第14章 统计函数定义中的单词数

下一个项目是在一个函数定义中实现对单词数量的计数。很明显，这个任务能够用 count-word-region 函数的某个变种来实现。参见第13章，“计数：重复和正则表达式”。如果只是要在一个函数定义中实现单词计数，用 C-M-h(mark-defun) 命令来标记这个函数定义，然后调用 count-word-region 函数，就可以轻易地做到这一点。

然而，我还有更大的野心：即要对所有的 Emacs 源代码中的函数定义的单词和符号计数，并打印一个图形，来显示不同长度的函数定义各有多少：有多少包含了 40~49 个单词或者符号，有多少包含 50~59 个单词或者符号，如此等等。我经常很好奇，一个典型的函数究竟有多长？这个函数将告诉我们这一点。

用一个短语来描述的话，这个柱型图项目是令人生畏的；但是我们将它分成许多小的步骤，每一次解决一个步骤，这个项目就变得不可怕了。让我们首先看一看这个项目应当有那些步骤：

- 第一，要编写一个函数对单个函数定义计数。这个函数既要处理单词也要处理符号。
- 第二，编写一个函数将函数定义中的单词数目列出来写进一个文件中。这个函数能够使用第一步编写的 count-words-in-defun 函数。
- 第三，要编写一个函数列出每一个文件中的每个函数中的单词数。它自动地查找不同的文件，切换到这些文件，并对其中的函数定义进行统计计数。
- 第四，要编写一个函数将在第三步产生的数变换成一个表，这个表要适合作为一个图形打印出来。
- 第五，编写一个函数将结果作为一个图形打印出来。

14.1 计数什么？

当我们第一次考虑如何对一个函数定义进行单词统计计数时，首先遇到的一个问题就是（或者应该是）要对什么东西计数？当我们在涉及 Lisp 函数定义而说到“单词”一词时，大部分情况下我们实际上是在说“符号”。例如，下面的multiply-by-seven 函数包含了5个符号：defun、multiply-by-seven、NUMBER、* 和 7。另外，在函数文档字符串中，它包含了 4 个单词：“Multiply”、“NUMBER”、“by” 和 “seven”。符号number是重复的，因此这个函数定义实际上总共包含了 10 个单词和符号。

```
(defun multiply-by-seven (number)
  "Multiply NUMBER by seven."
  (* 7 number))
```

然而，如果将 multiply-by-seven 函数用 C-M-h(mark-defun) 打上标记，并调用 count-words-region 函数，将发现 count-words-region 指出这个函数定义有 11 个单词，而不是 10 个。这中间肯定发生了一些错误!

错误是双重的：其一，count-words-region 没有将“*”符号当做一个单词来处理，而是将其当做一个符号来处理，因此 multiply-by-seven 包含了3个单词。其二，其中的连字符“-”被当做单词之间的间空而不是单词之间的连字符：“multiply-by-seven”被当做“multiply by seven”。

产生这种混乱的原因是：count-words-region 中的正则表达式查询是一个单词一个单词地向前移动的。在 count-words-region 函数的正式版本中，正则表达式是：

```
"\\w+\\W*"
```

这个正则表达式定义了这样一种模式：一个或者多个构词要素（字符）之后可能跟着一个或者多个非构词要素。所谓的构词要素要根据语法来定义，这一点值得在下面单独介绍。

14.2 单词或者符号是由什么构成的?

Emacs 根据不同的*语法分类*来处理不同的字符。例如，正则表达式“\\w+”是指定一个或者多个*构词要素字符*的一种模式。构词要素字符是一种语法分类的成员。其他语法分类包括标点符号字符类（如句点和逗号），以及空白字符类（如空格和制表符）。（关于这方面更详细的资料，请参见《*GNU Emacs* 技术手册》和《*GNU Emacs Lisp* 技术手册》的“语法表”一节。）

语法表定义了哪些字符是属于何种类别的。通常，连字符不是一个构词要素字符。相反，它被定义为是符号名的一部分但不是单词的字符类。这意味着，count-words-region 函数像处理单词之间的空格一样，用同样的方式处理连字符“-”，这就是为什么count-words-region 函数将“multiply-by-seven”计为 3 个单词的原因。

有两种方式使 Emacs 将“multiply-by-seven”当做一个符号来计算：改变语法表或者改变正则表达式。

可以将连字符重新定义为一个构词要素字符。这是通过改变 Emacs 为每一种模式设置的语法表来实现的。这种改变能够实现我们的目的，只不过，连字符仅仅是最常用的非常规构词要素字符。当然也还有其他这类字符。

另外，能够重新定义 count-words-region 函数中使用的正则表达式，以使之正确识别这些符号。这个过程的优点是清楚明了，但是稍微有点复杂。

第一部分是相当简单的：必须至少与一个构词要素字符匹配，因面正则表达式必须是：

```
\\(\\w\\|\\s_\\)+
```

“\\(”符号是分组结构的第一部分。这个分组结构包含“\\w”或者它的替代项“\\s_”。它们由“\\|”分开。其中的“\\w”与所有的构词要素字符匹配，而“\\s_”则与作为符号名的一部分而不是构词要素的字符匹配。其后的“+”号表示构成单词或者符号的字符必须至少匹配一次。

然而，这个正则表达式的第二部分更加难以设计。我们所需要的是在第一部分的基础上追加一个或者多个非构词要素字符(这种字符是可选的)。开始的时候我认为我可以将它设计为下面这种样子：

```
\\(\\W\\|\\S_\\)*"
```

大写的“W”和“S”字符用于与既不是构词要素的字符也不是构成符号的字符匹配。不幸的是，这个表达式与任何一个不是构词要素的字符匹配或者与任何一个不是构成符号的字符匹配。它匹配所有的字符！

然后我注意到，在我的测试区域中，每个单词或符号都跟着空白（或者是空格，或者是制表符，或者是换行符）。因此我试图设计一个与单个或者多个空白匹配的模式，这个模式跟在与一个或者多个构词要素或者是构成符号的字符匹配的模式之后。这也失败了。单词或符号经常是被空格分隔开的，但是在实际的代码中，括号可能跟随在符号之后，标点符号可能跟随在单词之后。因此，最后我设计了一个模式，在这个查询模式中，构词要素的字符或者构成符号的字符之后跟上可选的除了空白之外的字符，然后再跟上数目不定的空白。

这就是整个正则表达式：

```
"\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*"
```

14.3 count-words-in-defun 函数

我们已经看到，有许多种方法可以用来编写 count-words-in-defun 函数。要编写一个 count-words-in-defun 函数，仅仅需要采用其中的一种就行了。

使用 while 循环编写的函数很容易理解，因此我将采用这种方案来编写这个函数。因为 count-words-in-defun 将是一个更为复杂的程序的一部分，所以它无需是交互的，也无需显示消息，而只要返回计数值就行了。这些考虑稍微简化了这个函数的定义。

另外一个方面，count-word-in-defun 函数将被用于一个包含函数定义的缓冲区内。因而，有理由要求这个函数决定当位点处于一个函数定义当中时这个函数是否被调用。如果是，则返回这个函数定义的计数。这给 count-words-in-defun 函数的设计增加了复杂性，但是却使我们无需传递参量给这个函数了。

基于这样的考虑，下面的函数定义模板是合适的：

```
(defun count-words-in-defun ()
  "documentation..."
  (set up...
     (while loop...)
   return count)
```

像平常那样，我们的任务就是填充模板中的这些空缺。

首先，建立适当的环境。

我们假定这个函数将在一个包含了函数定义的缓冲区内被调用。位点要么在一个函数定义当中，要么不在其中。为了使 count-words-in-defun 函数正常工作，位点必须移动到函数定义的开始，计数器必须从零开始。计数循环必须在位点到达函数定义的末尾时停止。

beginning-of-defun 函数朝后查询某一行开始处的起始定界符(如“(”)并将位点移动到那个位置，或者移动到这个函数查询的边界。实际上，这意味着 beginning-of-defun 函数将位点移动到一个函数定义的开始，否则的话就移动到缓冲区的开始处。我们可以使用

beginning-of-defun 函数将位点移动到我们需要的地方。

while 循环需要一个计数器来跟踪要计数的单词或者符号。let 表达式能够被用于创建一个这样的局部变量，并将它绑定到零。

end-of-defun 函数与 beginning-of-defun 函数类似，它将位点移动到函数定义的末尾。end-of-defun 函数也可以用作一个决定位点是否是函数定义末尾的表达式的一部分。

count-words-in-defun 函数很快就有模有样了：首先将位点移动到函数定义的开始处，然后创建一个局部变量来存放计数值，最后记录函数定义末尾的位置，这样 while 循环就可以知道何时将停止循环。

函数代码就像这样：

```
(beginning-of-defun)
(let ((count 0)
      (end (save-excursion (end-of-defun) (point))))
```

这些代码很简单。其中稍微复杂一点的是 end：它被绑定到函数定义末尾的位置，这个值是由 save-excursion 表达式返回的，这个表达式所返回的值是通过调用 end-of-defun 函数使位点暂时地移动到函数定义的末尾并记录下它的值而得到的。

完成基本环境的设置之后，count-words-in-defun函数的第二部分就是while 循环。

这个循环必须包含这样一个表达式，它将位点一个单词接一个单词、一个符号接一个符号地往前移动。还要有另外一个表达式对移动次数计数。只要位点继续往前移动，while循环中的真假测试表达式应当测试为“真”；当位点移动到函数定义末尾时，测试结果为“假”。我们已经重新定义了一个满足这种要求的正则表达式，因此这个 while 循环就很直截了当了：

```
(while (and (< (point) end)
            (re-search-forward
             "\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*" end t)
  (setq count (1+ count)))
```

这个函数定义的第三部分返回对单词和符号的计数值。这个部分是 let 表达式主体中的最后一个表达式，很简单，就是局部变量 count。这个表达式被求值时就返回这个变量的值。

将这些组合起来，count-word-in-defun 函数定义就是：

```
(defun count-words-in-defun ()
  "Return the number of words and symbols in a defun."
  (beginning-of-defun)
  (let ((count 0)
        (end (save-excursion (end-of-defun) (point))))
    (while
        (and (< (point) end)
             (re-search-forward
              "\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*"
              end t))
      (setq count (1+ count)))
    count))
```

如何测试这个函数？这个函数不是交互的，但是很容易将它包装一下使其成为交互的函数。可以使用与 count-words-region 函数同样的代码：

```
;;; Interactive version.
(defun count-words-defun ()
  "Number of words and symbols in a function definition."
  (interactive)
  (message
   "Counting words and symbols in function definition ... ")
(let ((count (count-words-in-defun)))
  (cond
   ((zerop count)
    (message
     "The definition does NOT have any words or symbols."))
   ((= 1 count)
    (message
     "The definition has 1 word or symbol."))
   (t
    (message
     "The definition has %d words or symbols." count)))))
```

让我们再使用 C-c= 组合键绑定到这个函数定义上：

```
(global-set-key "\C-c=" 'count-words-defun)
```

现在，我们能够试一试 count-words-defun 函数了：将 count-words-in-defun 函数和 count-words-defun 函数都安装好，并设置好绑定的键，然后将光标置于下面这个函数定义当中：

```
(defun multiply-by-seven (number)
  "Multiply NUMBER by seven."
  (* 7 number))
     ⇒ 10
```

万事大吉！这个函数定义有 10 个单词和符号。

下一个问题就是对一个文件中的几个函数定义中的单词和符号进行计数。

14.4 在一个文件中统计几个函数定义的单词数

一个文件(如“simple.el”)可以存放 80个或者更多的函数定义。我们的长远目标是计算许多文件的统计结果，但是第一步，首要目标就是收集一个文件的统计信息。

这些统计信息是一系列的数字，每一个数就是一个函数定义的长度。可以将这些数储存在一个列表中。

我们知道，我们将把从这个文件中得到的信息与从另外的文件中得到的信息进行整合；这意味着，对文件中所有的函数定义进行单词计数的函数，只需返回这些长度的列表，而无需也不应当显示其他信息。

单词计数命令包含一个将位点一个单词一个单词朝前移动的表达式，以及另外一个对移动

的次数进行计数的表达式。这些计算函数定义长度的函数，也可以用同样的方式工作，用一个表达式使位点一个函数定义一个函数定义地朝前移动，而另一个表达式来构造长度列表。

对问题的这种描述，使之成为编写函数定义的基础。很明显，将从文件的开始处进行计数统计，因此这个函数的第一个命令将是：(goto-char (point-min))。其次，要开始一个while 循环，这个循环中的真假测试可以是一个正则表达式查询，它查询下一个函数定义——只要查询成功，位点将朝前移动，然后这个循环体被求值。循环体需要一个表达式来构造长度列表。cons这个列表构造命令能够被用于创建这个长度列表。这就是所有要做的工作。

下面是这部分代码片断：

```
(goto-char (point-min))
(while (re-search-forward "^(defun" nil t)
  (setq lengths-list
        (cons (count-words-in-defun) lengths-list)))
```

我们剩下的任务就是查找包含函数定义的文件。

在前面的例子中，我们要么使用这份文档（Info 文件），要么在一些缓冲区（如草稿缓冲区）之间来回切换。

查找一个文件是一个我们迄今为止还没有讨论的新过程。

14.5 查找文件

要在 Emacs 中查找一个文件，可以使用 C-x C-f(find-file) 命令。这个命令几乎就是为长度问题设计的，但并不完全这样。

让我们看一看 find-file 函数定义的源代码（可以使用 find-tag 命令来找到一个函数的源代码）：

```
(defun find-file (filename)
  "Edit file FILENAME.
Switch to a buffer visiting file FILENAME,
creating one if none already exists."
  (interactive "FFind file: ")
  (switch-to-buffer (find-file-noselect filename)))
```

这个函数定义具有简短但是完整的文档说明。当你交互地使用这个函数时，其中的交互表达式提示你输入一个文件名。函数定义的主体包含两个函数：find-file-noselect 和 switch-to-buffer。

根据由 C-h f（describe-function命令）命令得到的函数定义的说明文档，find-file-noselect 函数将文件读入一个缓冲区并返回到原来的缓冲区。然而，这个缓冲区没有被选中。Emacs 并不切换到读入文件的那个缓冲区。这就是 switch-to-buffer 函数所要完成的任务：它将 Emacs 关注的缓冲区切换到另外一个缓冲区，并将后面这个缓冲区在当前窗口中显示。我们已经讨论过缓冲区的切换的问题。（参见2.3节，“切换缓冲区”。）

在这个柱型图项目中，当程序在确定文件中的函数定义的长度时，确实不需要在屏幕上显示每一个文件。除了使用 switch-to-buffer 函数之外，我们能使用 set-buffer 函数，

这个函数将计算机程序的关注焦点重新定向到另外一个缓冲区，但是并不将这个缓冲区显示在屏幕上。因此，我们必须编写自己的表达式，而不是调用 find-file 命令来完成我们的任务。

完成这项任务很简单，就是使用 find-file-noselect 和 set-buffer 函数。

14.6 lengths-list-file 函数详解

lengths-list-file 函数的核心是一个 while 循环，这个循环包含一个函数定义一个函数定义地朝前移动位点的函数，以及计算每一个函数定义中的单词和符号数的函数。这个核心部分，必须在完成其他相关任务(比如查找一个文件)的函数内部，，以确保位点从文件的开始移动。这个函数定义是：

```
(defun lengths-list-file (filename)
  "Return list of definitions' lengths within FILE.
The returned list is a list of numbers.
Each number is the number of words or
symbols in one function definition."
  (message "Working on `%s' ... " filename)
  (save-excursion
    (let ((buffer (find-file-noselect filename))
          (lengths-list))
      (set-buffer buffer)
      (setq buffer-read-only t)
      (widen)
      (goto-char (point-min))
      (while (re-search-forward "^(defun" nil t)
        (setq lengths-list
              (cons (count-words-in-defun) lengths-list)))
      (kill-buffer buffer)
      lengths-list)))
```

这个函数有一个参量，就是对其进行统计计数的文件名。这个函数定义中有四行说明文档，但是没有交互表达式。因为人们担心在他们看不到任何事情正在运转的时候计算机就崩溃了，因此函数体的第一行就输出一条消息。

函数定义中的下面一行，包含一个 save-excursion 函数，这个函数将 Emacs 的注意力在函数执行完之后返回到当前的缓冲区。当你将这个函数嵌入在另外一个函数当中，并假设位点保存在原来的缓冲区中时，这是很有用的。

在 let 表达式的变量列表中，Emacs 找到文件并将局部变量 buffer 绑定到存放这个文件的缓冲区。同时，Emacs 创建局部变量 lengths-list。

接下来，Emacs 将它的注意力切换到那个缓冲区。

随后，Emacs 使这个缓冲区成为只读的。在理想的情况下这一行是不需要的。对函数定义中的单词或者符号进行计数的这些函数都不应当改变缓冲区的内容。另外，即使它被改变，这个缓冲区也将不被保存。这一行完全是为了安全小心起见。其理由是，这个函数以及它调用的那些函数的处理对象是 Emacs 源代码，如果这些源代码不经心地被改变了，那将是很麻烦的事情。

我可以告诉你，当我在一次试验当中改变了我的 Emacs 源代码后发生的事情，才使我意识到这一行的必要性。

再后面是一个增宽命令，如果缓冲区变窄开启，这个命令就起作用了。这个函数通常是不需要的——如果缓冲区不存在，Emacs 将创建一个新的缓冲区；但是当一个缓冲区正在访问这个文件，Emacs 就返回这个缓冲区。在这种情况下，这个缓冲区可能变窄开启了，因此需要增宽。如果要使这个函数实现完全的“用户友好”，应该安排好保存这些原来的设置以及位点的位置，但是我们不要这样做。

(goto-char (point-min))表达式将位点移动到缓冲区的开始。

然后是一个while循环，在这个循环中完成这个函数的主要任务。在这个循环中，Emacs 确定每一个函数定义的长度并构造一个长度列表来保存这些信息。

Emacs 在完成计数统计之后删除这个缓冲区，这是为了节省 Emacs 的空间。我的Emacs 第19版中包含了超过 300 个有意思的源文件，对于每一个函数要逐一使用 lengths-list-file 进行计数。如果 Emacs 访问所有这些文件而不删除任何一个缓冲区，我的计算机将用完它的虚拟内存。

最后，let 表达式中的最后一个表达式是 lengths-list 变量，它的值是作为整个函数的返回值而返回的。

你可以通过用通常的办法安装这个函数来试一试它。将光标置于下面的表达式后并键入C-x C-e(eval-last-sexp) 即可：

```
(lengths-list-file "../lisp/debug.el")
```

(你可能需要改变这个文件的路径，如果这个 Info 文件和 Emacs 源文件都在邻近的地方（如 /usr/local/emacs/info 和 /usr/local/emacs/lisp），这里列出的这个路径是起作用的。要改变这个表达式，只需将它拷贝到 “*scratch*” 缓冲区并编辑它，然后对它求值就行了。)

在我的那个版本的 Emacs 中，我的计算机花了 7 秒钟才产生“debug.el”文件的长度列表。它是这样的一个列表：

```
(75 41 80 62 20 45 44 68 45 12 34 235)
```

注意，文件中最后一个函数定义的长度在列表的开始位置。

14.7 在不同文件中统计几个函数定义的单词数

在前面的章节，创建了一个返回单个文件中所有函数定义的单词和符号长度的列表。现在，要定义一个计算一系列文件中函数定义的长度列表的列表。

对文件列表中每一个元素的处理都是重复的，因此能用while循环来处理，也能用递归调用的方法来处理。

常规的方法是使用while循环。传递给函数的参量是一个文件列表。就像我们前面看到的（参见11.1节，“while循环和列表”），你能够编写一个while循环，如果这样一个文件列表包含了元素，这个循环的循环体就被求值；但是如果这个文件列表中没有元素，就退出循环体。为了使这个设计能够正常工作，循环体必须包含一个表达式，这个表达式使文件列表每当循环

体执行一次就减少一个元素，因此最终列表就成了一个空列表。常用的技术就是每次循环体被求值时，将列表的值设为这个列表的cdr的值。

这部分代码的模板是：

```
(while test-whether-list-is-empty
  body...
  set-list-to-cdr-of-list)
```

而且，我们记得，while 循环将返回 nil（也就是测试表达式的值），而不是其循环体中任何表达式的值。(循环体中表达式的求值是作为附带效果而被完成的。)然而，设置长度列表的表达式是循环体的一部分——这是我们要作为整个函数的返回值的。为了实现这一点，我们将 while 表达式放在 let 表达式中，并使 let 表达式的最后一个元素包含了长度列表。(参见11.1.3节中“使用增量计数器的例子”小节。)

基于上面的考虑，可以直接写出如下函数定义：

```
;;; Use while loop.
(defun lengths-list-many-files (list-of-files)
  "Return list of lengths of defuns in LIST-OF-FILES."
  (let (lengths-list)

;;; true-or-false-test
    (while list-of-files
      (setq lengths-list
            (append
             lengths-list

;;; Generate a lengths' list.
             (lengths-list-file
              (expand-file-name (car list-of-files)))))

;;; Make files' list shorter.
      (setq list-of-files (cdr list-of-files)))

;;; Return final value of lengths' list.
    lengths-list))
```

expand-file-name 函数是一个内置函数，它将一个文件名转换成文件的绝对的长路径形式。因而文件

```
debug.el
```

变成

```
/usr/local/emacs/lisp/debug.el
```

这个函数定义中，仅有的一个新元素就是至今没有学习过的 append函数，这个函数值得用一个小节专门讲述。

append 函数

append 函数将一个列表追加到另外一个列表之后，因而，

```
(append '(1 2 3 4) '(5 6 7 8))
```

就产生下面这个列表

```
(1 2 3 4 5 6 7 8)
```

这就是为什么要将由 lengths-list-file 产生的两个列表连接起来组成一个列表的原因。这个结果与 cons 函数正好形成对照：

```
(cons '(1 2 3 4) '(5 6 7 8))
```

这个表达式将其第一个参量当做一个元素插入到其第二个参量列表中，形成一个新的列表：

```
((1 2 3 4) 5 6 7 8)
```

14.8 在不同文件中递归地统计单词数

除了 while 循环之外，可以用递归的办法处理文件的每一个列表。lengths-list-file 函数的递归实现更短、更简单。

递归函数通常有这样几个部分："do-again-test" 真假测试表达式，"next-step" 表达式，以及递归调用。其中真假测试表达式决定函数是否继续调用自身，如果 list-of-files 列表仍包含有剩余的元素，就继续调用函数自身（即递归调用）；" next-step " 表达式将 list-of-files 列表重置为这个列表的 cdr，因此最终这个列表就变空了；面递归调用则在更短的列表上调用自身。整个函数比上面描述的更简短。

```
(defun recursive-lengths-list-many-files (list-of-files)
  "Return list of lengths of each defun in LIST-OF-FILES."
  (if list-of-files                          ; do-again-test
      (append
       (lengths-list-file
        (expand-file-name (car list-of-files)))
       (recursive-lengths-list-many-files
        (cdr list-of-files)))))
```

简言之，这个函数返回 list-of-files 列表中第一个元素的长度列表，并将这个列表追加到对 list-of-files 列表中剩余元素的递归调用的结果上。

下面是对这个递归函数 recursive-lengths-list-many-files的测试，并给出了 lengths-list-file 函数对每一个文件的统计结果（做比较用）。

将 recursive-lengths-list-many-files 和 lengths-list-file 两个函数安装好，如果需要，然后对下面的表达式求值。你可能需要改变其中的文件的路径，当这个 Info 文件以及 Emacs 源文件在它们通常的位置，下面的表达式是能够正常工作的。要改变这些表达式，就将它们拷贝到 "*scratch*" 缓冲区，编辑它们，然后对它们求值。

求值的结果用 "⇒" 符号表示（这些结果是对 Emacs第18.57版而言的，其他版本的 Emacs 的结果可能与此不同）。

```
(lengths-list-file
 "../lisp/macros.el")
     ⇒ (176 154 86)
```

```
(lengths-list-file
 "../lisp/mailalias.el")
     ⇒ (116 122 265)

(lengths-list-file
 "../lisp/makesum.el")
     ⇒ (85 179)
(recursive-lengths-list-many-files
 '("../lisp/macros.el"
   "../lisp/mailalias.el"
   "../lisp/makesum.el"))
      ⇒ (176 154 86 116 122 265 85 179)
```

recursive-lengths-list-files 函数产生了我们需要的结果。

下一步就是为以图形的形式显示列表中的数据做准备。

14.9 为图形显示准备数据

recursive-lengths-list-many-files 函数返回一个数的列表。其中的每一个数记录一个函数定义的长度。现在需要做的工作是将这些数据转换成一个适合图形显示的数据列表。这个新的列表将告诉人们有多少函数定义的长度少于10个单词和符号，有多少函数定义的长度介于 10 和 19 之间，又有多少函数定义的长度介于 20 和 29 之间，如此等等。

简而言之，需要遍历由 recursive-lengths-list-many-files函数产生的这个长度列表，并计算在每一个长度范围内的函数定义的个数，进而产生一个记录这些数据的列表。

基于前而已经完成的工作，我们能够轻松地预测：编写这样一个函数并不太难，这个函数只要不断地使用长度列表的 cdr 就可以访问这个列表的每一个元素，并测定这个元素属于哪一个长度范围，然后对那个长度范围的计数器加 1。

然而，在开始编写这样的函数之前，应该考虑首先将列表排序的好处。将列表排序后，列表中的元素就是从小到大一一排列的。首先，排序将使对每一个范围的计数更加简单，因为两个相邻的元素要么在同一个长度范围，要么在相邻的长度范围。其次，要检查一个排好序的列表，我们能发现最大和最小的数，因此可以决定需要考虑的最大和最小的长度范围。

14.9.1 对列表排序

Emacs 包含了一个对列表中元素排序的函数，这个函数就是 sort 函数（可能你已经猜到了）。sort 函数接收两个参量，一个是要被排序的列表，另外一个是一个谓词，这个谓词决定目标列表中的第一个元素是否小于第二个元素。

就像前面看到的（参见第 1.8.4节，“用一个错误类型的数据对象作为参量”），谓词就是一个决定某些特性是否为真的函数。sort 函数将根据谓词使用的情况记录一个列表。这就是说，sort 函数能够被用于对非数字列表排序，只要使用适当的非数字标准的谓词就行了——例如，它能按字母顺序对一个列表排序。

当对一个数字列表排序时，要使用 < 函数。例如，

```
(sort '(4 8 21 17 33 7 21 7) '<)
```

将产生下面这个列表：

```
(4 7 7 8 17 21 21 33)
```

（注意，在这个例子中，sort 函数的两个参量都使用了单引号，因此这些符号在作为参量传送给 sort 函数时不应被求值。）

对由recursive-lengths-list-many-files 函数返回的列表进行排序是直截了当的：

```
(sort
 (recursive-lengths-list-many-files
  '("../lisp/macros.el"
    "../lisp/mailalias.el"
    "../lisp/makesum.el"))
 '<)
```

将产生：

```
(85 86 116 122 154 176 179 265)
```

（注意，在这个例子中，sort 函数的第一个参量没有用单引号，这是因为这个表达式必须被求值以产生一个传递给 sort 函数的列表。）

14.9.2 制作一个文件列表

recursive-lengths-list-many-files 函数需要一个文件列表作为其参量。对于我们的测试例子，我们手工构建了这样一个文件列表。但是 Emacs Lisp 源代码目录太大，以致于我们无法手工构建其中所有文件的列表。确实，我们需要使用 directory-files 函数来为我们自动构建文件列表。

directory-files 函数接收三个参量：第一个参量是目录名，它是一个字符串；非空的第二个参量使函数返回目录中文件的绝对路径；第三个参量是一个可选项。如果这个可选项包含了一个正则表达式（而不是空），则只有路径名与正则表达式匹配的文件被返回。

因此，在我的计算机系统中，

```
(length
 (directory-files "../lisp" t "\\.el$"))
```

将告诉我，在我的计算机中的 Emacs 第19.25版源代码目录中包含 307 个“.el”的文件。

在 recursive-lengths-list-many-files 函数中，这个表达式应当是：

```
(sort
 (recursive-lengths-list-many-files
  (directory-files "../lisp" t "\\.el$"))
 '<)
```

我们直接的目标是创建一个列表，这个列表要告诉我们，有多少函数定义包含的单词和符号少于 10 个，有多少函数定义包含的单词和符号介于 10 到 19 之间，又有多少函数定义包含的单词和符号介于 20 和 29 之间，如此等等。采用一个排序后的数字列表，这就容易多了：计算

这个列表中有多少元素小于 10，然后移动到计算过的元素之后，再计算有多少元素小于 20，然后移动到这次计算过的元素之后，再计算有多少元素小于 30.....。每一个数，如10、20、30、40 等等，都比那个范围的值要大。我们称这些数的列表为 top-of-ranges 列表。

如果需要，我们能够自动地产生这个列表，但是人工写出这个列表更简单。下面就是：

```
(defvar 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)
 "List specifying ranges for `defuns-per-range'.")
```

要改变范围，编辑这个列表就行了。

下一步，需要编写一个函数来创建一个列表，这个列表的每一个元素分别记录落在每一个范围内的函数定义的数量。很明显，这个函数必须接收 sorted-lengths和top-of-ranges 列表作为其参量。

defuns-per-range 函数必须不断地完成两件事情：它必须根据当前范围的值计算落在这个范围内的函数定义的数量；而且它必须在计算完当前范围中的函数定义的数目之后移动到 top-of-ranges 列表中的更高的一个值。因为这两个操作中的任何一个都是不断反复的，所以可以使用 while 循环来完成它们。一个循环计算函数定义落在由当前值确定的某个范围内的数量，另外一个循环则依次选择每一个值（计数范围）。

对于每一个范围，列表 sorted-lengths 中都有几个元素。这意味着处理sorted-lengths 列表的循环，将在处理 top-of-ranges 列表的内部，就像一个小齿轮在一个大齿轮之中一样。

这个内层循环对某个范围内的函数定义的数目计数。它是一个简单的计数循环，我们在前面已经见到过（参见11.1.3节，“使用增量计数器的循环”）。这个循环中的真假测试表达式判断 sorted-lengths 中的元素值是否小于当前范围的最大值。如果是，这个函数对计数器加 1，并测试 sorted-lengths 列表中的下一个元素。

因此这个内层循环如下所示：

```
(while length-element-smaller-than-top-of-range
  (setq number-within-range (1+ number-within-range))
  (setq sorted-lengths (cdr sorted-lengths)))
```

外层的循环必须从 top-of-ranges 列表最小的一个元素开始，然后被依次设置为后续更高的一个元素。这可以用下面的一个循环实现：

```
(while top-of-ranges
  body-of-loop...
  (setq top-of-ranges (cdr top-of-ranges)))
```

将这两个循环连接起来就像下面所示：

```
(while top-of-ranges

  ;; Count the number of elements within the current range.
  (while length-element-smaller-than-top-of-range
    (setq number-within-range (1+ number-within-range))
    (setq sorted-lengths (cdr sorted-lengths)))

  ;; Move to next range.
  (setq top-of-ranges (cdr top-of-ranges)))
```

另外，在外层循环的每一次循环中，Emacs 应当在一个列表中记下在当前范围内的函数定义的数目（number-within-range）。可以使用 cons 函数来完成这一工作。（参见 7.2节，“cons函数”。）

cons 函数工作得很好。唯一的不足之处是：最高范围的函数定义的数目在列表的开头，而最低范围的函数定义的数目在列表的末尾。这是因为 cons 函数将新元素插入到列表的开头，并且两个循环是从长度列表的最低值开始的，defuns-per-range-list 将最终以最大值结束（即它以最大值作为其第一个元素）。但是我们将要从最小值开始打印柱型图，解决的办法是将 defuns-per-range-list 反转过来。使用 nreverse 函数可以做到这一点，这个函数的作用就是将一个列表中元素的顺序反转过来。

例如，

```
(nreverse '(1 2 3 4))
```

产生：

```
(4 3 2 1)
```

注意，nreverse 函数是“破坏性的”——也就是它改变了作为其参量的列表；这与car 和 cdr 函数形成对照，这两个函数是非破坏性的。在这个例子中，我们不需要原来的 defuns-per-range-list 列表，因此这个破坏性的函数对我们来说也没有什么问题。（reverse 函数提供一个反转的列表拷贝，而将原始的列表留下来。）

将这些统统整合起来，defuns-per-range 函数就像如下所示：

```
(defun defuns-per-range (sorted-lengths top-of-ranges)
  "SORTED-LENGTHS defuns in each TOP-OF-RANGES range."
  (let ((top-of-range (car top-of-ranges))
        (number-within-range 0)
        defuns-per-range-list)

    ;; Outer loop.
    (while top-of-ranges

      ;; Inner loop.
      (while (and
              ;; Need number for numeric test.
              (car sorted-lengths)
              (< (car sorted-lengths) top-of-range))
```

并直接返回“假”。但是如果 (car sorted-lengths) 表达式返回一个非空值，and 表达式就计算后续的<表达式，并将这个表达式的值作为 and 表达式的值返回。

通过采用这样一种方法，我们避免了一个错误。关于 and 函数的更详细的内容，参见12.4节“forward-paragraph：函数的金矿”。

下面是一个对 defuns-per-range 函数的简单测试。首先，对下面的表达式求值，使之绑定到 top-of-ranges 列表上。然后，绑定 sorted-lengths 列表，最后对 defuns-per-range函数求值。

```
;; (Shorter list than we will use later.)
(setq top-of-ranges
 '(110 120 130 140 150
   160 170 180 190 200))

(setq sorted-lengths
      '(85 86 110 116 122 129 154 176 179 200 265 300 300))

(defuns-per-range sorted-lengths top-of-ranges)
```

对上面这个表达式求值后返回的列表如下所示：

```
(2 2 2 0 0 1 0 2 0 0 4)
```

确实，在 sorted-lengths 列表中有 2 个元素小于 110，有 2 个元素介于110~119 之间，有 2 个元素介于 120~129 之间，等等。有 4 个元素等于或大于 200。