附录A the-the 函数

有时候，当用 Emacs 写文章的时候，要重复单词——就像在这个句子开头的“you you”一样[⊖]。最经常重复“the”这个单词，因此将用于查找重复单词的函数称为“the-the”函数。

第一步，可以使用下面的正则表达式来查找重复的单词：

```
\\(\\w+[ \t\n]+\\)\\1
```

这个正则表达式，与后接一个或多个空格、制表符以及换行符的单个或多个构词要素字符相匹配。然而，它不能查到在不同行上面的重复单词，因为第一个单词的末尾是一行的结束，面第二个单词的末尾是一个空格，这是不同的。（关于正则表达式的更加详细的信息，请参见本书第12章“正则表达式查询”，《*GNU Emacs* 技术手册》中“正规表达式的句法”一节，以及《*GNU Emacs Lisp* 技术手册》中“正则表达式”一节。）

你可能会想仅仅通过查找重复的构词要素字符来寻找重复的单词，但是这行不通，因为这可能匹配任意两个重复的并非单词的构词字符串——如在“with the”中两次出现了“th”。

另外一个可能的正则表达式，是查找后而跟着非构词字符的重复的构词字符。这里，“\\w+”与一个或者多个构词字符匹配，而“\\W*”则与零个或者多个非构词字符匹配。

```
\\(\\(\\w+\\)\\W*\\)\\1
```

同样，这个正则表达式也不符合我们的要求。

下面介绍一个我使用的正则表达式。这个正则表达式虽然并不完美，但是已经完全够用。其中，“\\b”与空字符串匹配，因为空字符串在一个单词的开始或者末尾出现。“[^@\n\t]+”则与一次或者多次出现的除了 @ 符号、空格、换行符和制表符之外的其他字符匹配。

```
\\b\\([^@ \n\t]+\\)[ \n\t]+\\1\\b
```

也可以写出更加复杂的正则表达式，但是我发现这个正则表达式已经完全够用，因此我就使用它。

下面就是 the-the 函数，我将它放在我的“.emacs”文件中，并与一个方便的全局键绑定在一起：

```
(defun the-the ()
  "Search forward for for a duplicated word."
  (interactive)
  (message "Searching for for duplicated words ...")
  (push-mark)
;; This regexp is not perfect
;; but is fairly good over all:
```

⊖ 这句话的英文原文是“Sometimes when you you write text you duplicate words——as with ‘you you’ near the beginning of this sentence”。

```
    (if (re-search-forward
         "\\b\\([^@ \n\t]+\\)[ \n\t]+\\1\\b" nil 'move)
        (message "Found duplicated word.")
      (message "End of buffer")))

;; Bind `the-the' to  C-c \
(global-set-key "\C-c\\" 'the-the)
```

下面是一个测试文本：

```
one two two three four five
five six seven
```

可以用其他的正则表达式替代上面这个函数中使用的正则表达式，看一看用它们来处理这个测试文本时有什么不同。

附录B kill 环的处理

kill 环是一个列表，这个列表可以通过使用rotate-yank-pointer 函数被转化为一个环。yank和yank-pop命令使用了 rotate-yank-pointer 函数。本附录描述了 rotate-yank-pointer函数，同时也介绍了 yank 和 yank-pop 命令。

B.1 rotate-yank-pointer 函数

rotate-yank-pointer 函数的作用，就是在 kill 环中改变 kill-ring-yank-pointer 变量所指的元素。例如，它能够将原本指向环中第二个元素的kill-ring-yank-pointer 改为指向环中第三个元素。

下面是 rotate-yank-pointer 函数的代码：

```
(defun rotate-yank-pointer (arg)
  "Rotate the yanking point in the kill ring."
  (interactive "p")
  (let ((length (length kill-ring)))

    (if (zerop length)

        ;; then-part
        (error "Kill ring is empty")

      ;; else-part
      (setq kill-ring-yank-pointer
            (nthcdr (% (+ arg
                          (- length
                             (length
                              kill-ring-yank-pointer)))
                       length)
                    kill-ring)))))
```

这个函数看起来很复杂，但是就像平常一样，这个函数是可以被一层一层地拆开而进行分析理解的。首先，来看一看总体结构：

```
(defun rotate-yank-pointer (arg)
  "Rotate the yanking point in the kill ring."
  (interactive "p")
  (let varlist
    body...)
```

这个函数接收一个参量，即 arg。这个函数同时有一个简短的文档说明字符串，而且它是

交互的，其中的“p”意味着函数的参量必须是一个前缀参量，这个参量值是一个数。

函数体是一个 let 表达式。这个 let 表达式本身有一个变量列表和表达式主体。

let 表达式声明了一个变量，这是在这个函数内能够使用的唯一一个变量。这就是 length 变量。这个变量的值等于 kill 环中元素的个数。给这个变量赋值的是 length 函数。（注意，这个函数与变量 length 有相同的名字，但是一个是函数名，一个是变量名，两者是截然不同的。这就像一个说英语的人，可以区分下面两个句子中“ship”一词的不同意思一样：“I must ship this package immediately.”和“I must get aboard the ship immediately.”。）

length 函数给出一个列表中元素的个数，因此 (length kill-ring) 表达式返回 kill 环中元素的个数。

rotate-yank-pointer 函数体

rotate-yank-pointer 函数体是一个 let 表达式，而 let 表达式的主体是一个 if 表达式。

这个 if 表达式的作用，是判断kill环中是否有内容（元素）。如果 kill 环是一个空列表，则 error 函数使整个函数停止求值并在回显区输出一条消息。另一方面，如果 kill 环中有内容（不是一个空列表），函数就执行它的任务。

下面是 if 表达式的 if 部和 then 部：

```
(if (zerop length)                      ; if-part
    (error "Kill ring is empty")        ; then-part
  ...
```

如果kill 环中没有元素，是一个空列表，它的长度值 length 必定是零，这样就会在回显区中输出一条消息：“Kill ring is empty”。这个 if 表达式中使用了 zerop 函数，当它测试的参量的值为零时，这个函数返回“真”。当这个 zerop 函数返回“真”时，if 表达式的 then 部被执行。这个 if 表达式的 then 部是一个以 error函数开始的列表。其中的 error 函数与 message 函数类似，它也在回显区输出一行消息。然而，除了输出一行消息之外，error 函数还使调用它的整个函数停止执行。在这个例子中，这就意味着，如果 kill 环长度为零的话，这个函数的其余部分就不再被求值了。

（就我的观点来看，用 error 作为函数名是有点误导性的，至少对人面言是如此。一个更好的名字可能是“cancel”。当然，你无法指向一个空列表，更不用说使一个指针在一个空列表上来回移动。从计算机的角度严格地说，“error”一词又是正确的。但是，如果仅仅是找出 kill 环是否为空的话，人还是希望尝试这种事情。这是一次探索）。

（即使是在计算机世界里，从人的角度来说，这次探素也并不是一种错误，因此不应当用“error”一词来表示。Emacs 中的代码暗示着，在探索中追求完美的人正在制造错误。这是不好的。即使计算机在完成同样的事情时，如果出现了一个“error”，像“cancel”这样的词也更能体现当时的情况。）

1. if 表达式的 else 部

这个 if 表达式的 else 部，当 kill 环不是一个空列表时，完成 kill-ring-yank-

pointer 的赋值工作。这部分代码是：

```
(setq kill-ring-yank-pointer
      (nthcdr (% (+ arg
                    (- length
                       (length kill-ring-yank-pointer)))
                 length)
              kill-ring)))))
```

这部分需要解释一下。很明显，在这里，用前面介绍的 nthcdr 函数将 kill-ring-yank-pointer 设置成等于 kill 环的 n 次CDR一个值。(参见8.5节，“copy-region-as-kill”。)但是这究竟是如何实现的呢?

在分析这部分代码的细节之前，让我们首先考虑一下 rotate-yank-pointer 函数的作用。

rotate-yank-pointer 函数改变 kill-ring-yank-pointer 的指向。如果 kill-ring-yank-pointer 开始时指向列表的第一个元素，调用一次 rotate-yank-pointer 函数就使它指向第二个元素。如果 kill-ring-yank-pointer 指向的是第二个元素，调用 rotate-yank-pointer 函数一次就使它指向第三个元素（而且，如果 rotate-yank-pointer 被给予一个大于 1 的参量，它就使指针一次跳过多个元素）。

rotate-yank-pointer 函数使用 setq 函数来重置 kill-ring-yank-pointer 指向的位置。如果 kill-ring-yank-pointer 指向 kill 环的第一个元素，那么在最简单的情况下，rotate-yank-pointer 函数必定使它指向第二个元素。换一种方式来说，kill-ring-yank-pointer 必须重置为等于 kill 环的 CDR 的一个值。

即，在这些情况下，

```
(setq kill-ring-yank-pointer
   ("some text" "a different piece of text" "yet more text"))

(setq kill-ring
   ("some text" "a different piece of text" "yet more text"))
```

下面的代码将完成同样的事情：

```
(setq kill-ring-yank-pointer (cdr kill-ring))
```

结果，kill-ring-yank-pointer将是这个样子：

```
kill-ring-yank-pointer
     ⇒ ("a different piece of text" "yet more text"))
```

在讨论的这个函数中，实际的setq 表达式使用 nthcdr 函数来完成这件事情。

就像前面已经看到的（参见7.3节“nthcdr”），nthcdr 函数反复地取一个列表的 CDR——即一个列表的 CDR 的 CDR 的 CDR...。

下面两个表达式产生同样的结果：

```
(setq kill-ring-yank-pointer (cdr kill-ring))

(setq kill-ring-yank-pointer (nthcdr 1 kill-ring))
```

然而，在 rotate-yank-pointer 函数中，nthcdr 函数的第一个参量是一个看起来相当复杂的表达式，这个表达式中有不少数学内容：

```
(% (+ arg
      (- length
         (length kill-ring-yank-pointer)))
   length)
```

如常所示，需要首先分析其中最内层的表达式，然后以常用的方式分析外层的表达式。

最内层的表达式是 (length kill-ring-yank-pointer)。这个表达式计算 kill-ring-yank-pointer的当前长度（记住，kill-ring-yank-pointer 是一个变量的名字，这个变量的值是一个列表）。

对长度的测量是在下面这个表达式中完成的：

```
(- length (length kill-ring-yank-pointer))
```

在这个表达式中，第一个 length 是一个变量，这个变量是 kill 环的长度，它的值在这个函数的开始就用let语句设置好了。（如果这个变量名用 length-of-kill-ring 来表示，就会更加清楚一些。但是，如果通篇阅读整个函数，而不要像现在这样将函数分成一小片一小片来分析，就会发现即使用这么短的一个变量名也是不会混淆的。）

因此，(- length (length kill-ring-yank-pointer)) 给出 kill 环的长度与 kill-ring-yank-pointer 指向的那个列表的长度之间的差值。

要弄清它们是如何在 rotate-yank-pointer 中工作的，让我们从分析当kill-ring-yank-poiter 像 kill-ring 变量那样指向 kill 环的第一个元素时的情况开始，来看一看当 rotate-yank-pointer 用参量 1 调用时的情况。

在这种情况下，变量 length 和表达式 (length kill-ring-yank-pointer) 的值将是相同的，因为变量 length 就是 kill 环的长度，而这时kill-ring-yank-pointer 也指向整个 kill 环。因此，下面这个表达式的值将是零。

```
(- length (length kill-ring-yank-pointer))
```

因为参量 arg 的值将是 1，这意味着下面这个表达式也为 1。

```
(+ arg (- length (length kill-ring-yank-pointer)))
```

最终，nthcdr 接收的参量就是下列表达式的值。

```
(% 1 length)
```

2. %余函数

要理解表达式 (% 1 length)，需要首先理解 %函数。根据这个函数定义的文档说明（这可以通过键入 C-h f % 得到），% 函数返回它的第一个参量被第二个参量除之后的余数。例如，5 被 2 除之后的余数是1。(5中有两个2，余数为1。)

对于不经常进行算术运算的人来说，理解一个小的数能被一个更大的数除并得到余数会很别扭。在刚才使用的例子中，是 5 被 2 除。可以将它反过来，并问 2 被 5 除会怎样？如果会使用分数的话，答案是 2/5 或者0.4。但是如果你只会使用整数，结果就会两样了。很明显，5 不

是 2 的任何整数倍，但是余数是什么？要回答这个问题，看一看小时候熟悉的例子：

- 5 被 5 除得 1，余数是 0；
- 6 被 5 除得 1，余数是 1；
- 7 被 5 除得 1，余数是 2。
- 类似地，10 被 5 除得 2，余数是 0；
- 11 被 5 除得 2，余数是 1；
- 12 被 5 除得 2，余数是 2。

按这样考虑的话，就会得到：

- 0 被 5 除得 0，余数是 0；
- 1 被 5 除得 0，余数是 1；
- 2 被 5 除得 0，余数是 2；

等等。

因此，在这个函数定义中，如果 length 的值是 5，则表达式就相当于

```
(% 1 5)
```

其值是 1（将光标置于表达式之后并键入 C-x C-e 就得到这个结果。确实，1 打印在回显区中）。

3. 在rotate-yank-pointer 中使用%

当kill-ring-yank-pointer 指向 kill 环的开始时，传递给 rotate-yank-pointer 的参量值是 1，则 % 表达式返回1：

```
(- length (length kill-ring-yank-pointer))
     ⇒ 0
```

因此，

```
(+ arg (- length (length kill-ring-yank-pointer)))
     ⇒ 1
```

从而，不管 length 的值是多少，下面的表达式总是返回 1。

```
(% (+ arg (- length (length kill-ring-yank-pointer)))
   length)
     ⇒ 1
```

根据这个表达式的结果，setq kill-ring-yank-pointer 表达式简化为：

```
(setq kill-ring-yank-pointer (nthcdr 1 kill-ring))
```

现在这个表达式就容易理解了。最初指向 kill 环第一个元素的 kill-ring-yank-pointer 变量现在则指向了第二个元素。

很明显，如果传递给 rotate-yank-pointer 的参量值为 2，那么 kill-ring-yank-pointer 被设置为 (nthcdr 2 kill-ring)；对于不同的参量值，都有不同的指向。

类似地，如果 kill-ring-yank-pointer 是从第二个元素开始的，那么它的长度比 kill 环的长度短 1，因此计算得到的余数就基于表达式 (% (+ arg 1) length)。这意味着，如果传递给rotate-yank-pointer的参量值是1，则kill-ring-yank-pointer 就从kill环的第二个元素移动到第三个元素。

4. 指向最后一个元素

最后一个问题是，如果 kill-ring-yank-pointer 指向最后一个元素，会发生什么事情？这时调用 rotate-yank-pointer 函数是否意味着不能从kill 环中得到任何东西呢？答案是否定的。这时发生的事情很复杂，并且也很有用——kill-ring-yank-pointer 指向了 kill 环的第一个元素。

让我们看一看这是怎么一回事。假设，kill 环的长度是 5，传递给 rotate-yank-pointer 的参量值是1。当 kill-ring-yank-pointer 指向 kill 环的最后一个元素的时候，它的长度是 1。代码就变成：

```
(% (+ arg (- length (length kill-ring-yank-pointer))) length)
```

当用各个变量的值来取代这些变量之后，上面这个表达式就是：

```
(% (+ 1 (- 5 1)) 5)
```

这个表达式从最内层的表达式开始，一步一步向外求值后得到：(- 5 1) 的值是 4；(+ 1 4) 的值是 5；5 被 5 除的余数是 0。因此 rotate-yank-pointer 将要完成的就是：

```
(setq kill-ring-yank-pointer (nthcdr 0 kill-ring))
```

这个表达式将使 kill-ring-yank-pointer 指向 kill 环的开始。

因此，连续调用 rotate-yank-pointer 函数的结果就是将 kill-ring-yank-poiner 从指向kill环中的第一个元素开始，一步一步地移动，直到指向最后一个元素；然后再跳回到第一个元素。这也就是为什么 kill 环被称为环的原因，即通过跳回到第一个元素，就好像这个列表没有终点一样！（环就是没有终点的）。

B.2 yank函数

在学习了 rotate-yank-pointer 函数之后，再学习yank 函数代码就相当容易了。这个函数中只有一处有些小技巧，就是计算传递给 rotate-yank-pointer 函数的参量值。

这部分代码是：

```
(defun yank (&optional arg)
  "Reinsert the last stretch of killed text.
More precisely, reinsert the stretch of killed text most
recently killed OR yanked.
With just C-U as argument, same but put point in front
(and mark at end).  With argument n, reinsert the nth
most recently killed stretch of killed text.
See also the command \\[yank-pop]."
(interactive "*P")
(rotate-yank-pointer (if (listp arg) 0
                       (if (eq arg '-) -1
                         (1- arg))))
(push-mark (point))
(insert (car kill-ring-yank-pointer))
(if (consp arg)
    (exchange-point-and-mark)))
```

稍微看一眼这个函数定义的代码，就能够轻易地理解最后几行。这几行的功能是：记录标记的位置；然后 kill-ring-yank-pointer 指向的第一个元素 (CAR) 被插入到缓冲区；再之后，如果传递给函数的参量是 cons ，就交换位点和标记的值以使位点置于插入文本的开始处而不是末尾。这个可选参量在说明文档中有所解释。另外，函数本身是被交互调用的，使用了 "*P" 参量。这意味着它不能在一个只读缓冲区中使用，而且传递给函数的参量是一个未经处理的前缀参量。

1. 传递参量

yank 函数中最困难的部分是理解关于传递给它的参量的有关计算。幸运的是，它并不是初看起来的那么困难。

这部分代码是两个 if 表达式，对这两个（或者其中的一个）if 表达式求值的结果，将产生一个数，并且这个数将成为传递给 rotate-yank-pointer 的参量。

加上注释，函数代码是：

```
(if (listp arg)                          ; if-part
    0                                    ; then-part
  (if (eq arg '-)                        ; else-part, inner if
      -1                                 ; inner if's then-part
    (1- arg))))                          ; inner if's else-part
```

这部分代码由两个 if 表达式组成，其中一个 if 表达式是另外一个if 表达式的 else部。

第一个或者外层的 if 表达式，测试传递给 yank 函数的参量是否是一个列表。很奇特的是，如果不带参量调用 yank 函数，这个测试总将返回“真”——这是因为这时 nil 将被作为可选参量传递给 yank 函数，而（listp nil）总是返回“真”（因为 nil 是一个空列表）。因此，如果没有参量传递给 yank，那么传递给 rotate-yank-pointer 的参量就是零。这意味着，就像我们希望的那样，这个指针不移动，而且 kill-ring-yank-pointer 当初指向的第一个元素被插入到缓冲区中。类似地，如果传递给 yank 的参量是 C-u，这将被读作一个列表，因此传递给 rotate-yank-pointer 函数的参量同样也是零。（C-u 产生一个未经处理的前缀参量（4），这是一个只有单个元素的列表）。同时，在函数的后面部分，这个参量将被读作一个 cons，因此位点将被置于插入文本的开始，标记将被置于插入文本的末尾。（interactive 中的 P 参量就是为这种情况设置的，即当没有提供可选参量或可选参量是 C-u 时提供这些值。）

外层 if 表达式的 then 部，处理没有可选参量或者可选参量是 C-u 的情况，而 else 部处理其他情况。外层 if 表达式的 else 部本身又是另外一个 if 表达式。

内层的 if 表达式测试参量是否是一个负号。（这是通过同时按下 META 和 - 键或者同时按下 ESC 和 - 键得到的。）在这种情况下，就将 -1 作为一个参量传递 给 rotate-yank-pointer 函数。这使 kill-ring-yank-pointer 朝后移动，这正是用户所期望的。

如果内层 if 表达式的真假测试结果为“假”（也就是参量不是一个负号），这个表达式的 else 部被求值。这就是表达式 (1- arg)。由于这两个 if 表达式的存在，因此这种情况只能发生在参量是一个正数或者一个负数的时候（而不仅仅是一个负号）。表达式 (1- arg) 所做的就是对参量值减1，并返回其结果。（1- 函数的作用是从其参量中减去 1。）这意味着，如果传递

给 rotate-yank-pointer 的参量是 1，它就被减至零，这就是说 kill-ring-yank-pointer 指向的第一个元素被插入缓冲区，正像用户所期望的那样。

2. 传递一个负参量

最后，如果传递一个负的参量值给余函数 % 和 nthcdr 函数，会发生什么情况？它们还能正常运转吗？

答案可以通过一个快速测试给出。当（% -1 5）被求值时，就返回一个负值，如果用一个负值调用 nthcdr 函数，它给出的结果就像是用一个零作为第一个参量来调用一样。这可以通过对下面的代码求值得到。

这里“⇒”表示前面代码求值后产生的结果。求值可以以通常方式进行，将光标置于代码之后并键入 C-x C-e（eval-last-sexp）。如果在GNU Emacs的 Info中阅读这份文档，就可以直接这么做。

```
(% -1 5)
     ⇒ -1

(setq animals '(cats dogs elephants))
     ⇒ (cats dogs elephants)

(nthcdr 1 animals)
     ⇒ (dogs elephants)
(nthcdr 0 animals)
     ⇒ (cats dogs elephants)

(nthcdr -1 animals)
     ⇒ (cats dogs elephants)
```

因此，如果一个负号或者一个负的数值被传递给 yank，kill-ring-yank-pointer 就反向移动直到回到列表的开始。然后它停留在那里。当它从列表末尾移动回列表开始时，绕了一圈，它就停下来了，这与其他情况不同。这很有意义，因为你经常要重新粘贴最近剪切的那块文本，但是你通常不会想要粘贴 30 次前删除命令剪切的文本。因此你需要移动到kill环的末尾，但是如果要返回到列表的开始时就无需绕一圈了。

顺便提一下，任何传递给 yank 的数之前如果有一个负号，它都将被当做 -1 处理。这明显地简化了编写程序的工作。你无需朝后一步一步地跳回 kill 环的开始，这也比编写一个函数以确定要朝后移动多少元素简单得多。

B.3 yank-pop函数

理解 yank 函数之后，再学习yank-pop 函数就容易了。为了节省篇幅，这里省略了函数文档，这个函数定义的代码如下：

```
(defun yank-pop (arg)
  (interactive "*p")
  (if (not (eq last-command 'yank))
      (error "Previous command was not a yank"))
  (setq this-command 'yank)
  (let ((before (< (point) (mark))))
```

```
(delete-region (point) (mark))
(rotate-yank-pointer arg)
(set-mark (point))
(insert (car kill-ring-yank-pointer))
(if before (exchange-point-and-mark))))
```

这是一个交互函数，使用了“p”参量，因此前缀参量是经过处理才传递给这个函数的。这个命令仅能在前一个 yank 函数之后使用，否则就产生一个错误消息。这种检查使用了 last-command 函数（这个函数的介绍，参见 8.5节“copy-region-as-kill”。）

其中，let 表达式根据位点在标记之前或者之后来设置变量 before 的值为“真”或者为“假”，然后删除介于位点和标记之间的区域。这个区域就是前一个 yank 命令插入的区域，并且这就是要被替代的文本。下一步，kill-ring-yank-pointer 移动使前面插入过的文本不再被插入。标记被设置到新文本插入的区域的开始，而且 kill-ring-yank-pointer 指向的第一个元素被插入到这个区域。在前一次的 yank 命令执行中，如果位点被置于插入文本之前，现在位点和标记就要交换位置，使位点再一次置于新插入的文本的开始。这个函数的所有工作就是如此。

附录C　带坐标轴的图

坐标轴有助于你理解图形的意义。它们表达比例尺的大小。在较早的一章中（参见第15章，“准备柱型图”），编写了打印图形的代码。这里，编写打印图形的水平和垂直坐标轴以及图形本身的代码。

由于往缓冲区中插入信息是往右下方进行的，因此新的图形打印函数应该首先打印 Y 轴即垂直轴，然后打印图形本身，最后打印 X 轴即水平轴。下面的顺序定下了这个函数的主要内容：

1) 建立代码。

2) 打印 Y 轴。

3) 打印图形。

4) 打印 X 轴。

下面是根据这个函数打印出来的一个完整的图形：

```
10 -
                *
                *  *
                *  **
                *  ***
 5 -       *    *******
         * *** *******
         ************
       ***************
 1 - *****************
     |   |    |    |
     1   5   10   15
```

在这个图形中，垂直轴和水平轴坐标都是用数字表示的。然而，在有些图形中，水平轴坐标是时间，并且用月份来表示更好，如下所示：

```
5 -       *
        * ** *
        *******
      ********** **
1 - ***************
    |     -       |
    Jan  June    Jan
```

确实，只要稍微思考一下，就能够容易地得到不同的垂直轴和水平轴坐标的表示方式。我们的任务变得复杂了。但是复杂孕育着混乱。与其允许这种混乱情况的出现，不如首先选择简单的坐标表示方式，然后再修改或改进它。

基于这些考虑，可以得出用于print-graph函数的下面框架：

```
(defun print-graph (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))
    (print-Y-axis height ... )
    (graph-body-print numbers-list)
    (print-X-axis ... )))
```

下面，将依次解决 print-graph 函数定义的各个部分。

C.1 print-graph 函数的变量列表

在编写 print-graph 函数时，第一个任务就是编写 let 表达式中使用的变量列表（在此将暂时不考虑如何使这个函数成为一个交互函数以及函数定义的说明文档）。

变量列表应当设置几个值。很明显，垂直轴的最高点必须至少是图形的最大高度，这意味着必须得到图形的最大高度这个信息。注意，在 print-graph-body 函数中也需要这个信息。因为没有必要在两个不同的地方两次计算图形高度值，因此应当改变前面已经定义的 print-graph-body 函数，直接利用这里计算出来的图形高度值。

类似地，打印 X 轴的函数和 print-graph-body 函数都需要得到符号的宽度值。可以在这里统一进行这种计算，并改变前面章节中定义的 print-graph-body 函数使之直接利用这里得到的值。

水平坐标轴的长度必须至少与图形一样长。然而，这个信息只有打印水平坐标轴的函数使用，因此无需在变量列表中计算。

基于这些考虑，就可以直接写出 print-graph 函数的 let 表达式中的变量列表：

```
(let ((height (apply 'max numbers-list)) ; First version.
      (symbol-width (length graph-blank)))
```

就像下面我们将看到的一样，这个表达式是不够的。

C.2 print-Y-axis 函数

print-Y-axis 函数的任务是为垂直坐标轴打印坐标，如下所示：

```
10 -


 5 -


 1 -
```

这个函数应当接收图形高度值作为参量，然后应构造并插入适当的数字和标记。

在图中很容易看出，Y 轴坐标应当是什么样的。但是具体说出来并为此编写一个函数定义，就不那么简单了。如果说需要一个数和每隔 5 行需要一个短线来表示垂直坐标轴，也不很正确：在“1” 和“5”之间只有 3 行(第2、3和4行)，但是在 “5”和“10”之间有 4 行(第6、7、8和9行)。更好的说法是需要一个数和一条短线来表示基线（数1），然后在第 5 行和行数为 5 的整数倍的行用一个数和一条短线来表示坐标。

下一个问题，是确定垂直坐标轴应当有多高。假设图形中最高一列的最大高度是 7，那么 Y 轴上的最大坐标应当是 “5-” 并且图形应当凸显在坐标上方吗？或者 Y 轴上的最大坐标应当是“7-”并且表示图形的顶端吗？或者最大坐标应当是“10-”(这是 5 的整数倍，而又刚好超过图形的最大高度)吗？

后一种选择更好。大多数图形是在长方形的区域中打印出来的。长方形的边是以 5 为步进距离的，如5、10、15 等等。但是，一旦决定为垂直坐标轴使用一个步进距离，就会发现在变量列表中计算高度的简单表达式是错误的。这个表达式就是(apply 'max numbers-list)。这个表达式返回精确的高度值，而不是最大值加上与最接近5的整数倍的差值。因此，就需要一个更为复杂的表达式。

就像在别的例子中一样，如果将复杂的问题分解成几个小问题，这个问题就变得简单了。

首先，考虑当图形的最大高度值正好是 5 的整数倍的情况——即当最大高度值正好是 5、10、15 等时的情况。在这种情况下，就可以直接使用它作为 Y 轴的高度值。

确定某个数是否为 5 的整数倍的一个相当简单的方法，是将它除以 5，并检查它是否有余数。如果没有余数，则这个数就是 5 的整数倍。因而，7 除以 5 余 2，因此 7 不是 5 的整数倍。用另外稍微不同的语言来说（这使人回想起小学课堂来），7 中有一个 5，余下 2。然而，10 中有两个 5，没有余数：10 是 5 的整数倍。

C.2.1 题外话：计算余数

在 Lisp 中，计算余数的函数是 %。这个函数返回它的第一个参量被其第二个参量除之后的余数。在 Emacs Lisp 中，无法用 apropos 来找到 % 函数：如果键入 M-x aproposRET remainder RET，不会得到任何相关的函数。了解 % 这个函数存在的唯一方法是阅读一本关于它的图书，比如这份文档，或者阅读Emacs Lisp 源代码。% 函数曾经被用于在附录B中描述的 rotate-yank-pointer 函数代码中。

通过对下面两个表达式求值，就能够体验一下 % 函数：

```
(% 7 5)

(% 10 5)
```

第一个表达式返回 2，而第二个表达式返回零。

要测试返回值是否为零或者是别的什么值，可以使用 zerop 函数。如果这个函数的参量（这个参量必须是一个数）的值是零，则这个函数返回 t。

```
(zerop (% 7 5))
     ⇒ nil
```

```
(zerop (% 10 5))
     ⇒ t
```

因此，如果图形的高度正好被 5 整除，下面的表达式将返回t。

```
(zerop (% height 5))
```

(当然，height 变量的值可以从(apply 'max numbers-list)表达式得到。)

另一个方面，如果 height变量的值不是 5 的整数倍，需要将其重置为比这个值稍大的 5 的整数倍的值。使用一些已经很熟悉的函数就可以直接得到它。首先将 height 变量的值除以 5 以确定其中有多少个 5。例如，12 中有两个 5。如果将这个商加 1，再乘以 5，就将得到最临近的比高度值大而又是 5 的整数倍的数值。12 中有两个 5，加 1 后等于 3，3 乘以 5 等于15，这是比 12 大的 5 的整数倍的数。因此 Lisp 表达式就是：

```
(* (1+ (/ height 5)) 5)
```

例如，如果对下面的表达式求值，其结果就是 15：

```
(* (1+ (/ 12 5)) 5)
```

在所有这些讨论中，都是使用 "5" 作为 Y 坐标轴坐标间距的，但是也可以使用其他的值。为了使程序更通用，应当用一个变量来取代上面的 "5"。我所能想到的关于这个变量的最好的名字，大概就是 Y-axis-label-spacing 了。使用这个变量和 if 表达式，就得到下面的代码：

```
(if (zerop (% height Y-axis-label-spacing))
    height
  ;; else
  (* (1+ (/ height Y-axis-label-spacing))
     Y-axis-label-spacing))
```

如果图形的高度正好是 Y-axis-label-spacing 变量的值的整数倍，这个表达式返回 height 变量本身的值，否则就返回稍高于图形高度又是变量 Y-axis-label-spacing 整数倍的数值。

现在可以将这个表达式放进 print-graph 函数的 let 表达式中（当然首先要设置 Y-axis-labe-spacing 变量的值）。

```
(defvar Y-axis-label-spacing 5
  "Number of lines from one Y axis label to next.")

...
(let* ((height (apply 'max numbers-list))
       (height-of-top-line
        (if (zerop (% height Y-axis-label-spacing))
            height
          ;; else
          (* (1+ (/ height Y-axis-label-spacing))
             Y-axis-label-spacing)))
       (symbol-width (length graph-blank))))
...
```

(注意 let*函数的使用：图形高度的初始值首先由 (apply 'max numbes-list) 表达式计算出来，然后用 height 变量的结果值计算图形高度的最终值。)

C.2.2 构造一个 Y 轴元素

当打印垂直坐标轴时，想要每5行插入像“5-”和“10-”这样的字符串。而且，要求数字和破折号分别对齐，因此短的数字（只有一位的数字）前面要加上空格。例如，如果有些字符串中使用了两位的数字，那么只有一位数字的串必须在数字前面加入一个空格。

为了求出数的长度，要使用 length 函数。但是这个函数只能工作在一个字符串上，不能对一个数字进行操作。因此必须将这个数字转换成一个字符串。这种转换是由 int-to-string 函数实现的。例如，

```
(length (int-to-string 35))
     ⇒ 2

(length (int-to-string 100))
     ⇒ 3
```

除此之外，在每一个坐标中，每一个数后面必须加上一个像破折号“-”这样的字符串，我们将这个字符串称为 Y-axis-tic 标记。这个变量用 defvar 定义：

```
(defvar Y-axis-tic " - "
  "String that follows number in a Y axis label.")
```

Y 轴坐标的长度等于 Y-axis-tic 标记的长度加上图形顶点的高度值的长度之和。

```
(length (concat (int-to-string height) Y-axis-tic)))
```

这个值将由 print-graph 函数在它的变量列表中计算出来，并存放在 full-Y-label-width 变量中，供其他函数使用（注意当初并没有想到要在变量列表中包括这个值）。

要打印一个完整的垂直坐标轴的坐标，就要打印一个数字、一个标记，以及这两者之前可能还要根据数字的长度加上一个或者更多的空格。因此坐标包含三个部分：（可选的）空格、数字和标记符号。有这样几个参量传递给这个函数：特定行的数值，最高一行的宽度值（这是由 print-graph 函数计算的）。

```
(defun Y-axis-element (number full-Y-label-width)
  "Construct a NUMBERed label element.
A numbered element looks like this '  5 - ',
and is padded as needed so all line up with
the element for the largest number."
(let* ((leading-spaces
        (- full-Y-label-width
           (length
            (concat (int-to-string number)
                    Y-axis-tic)))))
  (concat
   (make-string leading-spaces ? )
```

```
    (int-to-string number)
    Y-axis-tic)))
```

这个 Y-axis-element 函数将前导空格(如果有)、数字和标记符号组合起来构成坐标。

前导空格的个数，是这个函数将坐标的实际长度——数字长度与标记符号长度之和——从需要的坐标总长度中减去而得到的。

空格是用 make-string 函数插入到字符串中的，这个函数接收两个参量：第一个参量告诉你字符串的长度应当是多少，第二个参量就是要插入的符号。这个符号是用特殊形式表示的。在这个例子中，就是使用问号后接上一个空格表示的“? ”。关于这个问题的详细资料可以参见《*GNU Emacs Lisp* 技术手册。》

int-to-string 函数被用在连接字符串的表达式中，它将一个数字转换成一个字符串，这个字符串将与前导空格和坐标标记符号连接起来。

C.2.3 创建 Y 坐标轴

前面的函数为构造一个特殊函数提供了全部工具，这个特殊函数的作用是为 Y 坐标轴产生带有数字、空格和标记符号的坐标。

```
(defun Y-axis-column (height width-of-label)
  "Construct list of Y axis labels and blank strings.
For HEIGHT of line above base and WIDTH-OF-LABEL."
  (let (Y-axis)
    (while (> height 1)
      (if (zerop (% height Y-axis-label-spacing))
          ;; Insert label.
          (setq Y-axis
                (cons
                 (Y-axis-element height width-of-label)
                 Y-axis))
        ;; Else, insert blanks.
        (setq Y-axis
              (cons
               (make-string width-of-label ? )
               Y-axis)))
      (setq height (1- height)))
    ;; Insert base line.
    (setq Y-axis
          (cons (Y-axis-element 1 width-of-label) Y-axis))
    (nreverse Y-axis)))
```

在这个函数中，从 height 变量的值开始，反复地减去 1。每减去一次，测试这个值是否是 Y-axis-label-spacing 的整数倍。如果是，就用 Y-axis-element 函数构造一个带数字的坐标；如果不是，就用 make-string 函数构造一个空白坐标。基线是由一个数字和一个坐标标记符号组成的。

C.2.4 print-Y-axis 函数的最后形式

由 Y-axis-column 函数构造的列表，被传递到 print-Y-axis 函数，后面这个函数将一个列表作为一列插入到缓冲区中：

```
(defun print-Y-axis
  (height full-Y-label-width &optional vertical-step)
  "Insert Y axis using HEIGHT and FULL-Y-LABEL-WIDTH.
Height must be the  maximum height of the graph.
Full width is the width of the highest label element.
Optionally, print according to VERTICAL-STEP."
;; Value of height and full-Y-label-width
;; are passed by 'print-graph'.
(let ((start (point)))
  (insert-rectangle
   (Y-axis-column height full-Y-label-width vertical-step))
  ;; Place point ready for inserting graph.
  (goto-char start)
  ;; Move point forward by value of full-Y-label-width
  (forward-char full-Y-label-width)))
```

print-Y-axis 函数使用 insert-rectangle 函数在一个缓冲区中插入 Y 轴坐标，而这个 Y 轴坐标是由 Y-axis-column 函数创建的。除此之外，它随后将位点置于正确的位置以便打印图形本身。

可以测试 print-Y-axis：

1) 安装

```
Y-axis-label-spacing
Y-axis-tic
Y-axis-element
Y-axis-columnprint-Y-axis
```

2) 拷贝下面的表达式：

```
(print-Y-axis 12 5)
```

3) 切换到“*scratch*”缓冲区，并将光标置于需要开始打印坐标的位置。

4) 键入M-:(eval-expression)

5) 将 graph-body-print 表达式插入到小缓冲区，并键入 C-y (yank)。

6) 按RET键对这个表达式求值。

Emacs 将垂直地打印 Y 坐标轴，最上面的一个坐标是“10 -”(print-graph 函数将传递 height-of-top-line 的值，在这个例子中是 15)。

C.3 print-X-axis 函数

X 坐标轴的坐标与 Y 坐标轴类似，不同的只是其坐标标记在数字上方，像下面这个样子：

```
|   |   |   |
```

```
1   5   10   15
```

第一个坐标标记符号在图形第一列的下方，这个标记符号前面有一些空格。这些空格是为了打印 Y 轴坐标面产生的。第二、第三和第四个标记符号是等距排列的，其间隔根据 X-axis-label-spacing 变量的值确定。

X 轴的第二行是坐标值，第一个数值之前也有空格，各个数值之间的间隔根据X-axis-label-spacing 变量的值确定。

变量 X-axis-label-spacing 的值应当以 symbol-width 为单位计算，因为你可能要改变图形符号的宽度，而不想改变坐标的形式。

print-X-axis 函数与 print-Y-axis 函数多少有些相似之处，只是它需要打印两行：一行是坐标标记符号，一行是坐标值。因此将为这两行分别编写函数来打印它们，然后将它们组合在 print-X-axis 函数中。

这个过程分为三步：

1) 编写一个打印 X 轴坐标标记符号的函数：print-X-axis-tic-line。

2) 编写一个打印 X 轴坐标值的函数：print-X-axis-numbered-line。

3) 编写一个名为 print-X-axis 的函数来打印这两行，这个函数使用 print-X-axis-tic-line 和 print-X-axis-numbered-line 函数。

X 轴标记符号

第一个函数应当打印 X 轴的标记符号。必须定义这个标记符号以及它们之间的间距：

```
(defvar X-axis-label-spacing
  (if (boundp 'graph-blank)
      (* 5 (length graph-blank)) 5)
  "Number of units from one X axis label to next.")
```

(注意，graph-blank 变量的值是由另外一个变量定义表达式 defvar 定义的。boundp 预先检查 graph-blank 变量是否已经设置了初始值；如果没有设置初始值，则 boundp 返回 nil。如果 gragh-blank 变量已经取消了绑定，而又没有使用这个条件表达式，将接收到一个出错消息："Symbol' s value as variable is void"。)

```
(defvar X-axis-tic-symbol "|"
  "String to insert to point to a column in X axis.")
```

定义这个变量是为了打印出如下标记：

```
|   |   |   |
```

第一个坐标标记符号是缩进的，因此它位于图形的第一列之下，之所以要缩进是为了留出空间打印 Y 轴坐标。

一个 X 轴标记符号元素包含从一个标记符号到另外一个标记符号之间的空格以及这个标记符号本身。空格的数目由标记符号本身的宽度和 X-axis-label-spacing 的值决定。

这部分的代码就是：

```
;;; X-axis-tic-element
...
```

```
(concat
 (make-string
  ;; Make a string of blanks.
  (-  (* symbol-width X-axis-label-spacing)
      (length X-axis-tic-symbol))
  ? )
 ;; Concatenate blanks with tic symbol.
 X-axis-tic-symbol)
...
```

随后，需要确定第一个标记符号之前需要缩进多少，以确定最初的空格数。这要使用由 `print-graph` 函数传递来的 `full-Y-label-width` 变量的值。

设置 `X-axis-leading-spacing`变量（缩进空格数）的值的代码是：

```
;; X-axis-leading-spaces
...
(make-string full-Y-label-width ? )
...
```

同时也需要确定水平坐标轴的长度（即数字列表的长度），以及在水平坐标轴上打印的坐标标记的个数：

```
;; X-length
...
(length numbers-list)

;; tic-width
...
(* symbol-width X-axis-label-spacing)

;; number-of-X-tics
(if (zerop (% (X-length tic-width)))
    (/ (X-length tic-width))
  (1+ (/ (X-length tic-width))))
```

有了上面这部分代码，就可以直接写出用于打印 X 轴标记符号行的函数：

```
(defun print-X-axis-tic-line
  (number-of-X-tics X-axis-leading-spaces X-axis-tic-element)
  "Print tics for X axis."
    (insert X-axis-leading-spaces)
    (insert X-axis-tic-symbol)  ; Under first column.
;; Insert second tic in the right spot.
(insert (concat
         (make-string
          (-  (* symbol-width X-axis-label-spacing)
              ;; Insert white space up to second tic symbol.
              (* 2 (length X-axis-tic-symbol)))
          ? )
         X-axis-tic-symbol))
```

```
;; Insert remaining tics.
(while (> number-of-X-tics 1)
  (insert X-axis-tic-element)
  (setq number-of-X-tics (1- number-of-X-tics))))
```

打印坐标数字行的函数也很直接、简单：

首先，创建数字元素，每一个元素都是由一个数字加上其前导空格组成：

```
(defun X-axis-element (number)
  "Construct a numbered X axis element."
  (let ((leading-spaces
         (-  (* symbol-width X-axis-label-spacing)
             (length (int-to-string number)))))
    (concat (make-string leading-spaces ? )
            (int-to-string number))))
```

接下来，创建打印坐标数字行的函数，在图形第一列的下面打印坐标“1”：

```
(defun print-X-axis-numbered-line
  (number-of-X-tics X-axis-leading-spaces)
  "Print line of X-axis numbers"
  (let ((number X-axis-label-spacing))
    (insert X-axis-leading-spaces)
    (insert "1")
    (insert (concat
             (make-string
              ;; Insert white space up to next number.
              (-  (* symbol-width X-axis-label-spacing) 2)
              ? )
             (int-to-string number)))
;; Insert remaining numbers.
(setq number (+ number X-axis-label-spacing))
(while (> number-of-X-tics 1)
  (insert (X-axis-element number))
  (setq number (+ number X-axis-label-spacing))
  (setq number-of-X-tics (1- number-of-X-tics)))))
```

最后，要编写print-X-axis函数，这个函数使用print-X-axis-tic-line 函数和print-X-axis-numbered-line 函数。

这个函数必须确定由 print-X-axis-tic-line 函数和 print-X-axis-numbered-line 函数使用的局部变量的值，然后还必须调用这两个函数。同样，这个函数还要在这两行之间输出一个换行符，以分隔这两行的内容。

这个print-X-axis 函数由一个定义了 5 个局部变量的变量列表以及对上面这两个函数的调用组成。

```
(defun print-X-axis (numbers-list)
  "Print X axis labels to length of NUMBERS-LIST."
  (let* ((leading-spaces
```

```
        (make-string full-Y-label-width ? ))
      ;; symbol-width is provided by graph-body-print
      (tic-width (* symbol-width X-axis-label-spacing))
      (X-length (length numbers-list))
      (X-tic
       (concat
        (make-string
         ;; Make a string of blanks.
         (-  (* symbol-width X-axis-label-spacing)
             (length X-axis-tic-symbol))
         ? )
        ;; Concatenate blanks with tic symbol.
        X-axis-tic-symbol))
      (tic-number
       (if (zerop (% X-length tic-width))
           (/ X-length tic-width)
         (1+ (/ X-length tic-width)))))
  (print-X-axis-tic-line tic-number leading-spaces X-tic)
  (insert "\n")
  (print-X-axis-numbered-line tic-number leading-spaces)))
```

可以这样测试 print-X-axis 函数：

1) 安装 X-axis-tic-symbol、X-axis-label-spacing、print-X-axis-tic-line 以及 X-axis-element、print-X-axis-numbered-line 和 print-X-axis。

2) 复制下面的表达式：

```
(progn
 (let ((full-Y-label-width 5)
       (symbol-width 1))
   (print-X-axis
    '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16))))
```

3) 切换到“*scratch*”缓冲区，将光标置于要绘制坐标轴的开始处。

4) 键入M-:(eval-expression)。

5) 用C-y (yank) 命令将测试表达式粘贴到小缓冲区。

6) 键入RET键来对这个表达式求值。

Emacs 将打印下面这样的水平坐标轴：

```
|   |    |    |    |
1   5   10   15   20
```

C.4 打印整个图形

现在已经准备好打印整个图形。

用于打印图形以及正确的坐标的函数遵循在本附录前面已提出的框架结构，但是还有一些增加的内容。

下面就是这个函数的具体结构：

```
(defun print-graph (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))
    (print-Y-axis height ... )
    (graph-body-print numbers-list)
    (print-X-axis ... )))
```

这个函数的最终结构与计划的结构有两点不同之处：第一，它包含一些在变量列表中计算的变量值；第二，它有一个可选的参量来定义每一坐标的增量。后面这一点是必须的，否则有些图形可能有太多的行以至于无法在一张纸或者在一个屏幕上显示打印出来。

这个新特征需要对 Y-axis-column 函数作一点改变，增加 vertical-step 函数到其中。Y-axis-column 函数最终就是：

```
;;; Final version.
(defun Y-axis-column
  (height width-of-label &optional vertical-step)
  "Construct list of labels for Y axis.
HEIGHT is maximum height of graph.
WIDTH-OF-LABEL is maximum width of label.
VERTICAL-STEP, an option, is a positive integer
that specifies how much a Y axis label increments
for each line.  For example, a step of 5 means
that each line is five units of the graph."
  (let (Y-axis
        (number-per-line (or vertical-step 1)))
    (while (> height 1)
      (if (zerop (% height Y-axis-label-spacing))
          ;; Insert label.
          (setq Y-axis
                (cons
                 (Y-axis-element
                  (* height number-per-line)
                  width-of-label)
                 Y-axis))
        ;; Else, insert blanks.
        (setq Y-axis
              (cons
               (make-string width-of-label ? )
               Y-axis)))
      (setq height (1- height)))
    ;; Insert base line.
    (setq Y-axis (cons (Y-axis-element
                        (or vertical-step 1)
                        width-of-label)
```

```
                        Y-axis))
    (nreverse Y-axis)))
```

图形的最大高度值和打印图形用的符号的宽度，由 print-graph 函数在它的 let 表达式中计算出来，因此 graph-body-print 函数必须作些改变以接收这两个值。

```
;;; Final version.
(defun graph-body-print (numbers-list height symbol-width)
  "Print a bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values.
HEIGHT is maximum height of graph.
SYMBOL-WIDTH is number of each column."
(let (from-position)
  (while numbers-list
    (setq from-position (point))
    (insert-rectangle
     (column-of-graph height (car numbers-list)))
    (goto-char from-position)
    (forward-char symbol-width)
    ;; Draw graph column by column.
    (sit-for 0)
    (setq numbers-list (cdr numbers-list)))
  ;; Place point for X axis labels.
  (forward-line height)
  (insert "\n")))
```

最后，print-graph 函数的代码是：

```
;;; Final version.
(defun print-graph
  (numbers-list &optional vertical-step)
  "Print labelled bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values.

Optionally, VERTICAL-STEP, a positive integer,
specifies how much a Y axis label increments for
each line.  For example, a step of 5 means that
each row is five units."
  (let* ((symbol-width (length graph-blank))
         ;; height is both the largest number
         ;; and the number with the most digits.
         (height (apply 'max numbers-list))
         (height-of-top-line
          (if (zerop (% height Y-axis-label-spacing))
              height
            ;; else
            (* (1+ (/ height Y-axis-label-spacing))
               Y-axis-label-spacing)))
         (vertical-step (or vertical-step 1))
         (full-Y-label-width
```

```
          (length
           (concat
            (int-to-string
             (* height-of-top-line vertical-step))
            Y-axis-tic))))
  (print-Y-axis
   height-of-top-line full-Y-label-width vertical-step)
  (graph-body-print
   numbers-list height-of-top-line symbol-width)
  (print-X-axis numbers-list)))
```

C.4.1 测试 print-graph 函数

可以用一个数字列表来测试 print-graph 函数：

1) 安装最终版本的 Y-axis-column、graph-body-print和print-graph 函数（以及其他另外的代码）。

2) 拷贝下面的表达式：

```
(print-graph '(3 2 5 6 7 5 3 4 6 4 3 2 1))
```

3) 切换到 “*scrarcs*” 缓冲区，将光标置于要绘制坐标轴的开始处。

4) 键入M-:(eval-expression)。

5) 用C-y (yank) 命令将测试表达式粘贴到小缓冲区。

6) 键入RET键来对这个表达式求值。

经过以上几步，Emacs 将打印出下面这样的带坐标的图形来。

```
10 -


          *
         **   *
 5 -    ****  *
        **** ***
      * *********
      ***********
 1 - ************

      |   |    |    |
      1   5   10   15
```

在另一方面，如果给这个 print-graph函数传递一个 vertical-step 的值为 2 的参量，对下面的表达式求值：

```
(print-graph '(3 2 5 6 7 5 3 4 6 4 3 2 1) 2)
```

就会得到下面的这个图形：

（问题：垂直坐标轴的底端上显示的“2”是一个 bug 吗？如果你认为它是一个 bug，并认为应

当在这个位置打印“1”（甚至是“0”），你可以修改函数代码。）

```
20 -


          *
         **    *
10 -    ****   *
         **** ***
       * *********
       ************
 2 -  **************

      |    |     |     |
      1    5    10    15
```

C.4.2 绘制函数中单词和符号数的图形

现在是打印函数定义中单词和符号数的出现次数的时候了：这个图形显示有多少函数定义中有少于 10 个单词和符号，有多少函数定义中有 10~19 个单词和符号，有多少函数定义中有 20~29 个单词和符号，等等。

这是一个多步骤的过程。首先要确保已经加载了所有需要的代码。

如果已经设置过 top-of-ranges 变量的值，最好是重新设置 top-of-ranges 的值。这只要对下面的表达式求值就行了：

```
(setq top-of-ranges
 '(10  20  30  40  50
   60  70  80  90 100
  110 120 130 140 150
  160 170 180 190 200
  210 220 230 240 250
  260 270 280 290 300)
```

然后，创建一个有关每一段中单词和符号的个数的列表。

对下面的表达式求值：

```
(setq list-for-graph
      (defuns-per-range
        (sort
         (recursive-lengths-list-many-files
          (directory-files "/usr/local/emacs/lisp"
                           t ".+el$"))
         '<)
        top-of-ranges))
```

在我的计算机中，这个计算过程需要大约一个小时。它检查我计算机中 19.23版的 Emacs 的所有 303 个 Lisp 文件。经过这个计算之后，list-for-graph 的值是：

```
(537 1027 955 785 594 483 349 292 224 199 166 120 116 99
90 80 67 48 52 45 41 33 28 26 25 20 12 28 11 13 220)
```

从这个列表看出：在我机器中的 Emacs 有 537 个函数定义只有少于 10 个单词和符号；有 1027 个函数定义有 10~19 个单词和符号；有 955 个函数定义有 20~29 个单词或符号，等等。

很明显，仅仅看一看这个列表，也可以看出大多数函数定义包含 10~30 个单词和符号。

现在就可以将它们以图形的方式打印出来。在此不需要打印一个高度是1030 行的图形，相反应当打印一个小于 25 行的图形。这个高度的图形能够在几乎所有的终端上显示，并可以方便地打印到纸上。

这意味着 `list-of-graph` 列表中的每一个值必须缩小 50 倍。

下面就是一个完成这个任务的简短的函数，它使用了两个我们没有用过的函数：`mapcar` 和 `lambda`。

```
(defun one-fiftieth (full-range)
  "Return list, each number one-fiftieth of previous."
 (mapcar '(lambda (arg) (/ arg 50)) full-range))
```

1. `lambda` 表达式

`lambda` 是一个匿名函数的符号，匿名函数就是没有函数名的函数。每当使用一个匿名函数，都必须将它的整个函数定义包含在内。

因此，

```
(lambda (arg) (/ arg 50))
```

这个函数定义就是说："返回 `arg` 除以 50 的结果"。

例如，在前面，有一个称为 `multiply-by-seven`的函数，这个函数将它的参量乘以 7。这个匿名函数与之相似，只是它将参量除以 50。而且这个函数没有函数名。`multiply-by-seven` 函数对应的匿名函数是：

```
(lambda (number) (* 7 number))
```

(参见 3.1节，"`defun`特殊表"。)

如果要将 3 乘以 7，可以这样编写代码：

```
(multiply-by-seven 3)
 \_______________/ ^
         |         |
        函数       参量
```

这个表达式返回 21。

类似地，可以这样做：

```
((lambda (number) (* 7 number)) 3)
 \____________________________/ ^
               |                |
            匿名函数            参量
```

如果我们要将 100 除以 50，可以这样编写表达式：

```
((lambda (arg) (/ arg 50)) 100)
 \---------------------/  \_/
           |               |
       匿名函数          参量
```

这个表达式返回 2。100 被传递给这个匿名函数，这个匿名函数将其参量除以 50。

关于 lambda 的更多的内容，可以参见《*GNU Emacs Lisp*技术手册》中的“lambda 表达式”一节。Lisp 和 lambda 表达式都是从 lambda 微积分中演化出来的。

2. mapcar 函数

mapcar 是一个这样的函数，它依次用其第二个参量中的每一个元素调用第一个参量。第二个参量必须是一个列表。

例如，

```
(mapcar '1+ '(2 4 6))
     ⇒ (3 5 7)
```

函数 1+ 将其参量加1，在上面这个例子中，1+ 函数作用在作为 mapcar的第二个参量的列表的每一个元素上，并产生一个新的列表。

与这个函数形成对照的是，apply 函数将其第一个参量作用在其余参量上。（参见第15章“准备柱型图”中关于 apply 的说明。）

在除以50的函数（one-fiftieth）中，第一个元素是匿名函数：

```
(lambda (arg) (/ arg 50))
```

而第二个参量是 full-range 变量，这个变量将被绑定到 list-for-graph。

因此整个表达式就是：

```
(mapcar '(lambda (arg) (/ arg 50)) full-range))
```

关于 mapcar 函数更详细的说明，可以参见《*GNU Emacs Lisp* 技术手册》中的“映射函数”一节。

使用 one-fiftieth 函数，可以产生一个其中每一个元素都是 list-for-graph 列表中相应元素的 1/50 的列表。

```
(setq fiftieth-list-for-graph
      (one-fiftieth list-for-graph))
```

最后的列表就是：

```
(10 20 19 15 11 9 6 5 4 3 3 2 2
1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 4)
```

这个列表几乎就是要打印的列表了！（我们同样注意到，在这个列表中丢失了一些信息：许多是 0，这些 0 意味着少于 50 个单词或符号的函数定义，而不一定意味着没有单词或符号的函数定义。）

3. 暗藏的另一个bug

在上面已经说过“这个列表几乎就是要打印的列表了！”。当然，在 print-graph 函数中

有一个 bug。这个函数有一个可选的参量 vertical-step，但是没有 horizontal-step 参量。而且列表 top-of-range 中的元素从 10到 300，每一个元素之间的间隔是 10。但是 print-graph 函数将只会每隔1打印一列。

这是一个暗藏的 bug 的典型例子，这个 bug 被忽略了。这不是那种你只阅读代码就可以发现的 bug，因为它并不在代码之中，它是一个忽略了的特性。最好的办法就是尽可能早地、尽可能多地测试你的代码。并尽可能地编写易于理解、易于修改的代码。要尽可能地时刻提醒自己，代码总是要重新编写的。这是一个不错的格言。

在这个例子中，print-X-axis-numbered-line 函数需要重新编写，然后 print-X-axis 和 print-graph 函数也需要修改。但这并不需要作很多的改变。其中一个细节是：X 轴坐标值需要与坐标标记符号一一对齐。这需要好好思考一下。

下面是修改后的 print-X-axis-numbered-line 函数：

```
(defun print-X-axis-numbered-line
  (number-of-X-tics X-axis-leading-spaces
   &optional horizontal-step)
  "Print line of X-axis numbers"
  (let ((number X-axis-label-spacing)
        (horizontal-step (or horizontal-step 1)))
    (insert X-axis-leading-spaces)
    ;; Delete extra leading spaces.
    (delete-char
     (- (1-
         (length (int-to-string horizontal-step)))))
    (insert (concat
             (make-string
              ;; Insert white space.
              (-  (* symbol-width
                     X-axis-label-spacing)
                  (1-
                   (length
                    (int-to-string horizontal-step)))
                  2)
              ? )
             (int-to-string
              (* number horizontal-step))))
    ;; Insert remaining numbers.
    (setq number (+ number X-axis-label-spacing))
    (while (> number-of-X-tics 1)
      (insert (X-axis-element
               (* number horizontal-step)))
      (setq number (+ number X-axis-label-spacing))
      (setq number-of-X-tics (1- number-of-X-tics)))))
```

如果你是在 Info 中阅读这份文档的，你可以看到 print-X-axis 和 print-graph 函

数的新的版本，并可以对它们求值。如果你是在阅读一本打印出来的书，就可以在这里看到这些函数的改变的部分(全部文本太长，以至无法全部打印出来)。

```
(defun print-X-axis (numbers-list horizontal-step)
  ...
    (print-X-axis-numbered-line
     tic-number leading-spaces horizontal-step))
(defun print-graph
  (numbers-list
   &optional vertical-step horizontal-step)
  ...
    (print-X-axis numbers-list horizontal-step))
```

C.4.3 打印出来的图形

作了上述修改并安装这些函数之后，可以这样调用 pring-graph 函数：

```
(print-graph fiftieth-list-for-graph 50 10)
```

下面就是打印出来的结果：

```
1000 -  *
        **
        **
        **
        **
 750 -  ***
        ***
        ***
        ***
        ****
 500 - *****
       ******
       ******
       ******
       *******
 250 - ********
       *********                    *
       ***********                  *
       **************               *
  50 - ****************** *         *
       |   |    |    |    |    |    |    |
      10  50  100  150  200  250  300  350
```

从图中可以看到，函数定义中单词和符号数最多集中于10~19之间。

Powered by xiaoguo’s publishing studio
QQ:8204136