第15章 准备柱型图

我们的目标是构造一个柱型图，用来显示 Emacs Lisp 源代码中各种长度的函数定义的数目。

作为一个实际的问题，如果你正在构造一个图，你可能要用一个像 gnuplot 这样的程序来完成它。（gnuplot 还被很好地集成在 GNU Emacs 中。）然而，在这个例子中，我们要从头创建一个函数，并且在设计这个函数的过程中我们将重新熟悉前面已学习过的内容，同时学习更多的东西。

在这一章，将首先编写一个简单的打印图形的函数。这第一个函数定义是一个*函数原型*，这个迅速编成的函数，可以使我们初窥打印图形这一未知的领域。我们将发现神龙，或者发现它们是虚构的。对这个领域仔细观察一番之后，我们将感到更加自信，然后要增强这个函数的功能使之自动打印坐标轴。

由于 Emacs 的柔性设计以及它能在各种类型的终端（包括字符终端）上很好地工作，因此我们将输出的图形用符号打印出来。星号将能完成这一工作。当我们增强打印图形的函数的功能时，我们将使用户可以选择他们自己喜爱的符号。

能够调用 graph-body-print这个函数，它将接收numbers-list作为其唯一的参量。在这一阶段，将不给出图形的坐标，而仅仅打印图形本身。

graph-body-print 函数根据 numbers-list 列表中的每一个元素插入一个由星号组成的垂直列。每一列的高度由 numbers-list 中相应元素的值决定。

插入垂直列的操作是反复执行的，这意味着 graph-body-print 这个函数既可以用 while 循环完成，也可以用递归方法完成。

第一个挑战是如何打印一个由星号组成的列。通常，在 Emacs 中，我们是在水平方向上打印字符的，一行一行地打印。为了打印垂直列，有两种方法可以采纳：编写自己的列输出函数，或者是找一个 Emacs 已有的函数。

要看看 Emacs 中是否有这样的一个函数，可以使用 M-x apropos 命令。这个命令很像 C-h a(command-apropos) 命令，只是后者只查找作为命令的函数。M-x apropos 命令则列出与一个正则表达式匹配的所有符号，包括非交互函数。

这里所要查找的是一些打印或插入列符号的命令。很可能这些函数的函数名将包括“print”、“insert”或者“column”这样的单词。因而，能够简单地输入 M-x apropos RET print\|insert\|column RET并看一看结果。在我的系统中，这个命令的执行需要一定的时间，然后产生一个含有 79个函数和变量的列表。扫描这个列表，只有 insert-rectangle 这个函数看起来比较适合做这项工作。确实，这就是我们寻找的那个函数。它的函数说明文档说：

```
insert-rectangle:
Insert text of RECTANGLE with upper left corner at point.
RECTANGLE's first line is inserted at point,
its second line is inserted at a point vertically under point, etc.
```

```
RECTANGLE should be a list of strings.
```

可以简单地测试一下这个函数，以确定它是否就是我们希望的那个函数。

将光标置于下面的insert-rectangle表达式之后并键入 C-u C-x C-e (evel-last-sexp) 命令，将得到随后的结果。这个函数将字符串“first”、“second”和“third”插入位点以及位点下面的位置。最后这个函数返回 nil。

```
(insert-rectangle '("first" "second" "third"))first
                                               second
                                               third
nil
```

当然，我们不希望将 insert-rectangle 表达式的这些文本内容插入到我们正在打印图形的缓冲区中，但是我们将从我们编写的函数中调用这个函数。然而，我们将不得不确保位点位于缓冲区中 insert-rectangle函数将插入字符列的适当位置。

如果你在 Info 中阅读这份文档，将缓冲区切换到诸如“*scratch*”缓冲区等缓冲区中，并将位点置于该缓冲区中的适当位置，然后键入 M-:，在消息缓冲区中的提示符下输入 insert-rectangle 表达式并键入回车键，这样你就可以看到这个函数是如何工作的了。这样做是使 Emacs 在消息缓冲区中对表达式求值，但是使用了“*scratch*”缓冲区中位点的值。(M-:是 eval-expression 的绑定键。)

我们发现：当我们这样做时，位点最后将位于最后插入的一行的末尾——这就是说，作为这个函数的附带效果，这个函数将位点移动了。如果我们这时（位点移动到这个位置时）试图重复使用这个函数，下一列将插入到原来插入的列的右下方。我们不能这样做！如果我们将要制作一个柱型图，每一列都应当伴着另一列。

因此我们发现：插入列的每一次 while 循环，必须重新将位点置于我们需要的位置，面且这个位置必须是一列的顶部而不是底部。而且，我们记得：当打印一个图形时并不希望所有的列都是一样高的。这意味着每一列的顶部，可能与前面各列的顶部不一样高。因此，不能简单地在每一次循环中将位点置于同一行上.....

我们正计划用星号打印柱型图。每一列上星号的数量是由 numbers-list 列表中的当前元素确定的。需要构建一个星号的列表，这个列表给出每一列上星号的正确数目，供 insert-rectangle 函数调用。如果这个列表仅仅包含所需的星号的数目，那么我们将正确定位，给出从打印的图形的底部开始向上究竟每一列占据多少行的高度。这是很难的。

另外，如果能够找出一种方法给 insert-rectangle 函数传递一个等长度的列表，那么就能将位点置于同一行上，只需朝右移动一列就可以了。然面，如果这样做了，传递给 insert-rectangle 函数的列表中的某些元素必须是空格面不是星号。例如，如果图形的最大高度是 5，但是这一列的高度是 3，那么 insert-rectangle 函数就需要这样一个列表：

```
(" " " " "*" "*" "*")
```

最后这个方案不太难，只要能确定列的高度就行。有两种方法供我们确定某一列的高度：一是根据需要确定图形的高度；一是扫描整个数值列表，用其中最大的数作为图形的最大高度。如果后面一个方案难以实现，前一个方案将是最简单的，但是在 Emacs 中，有一个内置函数可

以用来确定一个列表中的最大元素。我们可以使用这个函数，这个函数就是 max 函数，它返回其参量中最大的一个值。这个函数的参量必须是数字。例如，

```
(max  3 4 6 5 7 3)
```

将返回 7。（与max函数相对应的一个函数是 min函数，这个函数返回其参量中最小的值。）

然而，我们不能简单地在 number-list 列表上调用 max 函数。max 函数的参量是具体的数字，而不是数的列表。因而，下面的表达式，

```
(max  '(3 4 6 5 7 3))
```

将产生一个错误消息：

```
Wrong type of argument:  integer-or-marker-p, (3 4 6 5 7 3)
```

需要使用一个函数将参量列表传递给其他函数。这个函数就是 apply。这个函数将它的第一个参量（这个参量是一个函数）应用到其余的参量上，最后一个参量是一个列表。

例如，

```
(apply 'max 3 4 7 3 '(4 8 5))
```

返回8。

（顺便说一下，我不知道没有书的话你是如何学习类似的函数的。通过猜测其名字的一部分，然后使用 apropos 来查找它们，有可能发现其他的函数(如 search-forward 或者 insert-rectangle)。即使 apply 这个函数名具有明显的喻意——将其第一个参量应用到其余参量上——我仍然怀疑一位新手是如何在使用 apropos 或者其他工具的时候想到这个特定的单词的。当然，我可能错了，毕竟这个函数最初是由发明它的那个人命名的。）

这个函数第二个以及后续的参量是可选的，因此可以使用 apply 函数来调用一个函数，并将一个列表的元素传递给这个函数。因此像下面这样也将返回 8：

```
(apply 'max '(4 8 5))
```

后面这种方法就是我们将使用 apply 函数的方法。recursive-lengths-list-many-files 函数返回一个长度列表，对这个列表可以使用 max（也可以将max函数用于排序后的长度列表，当然列表是否排序在此是无关紧要的）。

因此，查找图形的最大高度的表达式就是：

```
(setq max-graph-height (apply 'max numbers-list))
```

现在我们能回到如何为图形的每一列创建一个字符列表的问题上了。知道了图形的最大高度和在这一列中的星号的数目，函数就应该返回一个供 insert-rectangle 命令使用的字符串列表了。

每一列都是由星号和空格组成的。因为函数接受的参量是列的最大高度和这一列的星号数量，所以空格的数目就是最大高度减去星号的数量。一旦给定了空格和星号的数量，用两个 while 循环就能构造 insert-rectangle 函数需要的列表：

```
;;; First version.
(defun column-of-graph (max-graph-height actual-height)
  "Return list of strings that is one column of a graph."
  (let ((insert-list nil)
```

```
      (number-of-top-blanks
       (- max-graph-height actual-height)))

    ;; Fill in asterisks.
    (while (> actual-height 0)
      (setq insert-list (cons "*" insert-list))
      (setq actual-height (1- actual-height)))

    ;; Fill in blanks.
    (while (> number-of-top-blanks 0)
      (setq insert-list (cons " " insert-list))
      (setq number-of-top-blanks
            (1- number-of-top-blanks)))

    ;; Return whole list.
    insert-list))
```

如果安装了这个函数并对下面这个表达式求值，你将看到它返回一个你需要的列表：

```
(column-of-graph 5 3)
```

返回

```
(" " " " " "*" "*" "*")
```

column-of-graph 函数有一个重要问题：图形中各列的空格和星号所用的符号都是“硬编码”（hard-coded）的，在函数中就是空格和星号。对于一个原型函数来说，这很好。但是你或者其他用户可能希望使用其他符号。例如，在测试图形函数时，你可能希望用一个句点来代替其中的空格，以确保位点每一次都由 insert-rectangle 函数正确设置。或者你可能希望使用“+”符号或者其他符号来代替星号。你甚至可能希望打印出来的图形每一列都占用超过一列的宽度。这个函数就应当更具伸缩性和弹性。用其他符号来代替空格和星号的途径是使用两个称为 graph-blank 和 grapp-symbol的变量，并单独定义这两个变量。

而且，函数文档还没有写好。基于这些考虑，我们编写了第二个函数定义：

```
(defvar graph-symbol "*"
  "String used as symbol in graph, usually an asterisk.")

(defvar graph-blank " "
  "String used as blank in graph, usually a blank space.
graph-blank must be the same number of columns wide
as graph-symbol.")
```

（关于 defvar 的详细资料，参见8.4节“用defvar初始化变量”。）

```
;;; Second version.
(defun column-of-graph (max-graph-height actual-height)
  "Return list of MAX-GRAPH-HEIGHT strings;
ACTUAL-HEIGHT are graph-symbols.
The graph-symbols are contiguous entries at the end
of the list.
```

```
The list will be inserted as one column of a graph.
The strings are either graph-blank or graph-symbol."

  (let ((insert-list nil)
        (number-of-top-blanks
         (- max-graph-height actual-height)))
;; Fill in graph-symbols.
(while (> actual-height 0)
  (setq insert-list (cons graph-symbol insert-list))
  (setq actual-height (1- actual-height)))

;; Fill in graph-blanks.
(while (> number-of-top-blanks 0)
  (setq insert-list (cons graph-blank insert-list))
  (setq number-of-top-blanks
        (1- number-of-top-blanks)))

;; Return whole list.
insert-list))
```

如果愿意，我们可以再一次重写 column-of-graph 函数，让用户选择打印一个线型图或者柱型图。这一点并不难做到。线型图与柱型图的不同在于柱型图中顶端与每一个柱条之间是空白部分。要构造一个线型图的一列，函数首先要构造一个空白列表，这个空白列表的长度比当前列的值小 1，然后用cons 函数在这个列表中增加一个图形符号，最后使用 cons 函数往这个列表中增加顶端空白部分。

编写这样一个函数是很容易的，但是由于我们不需要编写这个函数，因此我们不会做这份工作。但是这项工作应当被完成，并且如果要编写这样的函数，可以在 column-of-graph 的基础上进行改写而得到。更为重要的是，值得提醒一下，只要做些很小的修改就行了。如果你愿意编写这个函数，更改或者增强的部分是很简单的。

现在，最后，我们回到第一个打印图形的实际的函数上来。这个函数打印图形的主体部分，但是没有垂直轴和水平轴的坐标，因此可以称这个函数为 graph-body-print。

15.1 graph-body-print函数

经过前面章节的准备工作之后，graph-body-print 函数就很容易编写了。这个函数将一列一列地打印由星号和空格组或的列，每一列都对应着一个长度列表中的一个元素，这个列表中的每一个元素定义了该列中星号的数量。这是一个重复的动作，这意味着为此我们可以使用一个递减的 while 循环或者递归函数。在这一节，我们将编写使用 while 循环的函数定义。

column-of-graph 函数需要图形的高度作为其参量，因此应当确定并用一个局部变量记录图形的高度。

这样 graph-body-print 函数的函数定义的模板就像如下所示：

```
(defun graph-body-print (numbers-list)
  "documentation..."
```

```
(let ((height  ...
       ...))

  (while numbers-list
    insert-columns-and-reposition-point
    (setq numbers-list (cdr numbers-list)))))
```

需要往这个函数定义模板中填写相应的代码。

很明显，能够使用(apply' max numbers-list)表达式来确定图形的高度。

函数中的while循环将每次一个地遍历 numbers-list 列表中的每一个元素。每当 numbers-list 列表由(setq numbers-list (cdr numbers-list))表达式不断地删去前面的一个元素而变得越来越短时，这个列表的每个实例的 car 就是传递给 column-of-graph 的参量的值。

在 while 的每一次循环中，insert-rectangle 函数插入由 column-of-graph 函数返回的列表。由于 insert-rectangle 函数将位点移动到插入的矩形的右下方，因此我们需要在插入矩形时保存当前位点的位置，在插入矩形之后将位点从矩形的右下方移回原来的位置，然后在水平方向上移动到下一个位置。

如果插入的列是一个字符宽的，就像使用空格和星号表示的柱型图那样，重新定位位点的位置的表达式就是 (forward-char 1)。然而，柱型图中列的宽度可能大于一个字符。这意味着重新定位位点位置的表达式应当写成 (forward-char symbol-width)。变量 symbol-width 本身就是 graph-blank 的长度，并可以用 (length graph-blank) 表达式得到。将 symbol-width变量绑定到图形列宽度值的最佳位置是在 let表达式的变量列表中。

基于这些考虑，graph-body-print 函数的函数定义就应当如下所示：

```
(defun graph-body-print (numbers-list)
  "Print a bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values."

  (let ((height (apply 'max numbers-list))
        (symbol-width (length graph-blank))
        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")
))
```

在这个函数定义中，一个意想不到的表达式是 while 循环中的 (sit-for 0)。这个表达式使打印图形的操作比其他方式更为有趣。这个表达式使 Emacs "停下来"，或者在一段时间内什么也不做，然后重绘屏幕。将这个表达式放在这里，就使 Emacs 一列一列地显示图形。没有这一个表达式，Emacs 将在函数结束退出时才将图形显示出来。

可以用一个简短的数字列表来测试 graph-body-print 函数。

1) 安装 graph-symbol、graph-blank、column-of-graph 和 graph-body-print。

2) 拷贝下面的表达式：

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

3) 切换到 "*scratch*" 缓冲区，将光标置于需要绘制图形的位置。

4) 键入M-: (eval-expression)。

5) 在小缓冲区中用 C-y (yank) 命令找到 graph-body-print 表达式。

6) 按回车键对 graph-body-print 表达式求值。

Emacs 将打印如下图形：

```
         *
    *    **
    *   ****
   *** ****
  ********* *
 ***********
************
```

15.2 recursive-graph-body-print函数

graph-body-print 函数也可以用递归的方法来编写。在这种情况下，这个函数被分成两个部分：外层部分使用 let表达式来决定几个变量的值，如图形高度的最大值等，这些值都是只要开始时定下来就行了；内层的一个函数则递归地打印图形。

外层的函数并不复杂：

```
(defun recursive-graph-body-print (numbers-list)
  "Print a bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values."
  (let ((height (apply 'max numbers-list))
        (symbol-width (length graph-blank))
        from-position)
    (recursive-graph-body-print-internal
     numbers-list
     height
     symbol-width)))
```

其中的递归调用部分有点儿复杂。它由四个部分组成：一个 "do-again-text " 真假测试表达式、打印图形的代码、递归调用以及 "next-step" 表达式。真假测试表达式是一个 if 表达式，

决定 numbers-list 列表是否还包含有元素，如果有，则函数继续打印一列图形，并再次调用自己。这个函数根据“next-step”表达式产生的值来递归调用自身。“next-step”表达式产生一个更短的 numbers-list 列表。

```
(defun recursive-graph-body-print-internal
  (numbers-list height symbol-width)
  "Print a bar graph.
Used within recursive-graph-body-print function."

  (if numbers-list
      (progn
        (setq from-position (point))
        (insert-rectangle
         (column-of-graph height (car numbers-list)))
        (goto-char from-position)
        (forward-char symbol-width)
        (sit-for 0)      ; Draw graph column by column.
        (recursive-graph-body-print-internal
         (cdr numbers-list) height symbol-width))))
```

安装后，就可以测试这个表达式了。下面是一个例子：

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

对这个表达式求值就产生下面的打印结果：

```
                *
               **   *
              ****  *
              **** ***
            * *********
            ************
            *************
```

graph-body-print 函数或者 recursive-graph-body-print函数都只是打印图形的主体部分。

15.3 需要打印的坐标轴

一个图形总是需要坐标轴的，有了它你才能确定方向。对于一次性的项目，有足够的理由使用Emacs的图形模式来绘制坐标轴。但是，对于一个图形打印函数而言，就需要多次绘制坐标轴。

基于这个原因，我已经编写了对基本的 graph-body-print 函数的功能扩展部分，这个部分自动打印垂直与水平坐标轴的坐标。因为坐标轴打印函数没有包含什么新的东西和内容，我就将它放在附录C“带坐标轴的图”中介绍了。

15.4 练习

编写一个线型图打印函数。