all! This is because `\char` is a primitive and is thus executed in the stomach. Executing `\char` always creates a character of text, which is typeset.

▪ Code 14 (comment) causes the rest of the input line to be ignored, is itself ignored, and no end-of-line character is appended to the line.

▪ A space following an active character is not ignored.

▪ Category 9 is always ignored, but can be used to delimit a control sequence name. Thus if we change the catcode of 'x' to 9 (by `\catcode'\x=9`), future occurrences of 'x' will be ignored, and the string '`NxO`' will be typeset as 'NO'. However, a macro `\abc` can now be expanded by saying `\abcx...`. The ignored character 'x' serves to delimit the name `\abc` the way a space normally does. When TEX starts, the only ignored character is the ASCII ⟨null⟩ (`^^@`).

**Exercise 4:** What's the reason for category 9 (in other words, why type a character and then ignore it)?

**Answer.** When TEX is used to produce graphics, we sometimes want to suppress all spaces (see example on [390]). This can be done by `\catcode'\ =9`. A carriage return can be ignored by saying

```
\catcode'\
=9
```

on two separate lines. Also, certain control characters can sometimes be added to a file when it is transmitted between computers, and they should be ignored by TEX.

▪ Each blank line becomes a `\par` token, so there may be several consecutive such tokens, of which only the first is executed by the stomach.

Category 15 (invalid character) is initially assigned to the ASCII ⟨delete⟩ (`^^?`). TEX complains when it reads an invalid character.

## — 16. Commands —

Many different commands are available in TEX, and they can be classified in two ways:

▪ A command can be classified as a primitive, a character, or user-defined. The latter category is further classified into a macro or an active character.

▪ A command can also be classified as either horizontal, vertical, or neither.

Beginners learn very quickly that a command should start with a '\'. However, an active character is also a command, and even a character of text (catcodes 11, 12) is one (see [267]). When a command is used very often, we want its name to be as short as possible, so we define it as an *active character* (catcode 13). It then becomes a one-character command, without even a '\'.

It is useful to consider a character of text (other than a space) a command. Such a command tells TEX to start a new paragraph or, if it is already in H mode, to typeset the character. A character is thus a *horizontal command*. If we want TEX to treat a character as not horizontal, we can place it in an `\hbox`. Interestingly, an `\hbox` is not inherently horizontal (but neither is it inherently vertical).

A command starting with a '\' is called a *control sequence*. If it consists of letters only (catcode 11), it is called a *control word*; if it consists of a non-letter (any catcode $\neq$ 11), it is called a *control symbol*. Any spaces (or end of line) following a control word are ignored by TEX since it assumes that they are there only to delimit the word. However, a control symbol can only have one character following the '\', so there is no need for any delimiters. Spaces following a control symbol are not ignored, and a situation such as '`\?1`' is interpreted as the control symbol `\?` (normally undefined), followed by a '`1`' (which is normally typeset).

The ignore-space rule above, however, applies only to characters coming from the input file; if a control sequence comes from a token list [39], a space following it will not be ignored.

**Exercise 5:** Devise tests to prove the preceding statements!

**Answer.**

```
\def\abc{`} \abc ' \abc\ '
\def\?{\message{ok}} `\?1' `\? '
\toks0={\abc}
\toks0=\expandafter{\the\toks0 '} \showthe\toks0
```

The concept of a delimiter is worth a little discussion. If we want to expand a macro `\abc` we can say '`\abc␣`'. However, `\abc1...` is also okay. The '\' tells TEX that a control sequence starts, the '`a`' tells it that this is a control word (just letters), and the '`1`' delimits the string of letters. After TEX reads the '`1`' it backspaces over it, executes `\abc`, and rereads the '`1`'. The point it that the '`1`' is read twice and, when it is first read, it is not assigned a catcode.

This is a subtle point that may, sometimes, lead to errors. Consider the following:

```
\def\abc{\catcode`\%=11 }
\abc%
\abc%
```

The first line defines `\abc` as a macro that makes the '`%`' a letter. The second line uses '`%`' to delimit the string of letters `abc`. The '`%`' is thus read twice. When it is first read, it is not assigned a catcode, and `\abc` has not been expanded yet. When the '`%`' is read again, `\abc` has already been expanded, so the '`%`' is assigned a catcode of 11, which causes it to be typeset. When the third line is input, the '`%`' already has catcode 11, so it is considered a letter. TEX thus ends up with the string '`abc%`', and tries to expand macro `\abc%` which is normally undefined. The result is the message `! Undefined control sequence \abc%`.

**Exercise 6:** Why is there a space following the '11' in the definition of `\abc` above, and what happens without that space.

**Answer.** This is a result of the general rule that says that a number should normally be terminated by a space. TEX considers the space a terminator, so it does not get typeset. To better understand this rule, try the following:

```
\count20=20%
1stop
```

This example sets `\count20` to 201 and typesets 'stop'. The reason is that, after reading the '20', TEX reads ahead, hoping to find more digits. It finds the '1' since the '%' does not terminate a number.

Without the space, our three-line example is executed in the following steps:
**1.** The first line is read, and `\abc` is defined.
**2.** The second line is scanned. The '%' is read, which terminates the name `\abc`. The macro is expanded and the last thing, of course, is the '11'. Therefore, before executing the expansion, TEX reads the next character, hoping to find more digits. Since the next character is the '%', TEX skips to the next line and starts expanding `\abc`, still hoping to find more digits. The expansion of `\abc` on the third line, however, does not start with a digit.
**3.** At this point, TEX realizes that there are no digits following the '11' (from the second line). It therefore executes `\abc` from the second line, which changes the catcode of '%'.
**4.** Next, `\abc` from the third line is executed, which does not change a thing.
**5.** The '%' of the third line is reread and, since it is now considered a letter, it is typeset. (End of answer.)

There is an important difference between a character and a control sequence. A character has a catcode attached to it, which tells the gullet and the stomach what to do with the character. A control sequence has no catcode and may be redefined at any time, so the gullet has to look up the current definition before it can expand the control sequence.

A command may have arguments. Thus `\kern` must be followed by a dimension, and a '`^`' in math mode must be followed by the superscript. Sometimes the arguments are optional (the '`=`' in an assignment is a typical example), and sometimes there is a choice of arguments (`\leaders` should be followed [281] by either a ⟨box⟩ or a ⟨rule⟩, so it can be followed by one of the following: `\box15 \copy16 \vsplit17 \lastbox \hbox \vbox \vtop \hrule` or `\vrule`). The `\font` command [16] makes for an interesting example. It starts with `\font`\⟨name⟩=⟨file name⟩, followed by the optional arguments '`at` ⟨size⟩' or '`scaled` ⟨factor⟩'. The words '`at`', '`scaled`' are called *keywords*, and don't have a '\'. See [61] for a complete list of keywords.

In a command such as `\setbox0=\hbox{...}`, the arguments are '`0=\hbox{...}`'.

## — 17. Assignments —

An assignment [275] is any command that assigns a new meaning to a control sequence or to an internal quantity. Examples are `\def`, `\hsize=...`, `\font\abc=...`, `\setbox0=...` & `\advance\x...`. Note that the '=' is always optional. Assignments are examples of commands that are executed in the same way regardless of the current mode.

**Exercise 7:** What other commands are executed in a mode-independent way?

**Answer.** See list on [279–281].

## — 18. Lists —

This is another important concept. The contents of a box is a list, as is the contents of a math formula. A list is made up of items such as boxes, glue & penalties.

TEX is assembling a horizontal list when it is in H mode (building paragraphs) or in RH mode (building an `\hbox`). The items of such a list are strung horizontally, left to right, and must be H mode material [95]. In H mode, a list is terminated when TEX reads a `\par` (or a blank line), or anything that's vertical in nature (such as a `\vskip`). In RH mode, TEX terminates a list when it finds the '`}`' of the `\hbox`. If it finds inherently vertical material in this mode, it issues an error. Examples of horizontal commands are `\vrule`, `\valign`, `\char` and a character of text (see [283] for the complete list).

A character of text is a horizontal command whose meaning is: Add me to the current horizontal list or, if there is no current H list, start one with me as the first item.

TEX is assembling a vertical list when it is in V mode (between paragraphs) or in IV mode (building a `\vbox`). The items of such a list are stacked vertically, top to bottom, and must be V mode material [110]. In V mode, a list is terminated when TEX sees an inherently horizontal command, such as an `\hskip` or the first character of the next paragraph. In IV mode, TEX terminates a list when it finds the '`}`' of the `\vbox`. If it finds inherently horizontal material in this mode, it issues an error. Examples of vertical commands are `\hrule`, `\halign` & `\end` (see [286] for the complete list).

Kern is an interesting example of an item that may appear in either horizontal or vertical lists. Even though the same command, `\kern`, is used, it has different meanings in those lists. Kern is essentially rigid glue with the difference that TEX does not break a line or a page at a kern. Thus if two boxes are separated by a kern, they will remain tied. Of course, if the kern is followed by glue, a break is possible at the glue.

## — 19. Whatsits —

A whatsit is an item that may appear in either a horizontal or a vertical list. It has no dimensions and signifies an operation that should be delayed. The paragraph builder and page builder scan lists submitted to them and execute certain whatsits. There are three types of whatsits:

▪ `\special`. This command requires two arguments. They are first stored in the MVL, and end up being written, at `\shipout` time, to the DVI file. They are interpreted and executed by the printer driver. Individual printer drivers support different `\special` arguments, so `\special` is an example of a non-compatible command. A document using it may only be printed with certain printer drivers. A typical example is `\special{postscript` ⟨postscript commands⟩`}`. When the printer driver finds this in the DVI file, it sends the ps commands to the printer, so that special printing effects can easily be achieved.

▪ The three non-immediate output commands `\openout`, `\closeout` & `\write`. They are also stored in the MVL and are executed later, at `\shipout` time. The reason for their delayed execution is that they may have to write the page number on an output file, and that number is only known in the output routine.

▪ The `\language` and `\setlanguage` commands [455] also produce whatsits each time the language is switched in the midst of a paragraph (e.g., from Icelandic to Serbo-Croatian). These whatsits are stored in memory with the rest of the current paragraph, while the paragraph text is being read in H mode. When the paragraph builder typesets the paragraph (determines the line breaks), the whatsits are used to select the set of hyphenation rules appropriate for each language.

## — 20. Parameters —

Many numeric values are used by TEX, that a user may want to examine or modify. Those values are, therefore, given names, and are considered *parameters* of TEX (different from macro parameters). They are all listed on [272–275]. A typical example of a parameter is `\hsize`. Its value is a dimension, so it may be used whenever a dimension is necessary, as in `\hbox to\hsize` or `\hskip\hsize`. It may also be assigned a new value by `\hsize=3.5in` (however, the '=' is optional).

The rule is that the value of a parameter is used if its name appears in a context where such a value is needed. The value is changed if the name appears in any other context. A common error is a sentence such as 'The width of a line is normally `\hsize` but,...' When TEX sees `\hsize`, it is in the midst of typesetting our sentence, so it is in a context where it does not need a dimension. It assumes, therefore, that `\hsize` is an attepmt to modify `\hsize`, and reads ahead, expecting the new value. Finding the word 'but' instead, it complains of a missing number. The correct sentence should, of course, be 'The width of a line is normally `\the\hsize` but,....'

## — 21. Macros —

This material is intended for users who rarely use macros but are otherwise experienced. A macro is a list of tokens that's been given a name, typically because it is used a lot in certain documents. For example, if the following is used many times in a document '`\medskip$\bullet$\enskip`', it can be given a name, such as `\section`, and defined as a macro by '`\def\section{\medskip$\bullet$\enskip}`'. After this definition, the macro can be *expanded* by simply saying '`\section`'. To expand a macro means to expand the tokens that constitute it. Those tokens are also called *the replacement text* of the macro, since they replace the macro name during expansion.

However, the feature that makes macros so useful and powerful is the use of parameters. By using parameters, each expansion of the same macro may be different. Our macro above may be extended by adding a parameter, such as the name of the section. After '`\def\section#1{\medskip$\bullet$\enskip{\it#1}}`', each expansion must supply some text that will be typeset as the name of the new section. Thus, e.g., '`\section{Advanced Techniques}`' will expand the tokens of the macro, and replace the parameter `#1` by the argument 'Advanced Techniques'.

Note the difference between *parameters*, which are formal and have no fixed value, and *actual arguments* which are text and commands that TEX can process. The notion of parameters also generalizes the concept of a token. Up until now, a token was either a control sequence or a single character. From now on, a token can also be a macro parameter, something of the form '`#x`', where `x` is one of the digits $1, 2, \ldots, 9$. A macro can have up to 9 parameters and, each time it is expanded, arguments must be supplied to replace each parameter.

Our first macro is now extended by adding another parameter, '`#2`', whose value is the section number. Defining '`\def\section#1#2{\medskip$\bullet$\enskip{\bf#2.\thinspace}{\it#1}}`', each expansion must have the form '`\section{Advanced Techniques}6`'. The second argument, '6', is not enclosed in braces because of the following rule: "The argument of a macro is a single token (the next token in the input stream, except that it cannot be any of the braces), unless it is delimited or enclosed in braces." The first argument 'Advanced Techniques' is long, so braces are used to indicate its boundaries. The second argument is a single character (a token), so no braces are necessary; but what is a delimited argument?

When a macro is defined, each of the parameters `#1`, `#2`, can be followed by any characters (except, of course, '`#`' and '`{`') which are its delimiter. If this is done, then each argument in any of the expansions must be followed by the same characters. Our `\section` macro is now extended to include delimiters
'`\def\section#1;#2.{\medskip$\bullet$\enskip{\bf#2.\thinspace}{\it#1}}`' Each expansion of the new macro must look like '`\section Advanced Techniques;26.`' The first argument is everything up to, but not including, the first ';'. The second argument is everything following the ';' up to the next period. The delimiters themselves are skipped and do not become part of the actual arguments.

A simple example of delimiters is '`\def\test1#1#2. #3\end{...}`'. The first parameter has no delimiter, so it must be either a single token or braced. However, it must be *preceded* by a '`1`'. The second token is delimited by the two characters '`.␣`' (not just a period), and the third one, by the characters `\end`. Since the delimiters are skipped, the `\end` is not expanded (what would happen if it were?) If we now expand '`\test10123. 45\end`', the parameters will be '0', '123', '45'. If the expansion does not provide the right delimiters—such as in '`\test210123. 45\end`', '`\test10123.45\end`'—TEX issues the error message

```
! Use of \test doesn't match its definition.
```

When a parameter is delimited, the argument may contain braces, but they must be balanced. Thus in: `\def\x#1\end{\message{'#1'}} \x 1{2\end3}4\end5`, the argument is '`1{2\end 3}4`'.

Delimited parameters may get complex and confusing (see example on [203]), so the `\message` command may be used to find out the precise value assigned to each parameter
`\def\test1#1#2. #3\end{\message{arg1=#1; arg2=#2; arg3=#3}...}`. Another useful debugging tool is the `\tracingmacros` command, discussed among advanced macro features.

In addition to `\def`, macros can also be defined by `\let`, `\chardef`, and as active characters. It has already been mentioned that an active character is a control sequence whose name is limited to a single character, without even a '\', but whose replacement text can be of any length. The `\chardef` command [44] is, in a sense, the opposite of an active character. It defines a control sequence whose replacement text is limited to just one number (in the range 0–255), but whose name can be any valid cs name.

Thus `\chardef\abc=98` or `` \chardef\abc=`\b `` define `\abc` as a macro whose replacement text is the character code of 'b'. It is equivalent to `\def\abc{\char98}`. Also `\chardef\active=13` defines `\active` as a macro whose value is the number 13.

The `\let` command is still different. Its general form is `\let` ⟨control sequence⟩=⟨token⟩. It defines the control sequence as being identical to the token. Thus defining

```
\def\a{X}
\let\g=\a \def\k{\a}
\g\k
```

produces 'XX'. If we now redefine `\a`, the meaning of `\k` will change (since it was defined by `\def`) but `\g` will not change. Thus `\def\a{*} \g\k` produces 'X*'. The difference between `\def` and `\let` is now clear. `\let\g=\a` creates `\g` as an independent macro that does not change if `\a` is modifed. In contrast, `\def\k{\a}` Creates `\k` as a macro pointing to `\a`, so it is always dependent on `\a`.

## — 22. Formats —

When TEX starts, there are no user-defined macros, and the only commands available are the primitives. There are about 300 of them. The primitive `\def` can be used to define macros, which are then added to the repertoire of available commands. When `\def` is executed in the stomach, it loads the replacement text of the new macro into a special table in memory.

When a large document, such as a book, is developed, many macros may have to be defined. Each time the document is typeset, all the definitions have to be prepared by the stomach and loaded in memory, which may be time consuming. In such a case, it is better to convert the macros into a *format*. A format is a collection of commands (macros and primitives), written in a special way on a file, to facilitate rapid loading.

A complete TEX system includes a program called `INITEX` [39] that is used to install TEX. `INITEX` is like TEX but can also prepare formats and hyphenation tables, and perform other tasks. To prepare a format, `INITEX` should be run, the relevant macros should be defined, and the `\dump` command [283] executed. This command dumps the table containing the macros from memory onto a file. That file is called a format file, and it can later be loaded fast simply be copying its contents directly into memory. A format file is loaded, like any other file, by the `\input` command.

A very useful format, the `plain` format, comes with every TEX implementation. It is written on file `plain.fmt` and it consists of about 600 macros that are described throughout the TEXbook, and are useful for general purpose typesetting. The `plain` format is also listed in [App. B]. Many commonly used commands, such as `\bye` % `\smallskip`, are part of this format.

When macros are developed for a specific document, they are normally used together with the `plain` format. It is easy to create a new format containing all the `plain` macros plus any user-defined ones. See [344] for directions.

TEXtures, a Macintosh TEX implementation, makes it particularly easy to create formats since it can execute the `\dump` command without any need for `INITEX`. It is also easy to load any existing format in TEXtures at the flick of the mouse, without having to execute any commands.

## — 23. References —

**1.** Knuth. D. E, *Computers and Typesetting*, vol. E, Addison-Wesley, 1986.
**2.** Henderson, D, *Outline fonts with* METAFONT, TUGboat **10**(1) April 1989, 36.
**3.** Wujastyk, D, *The many faces of TEX*, TUGboat **9**(2) August 1988, 131.
**4.** Tobin, G. K. M, *The OCLC roman family of fonts*, TUGboat **5**(1) May 1984, 36.
**5.** Billawala, N, *Metamarks: Preliminary Studies for a Pandora's Box of Shapes*, Report STAN-CS-89-1256, Computer Science Department, Stanford University, 1989.
**6.** Siegel, D. R, *The Euler Project at Stanford*, Department of Computer Science, Stanford University, 1985
**7.** Haralambous, Y, *Typesetting Old German*, TUGboat **12**(1) March 1991, 129.
**8.** Fuchs, D, *The Format of TEX's* DVI *Files, Version 1*, TUGboat **2**(2), July 1981, 12.

Do not ye yet understand,
that whatsoever entereth in at the mouth
goeth into the belly,
and is cast out into the draught?
— *Mathew 15:17*

Now there are times when a whole generation is caught . . . between two ages, between two modes of life and thus loses the feeling for itself, for the self-evident, for all morals, for being safe and innocent.
— *Hermann Hesse, Steppenwolf*