TEXnical Programming

Slurp
April, 2023

- **TEX engines and formats: TEX vs LATEX**
- **Typesetting in plain TEX**
- **Macros and primitives**
- **Repeated macros**
- **Lists**

TeX Engines and Formats

TeX refers mainly to three things:

- The TeX language, a typesetting language made up of *primitives* like `\def` and `\hskip`.
- The TeX engine (`tex`), a program for compiling files written in the TeX language.
- The plain TeX format, a bare-bones TeX format.

TeX formats are self-contained macro packages whose intention it is to define macros for the user's use in creating documents.

Some well-known formats are plain TeX, LaTeX, and ConTeXt.

Macro packages are not considered formats since they are not self-contained and their intention is to provide further abstraction on top of an existing format.

The format can be specified during the invocation of the `tex` command on the command line.

TEX engines are programs which compile TEX code into output. Historically this output was originally a `dvi` file, but nowadays most output is a `pdf` file.

Some well-known formats are ε-TEX, pdfTEX, LuaTEX, and XƎTEX.

Engines tend to add primitives to the base TEX language: for example, ε-TEX adds the `\numexpr` and related primitives, pdfTEX adds the `\pdfliteral` primitive among others.

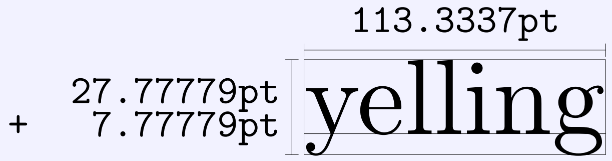
Note that invoking the program engine in the command line by itself also specifies the format: `pdftex` runs the pdfTEXengine with the plain TEX format, while `pdflatex` runs the pdfTEXengine with the LATEX format.

Questions?

Typesetting in Plain TeX

TEX creates its pages by creating a vertical list and filling it with boxes, glue, leaders, penalties, kerns, etc. There are two types of boxes, horizontal boxes and vertical boxes.

Boxes have width, height, and depth. The height of a box is visually its positive vertical displacement relative to the baseline, and its depth is its negative vertical displacement relative to the baseline. For example:



`\hbox`

`\hbox to/spread` ⟨*dimension*⟩`{`⟨*horizontal material*⟩`}`: Creates a horizontal box, if *to dimension* is specified then the width of the box is *dimension*.

If *spread dimension* is specified then the width of the box is *dimension* more than its natural dimension.

The height and depth of the box is equal to the height and depth of its contents.

`\vbox`

`\vbox to/spread` ⟨*dimension*⟩`{`⟨*horizontal material*⟩`}`: Creates a vertical box, if *to dimension* is specified then the height of the box is *dimension*.

If *spread dimension* is specified then the height of the box is *dimension* more than its natural dimension.

The width of the box is the width of its contents, and the depth of the box is the depth of the final box in it.

The box's baseline is the same as that of the final box inside, to align to the top box, use `\vtop`.

Between two boxes there is something called *glue*, which connects the boxes. Glue is one type of blank space (the other being a kern).

Glue has three attributes: its natural length, its maximum stretchiness, and its maximum shrinkage. Glue stretches and shrinks only when it needs to, and TEX uses these attributes in order to fit material into widths the material couldn't properly fit into in its natural width.

For example:

```
\hbox to 5cm{hello\hskip 3cm plus 2cm}there
```

Creates

hello there

Without the `plus2cm` we'd get the same output but with an overfull hbox warning.

The amount of stretchiness and shrinkage can be infinite.

All the dimensions may be negative as well.

`\hskip`	`\hskip` ⟨*natural length*⟩ `plus` ⟨*stretch*⟩ `minus` ⟨*shrink*⟩: Adds horizontal glue with the specified natural length, maximal stretch and shrinkage. The stretch and shrink are optional.
`\vskip`	`\vskip` ⟨*natural length*⟩ `plus` ⟨*stretch*⟩ `minus` ⟨*shrink*⟩: Adds vertical glue with the specified natural length, maximal stretch and shrinkage. The stretch and shrink are optional.
`\kern`	`\kern` ⟨*dimension*⟩: Adds a kern whose dimension is *dimension*. Kerns, unlike glue are nonbreaking, nonstretching, and nonshrinking. The orientation of the kern (horizontal or vertical) is inferred by the context.

TEX has `3` orders of infinities for glue stretching:

`fil` First order `fil`: `\hskip 0pt plus 1fil\relax` creates glue which has no natural length but has infinite stretchiness. A primitive version, `\hfil`, exists as well in place of the code above.

`fill` Second order `fill`: `\hskip 0pt plus 1fill\relax` creates glue which also has no natural length and infinite stretchiness. It takes precedent over first order infinities. A primitive version, `\hfill`, exists as well.

`filll` Third order `filll`: Same as the other two, but takes precedent over both of them. No primitive version exists.

Vertical versions of `\hfil` and `\hfill` exist, `\vfil` and `\vfill`.

Another important primitive is `\hss` which can both shrink and stretch infinitely. It is analogous to `\hskip 0pt plus 1fil minus 1fil`.

It too has a vertical version `\vss`.

```
\def\line{\hbox to \hsize}
\def\centerline#1{\line{\hfil#1\hfil}
\def\rightline #1{\line{\hfil#1}}
\def\leftline  #1{\line{#1\hfil}
\def\rlap#1{\hbox to 0pt{#1\hss}}
\def\llap#1{\hbox to 0pt{\hss#1}}
```

`\line` creates a box which spans the entire line.
`\centerline` centers input relative to the line.
`\rightline` and `\leftline` right and left-justify input respectively.
`\rlap` typesets input and then seems to move back as if it hadn't been typeset.
`\llap` moves back the width of its material and then typesets it.

```
\centerline{Centered Text}
\rightline{Right-Justified}
\leftline{Left-Justified}

\quitvmode\llap{outside}\hfill1\llap{0} and \rlap{1}0
   \hfill\rlap{outside}
```

```
                        Centered Text
                                            Right-Justified
   Left-Justified

outside                   0 and 0                            outside
```

`\hrule`

`\hrule width` ⟨*dimen*⟩ `height` ⟨*dimen*⟩ `depth` ⟨*dimen*⟩

Creates a horizontal rule (a horizontal line). All of the dimensions are optional. If the width is not specified, then the width spans the width of the smallest box enclosing the rule. If the height is not specified, it is by default `0.4`pt. If the depth is not specified, it is by default `0`pt.

This must be used in vertical mode (not horizontal mode, it is a horizontal rule because it is generally used to make horizontal lines in vertical mode).

`\vrule`

`\vrule width` ⟨*dimen*⟩ `height` ⟨*dimen*⟩ `depth` ⟨*dimen*⟩

Creates a vertical rule (a vertical line). All of the dimensions are optional. If the width is not specified, it is by default `0.4`pt. If the height or the depth are not specified, they span the height or depth of the smallest box enclosing the rule.

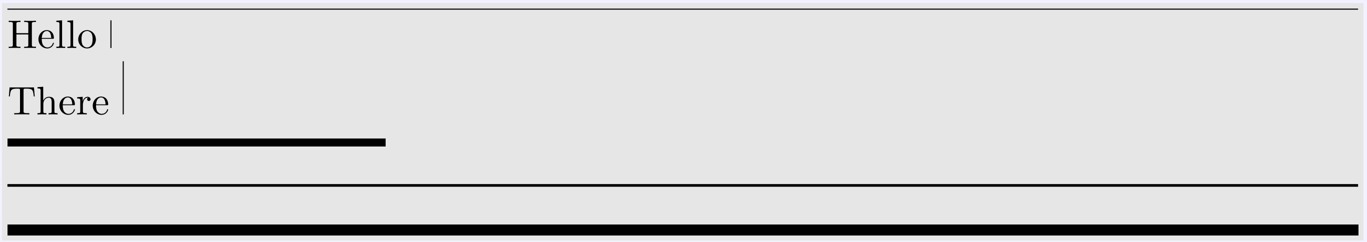
This must be used in vertical mode.

```
\hrule
Hello \vrule

There \vrule \vbox to 20pt{}

\vbox{\line{}\hrule height 3pt}

\vskip.5cm
\hrule height 1pt
\vskip.5cm
\hrule height 2pt
\hrule height 2pt
```



```
Hello

There
```

```
\def\boxed#1{\vbox{\hrule \hbox{\vrule #1\vrule}\hrule}}
\def\badbox#1{\hbox{\vrule \vbox{\hrule #1\hrule}\vrule}}

\boxed{Typesetting in plain \TeX}

\vskip.5cm
\badbox{Typesetting in plain \TeX}
```

Typesetting in plain TEX

Typesetting in plain TEX

```
\def\spas{3pt}
\def\spacebox#1{%
    \hbox{%
        \vrule%
        \vbox{%
            \hrule \vskip\spas%
            \hbox{\hskip\spas #1\hskip\spas}%
            \vskip\spas \hrule%
        }%
        \vrule%
    }%
}

\spacebox{Typesetting in plain \TeX}
```

$\boxed{\text{Typesetting in plain }\TeX}$

TEX has box registers which you can use to store boxes. The number of box registers depends on which engine you use (the original TEX engine had `256` from `\box0` to `\box255`).

`\newbox`	`\newbox`⟨*control sequence*⟩: Allocates a box and sets *control sequence* to reference this box.
`\setbox`	`\setbox`⟨*box number*⟩ `=` ⟨*box*⟩: Sets the *box number*-th box register to be equal to `box`.
`\box`	`\box`⟨*box number*⟩: uses the *box number* box register. After using this, the box register is emptied.
`\copy`	`\copy`⟨*box number*⟩: like `\box` but the register is not emptied after its use.
`\unhbox`	`\unhbox`⟨*box number*⟩: uses the *box number* box register and removes a level of boxing.

```
\setbox0=\hbox{A} \setbox1=\hbox{\unhbox0 B}
```

Makes `\box1` equal to `\hbox{AB}`.

The box register must be a horizontal box (`\hbox`). For vertical boxes, use `\unvbox`. There also exist `\unhcopy` and `\unvcopy`.

You can access the width, height, and depth of a box register using `\wd`, `\ht`, and `\dp`. The use being `\wd`⟨*box register*⟩.

In order to print a TEX variable (like the dimensions of a box), you must prepend it with the primitive `\the`.

```
\setbox0=\hbox{Typesetting in plain \TeX}
Width: \the\wd0, Height: \the\ht0, Depth: \the\dp0
```

```
Width: 165.91646pt, Height: 10.41666pt, Depth: 3.22916pt
```

Also, notice that something like

```
\hbox{A}B
```

Gives

```
A
B
```

This is because when TEX boxes `\hbox{A}` it is in vertical mode, and so the box is placed into the vertical list. Then when it reads `B`, it enters horizontal mode, then boxes the horizontal material (which is just the `B` here) and places that into the vertical list.

So the vertical list looks like

```
\vbox{
    \hbox{A}
    \hbox{B}
}
```

So what we'd want to do is exit vertical mode before the `\hbox`, so its added to the horizontal list instead of the vertical one. Ie. we want the vertical list to look like:

```
\vbox{
    \hbox{\hbox{A}B}
}
```

In order to enter horizontal mode, we need to add horizontal material.

We can do this by unhboxing a void register, since `\unhbox` adds horizontal material.

TEX provides a box register which is meant to be always void, `\voidb@x`, so you can do

```
\unhbox\voidb@x\hbox{A}B
```

```
AB
```

TEX also provides the macro `\leavevmode` which is short for this.

pdfTEX provides the primitive `\quitvmode` which achieves the same thing.

```
\def\presentbox#1{{%
    \setbox0=\hbox{#1}%
    \hbox{%
        \vbox{\tt%
            \hbox{Width: \the\wd0}%
            \hbox{Height: \the\ht0}%
            \hbox{Depth: \the\dp0}%
        }%
        \quad\boxed{%
            \rlap{#1}%
            \vrule width\wd0 height .5pt depth 0pt%
        }%
    }%
}}

\presentbox{Typesetting in plain \TeX}
```

```
Width:   165.91646pt
Height:   10.41666pt
Depth:   3.22916pt      Typesetting in plain TEX
```

```
\def\centersym#1#2{\quitvmode{%
   \setbox0=\hbox{#1}%
   \rlap{#1}\hbox to \wd0{\hss#2\hss}%
}}

\centersym{$\bigcup$}{$\cdot$}
```

⊍

Leaders are a generalization of TEX's concept of glue. The purpose of leaders are to *lead* your eyes across the page.

This . . . . is an example of leaders.

The general use of leaders is

`\leaders`⟨*box or rule*⟩⟨*glue*⟩

For example the above leaders was created by

```
\leaders\hbox to 1em{\hss.\hss}\hfill
```

Note that if a line ends with leaders or glue, it is removed (it doesn't really make sense to end a line with a blank space). So the following code:

```
\quitvmode\leaders\hbox to 1em{\hss.\hss}\hfill

Next line
```

yields:

```
Next line
```

There is no leaders here because the paragraph ends with leaders. In order to get around this you can simply place an empty box, or an empty kern after the leaders.

Notice the `\quitvmode` before the
`\leaders`, this is necessary because `\leaders` works in vertical mode as well, but its glue must be vertical glue (ie `\vskip`, `\vfill`, etc).

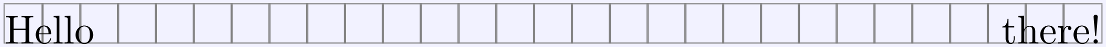
`\leaders`

`\leaders` places leaders aligned with (what seems like) an infinite grid of boxes (either aligned with a row or column of the grid, depending on the horizontal/vertical context).

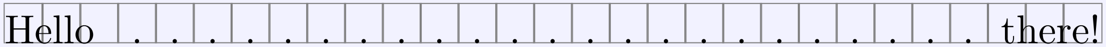
This grid of boxes has the width/height of the box/rule in the leaders, and starts left aligned with the smallest enclosing box of the material.
`\leaders` starts placing the leaders at the first box fully enclosed in the available space, and continues to the last available box (which is placed according to the input glue)

So for example doing something like

```
Hello\leaders\hbox to.5cm{\hss.\hss}\hfill there
  !
```



So the first box is placed in the 4th box (the first box which isn't occupied), and the last in the 4th to last (the last box which isn't occupied).



`\cleaders` The process here is simpler, the boxes or rules are all packed tightly next to each other and an equal amount of glue is placed on either ends of the leaders:

leaders : . . . . end
cleaders: . . . . end

`\xleaders` The boxes or rules here are all spaced equally apart. If there are q boxes/rules, then an equal amount of glue is placed in the $q + 1$ spots between/around them.

leaders : . . . . end
xleaders: . . . . end

Notice that while the placement of the boxes with `\leaders` is independent of the surrounding material, so multiple `\leaders` will have aligned boxes:

Typesetting in plain TeX . . . .

Typesetting in plain . . . . TeX

Typesetting in . . . . plain TeX

Typesetting . . . . in plain TeX

. . . . Typesetting in plain TeX

While `\cleaders` and `\xleaders` both are:

Typesetting in plain TeX . . . .

Typesetting in plain . . . . TeX

Typesetting in . . . . plain TeX

Typesetting . . . . in plain TeX

. . . . Typesetting in plain TeX

An important part of TEX is the concept of *alignment*.

In LATEX this takes the form of environments like `tabular` and `array`, but alignment is even more powerful.

The main TEX primitive which allows for alignment is `\halign`.

The input of `\halign` comes in two parts: the preamble and the actual alignment material. The preamble of an alignment dictates how to align the material, it is simply a list of what material to place around the alignment material.

```
\tabskip=1cm
\halign{#\hfil&\hfil#\hfil&\hfil#\cr
   Left Aligned&Center Aligned&Right Aligned\cr
   Typesetting&in plain&\TeX\cr
}
```

```
Left Aligned     Center Aligned     Right Aligned
Typesetting         in plain                  TEX
```

Notice here that as opposed to LATEX where `\\` delimits the rows in an alignment, `\halign` uses `\cr` (carriage return).

Here the preamble is `#\hfil&\hfil#\hfil&\hfil#`

This means for the first entry of each row, the template is `#\hfil`, this right aligns it. What's significant about this is that the width of each entry is the maximum width of the all the entries in the column.
Similarly the second and third templates center and left-align their entries respectively.

`\tabskip` is the glue added between every column (as well as before the first and after the last column).

`\tabskip` can also be altered within the preamble of an alignment.

The glue inserted before the first column is equal to the value of `\tabskip` when `\halign` is called, and subsequent changes of `\tabskip` within the preamble of the alignment alter the glue inserted after the current and subsequent columns.

```
\tabskip=0pt
\halign{#\hfil\tabskip=1cm&\hfil#\hfil&\hfil#\tabskip=0pt\cr
   Left Aligned&Center Aligned&Right Aligned\cr
   Typesetting&in plain&\TeX\cr
}
```

```
Left Aligned      Center Aligned      Right Aligned
Typesetting          in plain                   TEX
```

```
    Left Aligned      Center Aligned      Right Aligned
    Typesetting          in plain                   TEX
```

For comparison, above is the output of the previous alignment.

plain TeX provides the macro `\ialign` which sets `\tabskip` to `0` and calls `\halign`.

We can also set the width of `\halign` similar to an `\hbox` via

`\halign to` ⟨*width*⟩`{...}`

Instead of `\cr`, we can use `\crcr` which does the same thing, but if `\crcr` comes after another `\cr`, it does nothing.

Thus we can create a macro which is similar to LaTeX's `align` environment

```
\def\align#1{%
   \tabskip=0pt plus 1fil\relax%
   \halign to \hsize{%
      \hfil$\displaystyle##{}$\tabskip=0pt&%
      $\displaystyle{}##$\hfil\tabskip=0pt plus 1fil\cr%
      #1\crcr%
   }%
}

\align{
   \sum_{n=1}^\infty a_n &= 1\cr
   n! &= \prod_{i=1}^n i\cr
}
```

$$\sum_{n=1}^{\infty} a_n = 1$$

$$n! = \prod_{i=1}^{n} i$$

We can use the `\openup` macro to change the amount of glue added between lines (to "open up" the lines). This is used mostly in the context of alignment.

```
First paragraph

{\openup2\jot\halign{#\cr Hello\cr There\cr}}

Second paragraph

\halign{#\cr Hello\cr There\cr}

Third paragraph
```

```
First paragraph

Hello

There
Second paragraph
Hello
There
Third paragraph
```

`\jot` is simply a dimension (set equal to `3`pt by plain TeX), it is customary to use `\openup` in terms of `\jot`s.

`\openup` is cumulative: `\openup1\jot\openup-1\jot`, has the same effect as if nothing had been done.

`\openup` changes the amount of glue added between lines, so it is necessary to keep its changes local by placing it inside a group (`{...}`).

Tokens inside the preamble of an alignment are not expanded, unless they are preceded by the `\span` primitive, which expands (ex-span-ds) the next token.

`\span` has an entirely different meaning within the alignment material, it takes the place of `&` and merges the two entries into one box whose width matches with the width (including the inter-column glue) of the columns.

Within an alignment entry `\omit` omits the current template, and instead the entry uses the trivial template `#`.

Thus `\span` and `\omit` can be used in conjunction for cells which span multiple columns.
Another useful primitive is `\noalign{`$\langle$*vertical material*$\rangle$`}` which can come after `\cr` and adds *vertical material* in place of interline glue.

```
{\tabskip=0pt
\openup1\jot
\halign{#\hfil\tabskip=5pt&\hfil#\tabskip=0pt\cr
\noalign{\hrule\vskip3pt}
\omit\span\omit\hfil Alignment Example\hfil\cr
\noalign{\vskip3pt\hrule\vskip\lineskip}
Typesetting&in\cr
plain&\TeX\cr}}
```

Alignment Example	
Typesetting	in
plain	TEX

If you add a `&` before a template in an alignment's preamble, it is as if that template and all subsequent templates are repeated infinitely.

So for example the preamble

```
\hfil#&&\hfil#&#\hfil
```

Creates an alignment where the first template right-aligns, so does the next, the one after left-aligns, then right-aligns, then left-aligns, and so on (`RRLRLRL...`)

Finally, there is another alignment primitive `\valign` which does its alignment vertically; instead of aligning by rows, it aligns by columns.

The entries in `\valign` are in vertical mode. It is much less common than `\halign`.

TEXnical Programming

Finally (just kidding, we have a lot more to go), let's bring together some of the concepts we've just covered and revisit our `\spacebox` macro.

We can simplify it using alignment:

```
\def\spas{3pt}
\def\alignedbox#1{%
   \vbox{\offinterlineskip%
      \tabskip=0pt%
      \halign{##\tabskip=\spas&###&##\tabskip=0pt\cr
         \noalign{\hrule}
         \vrule&%
         \tabskip=\spas\valign{##\cr\hbox{#1}\cr}&%
         \vrule\cr
         \noalign{\hrule}
      }%
   }%
}

\alignedbox{Typesetting in plain \TeX}
```

Typesetting in plain TEX

Just kidding, that wasn't simpler.

But alignment is more versatile. Instead of boring rules, we will use leaders to create patterned lines, for example:

```
\def\hdotsline{\xleaders\hbox to 3pt{\hss.\hss}\hfil}
\def\vdotsline{\xleaders\vbox to 3pt{\vss\hbox{.}\vss}\vfil}
\def\dotbox#1{%
    \vbox{\offinterlineskip%
        \tabskip=0pt%
        \halign{##\cr
            \noalign{\hdotsline}
            \valign{##\cr
                \vdotsline\cr
                \hbox{#1}\cr
                \vdotsline\cr
            }\cr
            \noalign{\hdotsline}
        }%
    }%
}

\dotbox{Typesetting in plain \TeX}
```

Typesetting in plain TeX

The primary use of `\valign` here is that it puts its entries in vertical mode whose height is the maximum height in the row.

`\dotbox` doesn't add space around the text, doing so isn't terribly hard though.

TeX utilizes many many registers to control the look of its output. Among them are some registers containing glue and dimensions which controls the space around paragraphs and lines.

`\parindent`	The width of indentation before the first line of a paragraph. Plain TeX sets this to `20pt`. `\parindent` is a dimension.
`\parskip`	The inter-paragraph glue. Plain TeX sets this to `0pt plus 1pt`.
`\leftskip`	The glue added to the left of every line. Plain TeX sets this to `0pt`.
`\rightskip`	The glue added to the right of every line. Plain TeX sets this to `0pt`.

For example, if we'd like to center a paragraph we could set the left and right glue to have infinite stretchability.

```
{\parindent=0pt
\leftskip=0pt plus 1fill \rightskip=0pt plus 1fill\relax
It is not the plan of this essay to discuss the
millennium-old problem of faith and reason. Theory is not
my concern at the moment. I want to instead focus attention
on a human-life situation in which the man of faith as an
individual concrete being, with his cares and hopes,
concerns and needs, joys and sad moments, is entangled.
\par
}
```

It is not the plan of this essay to discuss the millennium-old problem of faith
and reason. Theory is not my concern at the moment. I want to instead
focus attention on a human-life situation in which the man of faith as an
individual concrete being, with his cares and hopes, concerns and needs, joys
and sad moments, is entangled.

Notice that `\par` (which ends the current paragraph) must be in the same group as the paragraph. Otherwise the paragraph would be ended outside the group once `\leftskip` and `\rightskip` have already reverted back to their original values.

TEX also provides registers for further altering the shape of a paragraph with `\hangindent` and `\hangafter`.

`\hangindent` is a dimension which specifies the dimension of the "hanging indentation".

`\hangafter` is a number which specifies which lines will be indented.

Suppose `\hangafter` is n, if $n \geq 0$ then the indented lines are $n + 1$, $n + 2$, and so on until the end of the paragraph.

If $n < 0$ then the indented lines are 1, 2, $\ldots$, $|n|$.

```
{\hangindent=1cm \hangafter=-2
Therefore, whatever I am going to say here has been derived
not from philosophical dialectics, abstract speculation, or
detached impersonal reflections, but from actual situations
and experiences with which I have been confronted. Indeed,
the term ``lecture'' also is, in this context, a misnomer.
\par}
```

Therefore, whatever I am going to say here has been derived not from philosophical dialectics, abstract speculation, or detached impersonal reflections, but from actual situations and experiences with which I have been confronted. Indeed, the term “lecture” also is, in this context, a misnomer.

Questions?

Macros and Primitives

While many modern languages provide some form of functions (or procedures, or subroutines), TEX does not. Instead TEX is a macro-based language.

This makes sense since TEX's purpose is to typeset, not program. A macro is simply something which takes input and swaps it with some output.

Some other languages, like C, also have macros. For example

```
#define MACRO(a) (Input is a)
MACRO(hello)
```

Will create a file containing the line `Input is hello` if the C preprocessor is run on it.
TEX macros are similar: they take input and swap it with some output within the document. For example

```
\def\macro#1{Input is #1}
\macro{hello}
```

Will create output with the line `Input is hello` once it is compiled.

But this is a simple example; TEX macros are far more powerful than C's. In fact, TEX is Turing Complete, meaning any algorithm can be written using TEX.

TEX macros are marked by the use of a *escape character*, which is usually the backslash `\`.

Calling these *macros* is actually incorrect, what follows an escape character is a *control sequence*. There are two types of control sequences: a *control word* which is a sequence of *letters* followed by a non-letter, for example `\sqrt`. And a *control letter* is a control sequence consisting of a single non-letter, for example `\"`.

A critical difference between control words and letters is that TEX will always ignore a space after a control word (can you think of why?) while it will not ignore a space after a control letter.

Some of TEX's control sequences are *primitives*, these are control sequences which are not decomposable into simpler functions. TEX's primitives form the backbone on which the rest of TEX is defined. Some examples of primitives are `\input` and `\hbox`.

You can define your own control sequences, called *macros* using primitives like `\def`. This section will focus on this.

The general usage of the `\def` primitive is:

`\def`⟨*control sequence*⟩⟨*parameter text*⟩`{`⟨*replacement text*⟩`}`

Which defines a control sequence which matches text to *parameter text* and replaces it with the *replacement text*. A simple example is as follows:

```
\def\silly ABC{abc}

\silly ABCDEFG
```

abcDEFG

Within the **`parameter text`** of a macro we can define parameters using the special character **`#`**. A parameter is denoted by **`#N`** where `N` is a number between **`1`** and **`9`**, and the parameters must be ordered sequentially.

The parameter text of a macro is matched lazily, ie. the first instance of a pattern will be matched. For example:

```
\def\sillier A#1C{a(#1)c}

\sillier ABCABC
```

a(B)cABC

When pattern-matching for a macro, TEX will consider groups of tokens contained within braces (`{...}`) to be a single atomic unit, and will not match it or the tokens contained within it to a non-parameter.

If no delimiter follows a parameter in the parameter text, the parameter is whatever the next group of tokens is. That is, if the next tokens are contained within braces: `{...}`, then the whole group is taken as the parameter (without the braces), and if it's a single token then that token is taken.

For example:

```
\def\silliness#1{(#1)}
\def\sillier A#1C{a(#1)c}

\silliness{abc}def  % #1 = abc

\silliness abcdef   % #1 = a

\sillier AB{BC}C    % #1 = B{BC}
```

(abc)def

(a)bcdef

a(BBC)c

A small note to remember is that while:

```
\def\macro#1#2{...}
\def\macro #1#2{...}
```

are equivalent,

```
\def\macro#1#2{...}
\def\macro #1 #2 {...}
```

are not. The second `\macro`'s parameters must be delimited with spaces, while the first's are not.

Before we get to writing some funky macros, let's first discuss a handful of very useful TEX primitives.

`\expandafter`	`\expandafter`⟨*token1*⟩⟨*token2*⟩: TEX ignores *token1* at first, expands *token2* (once) and then places *token1* before the expansion.
`\csname` `\endcsname`	`\csname...\endcsname`: TEX reads the tokens between `\csname` and `\endcsname` and totally expands them. After this expansion, only ASCII must remain and TEX replaces this with a control sequence token whose name is this expansion. If this control sequence is undefined currently, this control sequence token is defined to be `\relax`.
`\string`	`\string`⟨*token*⟩: TEX reads *token* without expanding it, and if it is a control sequence, it converts it to a token list of characters (of category `other` other than spaces) which corresponds to its name, following the escape character token. If *token* is a character, it is preserved.

```
\def\gobble#1{}
\def\strip{\expandafter\gobble\string}

\def\initstring#1#2{%
   \expandafter\edef\csname s@#1 \endcsname{#2}}  % Why is \ex
     pandafter necessary?
\def\printstring#1{\csname s@#1 \endcsname}
\def\appendstring#1#2{%
   \expandafter\edef\csname s@#1 \endcsname{\printstring{#1}
     \printstring{#2}}}

\initstring{first string}{Hello}
\initstring{second string}{ World!}
\appendstring{first string}{second string}
\printstring{first string}
```

Hello World!

As you may be aware, certain characters in TeX are assigned special purposes: curly braces (`{...}`) begin and end groups, dollar signs start and end math mode, tildes (`~`) create non-breaking spaces, etc.

While it may seem like this is the case, it is not. No character is assigned a special purpose, as this implementation would not be robust: there are many times when you may want a character to have a different purpose (eg. verbatim environments). Instead each character is assigned a *category code* (catcode) and each category code has a special purpose.

Thus while certain characters may have special attributes, they are not inherent to the character: they are inherent the catcode assigned to the character.

The following is a list of all category codes, their purpose, and "canonical" examples of characters with those category codes.

0	Escape character (eg. `\`) An escape character denotes the beginning of a control sequence, which is an escape character (catcode `0`) followed by a string of letters (catcode `11`) or a single non-letter.
1+2	Begin (catcode `1`) and end (catcode `2`) group (eg. `{` and `}`) These characters begin and end nested groups, which are used for locality and parameters.
3	Math shift (eg. `$`) This begins or ends math (or display math) mode.
4	Alignment tab (eg. `&`) This is the character used in alignment primitives to begin new tabs.
5	End of line (eg. the carriage return/newline character) When TEX encounters an end-of-line character, depending on its current state it may add a space or end the paragraph or nothing.

6 Parameter (eg. `#`)

This is the character used to denote parameters within macros.

7+8 Superscript (catcode `7`) and subscript (catcode `8`) (eg. `^` and `_`)

These are the characters used to create superscripts and subscripts in math mode.

9 Ignored (eg. NUL byte)

These characters are ignored by TEX.

10 Space (eg. the space character)

Space characters make TEX insert a space if it is not currently skipping spaces.

11 Letter (eg. a. . . z, A. . . Z)

These are the characters which can be used in control words.

12 Other (eg. `@`)

An Other character is a character with no other category code. Its special purpose is that it has no special purpose. It is therefore useful for example when creating verbatim environments.

13 Active (eg. `~`)

These are characters which act like macros, they can be defined with `\def` and `\let`.

14 Comment (eg. `%`)

Everything after this character until the end of the line (inclusive) are ignored.

15 Invalid (eg. DELETE)

These characters are ignored but make TEX print an error message.

Category codes can be accessed and altered using the `\catcode` primitive. `\catcode` should be followed by an integer corresponding to the character code of the character whose category code you'd like to access. TeX uses ASCII codes for its character codes, but some unicode-supporting engines use unicode.

For example since the ASCII value of 'a' is `97`, doing `\catcode97=13` changes 'a' to be an active character.

But remembering ASCII by heart is a pain, and fortunately not necessary. When TeX is looking for a number and encounters a backtick (`) followed by a character or a control sequence of one character, it uses the character code of that character.

So for example we could have done ``\catcode`a=13`` or ``\catcode`\a=13``.

The reason for allowing control sequences as well is because a construct like ``\catcode`\=12`` will not set `\`'s category `12`, rather it will set `=`'s category code. But ``\catcode`\\=12`` will work.

The most common use of catcode manipulation is changing the catcode of `@` (and to a lesser extent, `_`). This is commonly done by package authors to create internal macros which they want to hide from the user. This is done by changing the catcode of `@` to `11` (letter) and creating macros which have `@` in their name. At the end of the package the catcode of `@` is reverted to `12` (other), its original value, and so users cannot use these macros without themselves changing the catcode of `@`.

This is done in LaTeX with the macros `\makeatletter` and `\makeatother`. These essentially expand to `\catcode‘@=11` and `\catcode‘@=12` respectively.

Suppose you have a macro like

```
\def\makeatletter{\catcode`\@=11}
```

While this seems like a fine macro, it is actually problematic. For example suppose we have a macro called `\version@name` which contains a number, say `3`. Take a look at the following code:

```
\makeatletter
\version@name
```

This will actually fail with the error message

```
! Undefined control sequence
l.X \makeatletter\version
                         @num
```

This is because TEX continues reading for numbers after the `11` in `\makeatletter`, and so it comes across `\version@num`. Since the category code of `@` is still `12`, it reads this as the control sequence `\version` followed by `@num`. So it attempts to expand `\version` to see if it contains a number, but since `\version` doesn't exist, it errors.

Now suppose it was called `\versionname` instead. This still errors, but for a different reason:

```
! Invalid code (113), should be in the range 0..15.
```

If we follow the process above, notice that TeX continues reading after the `11`, encounters `\versionnumber` which it expands and finds a `3`. Thus it appends this `3` to the current number it is reading to get `113`. So it attempts to set the category code of `@` to `113`, which is an invalid category code.

We can avoid all of these issues with a slight redefinition to our `\makeatletter`. These issues stem from the fact that TeX continues to scan even after the `11`, so all we need to do is stop it from scanning past the `11`. This can be done by inserting a `\relax` after the `11`. `\relax` is a TeX primitive which is unexpandable but does nothing, so it stops TeX's scan without inserting anything into the stream.

```
\def\makeatletter{\catcode`@=11\relax}
```

In general with constructs where TeX is scanning ahead to look for parameters, it is a good idea to place a `\relax` after the parameter to minimize the risk of issues like this.

Another common issue with catcodes is the issue of *frozen catcodes*. This issue stems from the fact that TEX will freeze catcodes in certain situations, namely when parameterizing tokens and in the definition of a macro (the parameter and replacement text).

Let's look at the following situation: suppose we have some macro called `\m@cro` and the following code:

```
\def\identity#1{#1}
\identity{{\catcode`@=11\relax \m@cro}}

\def\macro{{\catcode`@=11\relax \m@cro}}
\macro
```

The double braces here are in order to keep catcode alterations local.

Both of these calls will fail due to catcode freezing. The first one fails because the catcode of `@` in `\m@cro` is `12` (other) when the parameter is scanned, and so `@`'s catcode is frozen at `12` and so TEX sees `\m@cro` as the control sequence `\m` followed by `@cro`. Since `\m` is not defined, it will error.

Similarly, `\macro` fails because when the definition of `\macro` is read, `@` has catcode `12` (other) and so when TEX tokenizes the replacement text of `\macro`, the catcode of `@` is frozen at `12`.

TeX also allows you to use `^^` to insert ASCII characters by following the `^^` with two lowercase hexadecimal digits. So for example `^^61` inserts the character whose ASCII value is `61` in hex (or `97` in decimal: a)

There is another convention where you follow `^^` with a single character and if that characters internal code is between `64` and `127` then `64` is subtracted from the code. Otherwise `64` is added to the code. So for example `^^M` gives ASCII character `77` − `64` = `13` which is the carriage return character.

```
{
    \catcode`^^0d=12 % Why are comments necessary?
    \catcode`@=11 %
    \gdef\@getline#1#2^^0d{\egroup#1{#2}}% ^^0d = \n
    \gdef\getline#1{\bgroup\catcode`^^0d=12\relax\@getline
      {#1}}%
}

\def\parenthesize#1{(#1)}
\getline\parenthesize Hello there!
```

(Hello there!)

TEX, like any programming language, provides support for conditionals. TEX conditionals have the form

`\ifX`⟨*condition*⟩ `...` `\fi`

where `\ifX` is any one of the TEX conditional primitives. TEX also has support for `\else`s.

`\if`

`\if`⟨*token1*⟩⟨*token2*⟩: `\if` expands whatever comes after it until it finds two unexpandable checks if both tokens have the same character code, that is they represent the same character.

If one of the tokens is a control sequence, if it has been `\let` equal to some non-active character then it is considered to have the same category code and character code as that character (at the time that it was `\let`). Otherwise it is considered to have category code and character code incompatible with normal characters, but comparing two macros will be true.

`\ifcat`

`\ifcat`⟨*token1*⟩⟨*token2*⟩: the same rules apply to `\ifcat` as `\if`, but the category codes are compared instead of the character codes.

In order to supress the expansion of active characters, you must prepend them with `\noexpand`.

`\ifx`

`\ifx`⟨*token1*⟩⟨*token2*⟩: the tokens are *not* expanded. `\ifx` checks that both the category code and the character code of the tokens match if they are characters (or `\let` equal to characters). If the tokens are control sequences, `\ifx` checks that they have the same expansion (if they are primitives then it checks that they are the same primitive).

`\ifnum`

`\ifnum`⟨*num1*⟩⟨*relation*⟩⟨*num2*⟩: compares two integers *num1* and *num2*. *relation* must be either `=`, `<`, or `>` with category code `12`.

The following powerful primitives give us tools to change the flow of expansion.

`\expanded`	`\expanded{`⟨*tokens*⟩`}`: This will totally expand *tokens*, similar to `\edef` but `\expanded` is itself expandable. This is a pdfTEX primitive.
`\unexpanded`	`\expanded{`⟨*tokens*⟩`}`: The expansion is *tokens*, this means that when placed inside an expansion-only environment like `\expanded`, *tokens* are not expanded. This is a ε-TEX primitive.
`\futurelet`	`\futurelet\`⟨*macro*⟩⟨*token1*⟩⟨*token2*⟩: This is effectively the same as `\let\`⟨*macro*⟩`=`⟨*token2*⟩⟨*token1*⟩⟨*token2*⟩ This is very useful when you want to look ahead at the next token in the stream without removing it from the stream.
`\aftergroup`	`\aftergroup`⟨*token*⟩: *token* is inserted after the current group.
`\afterassignment`	`\afterassignment`⟨*token*⟩: *token* is inserted after the next assignment (`\def`, `\let`, a register assignment like `\setbox`, etc).

Let us now show some uses of `\futurelet` and `\afterassignment`. A good example of these primitives are the handy LATEX macros `\@ifnextchar` and `\@ifstar`.
The general uses are as follows:

`\@ifnextchar` `\@ifnextchar`⟨*token1*⟩`{`⟨*true*⟩`}{`⟨*false*⟩`}`⟨*token2*⟩: if *token1* and *token2* are equal then *true* is inserted into the sream (it is run), otherwise *false* is. This is done without removing *token2* from the stream.

`\@ifstar` `\@ifstar{`⟨*true*⟩`}{`⟨*false*⟩`}`⟨*token*⟩: if *token* is an asterisk (`*`), *true* is inserted into the sream (it is run), otherwise *false* is.

As opposed to just running `\@ifnextchar*`, the asterisk is removed in the case that *token* is an asterisk.

A naive implementation of