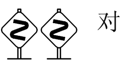
在我们讨论 TEX 的 `\if...` 命令的指令集前, 来看另一个例子, 这样一般思路就清楚了。假定 `\count` 寄存器 `\balance` 存放的是某人付所得税后的余额; 这个量用美分来表示, 并且它可以是正的, 负的或者是零。我们的简单目的就是编写一个宏, 它要为税务机关按照 balance 的值生成一份报告, 此报告要作为通知书的一部分寄给此人。正 balance 的报告与负的差别很大, 因此我们可以用 TEX 的条件文本来做:

```
\def\statement{\ifnum\balance=0 \fullypaid
  \else\ifnum\balance>0 \overpaid
       \else\underpaid
       \fi
  \fi}
```

这里, `\ifnum` 是一个比较两个数的条件命令; 如果 balance 为零, 宏 `\statement` 就只剩下 `\fullypaid`, 等等。



对这个结构中 `0` 后面的空格要特别注意。如果例子中的给出的是

```
...=0\fullypaid...
```

那么 TEX 在得到常数 `0` 的值之前要展开'`\fullypaid`', 因为 `\fullypaid` 可能以 `1` 或其它东西开头而改变这个数字。(毕竟, 在 TEX 看来, '`01`'是完全可以接受的 ⟨number⟩。) 在这种特殊情况下, 程序照样工作, 因为待会我们就可以看到, `\fullypaid` 的开头是字母 `Y`; 因此, 落掉空格后唯一引起的问题就是 TEX 处理变慢, 因为它要跳过的是整个展开的 `\fullypaid`, 而不仅仅是一个未展开的单个记号 `\fullypaid`。但是在其它情形, 象这样落掉空格可能使得 TEX 在你不希望展开宏时把它展开, 并且这样的反常会得到敏感而混乱的错误。要得到最好的结果, *总是要在数值常数后面放一个空格*; 整个空格就告诉 TEX 这个常数是完整的, 而这样的空格从来不会程序在输出中。实际上, 当不在常数后面放空格时, TEX 实际上要做更多的事, 因为每个常数都要持续到读入一个非数字字符为止; 如果这个非数字字符不是空格, TEX 就把你的确有的记号取出来并且备份, 以便下次再读。(另一方面, 当某些其它字符紧跟常数时, 作者常常忽略掉空格, 因为额外的空格在文件中挺难看的; 美感比效率更重要。)

▶ **练习20.10**

继续看看税务机关的例子, 假定 `\fullypaid` 和 `\underpaid` 的定义如下:

```
\def\fullypaid{Your taxes are fully paid---thank you.}
\def\underpaid{{\count0=-\balance
  \ifnum\count0<100
    You owe \dollaramount, but you need not pay it, because
    our policy is to disregard amounts less than \$1.00.
  \else Please remit \dollaramount\ within ten days,
    or additional interest charges will be due.\fi}}
```

按此编写宏 `\overpaid`, 假定 `\dollaramount` 是一个宏, 它按美元和美分输出的 `\count0` 的内容。你的宏应该给出的是: a check will be mailed under separate cover, unless the amount is less than $1.00, in

which case the person must specifically request a check.

▶ **练习20.11**
编写宏 `\dollaramount`, 这样税务机关的 `\statement` 就完整了。

现在, 我们完整地总结一下 TEX 的条件命令。其中有一些本手册还未讨论的东西。

- `\ifnum`⟨number$_1$⟩⟨relation⟩⟨number$_2$⟩ (比较两个整数)

⟨relation⟩ 编写必须是'$<_{12}$', '$=_{12}$'或'$>_{12}$'。两个整数按通常的方法进行比较, 因此所得结果为真或假。

- `\ifdim`⟨dimen$_1$⟩⟨relation⟩⟨dimen$_2$⟩ (比较两个尺寸)

它与 `\ifnum` 类似, 但是比较的是两个 ⟨dimen⟩ 的值。例如, 要检验 `\hsize` 的值是否超过 100 pt, 可以用'`\ifdim\hsize>100pt`'。

- `\ifodd`⟨number⟩ (奇数测试)

如果 ⟨number⟩ 为奇数则真, 为偶数则假

- `\ifvmode` (垂直模式测试)

如果 TEX 处在垂直模式或者内部垂直模式则真(见第十三章)。

- `\ifhmode` (水平模式测试)

如果 TEX 处在水平模式或受限水平模式则真(见第十三章)。

- `\ifmmode` (数学模式测试)

如果 TEX 处在数学模式或列表数学模式则真(见第十三章)。

- `\ifinner` (内部模式测试)

如果 TEX 处在内部垂直模式或受限水平模式或(非列表)数学模式则真(见第十三章)。

- `\if`⟨token$_1$⟩⟨token$_2$⟩ (测试字符代码是否相同)

TEX 将把 `\if` 后面的宏展开为两个不能再展开的记号。如果其中一个记号是控制系列, 那么 TEX 就把它看作字符代码为 256 和类代码为 16, 除非此控制系列的当前内容被 `\let` 为等于非活动字符记号。用这种方法, 每个记号给出一个(字符代码, 类代码)对。如果字符代码相等, 条件成立, 而与类代码无关。例如, 在给出 `\def\a{*}`, `\let\b=*` 和 `\def\c{/}` 后, '`\if*\a`'和'`\if\a\b`'为真, 但是'`\if\a\c`'为假。还有, '`\if\a\par`'为假, 但是'`\if\par\let`'为真。

- `\ifcat`⟨token$_1$⟩⟨token$_2$⟩ (测试类代码是否相同)

它就象 `\if` 那样, 但是测试的是类代码, 而不是字符代码。活动字符的类代码是 13, 但是为了让 `\if` 或 `\ifcat` 在得到这样的字符时强制展开, 必须给出'`\noexpand`⟨active character⟩'。例如, 在

```
\catcode`[=13 \catcode`]=13 \def[{*}
```

之后, 测试'`\ifcat\noexpand[\noexpand]`'和'`\ifcat[*`'为真, 而测试'`\ifcat\noexpand[*`'为假。

- \ifx⟨token$_1$⟩⟨token$_2$⟩ (测试记号是否相同)

在这种情况下, 当 TEX 遇见这两个记号时, 不展开控制系列。如果(a) 两个记号不是宏, 并且它们都标识同一(字符代码, 类代码)对或同一 TEX 原始命令, 同一 \font 或 \chardef 或 \countdef 等等; (b) 两个记号是宏, 并且它们对 \long 和 \outer 都处在相同的状态, 以及它们有同样的测试和"顶级"展开, 那么条件为真。例如, 设置'\def\a{\c} \def\b{\d} \def\c{\e} \def\d{\e} \def\e{A}'后, 在 \ifx 测试中, \c 和 \d 相等, 但是 \a 和 \b, \d 和 \e 不相等, \a, \b, \c, \d, \e 的其它组合也不相等。

- \ifvoid⟨number⟩, \ifhbox⟨number⟩, \ifvbox⟨number⟩ (测试一个盒子寄存器)

⟨number⟩ 应该在 0 和 255 之间。如果 \box 是置空的, 或者包含一个 hbox 或 vbox 时条件为真(见第十五章)。

- \ifeof⟨number⟩ (测试文件是否结束)

⟨number⟩ 一个在 0 和 15 之间。条件为真, 除非相应的输入流是开的并且没有读完。(见下面的命令 \openin。)

- \iftrue, \iffalse (永远为真或为假)

这些条件有预先确定的结果。但是它们却非常有用, 见下面的讨论。

最后, 有一个多条件结构, 它与其它的不同, 因为它有多个分支:

- \ifcase⟨number⟩⟨text for case 0⟩\or⟨text for case 1⟩\or $\cdots$
  \or⟨text for case n⟩\else⟨text for all other cases⟩\fi

在这里, $n+1$ 种情形由 n 个 \or 分开, 其中 n 可以是任意非负数。⟨number⟩ 选择要使用的文本。 \else 部分还是可选的, 如果你不想在 ⟨number⟩ 为负数或大于 n 的情形下给出某些文本的话。

▸ **练习20.12**

设计一个宏 \category, 在其后面输入单个字符后, 它用符号输出此字符的当前类代码。例如, 如果使用 plain TEX 的类代码, '\category\\'将展开为'escape', '\category\a'将展开为'letter'。

▸ **练习20.13**

用下列问题测验一下自己, 看看你是否掌握了这些模糊的情形: 在设置定义'\def\a{} \def\b{**} \def\c{True}'后, 下面哪些是真的? (a) '\if\a\b'; (b) '\ifcat\a\b'; (c) '\ifx\a\b'; (d) '\if\c'; (e) '\ifcat\c'; (f) '\ifx\ifx\ifx'. (g) '\if\ifx\a\b\c\else\if\a\b\c\fi\fi'.

注意, 所有的条件控制系列都以 \if... 开头, 并且它们都要匹配 \fi。这种 \if... 与 \fi 配对的约定使程序中条件控制系列的嵌套更好分清。\if...\fi 的嵌套与 {...} 的嵌套无关; 因此, 可以在条件控制系列中间开始或结束一个组, 也可以在组中间开始或结束一个条件控制系列。编写宏的大量经验表明, 这种无关性在应用中很重要; 但是如果不仔细也会出现问题。

有时候要把信息从一个宏传到另一个, 而实现它有几种方法: 把它作为一个变量来传递, 把它放在一个寄存器中, 或者定义一个包含此信息的控制系列。例如, 附录 B 中的宏 \hphantom,

\vphantom 和 \phantom 非常相似, 因此作者希望把它们三个的所有部分放在另一个宏 \phant 中。用某种方法来告诉 \phant 所要的是哪类 phantom。第一种方法是定义象下面这样的控制系列 \hph 和 \vph:

```
\def\hphantom{\ph YN} \def\vphantom{\ph NY} \def\phantom{\ph YY}
\def\ph#1#2{\def\hph{#1}\def\vph{#2}\phant}
```

之后 \phant 可测试'\if Y\hph'和'\if Y\vph'。这可以用, 但是有几个更有效的方法; 例如, '\def\hph{#1}' 可以用'\let\hph=#1'来代替, 以避免展开宏。因此, 一个更好的方法是:

```
\def\yes{\if00} \def\no{\if01}
\def\hphantom{\ph\yes\no}...\def\phantom{\ph\yes\yes}
\def\ph#1#2{\let\ifhph=#1\let\ifvph=#2\phant}
```

之后 \phant 可测试'\ifhph'和'\ifvph'。(这种结构出现在 TEX 语言中有 \iftrue 和 \iffalse 之前。) 想法很好, 因此作者就开始把 \yes 和 \no 用在其它情形中。但是接着有一天, 一个复杂的条件控制系列失败了, 因为它把象 \ifhph 这样的测试放在另一个条件文本中了:

```
\if...  \ifhph...\fi  ... \else  ...  \fi
```

能看出问题吗? 当执行最外层条件的 ⟨true text⟩ 时, 所有的都很顺利, 因为 \ifhph 是 \yes 或 \no, 并且它展开为 \if00 或 \if01。但是当跳过 ⟨true text⟩ 时, \ifhph 没有被展开, 因此第一个 \fi 错误地匹配到第一个 \if 上; 很快错误就都出来了。这时 \iftrue 和 \iffalse 就被加进 TEX 语言中, 来代替 \yes 和 \no; 现在, \ifhph 是 \iftrue 或 \iffalse, 因此不管它是否被跳过, TEX 都可正确匹配上 \fi。

为了便于构造 \if..., plain TEX 提供了一个叫 \newif 的宏, 这样在给出'\newif\ifabc'后, 就定义了三个控制系列: \ifabc(测试真假), \abctrue(测试为真)和 \abcfalse (测试为假)。现在, 附录 B 的 \phantom 问题就可以如下解决:

```
\newif\ifhph \newif\ifvph
\def\hphantom{\hphtrue\vphfalse\phant}
```

并且有 \vphantom 和 \phantom 的类似定义。不再需要宏 \ph 了; 还是 \phant 来测试 \ifhph 和 \ifvph。附录 E 中有由 \newif 生成的其它条件文本的例子。新的条件文本开始都设定为假。

注意: 不要在条件文本中给出象'\let\ifabc=\iftrue'这样的东西。如果 TEX 跳过这些命令, 就会认为 \ifabc 和 \iftrue 都要匹配一个 \fi, 因为 \let 没有被执行! 把这样的命令包在宏中, 这样 TEX 只有在不跳过要读入的文本时才能遇见'\if...'。

TEX 有 256 个"记号列寄存器"叫做 \toks0 到 \toks255, 因此记号列可以在不经过 TEX 读入器时很容易地传来传去。还有一个 \toksdef 指令, 使得, 比如,

```
\toksdef\catch=22
```

把 \catch 与 \toks22 等价起来。Plain TEX 提供了一个宏 \newtoks, 由它来分配新的记号列寄存器; 它类似于 \newcount。记号列寄存器的性质就象记号列参数 \everypar, \everyhbox, \output, \errhelp 等

等。为了给记号列参数或寄存器指定新值, 可以使用

⟨token variable⟩={⟨replacement text⟩}
或者 ⟨token variable⟩=⟨token variable⟩

其中 ⟨token variable⟩ 表示一个记号列参数, 或者是由 \toksdef 或 \newtoks 定义的一个控制系列, 或者是一个明确的寄存器名字'\toks⟨number⟩'。

经常使用宏这个便利工具的每个人都会遇到编写的宏出问题的时候。我们已经说过, 为了看看 TEX 是怎样展开宏和它找到的变量是什么, 我们可以设置 \tracingmacros=1。还有另一个有用的方法来看看 TEX 在做什么: 如果设置 \tracingcommands=1, 那么 TEX 将显示出它所执行的每个命令, 就象第十三章那样。还有, 如果设置 \tracingcommands=2, 那么 TEX 将显示所有条件命令及其结果, 以及实际执行或展开的非条件命令。这些诊断信息出现在 log 文件中。如果还设置了 \tracingonline=1, 那么在终端上也可以看到。(顺便说一下, 如果设置 \tracingcommands 大于 2, 那么得到的信息同等于 2 一样。) 类似地, \tracingmacros=2 将跟踪 \output, \everypar, 等等。

要知道宏命令出毛病的一个方法就是用刚才讨论的跟踪方法, 这样就能看到 TEX 每步在做什么。另一种就是掌握宏是怎样展开的; 现在我们来讨论这个规则。

TEX 的咀嚼过程把你的输入变成一个长记号列, 就象第八章讨论的那样; 其消化过程严格按照这个记号列进行。当 TEX 遇见记号列中的控制系列时, 要查找其当前的意思, 并且在某些情况下, 在继续读入之前要把此记号展开为一系列其它记号。展开过程作用的对象是宏和某些其它特殊的原始命令, 象 \number 和我们刚刚讨论过的 \if 这样。但是有时候, 却不进行展开; 例如, 当 TEX 处理一个 \def, 此 \def 的 ⟨control sequence⟩, ⟨parameter text⟩ 和 ⟨replacement text⟩ 不被展开。类似地, \ifx 后面的两个记号也不被展开。本章后面要给出一个完整列表, 在这些情况下不展开记号; 在实在没办法时, 你可以用它作为参考。

现在我们来看看不禁止展开时控制系列的展开情况。这样的控制系列分几种:

- 宏: 当宏被展开时, TEX 首先确定其变量(如果有的话), 就象本章前面讨论的那样。每个变量是一个记号列; 但记号作为变量而看待时, 它们不被展开。接着 TEX 用替换文本代替宏及其变量。

- 条件文本: 当 \if... 被展开时, TEX 读入必要的内容来确定条件的真假; 如果是假, 将跳过前面(保持 \if...\fi 的嵌套)直到找到要结束所跳过文本的 \else, \or 或 \fi。类似地, 当 \else, \or 或 \fi 被展开时, TEX 读入要跳过的任何文本的结尾。条件文本的"展开"是空的。(条件文本总是减少在后面的消化阶段所遇见的记号量, 而宏一般增加记号的量。)

- \number⟨number⟩: 当 TEX 展开 \number 时, 它读入所跟的 ⟨number⟩ (如果需要就展开记号); 最后的展开由此数的小数表示组成, 如果是负的前面要有'-'。

- \romannumeral⟨number⟩: 它与 \number 类似, 但是展开由小写 roman 数字组成。例如, '\romannumeral 1984'得到的是'mcmlxxxiv'。如果数字是零或负, 展开为空。

- \string⟨token⟩: TeX 首先读入 ⟨token⟩ 而不展开。如果控制系列记号出现, 那么它的 \string 展开由控制系列的名字组成(如果控制系列不单单是活动字符, 就把 \escapechar 包括进来作为转义符)。否则, ⟨token⟩ 就是字符记号, 并且其字符代码保持为展开后的结果。

- \jobname: 展开为 TeX 为此进程选定的名字。例如, 如果 TeX 把它的输出放在文件 paper.dvi 和 paper.log 中, 那么 \jobname 就展开为'paper'。

- \fontname⟨font⟩: 展开为对应于所给定字体的外部文件名; 比如, '\fontname\tenrm'将展开为'cmr10'(五个记号)。如果字体所用的不是其设计尺寸, 那么"at size"也出现在展开中。⟨font⟩ 是一个由 \font 定义的标识符; 或是 \textfont⟨number⟩, \scriptfont⟨number⟩ 或 \scriptscriptfont⟨number⟩; 或者是 \font, 它表示当前字体。

- \meaning⟨token⟩: TeX 把它展开为一系列字符, 它们是命令'\let\test=⟨token⟩ \show\test'在你的终端上显示的内容。例如, '\meaning A'一般展开的是'the letter A'; 在'\def\A#1B{\C}'后, '\meaning\A'展开的是'macro:#1B->\C '。

- \csname...\endcsname: 当 TeX 展开 \csname, 它要读入匹配的 \endcsname, 如果需要就展开记号; 在此展开中, 只有字符记号或保留下来。因此, 整个 \csname...\endcsname 文本的"展开"将是单个控制系列记号, 如果它当前没有定义, 则定义为 \relax。

- \expandafter⟨token⟩: TeX 首先读入 \expandafter 紧后面的记号, 而不展开它; 我们称此记号为 t。接着, TeX 读入 t 后面的记号(如果此记号有一个变量, 那么可能有更多的记号), 把它用其展开代替。最后 TeX 把 t 放在此展开的前面。

- \noexpand⟨token⟩: 展开为记号自己; 但是如果此记号是一个按 TeX 的展开规则一般要被展开的控制系列, 那么其含义与'\relax'一样。

- \topmark, \firstmark, \botmark, \splitfirstmark, 和 \splitbotmark: 展开为相应"标记"寄存器中的记号列(见第二十三章)。

- \input⟨file name⟩: 展开为空; 但是 TeX 做好准备从给定文件的读入内容, 之后再在当前文件中继续读入其它记号。

- \endinput: 展开为空。TeX 到达了 \input 行的结尾, 将停止从包含此行的文件中读入。

- \the⟨internal quantity⟩: 展开为一列记号, 它表示某个 TeX 变量的当前值, 讨论见下面。例如, '\the\skip5'将展开为'5.0pt plus 2.0fil'(17 个记号)。

\the 这个有用的命令有很多子情形, 因此我们要同时讨论它们。各种内部数字量都可以被提出来使用:

- \the⟨parameter⟩, 其中 ⟨parameter⟩ 是某个 TeX 整数参数(比如, \the\widowpenalty), 尺寸参数(比如, \the\parindent), 粘连参数(比如, \the\leftskip), 或 muglue 参数(比如, \the\thinmuskip)的名字。

- \the⟨register⟩, 其中 ⟨register⟩ 是某个 TeX 的整数寄存器(比如, \the\count0), 尺寸寄存器(比如, \the\dimen169), 粘连寄存器(比如, \the\skip255), 或 muglue 寄存器(比如, \the\muskip\count2)的名字。

- \the⟨codename⟩⟨8-bit number⟩, 其中, ⟨codename⟩ 表示 \catcode, \mathcode, \lccode, \uccode, \sfcode 或 \delcode。例如, \the\mathcode‘/ 得到的是斜线的当前数学代码(整数)。
- \the⟨special register⟩, 其中, 特殊寄存器为某个整数量 \prevgraf, \deadcycles, \insertpenalties, \inputlineno, \badness 或 \parshape(它只表示 \parshape 的行的数目); 或者是某个尺寸 \pagetotal, \pagegoal, \pagestretch, \pagefilstretch, \pagefillstretch, \pagefilllstretch, \pageshrink, \pagedepth。在水平模式下还可以指向一个特殊整数 \the\spacefactor, 在垂直模式下可用于一个特殊尺寸 \the\prevdepth。
- \the\fontdimen⟨parameter number⟩⟨font⟩, 它得到的是一个尺寸; 例如, 字体的参数 6 为其“em”的值, 因此, ‘\the\fontdimen6\tenrm’得到的是‘10.0pt’(6 个记号)。
- \the\hyphenchar⟨font⟩, \the\skewchar⟨font⟩, 它们得到的是定义给定字体的相应整数值。
- \the\lastpenalty, \the\lastkern, \the\lastskip, 它们得到的是当前列中最后一个项目的 penalty, kerning, 粘连或 muglue 的量, 如果此项目分别是 penalty, kern, 或粘连的话; 否则得到的是‘0’或‘0.0pt’。
- \the⟨defined character⟩, 其中 ⟨defined character⟩ 是一个控制系列, 它已经由 \chardef 或 \mathchardef 给定一个整数值; 结果就是此整数值, 用小数表示。

在到现在为止的所有情况下, \the 得到的结果是一系列 ASCII 字符记号。除了字符代码为 32 的记号(“空格”)的类代码为 10 外, 每个记号的类代码都是 12 (“其它字符”)。同样的类代码规则也适用于由 \number, \romannumeral, \string, \meaning, \jobname 和 \fontname 得到的记号。

在一些情形下, \the 得到的是非字符的记号, 是象 \tenrm 这样的字体标识符, 或者是任意记号列:

- \the⟨font⟩ 得到的是选择给定字体的字体标识符。例如, ‘\the\font’是对应于当前字体的控制系列。
- \the⟨token variable⟩ 得到的是一个变量当前值的记号列。例如, 可以展开‘\the\everypar’和‘\the\toks5’。

TEX 的原始命令‘\showthe’将把在展开定义中‘\the’所得到的东西显示在终端上; 展开前面加上‘> ’, 后面跟上句点。例如, 如果采用 plain TEX 的段落缩进, 那么‘\showthe\parindent’显示的是

```
> 20.0pt.
```

下面是以前说过的可展开的记号不被展开的所有情形的列表。某些情形中含有未讨论过的原始命令, 但是我们最后会给出它们的。在下列情形下展开被禁止:

- 当记号在错误修复期间被删除时(见第六章)。
- 当因为条件文本被忽略而记号被跳过时。
- 当 TEX 正在读入宏的变量时。
- 当 TEX 正在读入由下列定义的控制系列时: \let, \futurelet, \def, \gdef, \edef, \xdef, \chardef, \mathchardef, \countdef, \dimendef, \skipdef, \muskipdef, \toksdef, \read 和 \font。

- 当 TeX 正在读入下列控制系列的变量记号时: `\expandafter`, `\noexpand`, `\string`, `\meaning`, `\let`, `\futurelet`, `\ifx`, `\show`, `\afterassignment`, `\aftergroup`。
- 当 TeX 正在读入的是 `\def`, `\gdef`, `\edef` 或 `\xdef` 的参数文本时。
- 当 TeX 正在读入的是 `\def` 或 `\gdef` 或 `\read` 的替换文本; 或者是象 `\everypar` 或 `\toks0` 这样的记号变量的文本; 或者是 `\uppercase` 或 `\lowercase` 或 `\write` 的记号列。(当 `\write` 的记号列实际输出到一个文件时要被展开。)
- 当 TeX 正在读入对齐的前言时, 但是除了在原始命令 `\span` 的一个记号后, 或者正在读入 `\tabskip` 的 ⟨glue⟩ 时。
- 在数学模式开始的记号 $\$_3$ 紧后面时, 这是为了看看是否后面还跟着一个 $\$_3$。
- 在开始字母常数的记号 $`_{12}$ 的后面。

有时候你会发现自己要定义一个新宏, 它的替换文本由于当前情形而已经被展开了, 而不是简单地逐字复制替换文本。为此, TeX 提供了命令 `\edef` (被展开的定义), 以及 `\xdef` (它等价于 `\global\edef`)。其一般格式与 `\def` 和 `\gdef` 一样, 但是 TeX 盲目地按照上面的展开规则来展开替换文本的记号。例如, 看看下面这个定义:

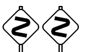
```
\def\double#1{#1#1}
\edef\a{\double{xy}} \edef\a{\double\a}
```

这里, 第一个 `\edef` 等价于'`\def\a{xyxy}`', 而第二个等价于'`\def\a{xyxyxyxy}`'。所有其它类型的展开也要进行, 包括条件文本; 例如, 在 TeX 给出 `\edef` 时处在数学模式的情况下

```
\edef\b#1#2{\ifmmode#1\else#2\fi}
```

的结果等价于'`\def\b#1#2{#1}`', 否则结果等价于'`\def\b#1#2{#2}`'。

由 `\edef` 和 `\xdef` 给出的被展开的定义要把记号展开到只剩下不能展开的记号, 但是由'`\the`'生成的记号列不再进一步展开。还有, '`\noexpand`'后面的记号不展开, 因为它的展开能力无效了。这两个命令可以用来控制展开哪些, 不展开哪些。



例如, 假设你要把 `\a` 定义为 `\b` (展开的)后面跟 `\c` (不展开) 再后面跟 `\d` (展开的), 并且假定 `\b` 和 `\d` 是无参数的简单宏。就可以用两种方法来实现:

```
\edef\a{\b\noexpand\c\d}
\toks0={\c} \edef\a{\b\the\toks0 \d}
```

甚至可以不用 `\noexpand` 或 `\the` 也可得到同样的效果; 对想多学习一些 TeX 展开原理的读者, 建议做一下下面三个练习。

▶ **练习20.14**

不用'`\noexpand`'和'`\the`', 找出定义前一段中 `\a` 的方法。

▶ **练习20.15**

继续讨论避免展开的例子, 假设要把 \b 完全展开到只剩下不能展开的记号, 但是你根本不希望展开 \c, 并且要把 \d 只展开一层。例如, 在定义 \def\b{\c\c}, \def\c{*} 和 \def\d{\b\c} 后, 要得到的是 \def\a{**\c\b\c} 这样的结果。怎样才能用 \the 来实现这个部分展开?

▶ **练习20.16**

不用 \the 或 \noexpand 来解决上一个练习。(这个练习很难。)

TeX 的原始命令 \mark{...}, \message{...}, \errmessage{...}, \special{...} 和 \write⟨number⟩{...} 都展开大括号中的记号列, 同 \edef 和 \xdef 基本一样。但是象 # 这样的宏参数字符在这样的命令中不用重复; 在 \edef 中用 ##, 而在 \mark 中只用 #。命令 \write 有点特殊, 因为它的记号列首先读入而不展开; 直到记号实际上被写入一个文件时才进行展开。

▶ **练习20.17**

比较下面两个定义:

```
\def\a{\iftrue{\else}\fi}
\edef\b{\iftrue{\else}\fi}
```

哪个得到未匹配的左括号? (有点技巧。)

TeX 可以同时从大约 16 个文件中读入各个文本行, 除了在 \input 后的文件之外。为了开始读入这样一个辅助文件, 应该使用

\openin⟨number⟩=⟨file name⟩

其中 ⟨number⟩ 在 0 和 15 之间。(Plain TeX 用命令 \newread 了分配输入流的数字 0 到 15, 它类似于 \newbox。) 在大多数 TeX 的安装时, 如果没有明确给出扩展名, 那么扩展名'.tex'将添加到文件名之后, 就象用 \input 一样。如果文件找不到, TeX 不给出错误信息; 它只把输入流看作没有打开, 并且你可以用 \ifeof 来测试这种状态。当不再使用某个文件时, 可以使用

\closein⟨number⟩

并且如果与输入流数字相对应的文件是打开的, 那么它将关闭, 即, 返回其初始状态。为了从一个打开的文件得到输入, 可以使用

\read⟨number⟩to⟨control sequence⟩

并且所定义的控制系列是一个无参数的宏, 其替换文本是从指定文件读入的下一行的内容。此行用第八章的程序按当前类代码转换为一个记号列。如果需要, 再读入其它行, 直到左右大括号的数目相同。 TeX 在要读入的文件结尾暗中添加了一个空行。如果 ⟨number⟩ 不在 0 和 15 之间, 或者如果这样的文件没有打开, 或者文件结束了, 那么输入就来自终端; 如果 ⟨number⟩ 不是负值, 那么 TeX 将给出提示符。如果你不使用 \global\read, 那么宏的定义将是局域的。

例如, 通过命令 \read 以及 \message (它把一个展开的记号列写在终端上和 log 文件中), 可以很容易实现与用户对话:

```
\message{Please type your name:}
\read16 to\myname
\message{Hello, \myname!}
```

在这种情况下, 命令 \read 将写入'\myname='并等待应答; 应答在 log 文件中重复出来。如果'\read16'变成'\read-1', 那么'\myname='就被省略了。

▶ **练习20.18**
刚刚给出的 \myname 例子效果并不好, 因为在行尾的 ⟨return⟩ 被转换为一个空格。看看怎样解决这个小问题?

▶ **练习20.19**
继续看前一个例子, 定义一个叫 \MYNAME 的宏, 它包含所有 \myname 的字母, 但是用的是大写字母。例如, 如果 \myname 展开为 Arthur, 那么 \MYNAME 展开就是 ARTHUR。假定在 \myname 的展开中只包含字母和空格。

附录 B, D, E 包含大量编写的宏的例子, 可以做很多事情。现在, 在本章结尾, 我们通过几个例子来说明作为编程语言, TEX 实际上可以怎样使用, 如果你要得到某些特殊的效果, 并且不在意计算机所耗的时间。

Plain TEX 中含有一个 \loop...\repeat 结构, 它象这样工作: 给出'\loop α \if... β \repeat', 其中 α 和 β 是任意系列的命令, 并且 \if... 是任意条件测试(无匹配的 \fi)。TEX 就首先执行 α; 接着如果条件为真, TEX 就执行 β, 并且再次从 α 开始重复整个过程。如果条件为假, 循环就停止。例如, 下面有一个小程序, 进行一段对话, 其中 TEX 等待用户输入'Yes'或'No':

```
\def\yes{Yes } \def\no{No } \newif\ifgarbage
\loop\message{Are you happy? }
  \read-1 to\answer
  \ifx\answer\yes\garbagefalse % the answer is Yes
    \else\ifx\answer\no\garbagefalse % the answer is No
      \else\garbagetrue\fi\fi % the answer is garbage
  \ifgarbage\message{(Please type Yes or No.)}
\repeat
```

▶ **练习20.20**
用 \loop...\repeat 原理了编写一个一般的 \punishment 宏, 它重复任意给定段落任意给定次。例如,

```
\punishment{I must not talk in class.}{100}
```

将得到练习 20.1 所要的结果。