
 在我们讨论  $\text{T}_{\text{E}}\text{X}$  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  $\text{T}_{\text{E}}\text{X}$  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  $\text{T}_{\text{E}}\text{X}$  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  $\text{T}_{\text{E}}\text{X}$  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  $\text{T}_{\text{E}}\text{X}$  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  $\text{T}_{\text{E}}\text{X}$  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  $\text{T}_{\text{E}}\text{X}$  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,  $\text{T}_{\text{E}}\text{X}$  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,  $\text{T}_{\text{E}}\text{X}$  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  $\text{T}_\text{E}\text{X}$  的条件命令。其中有一些本手册还未讨论的东西。

- `\ifnum<number1><relation><number2>` (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<dimen1><relation><dimen2>` (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 100pt, 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  $\text{T}_\text{E}\text{X}$  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  $\text{T}_\text{E}\text{X}$  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  $\text{T}_\text{E}\text{X}$  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  $\text{T}_\text{E}\text{X}$  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<token1><token2>` (测试字符代码是否相同)

$\text{T}_\text{E}\text{X}$  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  $\text{T}_\text{E}\text{X}$  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\aa{*}`, `\let\b=*` 和 `\def\c{/}` 后, '`\if*\a`' 和 '`\if\aa\b`' 为真, 但是 '`\if\aa\c`' 为假。还有, '`\if\aa\par`' 为假, 但是 '`\if\par\let`' 为真。

- `\ifcat<token1><token2>` (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(token1)<(token2)` (测试记号是否相同)

在这种情况下, 当  $\text{T}_{\text{E}}\text{X}$  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  $\text{T}_{\text{E}}\text{X}$  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\af{c} \def\b{d} \def\cf{e} \def\df{e} \def\ef{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 ...`  
`\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  $\text{T}_{\text{E}}\text{X}$  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\af{ } \def\b{**} \def\cf{True}`后, 下面哪些是真的? (a) `\ifa\b`; (b) `\ifcat\af\b`; (c) `\ifx\af\b`; (d) `\if\cf`; (e) `\ifcat\cf`; (f) `\ifx\ifx\ifx`. (g) `\if\ifx\af\b\cf\else\if\af\b\cf\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; 但是如果不小心也会出现问题。

 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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, T<sub>E</sub>X 都可正确匹配上 `\fi`。

 为了便于构造 `\if...`, plain T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X 只有在不跳过要读入的文本时才能遇见‘`\if...`’。

 T<sub>E</sub>X 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 T<sub>E</sub>X 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 T<sub>E</sub>X 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  $\text{\TeX}$  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  $\text{\TeX}$  在做什么: 如果设置 `\tracingcommands=1`, 那么  $\text{\TeX}$  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  $\text{\TeX}$  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  $\text{\TeX}$  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。

  $\text{\TeX}$  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  $\text{\TeX}$  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  $\text{\TeX}$  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。

 现在让我们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,  $\text{\TeX}$  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  $\text{\TeX}$  用替换文本代替宏及其变量。
- 条件文本: 当 `\if...` 被展开时,  $\text{\TeX}$  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,  $\text{\TeX}$  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  $\text{\TeX}$  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 `'mcmclxxxiv'`。如果数字是零或负, 展开为空。