GNU技术文档精粹

HZ BOOKS

GNU Emacs Lisp 编程入门

Programming in Emacs Lisp: An Introduction

(美) Robert J. Chassell 著
毛文涛 吕芳 译
洪峰 审校

机械工业出版社
China Machine Press

Free Software Foundation

GNU技术文档精粹

GNU Emacs Lisp编程入门

(美) Robert J. Chassell 著
毛文涛、吕芳 译
洪峰 审校

机械工业出版社
China Machine Press

本书的作者罗伯特·卡塞尔是自由软件基金会的合创人之一，也是理查德·斯托曼博士青年时期结交的挚友，他精通GNU Emacs Lisp的每一个方面。本书是一本GNU Emacs Lisp的编程入门，全书循序渐进地介绍了GNU Emacs Lisp编程的各种基础知识和方法，文笔流畅、讲解透彻，对GNU Emacs用户提高对它的理解和运用帮助极大。

图书在版编目（CIP）数据

GNU Emacs Lisp编程入门 /（美）卡塞尔（Chassell, R. J.）著；毛文涛等译. -北京：机械工业出版社，2001.5

（GNU技术文档精粹）

书名原文：Programming in Emacs Lisp: An Introduction

ISBN 7-111-08862-X

Ⅰ.G… Ⅱ.①卡… ②毛… Ⅲ.LISP语言-程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2001）第19381号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：姚蕾

北京昌平奔腾印刷厂印刷 ·新华书店北京发行所发行

2001年5月第1版第1次印刷

787mm×1092mm 1/16 · 13.5印张

印数：0 001-5 000册

定价：38.00元

致中国读者

Calendars, email, writing in general, programming, and debugging programs: GNU Emacs gives you tools for all these actions, and gives you even more. GNU Emacs is truly an integrated environment. Emacs Lisp is the language in which most of Emacs is written. It is a simple yet powerful language that is easily understood and learned.

FSF-CHINA led by Hong Feng is doing us all a great service by translating this document from English into Chinese. The work will bring the joy, the efficiency, and the power of GNU Emacs to many people. I hope you will gain as much from reading this book as I did from writing it.

My best wishes to you. Bob Chassell Robert J. Chassell

译者序

GNU Emacs 长期以来一直是自由软件基金会的旗舰产品。它是由理查德·斯托曼（Richard Stallman）博士为 GNU 工程开发的第一个自由软件。在所有目前已开发的 GNU 软件中，GNU Emacs 的作用和地位是非常突出的，因为几乎所有其他的自由软件基金会的工具都是用GNU Emacs 编写出来的。

从编程实践上看，GNU Emacs 有许多特点。其中最为突出的一个特点是斯托曼在创造 GNU Emacs 编辑器时非常巧妙地揉合了用Lisp 语言和 C 语言编写的代码。

斯托曼利用 Lisp 语言编写 GNU Emacs 的大部分代码不是偶然的。Lisp 语言发明于20世纪50年代，并广泛地应用于人工智能研究领域，而斯托曼早年曾经在麻省理工学院人工智能实验室工作过很长时间，所以他非常熟悉 Lisp 语言的优点。Lisp 是解释性的语言，用 Lisp 开发的程序具有良好的可读性，因此将它用于处理文本编辑这样的任务是非常合适的。当然解释性的语言与硬件直接作用时其效率与编译性的语言相比则显得不高，这样的任务还是由 C 语言代码来完成比较合适。

斯托曼的一个天才构想就是利用 C 语言编写与硬件直接作用的 GNU Emacs 模块(如显示模块)，而绝大多数文本编辑模块则统统利用 Lisp 语言来编写。Lisp 语言是一种功能全而的编程语言，其解释器被嵌入了 GNU Emacs 中后，用户便可以用它自行对 GNU Emacs 进行定制。这一几乎无限的灵活性是其他编辑器很难做到的。在 GNU Emacs中，Emacs的Lisp代码模块和 C 代码模块组织良好，它们相互取长补短，相得益彰。

为了保持源代码的可读性与一致性，斯托曼将 GNU Emacs 中的 C 语言代码模块的函数名写得很像 Lisp 函数名。如果不仔细研究模块底层的细节，那么实在很难将它们两者区分开。实际上，如果没有特殊的目的，作者希望你不要去辨认它们之间的区别。这样，在扩充和维护代码时，工作就会变得容易多了。经过这么一番匠心独运的安排，GNU Emacs 成为了一种“高级的、自带文档的、可定制的和可扩充的实时显示的编辑器”。难怪如此众多的自由软件开发人员整天都可以坐在计算机旁运行 GNU Emacs 而乐此不疲。

自从 GNU Emacs 问世以来，众多专家一致评价说：斯托曼的这一天才的泛对称设计思想极富艺术性，具有方法论研究的永久价值。

由于使用GNU Emacs 的开发人员数量众多，运行的平台又很广泛，因此各种使用GNU Emacs Lisp 编写的增强功能包也源源不断地产生了，有些功能包还成为标准GNU Emacs 发行版本的一部分。GNU Emacs 今天仍处在不断的演进和完善过程中，它代表着一种文化。这一切都是由于斯托曼当年的天才设计思想所引发的良好局而。

当然，要了解和运用这些新功能，或者自己动手开发所希望的特性来增强和扩充 GNU Emacs，使用 GNU Emacs Lisp 编程是必不可少的，而通过本书学习编程可以说是一个良好的起点。

本书的作者，罗伯特·卡塞尔（Robert Chassell）先生是自由软件基金会的合创人之一，也是斯托曼青年时期结交的挚友。他精通 GNU Emacs Lisp 的每一个方面。在这本编程入门著作中，他循序渐进地介绍了 GNU Emacs Lisp 编程的各种基础知识和方法，文笔流畅、讲解透彻，对GNU Emacs 用户提高对它的理解和运用帮助极大。

另外，由于各种 Lisp 解释器大同小异，因此，一旦通过这一教程理解了 GNU Emacs Lisp 的工作原理，那么你所掌握的知识和技巧对于学习其他版本的 Lisp 语言（如 AutoLisp）或者现在日趋流行的Python等解释性语言也会具有触类旁通的指导意义。

这部著作的中文版是自由软件基金会中国研究院组织人员翻译的第一本 GNU自由软件文档。我们出版这一著作以及《*GNU Emacs* 技术手册》、《*GNU Emacs Lisp* 技术手册》等中文版的主要目的就在于更好地引导读者体会编写程序、分析自由软件源代码的乐趣。这对于在中国催生新的黑客具有极大的促进作用，因为我相信黑客道的真正本质就是“热爱编写程序、并享受通过编写程序而变得更加聪明这一过程”。

自由软件基金会中国研究院
洪 峰
fred@mail.rons.net.cn

前　言

GNU Emacs 文本编辑器的绝大多数代码是用一种被称为 Emacs Lisp 的编程语言编写的。用这种语言编写的代码就是这个软件——指令集——用户通过它向计算机发布命令以告诉计算机如何工作。Emacs 就是为使你能用 Emacs Lisp 编写新的代码并能方便地作为编辑器的扩展部分来安装而设计的。这也是为什么 Emacs被称作“可扩展的编辑器”的原因。

因为 Emacs 的确提供了比编辑更多的功能，它或许应当被称为“可扩展的计算环境”，但是这个词显得口气太大。同样，在 Emacs 中做的任何事情——查找玛雅年代和月相、简化多项式、调试代码、管理文件、阅读信件以及撰写图书——所有这些活动都是“编辑”这个词所包含的。

虽然人们经常将 Emacs Lisp 与文本编辑器联系到一起，但它却是一种完整的计算机编程语言。可以像使用任何其他编程语言一样使用它。

也许你希望理解编程；也许你希望扩展 Emacs；或者也许你希望成为一名程序员。这本入门教程就是为你开始 Emacs Lisp 之旅而设计的：引导你学习编程基础，更重要的是告诉你如何自学提高。

在整本书中，你将看到为数不多的几个程序例子，你可以在 Emacs 中运行它们。如果用 GNU Emacs 的 Info 阅读本书文档，可以在例子程序出现时运行它们。（这很容易做到，我们将在例子出现时作进一步解释。）同时，你也可以将这本教程作为一本印刷的图书一样，当你坐在计算机旁运行 Emacs 时阅读。（这就是我所喜欢的方式，我喜欢印刷出来的纸版图书。）如果你身边没有一个运行的Emacs，你仍旧可以阅读这本书，但是在这种情况下，最好将其作为一本小说或者是一本你未到过的某个国家的导游手册来阅读：这样读起来会较有趣，但是你的收获会与亲身体验不同。

本教程用许多篇幅介绍 GNU Emacs 用到的代码。教程的这些设计安排有两个目的：一是使读者熟悉真实的正在运行的代码；二是使读者熟悉 Emacs 工作的方式。弄清一个编辑器如何工作是很有趣的。同样，我希望读者养成浏览源代码的习惯。读者可以从中学习并开阔思路。有了 GNU Emacs，就像拥有一个龙穴宝藏一样。

除了将 Emacs 当做一个编辑器、将 Emacs Lisp 当做一门编程语言学习之外，书中的例子和导引将使读者通晓将 Emacs 作为Lisp 编程环境的机会。GNU Emacs 支持编程，并提供了你将乐于使用的工具，如 “M-”（这是调用 `find-tag` 命令的键）。你还可以学习缓冲区和其他对象，这些都是编辑环境的组成部分。学习 Emacs 的这些功能就像熟悉家乡周围的新路一样。

最后，我希望传授一些使用 Emacs 来学习编程时读者不知道的技巧。你可以经常用Emacs 来解决那些困扰你的问题，并用它们做一些新奇的事情。这种自力更生不仅是一种乐趣，更是一种优点。

读者对象

这本教程是作为入门读物为那些非编程人员编写的。如果你是一名程序员，可能并不满足

这本初级读物。原因在于你可能已经通过阅读参考手册成了专家。或许本书的组织方式已经使你失去兴趣。

一位评论过本书的编程专家曾这样对我说：

> 我更喜欢从参考手册中学习（编程）。我“潜入”每一个段落，并在段落之间跃出“水面”呼吸空气。
>
> 当到达一个段落结尾时，我假定这个主题已经结束。我知道了需要知道的所有东西（也可能存在这样的可能性，那就是下一个段落将对这个主题作更加详细的讲解）。我期望一份认真撰写的参考手册不要出现太多的冗余，并且它应指引我学习所希望的知识。

这本入门教程并不是为这类读者撰写的！

首先，我试图就每一件事情至少说上三遍：第一次介绍它；第二次在文中详细展现它的内容；第三次在不同的地方揭示它，或者复习一下。

其次，我几乎从不将一个主题的所有内容放在一个地方讲完，更不放在某一段中。以我思考的方式而言，那样做会给读者强加过重的负担。相反，我试图仅仅解释在那种情况下你需要知道的东西（有时会增加一点点附加信息，在后面读到这些附加信息的正式介绍时无需惊讶）。

阅读本书的时候，我并不指望你第一次就学会所有的东西。通常的情况是你仅需要对某些内容略微了解。我希望已经组织好本书，为你提供了足够的信息，并提醒你哪些是重要的线索，且着重讲述它们。

你应当“潜入”某些段落，除此以外没有其他方法。但是我已尽力减少这类段落。希望本书成为一座可以攀越的小山，而不是一座使人畏缩的高峰。

《*GNU Emacs Lisp* 编程入门》还有一个姊妹篇，那就是《*GNU Emacs Lisp* 技术手册》。那本手册比本书更详细。在那本手册中，关于任何一个话题的所有信息都集中在一个地方。如果你喜欢上面引用的那位程序员所欣赏的学习方法，那么应当掉头去阅读那本技术手册。当然，阅读完这本编程入门后，在编写自己的程序时，你会发现那本技术手册很有用。

Lisp 的历史

Lisp 是20世纪50年代晚期在麻省理工学院为研究人工智能而被首先发展起来的。Lisp语言的强大功能使之也能用于其他目的，比如编写编辑器命令。

GNU Emacs Lisp 在很大程度上得益于20世纪60年代在MIT编写的 MacLisp 。它同时也得益于在20世纪80年代成为标准的 Common Lisp。然而，Emacs Lisp 比 Common Lisp 简单得多（标准的 Emacs 发行版本中包含一个可选的扩展文件“`cl.el`”，它为 Emacs Lisp 增加了许多 Common Lisp 的特性）。

初学者注意

如果你不知道 GNU Emacs，阅读本书仍旧有益。但是，如果仅仅是想学习在计算机屏幕上如何操作，建议你学习 Emacs。可以通过在线教程自学如何使用 Emacs。为使用在线教程，按下组合键 `C-h t`（这意味着同时按下并释放 CTRL和 `h` 键，然后按下并释放 `t` 键）。

同样，我经常在提到 Emacs 的标准命令时列出激活该命令时应按下的键序列，然后在括号

中给出命令名，例如：M-C-\（indent-region）。这意味着indent-region命令通常是通过输入键序列：M-C-\ 来激活的（如果你愿意的话，可以改变激活这个命令的键序列，这称作“重新绑定”。参见16.11节，“键图”）。缩写 M-C-\ 意味着同时输入 META 键、CTRL 键和 \ 键。有时，像这样的一个组合键也叫做一个键和弦，因为它类似于在钢琴上演奏一个和弦。如果你的键盘没有 META 键，可以用前缀 ESC 键取代它。在这种情况下，M-C-\ 意味着按下并释放 ESC 键，然后同时按下并释放 CTRL 键和 \ 键。

如果用 GNU Emacs 的 Info 阅读这份文档，只用空格键就能翻阅整本书（可以输入 C-h i 然后选择 Info 来学习）。

关于术语的说明：当仅仅提到 Lisp 这个词时，常常是指各变种的Lisp；但是当提到 Emacs Lisp 时，就是特指 GNU Emacs Lisp 了。

致谢

感谢所有对这本书提供了帮助的人们。特别感谢 Jim Blandy、Noah Friedman、Jim Kingdon、Roland McGrath、Frank Ritter、Randy Smith、Richard M. Stallman 和 Melissa Weisshaus。同时要感谢 Philip Johnson 和 David Stampe耐心的鼓励。书中的所有错误都由我负责。

目 录

第1章 列表处理

对那些没有学过Lisp语言的人而言，Lisp是一种奇怪的编程语言。在Lisp代码中到处都是括号。有些人把Lisp这个词当成是“Lots of Isolated Silly Parentheses”（大量分离的愚蠢的括号）的缩写。但是这种说法是没有根据的。Lisp是指“LISt Processing”（列表处理），和通过把列表放置在括号之间来处理列表（甚至是列表的列表）的编程语言。括号标记了列表的边界。有时一个列表用一个单引号或表示标记“'”开头。列表是 Lisp 的基础。

1.1 Lisp 列表

在 Lisp 中，一个列表看起来像这个样子：`'(rose violet daisy buttercup)`。这个列表以单引号开始。这个列表也可以写成下面这种你可能比较熟悉的形式：

```
'(rose
  violet
  daisy
  buttercup)
```

在这个列表中，元素是四种不同的花的名称，它们之间用空格分隔开，并用括号括起来，就像花在一个用石头墙围起来的花园中一样。

列表中也可以有数字，如列表（+ 2 2）一样。这个列表有一个加号“+”，后接两个“2”，它们之间用空格分隔开。

在 Lisp 中，数据和程序都以同样的方式表示；也就是说，它们都是由空格分隔的、由括号括起来的单词、数字或者其他列表的列表。（因为如果一个程序看起来像数据，那它就很容易作为其他程序的数据；这是 Lisp 的一个很有用的特性。）(附带提一下，对前面这对括号而言，它不是 Lisp 列表，因为其中使用了标点符号“；”和“。”来分隔不同的元素。)

下面是另一个列表，这个列表中有另外一个列表：

```
'(this list has (a list inside of it))
```

这个列表的元件是单词“this”、“list”、“has”和内部列表“(a list inside of it)”。内部列表由“a”、“list”、“inside”、“of”和“it”几个词组成的。

1.1.1 Lisp 原子

在 Lisp 中，我们刚才说到的词被称作原子（*atom*）。这个术语来自原子一词的历史含义，即原子意味着“不可分”。只要提到 Lisp，我们在列表中使用的词就不可以再被分成更小的部分，这在程序中也一样。数字、单个字符(如“+”)都是如此。另一方面，不像原子，一个列表可以拆分成不同的部分。（参见第7章，“基本函数：*car*、*cdr* 和 *cons*”。）

在一个列表中，原子是由空格一一分隔的。原子可以紧接着括号。

从技术上说，Lisp中的一个列表有三种可能的组成方式：括号和括号中由空格分隔的原子；括号和括号中的其他列表；括号和括号中的其他列表及原子。一个列表可以仅有一个原子或者完全没有原子。一个没有任何原子的列表就像这样：()，它被称作空列表。与所有的列表都不同的是，可以把一个空列表同时看作既是一个原子，也是一个列表。

原子和列表的书面表示都被称作符号表达式（*symbolic expression*），或者更简洁地被称作*s-*表达式(*s-expression*)。表达式这个词，既可以指书面的表示，也可以指一个原子或者一个列表在计算机中的内部表示。人们常常无区别地使用表达式这个词。（同样地，在许多书中，表格（*form*）这个词也被看作是表达式的同义词）。

顺便说一下，构成我们的宇宙的原子是在它们被认为是不可分的时候命名的。但是，人们已经发现，物理上的原子不再是不可分的。原子的一部分可以被分出来，或者可以裂变成大致相等的两个部分。物理上的原子在它们的更真实的本质被发现之前就已被过早地命名。在 Lisp 中，某种类型的原子，例如一个数组，可以被分成更小的部分，但是分割数组的机制与分割列表的机制是不同的。只要是涉及列表操作，列表中的原子就是不可分的。

与英语中一样，Lisp原子的组成字母的意义与由这些字母构成的单词的含义是不同的。例如，代表“South American sloth”的单词“ai”与“a”和“i”这两个字母是完全不同的。

自然界中有许多种原子，但是在 Lisp 中只有几种原子：例如，数字(比如“37”、“511”或“1729”)和符号(比如“+”、“foo”和“forward-line”)。以上列出的这些单词都是符号。在 Lisp 的日常使用习惯中，“原子”一词不太常用，因为程序员经常试图更明确地表示他们处理的原子类型。Lisp编程几乎都是关于列表中的符号的(且有时是关于数字的)。（附带说明一下，上述3个单词[⊖]是 Lisp 中一个正确的列表，因为它包含的是原子。在这种情况下，原子是一些由空格分隔、用括号括起来的符号，其中没有任何对 Lisp 而言是非法的标点符号。）

另外，双引号中的文本——不论是句子或者是段落——都是一个原子。下面是一个这样的例子：

```
'(this list includes "text between quotation marks.")
```

在 Lisp 中，所有用双引号括起来的文本，包括标点符号和空格，都是单个原子。这种原子被称作串(*string*)（代表“字符串”之意），并且它是一种事物的分类，以便让计算机能够打印出可供阅读的信息。字符串是不同于数字和符号的一种原子，在使用上也是不同的。

1.1.2 列表中的空格

列表中空格的数量无关紧要。从 Lisp 语言的角度来说，

```
'(this list
  looks like this)
```

与下面的列表完全等价：

```
'(this list looks like this)
```

⊖ 这里是指英文版中前面括号中的3个单词，即“(and sometimes numbers)”。——译者注

上面两个例子对 Lisp 而言是同一个列表。这个列表由符号“this”、“list”、“looks”、“like”、“this”按上面这种顺序组成。

多余的空格和换行符只不过是为了使人们易于阅读而设计的。当 Lisp 读取表达式时，它剔除了所有多余的空格（但是原子间至少需要一个空格以使原子分隔开来）。

这看起来很奇怪，我们已经看到的例子涵盖了 Lisp 列表的几乎所有情况。其他任何一个 Lisp 列表看起来都或多或少与上面的例子相似，只是列表可能更长更复杂。简要地说，列表放在括号之间，串放在引号之间，符号看起来像一个单词，而数字看起来就像一般的数字一样。（当然，对于特定的情况，方括号、大括号、句点和一些特殊的字符都是可以使用的，然而我们暂时先不去理会它们。）

1.1.3 GNU Emacs 帮助你输入列表

如果你在 GNU Emacs 中使用Lisp交互模式或是Emacs Lisp模式来输入一个 Lisp 表达式，那么你将可以使用多种命令来使 Lisp 表达式排成易于阅读的格式。例如，按TAB键会使光标所在的行自动缩排到适当的位置。用于在一个区域内正确缩排的常用命令是 M-C-\。设计缩排，是为了使读者看清列表的都些元素是属于哪个列表的——缩排时子列表的元素要比外围列表的元素更缩进一些。

另外，当输入一个右括号时，Emacs 立即使光标跳到与之配对的左括号处，因此就可以看清它到底是哪一个列表。这个功能非常有用，因为 Lisp 中输入的每一个列表必须有一对匹配的左括号和右括号（有关 Emacs 模式的详细情况可以参见《*GNU Emacs*技术手册》中“主要模式”一节）。

1.2 运行一个程序

Lisp 中的一个列表——任何列表——都是一个准备运行的程序。如果你运行它（在 Lisp 的术语中，这称为*求值*），计算机将完成三件事情：只返回列表本身；告诉你一个出错消息；或者，将列表中的第一个符号当做一个命令，然后执行这个命令。（当然，你通常真正希望的是上述三件事情中的最后一件）。

单引号“'”，也就是在前一节例子中的列表前面的引号，被称作一个*引用*（*quote*）。当单引号位于一个列表之前时，它告诉 Lisp 不要对这个列表做任何操作，而仅仅是按其原样。但是，如果一个列表前面没有引号，这个列表中的第一个符号就很特别了：它是一条计算机要执行的命令（在 Lisp 中，这些命令被称作*函数*）。上面说到的列表（+ 2 2）就没有引号在前面，因此 Lisp 将“+”号理解为一条指令，用来对这个列表的其余部分进行操作；在这种情况下，就是将其后续的数字相加。

如果在 GNU Emacs 的 Info 中阅读到这个列表，可以这样对它求值：将光标移到下面列表的右括号之后，然后按C-x C-e：

```
(+ 2 2)
```

你将看到数字 4 显示在回显区。（用术语来说，刚才做的就叫做：“对一个列表求值”。回显区是屏幕底部的那一行，它显示或者“回显”文本。）现在，对下面带引号的列表进行同样的操

作，将光标置于下面列表之后，然后按 C-x C-e:

```
'(this is a quoted list)
```

此时，将会看到（this is a quoted list）显示在回显区。

在这两种情况下，你所做的是给GNU Emacs内一个叫做 *Lisp* 解释器的程序一个命令，即给解释器一个命令，使之求值。Lisp 解释器的名字来自于由一个人来完成某项任务这个词，这个人给出一个表达式的值，即他解释了它。

同样可以对一个不是列表的一部分的原子(即不被括号括起来的原子)求值。同样，Lisp 解释器将人能理解的表达式翻译成计算机的语言。但是在讨论这个问题之前（参见1.7节，“变量”），我们先讨论在出错时 Lisp 解释器会做些什么。

1.3 产生错误消息

如果不小心出了错，也不要太担心。现在我们将给 Lisp 解释器一个命令，使之产生一个错误消息。这是一个无害的动作，确实，我们时常会有意识地产生错误消息。一旦理解了这种术语，错误消息是能提供有用信息的。与其说是错误消息，不如说是有助的消息。它们像是一个给在异国他乡的游客的路标，破译它们可能很艰难，但是一旦理解了，它们就成了指路明灯。

我们将要做的，就是对一个没有引号并且其第一个元素不是一个有意义的命令的列表求值。下面是一个与我们用到过的列表几乎完全相同的列表，但是它前面没有单引号。将光标移到它后面并输入 C-x C-e:

```
(this is an unquoted list)
```

这一次，将会看到下面的内容显示在回显区：

```
Symbol's function definition is void: this
```

(另外，终端可能对你发出鸣叫声——有些终端这样做，有些不这样。有些则闪烁。这仅仅是一个示警的装置。)只要键入任何键，消息都将迅速消失，哪怕是仅仅移动了光标。

根据已有的知识，我们几乎可以读懂这条错误消息。我们知道“Symbol”一词的意义。在这种情况下，它指列表中的第一个原子，就是“this”一词。上述错误消息中的“function”一词在前面已经出现过一次。它是非常重要的一个词。对我们的目的而言，可以将它定义为：一个“函数”(function)就是一组告诉计算机做什么的计算机指令（从技术上说，这个符号告诉计算机到什么地方去寻找这些指令，但是这是一个我们暂时可以忽略的复杂问题）。

现在，我们可以理解这条错误消息了：“Symbol's function definition is void:this”。其中的“Symbol”是指“this”。这个错误消息是指没有为“this”定义让计算机执行的任何指令。

这条错误消息中稍显奇怪的用词“function definition is void”，是 Emacs Lisp 实现方式的体现：即当一个符号没有一个对应的函数定义时，那个应当包含指令的位置就是“空的”（void）。

另一方面，因为可以成功地通过对表达式（+ 2 2）求值来执行 2加 2计算，那就是说，+ 号一定有一组计算机执行的指令，这些指令就是将 + 号后面的数字加起来。

1.4 符号名和函数定义

在已经讨论过的内容的基础上，我们可以结合介绍 Lisp 的另外一个特性。这个重要的特性就是，一个符号，如 + 号，它本身并不是计算机执行的指令本身。相反，符号或许是临时用于定位函数或者一组指令的。我们所看到的只不过是一个名字而已，通过这个名字可以找到相应的指令。人的名字起着同样的作用。我可以被叫做“Bob”，然而，我并不是“B”、“o”、“b”这几个字母，而是与这个特定的生命形式相联系的有意识的人。这个符号不是我，但是它可以用于指我。

在 Lisp 中，一组指令可以连到几个名字，例如，计算机的加法指令可以连接到符号“Plus”，也可以连接到符号“+”。在人类社会中，我可以叫做“Robert”，也可以叫做“Bob”，或者其他什么词。

另一方面，一个符号一次只能有一个函数定义与其连接，否则，计算机就会疑惑到底使用哪个函数。如果在人群中出现这种情况，那么只有一个人可以叫做“Bob”。然而，一个名字指向的函数定义是容易改变的。(参见3.2节“安装函数定义”。)

因为 Emacs Lisp 很大，它常以一定的方式将符号命名，这种命名方法可以确定函数属于 Emacs 的哪一个部分。因而，处理 Texinfo 的所有函数的名字都以“texinfo-”开头，所有用于阅读电子邮件的函数的名字都以“rmail-”开头。

1.5 Lisp 解释器

根据前面的内容，现在可以来说明在我们命令Lisp 解释器对一个列表求值时它做些什么了。首先，它查看一下在列表前面是否有单引号。如果有，解释器就为我们给出这个列表。如果没有引号，解释器就查看列表的第一个元素，并判断它是否是一个函数定义。如果它确实是一个函数，则解释器执行函数定义中的指令。否则解释器打印一个错误消息。

这就是 Lisp 工作的方式。简单极了。后面我们会介绍更复杂一些的内容，但是这些是基本的。当然，为了编写 Lisp 程序，需要知道如何书写函数定义并将它们连向函数名，以及如何使得这样做不使自己和计算机都搞混。

现在，我们介绍第一种复杂的情况。除了列表之外，Lisp 解释器可以对一个符号求值，只要这个符号前没有引号也没有括号包围它。在这种情况下，Lisp 解释器将试图像变量一样来确定符号的值。(参见1.7节，“变量”。)

出现第二种复杂的情况是因为一些函数异常并且以异常的方式运行。那些异常的函数被称作特殊表（*special form*）。它们用于特殊的工作，例如定义一个函数。但是这些特殊表并不多。在下面几章，你将接触几个更加重要的特殊表。

第三种也是最后一种复杂的情况是：如果 Lisp 解释器正在寻找的函数不是一个特殊表，而是一个列表的一部分，则Lisp 解释器首先查看这个列表中是否有另外一个列表。如果有一个内部列表，Lisp 解释器首先解释将如何处理那个内部列表，然后再处理外层的这个列表。如果还有一个列表嵌入在内层列表中，则解释器将首先解释那个列表，然后逐一往外解释。它总是首先处理最内层的列表。解释器首先处理最内层的列表是为了找到它的结果。这个结果可以由

包含它的表达式使用。

否则，解释器从左到右工作，一个表达式接一个表达式地进行解释。

字节编译

解释的另一个方面是：Lisp 解释器可以解释两种类型的输入数据：人可以读懂的代码（我们将着重关注这种代码）和经过特殊处理的、被称作字节编译（*byte compiled*）的代码。字节编译代码是人无法读懂的。字节编译代码比人能读懂的代码运行得更快。

可以通过运行一个编译命令（如byte-compile-file）将人能读懂的代码转换成字节编译代码。字节编译代码经常储存在一个文件中，这个文件以“.elc”作为扩展名，而不是以“.el”作为扩展名。可以在“emacs/lisp”目录中看到这两种文件。可以阅读的文件是扩展名为“.el”的文件。

实际上，用户可能做的绝大多数事情是定制或者扩展 Emacs，对于这些事情无需进行字节编译，在这里不讨论这个主题。关于字节编译的完整描述，可以参见《*GNU Emacs Lisp*技术手册》中的“字节编译”一节。

1.6 求值

当 Lisp 解释器处理一个表达式时，这个动作被称作“求值”。我们称，解释器计算表达式的值。在前面我已经数次使用了这个术语。这个词来自于日常用语之中，根据《*Webster's New Collegiate Dictionary*》的解释，它意指“确定值或数量，或者是“评价”。

完成表达式的求值后，Lisp 解释器几乎总是要返回一个值，这个值是计算机执行它在函数定义中找到的指令的结果，或者它将放弃那个函数并产生一个错误消息。（解释器也会发现，有时它进入了死胡同，也就是说，解释器执行另外一个函数，或者它试图一次又一次不断地重复它在一个“无穷循环”中的操作。这些操作并不常见，可以忽略它们。）通常，解释器返回一个值。

在解释器返回一个值的同时，它也可以做些其他什么事情，例如移动光标或者拷贝一个文件，这种动作称为*附带效果*（*side effect*）。我们认为的重要事情，如打印一个文件，对 Lisp 解释器而言常常是一个附带效果。这个行话显得有些奇特，但是它表明学习使用附带效果是相当容易的。

总之，对一个符号表达式求值几乎总是使 Lisp 解释器返回一个值，同时可能产生一个附带效果，不然，就产生一个错误消息。

对一个内部列表求值

如果是对一个嵌套在另一个列表中的列表求值，对外部列表求值时可以使用首先对内部列表求值所得的结果。这解释了为什么内层列表总是首先被求值的：因为它们的返回值被用于外部表达式。

通过对下面这个例子求值，可以深入理解这个过程。将光标置于下面的表达式的末尾，并键入 C-x C-e：

```
(+ 2 (+ 3 3))
```

数字 8 就会显示在回显区。

这里发生的事情就是，Lisp 解释器首先对内部列表（+ 3 3）求值，它的返回值是 6；然后解释器计算外部表达式的值，就像对列表（+2 6）求值一样，这次返回 8。至此，因为没有其他更多的外围表达式需要被求值，因此解释器将这个值打印到回显区。

现在，可以容易理解通过键入 C-x C-e 发出的命令的含义：这个命令的名称就是 eval-last-sexp。其中“sexp”是“symbol expression”(符号表达式)的缩写，“eval”是“evaluation”(求值)一词的缩写。这个命令就是指“对最近一个符号表达式求值”。

作为实验，可以将光标置于表达式下面一个空白行的开始，或者置于表达式中间，然后执行求值命令。

同样使用上面的表达式：

```
(+ 2 (+ 3 3))
```

如果将光标置于表达式下面一个空白行的开头并键入 C-x C-e，数字 8 仍将显示在回显区。现在将光标移动到表达式的内部。如果将光标移动到倒数第二个括号之后，即光标覆盖在最后一个括号上，执行求值命令将看到数字 6 显示在回显区。因为求值命令是对表达式（+ 3 3）求值的。

现在将光标立即置于一个数字之后。键入 C-x C-e，将得到这个数字本身。在Lisp 中，如果对一个数字求值，将得到这个数字本身——这就是数字区别于符号的地方。如果对一个以 + 号开头的列表求值，将得到计算机执行这个符号名所附带的函数定义中的一组指令的结果。如果一个符号本身被求值，那么就会发生一些不同的事情，这一点将在下一节介绍。

1.7 变量

在 Lisp 中，可以将一个值赋给一个符号，就像将一个函数定义赋给一个符号那样。这两者的含义是不同的。函数定义是一组指令，这组指令是由计算机执行的。另一方面，一个值，比如一个数字或者一个名字，是可以变化的（这就是为什么称其为变量的原因）。一个符号的值可以是 Lisp 中的任意表达式，如一个符号、一个数字、一个列表或者一个字符串。有值的一个符号通常被称作一个变量(*variable*)。

一个符号可以同时具有一个函数定义和一个值。这两者是各自独立的。这有点像“Cambridge”一词，既可以指那个在麻省的Cambridge市，也可以指其他赋予这个名字的信息，比如“伟大的编程中心”。

对这个问题的另外一种思考方式是：将符号设想为一个有许多抽屉的柜子。函数定义放在一个抽屉中，值放在另外的抽屉中，等等。放在抽屉中的值可以在不影响其他抽屉中存放的函数定义的情况下被改变，反过来也一样。

变量 fill-column 展示了一个具有值的符号，在每一个 GNU Emacs 缓冲区中，这个符号被赋予一些值，通常是 70 或者 72，但有时被赋予别的一些值。为了从这个符号中找到其中的值，对它本身求值即可。如果在 GNU Emacs 的 Info 中阅读这份文档，可以将光标移动到这个符号的后面，并键入C-x C-e：

```
fill-column
```

在键入C-x C-e之后，Emacs 将数字 72 打印在回显区中，这个值就是在我写这本书的时候为我设置的 fill-column 的值。对你而言，在你的 Info 缓冲区中这个值可能不一样。注意，作为一个变量的返回值被打印在回显区中，与执行一个函数定义的一组指令的返回值被打印在回显区中是完全一样的。从 Lisp 解释器的角度来看，一个返回值就是一个返回值而已。这个返回值究竟来自于何种表达式，这个问题一旦在这个值被求出后便已经不再重要了。

任何值都可以赋给一个符号，用术语来说，就是将变量与一个值绑定（*bind*）起来：绑定到一个数字，如 72；绑定到一个字符串，如“such as this”；绑定到一个列表，如(spruce pine oak)。甚至可以将一个变量绑定到一个函数定义上。

一个符号可以用几种方法与一个值绑定。其中一种方法请参见1.9节，“给一个变量赋值”。

注意，在我们对fill-column 变量求值时，这个单词的两边没有括号。这是因为我们并不希望将它当做一个函数名使用。如果 fill-column 是一个列表仅有的一个原子或者第一个原子，Lisp 解释器将试图寻找与之相联系的函数定义。但是fill-column 没有函数定义。试一试对下面的表达式求值：

```
(fill-column)
```

将得到这样一个错误消息：

```
Symbol's function definition is void: fill-column
```

符号无值时的错误消息

如果试图对一个没有赋值的符号求值，将收到一个错误消息。可以试一试 2 加 2 的加法。在下面的表达式中，将光标紧挨在 + 号后面，并在第一个 2 前而，键入 C-x C-e：

```
(+ 2 2)
```

将得到这样一个错误消息：

```
Symbol's value as variable is void: +
```

这个错误消息与我们看到过的错误消息“Symbol's function definition is void: this”不同。在这个例子中，没有被赋值的符号被当做一个变量。在前而那个情况下，符号“this”没有函数定义。

在这个关子加号的实验中，我们所做的是使 Lisp 解释器对 + 号求值，并寻找这个变量的值而不是寻找其函数定义。我们是通过将光标置于符号的后而而不是像前面那样置子闭合列表的括号后面来实现的。相应地，Lisp 解释器对前面的符号表达式求值，在这个例子中就是 + 号本身。

因为 + 号没有与之绑定在一起的值，而只有一个函数定义，因此错误消息就报告作为一个变量，+ 号的值是空的了。

1.8 参量

为了理解信息是如何传送给函数的，让我们再看看上面多次提到的那个函数：2加 2 之和。在 Lisp 中，这写成：

```
(+ 2 2)
```

如果对这个表达式求值，数字 4 将出现在回显区中。Lisp 解释器所做的是将加号后面的数字加起来。

由 + 号相加的数字被称为 + 函数的参量。这些数字就是给予或者传递给函数的信息。

“参量”（argument）[⊖]一词来自于它在数学中的应用，而不是指两个人之间的争论。相反，它指传递给函数的信息，在这个例子中就是传递给 + 函数。在Lisp 中，一个函数的参量是函数后面的原子或者列表。通过对传递给函数的原子或者列表求值，得到返回值。不同的函数需要不同数目的参量；有些函数根本不需要参量。[⊜]

1.8.1 参量的数据类型

应当传递给函数的数据的类型依赖于它使用什么信息。像+函数这样一个函数，其参量必须有数字类型的值，因为 + 意味着要将数字加起来。其他函数使用不同类型的数据作为它们的参量。

例如，concat函数将两个或者更多的字符串连接起来，产生一个新的字符串。这时参量的类型是字符串。将两个字符串“abc”和“def”连接起来就生成一个新的字符串“abcdef”。这可以通过对下面的表达式求值得到：

```
(concat "abc" "def")
```

求值得到的这个表达式的值是“abcdef”。

一个函数（如 substring），既使用字符串也使用数字作为参量。这个函数返回字符串的一部分，即函数第一个参量的一个子字符串。这个函数需要三个参量，第一个参量是一个字符串，第二个参量和第三个参量是指明子字符串开始和结束位置的数字。数字是指从字符串的首字符位置开始计数的（包括空格和标点符号）。[⊜]

例如，如果对下面的表达式求值：

```
(substring "The quick brown fox jumped." 16 19)
```

将在回显区中看到“fox”。在这个例子中，参量就是一个字符串和两个数字。

注意，传递给 substring 函数的字符串是一个单原子，虽然它由几个被空格分开的单词组成。Lisp 将引号中的所有内容作为串的一部分（包括空格和标点符号）进行计数。可以将 substring 函数当做一种“原子分裂器“，因为它接收其他不可分的原子，抽取其中的一部分。然而，substring函数仅能从一个字符串参量中抽取子字符串，而不是从其他类型的原子中抽取，如从一个数字或者一个符号抽取。

⊖ argument有时也译作“变元”，本书中采用“参量”的译法。——译者注

⊖ 追溯“argument”一词有两种不同含义的过程是有意思的，一种来源于数学，一种出自日常用语。按照《牛津英语词典》，“argument”一词采自拉丁语，表示“澄清，证实”；因此，按照这种词源线索，它具有“提供证据”的含义，即是“提供的信息”，这就是在Lisp中引伸出的意义。但是，按其他词源线索，它又表示“某种论断的方式，这种论断可能引起其他相反的断言”，这就是这个词所包含的争论的意思。(注意，这个单词同时具有两个不同的定义。对比之下，在Emacs Lisp中，一个符号不能同时具有两个不同的函数定义。)

⊜ 注意计数是从0开始的，在下例中，字符“T”就是第0个数字。—译者注

1.8.2 作为变量和列表的值的参量

参量可以是一个符号，对这个符号求值将返回一个值。例如，当符号fill-column被求值时，它返回一个数字。这个数字能被用于加法之中。将光标置于下面的表达式之后，并键入C-x C-e:

```
(+ 2 fill-column)
```

其返回值是一个数，它比你单独求 fill-column 的值大 2。对我来说，就是 74。因为 fill-column 的值是 72。

就像刚才看到的，参量可以是一个符号，当求值时这个符号返回一个值。另外，参量也可以是一个列表，当求值时这个列表返回一个值。例如，下面的表达式里，函数 concat 的参量是字符串“The”、“red foxes.”和列表(+ 2 fill-column)。

```
(concat "The " (+ 2 fill-column) " red foxes.")
```

如果对这个表达式求值，“The 74 red foxes.”将显示在回显区中。(注意，必须在“The”之后和“red ”之前输入空格，它们才能给出正确的结果。)

1.8.3 数目可变的参量

有些函数，如 concat、+ 和 *，可以有任意多个参量(* 是乘法符号)。用通常的方法对下面的表达式求值就可以看到这一点。在这本书中在回显区看到的内容将在“⇒”符号后面打印出来，可以将“⇒”符号读作“求值得”。

第一组中的函数没有参量：

```
(+)       ⇒ 0

(*)       ⇒ 1
```

在这一组，每个函数有一个参量：

```
(+ 3)     ⇒ 3

(* 3)     ⇒ 3
```

而在这一组，每个函数有三个参量：

```
(+ 3 4 5) ⇒ 12

(* 3 4 5) ⇒ 60
```

1.8.4 用一个错误类型的数据对象作为参量

当函数的一个参量被传送一个错误类型的数据时，Lisp 解释器产生一个错误消息。例如，+ 函数要求其参量都是数。作为一个试验，可以传送一个带引号的符号 hello 而不是一个数给它。将光标置于下面的表达式之后，并键入 C-x C-e:

```
(+ 2 'hello)
```

当这样做时，就会产生一个错误消息。这里所发生的是：+ 函数试图将数字 2和 'hello 的返回值相加。但是 'hello 的返回值是符号 hello 而不是一个数。只有数才能相加。因此 + 函数不能执行它的加法。

一般地说，在学习了如何阅读错误消息之后，错误消息将是有帮助的，具有提示作用。上例的错误消息是：

```
Wrong type argument: integer-or-marker-p, hello
```

这个错误消息的前面部分是很直接了当的，它就是说“Wrong type argument”(参量类型错误)。后续部分来自神秘术语“integer-or-marker-p”。这是试图告诉你 + 函数期望得到什么类型的参量。

符号integer-or-marker-p的意思是说，Lisp 解释器试图确定提交给它（也就是参量的值）的信息是一个整数（也就是整个数）或者是一个标记（表示一个缓冲区位置的一个特殊对象）。解释器所做的就是测试是否对传递给 + 函数的这个数进行加法运算。它同时也测试这个参量是否是某些叫做标记的东西，这是 Emacs Lisp 的一个特殊的特性。（在 Emacs 中，缓冲区中的位置是以标记来记录的。当执行 C-@ 或者 C-SPC 命令设置标记时，这个位置就被记录为一个标记。这个标记可以被当做一个数，就是从缓冲区开始处到这个位置为止的所有字符数。）在 Emacs Lisp 中，+ 函数可以将标记位置的值拿来当做一个数进行相加。

integer-or-marker-p中的“p”是早期 Lisp 研究人员编程实践的体现。这个“p”字符代表“predicate”（即谓词）。在早期的 Lisp 研究人员使用的术语中，一个谓词是指一个决定某些属性是否为真的函数。因此，“p”告诉我们integer-or-marker-p是一个函数名，这个函数决定当提供的参量是一个整数或者一个标记时是否为真。其他以“p”结尾的 Lisp 符号，包括zerop（这个函数测试参量值是否为零)和listp（这个函数测试参量是否是一个列表）。

最后，错误消息的最后部分是符号hello。这就是传送给 + 函数的参量的值。如果为这个 + 函数传递了正确类型的对象，这个值应当是一个数，如 37，而不是一个像 hello 这样的符号。但是，如果那样的话，你就不会得到一个错误消息了。

1.8.5 message 函数

像 + 函数一样，message 函数的参量数目是可以变化的。它被用于给用户发送消息。它如此有用，因此我们在这里特做一番讲解。

消息是打印在回显区中的。例如，通过对下面的列表求值，就能够在回显区中打印一条消息：

```
(message "This message appears in the echo area!")
```

双引号中的整个字符串是一个参量，它被打印出来。（在这个例子中应注意：引号中的消息本身将显示在回显区中，这是因为你看到的是 message 函数的返回值。在使用 message 函数的绝大多数情况中，在回显区中打印消息只是一个附带作用，而打印出来的消息则是没有引号的。示例请参见3.3.1节，“交互的multiply-by-seven函数”。

然而，如果在带引号的字符串中加有“%s”，message 函数将不打印“%s”，而是去找紧跟在这个字符串后面的参量。它先对第二个参量求值，并将这个值打印到字符串中“%s”出现

的位置。

将光标置于下面的表达式后并键入 C-x C-e，就可以看到上面说的这种情况：

```
(message "The name of this buffer is: %s." (buffer-name))
```

在 Info 中，“The name of this buffer is:*Info*”将出现在回显区。函数 buffer-name以一个字符串的方式返回缓冲区的名字，message 函数将这个字符串插入以取代 %s。

为了输出一个十进制数，可以用类似于“%s”的方式，但使用“%d”来实现。例如，为了在回显区中打印一条告知 fill-column 值的消息，对下面的表达式求值即可：

```
(message "The value of fill-column is %d." fill-column)
```

在我的系统中，当对这个列表求值时，“The value of fill-column is 72.”出现在我的回显区中。

如果在带引号的字符串中有多于一个的“%s”，字符串后的第一个参量的值输出到第一个“%s”的位置，字符串后的第二个参量的值输出到第二个“%s”的位置，以此类推。例如，如果对下面的列表求值：

```
(message "There are %d %s in the office!"
         (- fill-column 14) "pink elephants")
```

一个相当古怪的消息将显示在回显区中。在我的系统上，它是：“There are 58 pink elephants in the office!”。

表达式（- fill-column 14）被求值，其结果在同样的位置替换“%d”，双引号中的字符串“pink elephants”被当做一个参量并替换“%s”。（这就是说，双引号中的串求值后就是它本身，就像一个数一样。）

最后，这里有一个稍微复杂一点的例子。它不仅展示一个数的计算，同时也展示如何能够在一个表达式内部使用另外一个表达式来产生用于替换“%s”的文本：

```
(message "He saw %d %s"
         (- fill-column 34)
         (concat "red "
                 (substring
                  "The quick brown foxes jumped." 16 21)
                 " leaping."))
```

在这个例子中，message 函数有三个参量：字符串“He saw %d %s”、表达式(-fill-column 34)、和一个以 concat 函数开始的表达式。对表达式（- fill-column 34）求值返回的结果被插入以取代“%d”的位置，而以concat函数开始的表达式求出的值被插入以取代“%s”的位置。

当我对这个表达式求值时，消息“He saw 38 red foxes leaping.”显示在我的回显区中。

1.9 给一个变量赋值

有几种方法给一个变量赋值。其中一种方法是使用 set 函数或者使用 setq 函数。另外一

种方法是使用 let 函数（参见3.6节，“let函数”）。（这个过程用术语来说，就是将一个变量绑定到一个值上。）

下面几小节不仅描述 set 和 setq 函数是如何工作的，而且展示参量是如何被传送的。

1.9.1 使用 set 函数

为了将符号flowers的值设置为列表 '（rose violet daisy buttercup），将光标置于下面的表达式之后并键入C-x C-e来对表达式求值：

```
(set 'flowers '(rose violet daisy buttercup))
```

列表（rose violet daisy buttercup）将出现在回显区中。这是 set 函数返回的值。作为一个附带效果，符号 flowers 被绑定到一个列表上，也就是列表作为值被赋给可以被当做变量的符号 flowers。顺便说一下，这个过程，展示了 Lisp 解释器的附带效果（赋值）如何能成为我们感兴趣的主要作用。这是因为每一个 Lisp 函数如果不产生一个错误消息的话，它就必须要返回一个值，但是如果为函数设计一个附带效果的话，它将只有一个附带效果。

对 set 表达式求值之后（即赋值之后），能对符号 flowers 求值，它将返回你刚设置的值。下面就是这个符号。将光标置于它后面并键入 C-x C-e：

```
flowers
```

当对 flowers求值时，列表（rose violet daisy buttercup）显示在回显区中。

附带提一下，如果对带单引号的变量求值，在回显区看到的将是这个符号flowers本身。下面是带引号的符号，你可以试一试：

```
'flowers
```

同样要注意，当使用 set 函数时，需要将 set 函数的两个参量都用引号限定起来，除非你希望它们被求值。在这种情况下，我们不希望任何参量被求值，即不希望变量 flowers 被求值，也不希望列表(rose violet daisy buttercup)被求值，因此它们都带引号。（在使用 set 函数时，如果没有将第一个参量用单引号标明，第一个参量将在所有其他操作执行之前被求值。如果这样做了，而 flowers 又还没有一个值的话，将得到一个错误消息，即“Symbol's value as variable is void”；另一方面，如果 flowers 确实在求值后返回一个值，set 函数将试图设置这个返回的值。这确实是由函数完成的，但是很少这样做。）

1.9.2 使用 setq 函数

实际上，人们几乎总是将 set 函数的第一个参量用单引号标出。set 函数和其第一个带引号的参量的组合是如此常用，以致于它有一个自己的名字：setq 特殊表函数。这个特殊表就像set函数一样，不同之处只在于其第一个参量自动地带上单引号。因此，不必自己键入单引号了。同样，另外一个方便之处在于，setq 函数允许在一个表达式中将几个不同的变量设置成不同的值。

用 setq 函数将变量 carnivores的值设置成列表' (lion tiger leopard)，可以使用下面的表达式完成：

```
(setq carnivores '(lion tiger leopard))
```

这也可以用 set 函数完成，只是在 setq 函数中，变量前自动加上了单引号。(setq 中的“q”就是指引用quote)。用 set 函数，这个表达式是这样的：

```
(set 'carnivores '(lion tiger leopard))
```

同样地，setq 函数也可以用于给不同变量赋给不同值。第一个参量绑定到第二参量的值，第三个参量绑定到第四个参量的值，以此类推。例如，用下面的表达式将树的一个列表赋给符号 trees，将食草动物的一个列表赋给符号 herbivores：

```
(setq trees '(pine fir oak maple)
      herbivores '(gazelle antelope zebra))
```

(这个表达式也可以写在一行上，但是这可能无法打印一张纸上，而且人们发现格式化的列表更易于阅读。)

虽然我已经使用“赋值”一词，但是还有另外一种方式来理解 set 和 setq 函数。那就是，set 和 setq 函数将符号指向列表。后面这种思考方式很常用，在后续几章我们将至少在一个符号中用“指针”作为它名字的一部分。之所以选择这个术语，是因为符号有一个值，特别是一个列表赋给符号，或者用另一种方式说，就是符号“指向”这个列表。

1.9.3 计数

这虽有一个例子演示如何在计数器中使用 setq 函数。可以用这种方法对你的程序的某个部分重复多少次进行计数。首先，将一个变量赋值为 0，然后每当程序自行重复一次就给这个变量加 1。为达到这一目的，你需要一个作为计数器的变量和两个表达式：第一个表达式是将变量赋值为 0 的初始化 setq 表达式，第二个表达式是每次求值时对计数器加 1 的setq表达式。

```
(setq counter 0)                ; Let's call this the initializer.

(setq counter (+ counter 1))    ; This is the incrementer.

counter                         ; This is the counter.
```

(分号后面的内容是注释部分，参见3.2.1节，“改变函数定义”。)

如果对上而这些表达式中的第一个表达式，即对初始化表达式——(setq counter 0)求值，然后对第三个表达式，即对计数器——counter求值，数字 0 将显示在回显区。如果再对第二个表达式，即对递增器——(setq counter (+ counter 1))求使，计数器将得到值 1。因此，如果继续对计数器求值，数字1将显示在回显区中。每当对第二个表达式求值一次，计数器的值就将增加1。

当对递增器——(setq counter (+ counter 1))求值时，Lisp 解释器首先对最内层的列表求值，也就是先进行加法运算。为了对这个表达式求使，它必须对变量counter和数字 1 求值。当对变量 counter求值时，得到 counter变量的当前值。解释器将这个值和数字 1 传送给 + 函数，这个函数将它们加起来。所得的和作为内部列表的返回值传送给 setq 函数。这个函数将counter 变量设置为这个新值。因而，变量 counter 的值就被改变了。

1.10 小结

学习 Lisp 就像登山一样，最初的一段总是最陡峭的。现在，你已经登上几乎是最陡峭的那个部分了。继续向上攀登时，剩下的部分会越来越容易。

总之：

- Lisp 程序由表达式组成，表达式是列表或者单个原子。
- 列表由 0 个或者更多的原子或者内部列表组成，原子或者列表之间由空格分隔开，并由括号括起来。列表可以是空的。
- 原子是多字符的符号（如`forward-paragraph`）、单字符符号（如 + 号）、双引号之间的字符串、或者数字。
- 对数字求值就是它本身。
- 对双引号之间的字符串求值也是其本身。
- 当对一个符号求值时，将返回它的值。
- 当对一个列表求值时，Lisp 解释器查看列表中的第一个符号以及绑定在其上的函数定义。然后这个函数定义中的指令被执行。
- 单引号告诉 Lisp 解释器返回后续表达式的书写形式，而不是像没有单引号时那样对其求值。
- 参量是传递给函数的信息。除了作为列表的第一个元素的函数之外，通过对列表的其余元素求值来计算函数的参量。
- 当对一个函数求值时总是返回一个值（除非得到一个错误消息）。另外，它也可以完成一些被称作附带效果的操作。在许多情况下，一个函数的主要目的是产生一个附带效果。

1.11 练习

几个简单的练习：

- 通过对一个不在括号内的适当符号求值，产生一个错误消息。
- 通过对一个在括号内的适当符号求值，产生一个错误消息。
- 创建一个每次增加 2 而不是 1 的计数器。
- 写一个表达式，当对它求值时，它在回显区中输出一条消息。

第2章 求值实践

在学习用 Emacs Lisp 编写函数定义之前，花很少一点时间对已经学习过的各种表达式进行求值是很有帮助的。这些表达式是以函数作为其第一个（经常是仅有的）元素的列表。因为其中一些与缓冲区相联系的函数既简单又有趣，所以我们就从它们开始讲起。在这一章中，将对这样的一些表达式求值。在另一章中，我们将学习另外几段与缓冲区有关的函数的代码，看看那些函数是如何被编写的。

每当在 Emacs Lisp 中发出一个编辑命令时，比如一个移动光标或滚动屏幕的命令，就是在对一个表达式求值，这个表达式的第一个元素就是一个函数。这就是 Emacs 的工作方式。

当你击键时，你使 Lisp 解释器对一个表达式求值，于是你就得到了结果。即使是键入普通文本也是对 Emacs Lisp 的一个函数求值。在这种情况下，就是使用了 `self-insert-command` 函数，这个函数仅仅插入你输入的字符。通过键入键序列进行求值的函数被称为*交互函数*，或者是命令。如何使一个函数变成交互函数将在如何编写函数定义一章中讲解，参见3.3节，“使函数成为交互函数”。

除了键入键盘命令外，我们已经看到第二种对表达式求值的方法：将光标置于列表后，并键入 `C-x C-e`。在这一章的余下部分，我们就将这样做。对表达式求值还有其他方法，这些方法将在后续章节中描述。

除了用于求值实践外，在下面几节中演示的函数本身都是很重要的。学习这些函数可帮助你弄清：缓冲区和文件的区别、如何切换至一个缓冲区以及如何确定在其中的位置。

2.1 缓冲区名

`buffer-name` 和 `buffer-file-name`这两个函数显示文件和缓冲区之间的区别。当对表达式（`buffer-name`）求值时，缓冲区的名称将在回显区中出现。当对（`buffer-file-name`）表达式求值时，缓冲区所指的那个文件的名称将在回显区中出现。通常情况下，由（`buffer-name`）返回的名称与（`buffer-name`）所指的文件名称相同，由（`buffer-file-name`）返回的名称是文件完整的路径名。

文件和缓冲区是两个不同的实体。文件是永久记录在计算机中的信息（除非你删除了它）。而缓冲区则是 Emacs 内部的信息，它在 Emacs 编辑会话结束时(或当取消缓冲区时)就消失了。通常情况下，缓冲区包含了从文件中拷贝过来的信息，我们称这个缓冲区正在“访问”那个文件。这份拷贝正是你加工或修改的对象。对这个缓冲区的改动不会改变那个文件，除非你保存了这个缓冲区。当你保存这个缓冲区时，缓冲区中的内容被拷贝到文件中去，因此被永久地保存下来。

如果在 GNU Emacs 的 Info 中阅读本教程，可以通过将光标置于下面的表达式后并键入 `C-x C-e` 来对它们求值：

```
(buffer-name)

(buffer-file-name)
```

当我这样做时，“introduction.texinfo”是对（buffer-name）求值所返回的值，而“/gnu/work/intro/introduction.texinfo”是对（buffer-file-name）求值所返回的值。前者是缓冲区的名字，而后者是文件的名字。（在表达式中，括号告诉 Lisp 解释器将buffer-name 和buffer-file-name当做函数处理；如果没有括号，则解释器将它们当做变量来对这些符号求值。参见1.7节，“变量”。）

尽管缓冲区和文件有这些区别，但是你会经常发现人们在说一个缓冲区的时候是指文件，或者反过来。在实践中绝大多数人会说：“我正在编辑一个文件”，而不是说“我正在编辑很快就要存入文件的一个缓冲区”。从人们谈话的上下文中几乎总是可以知道人们真正所指的东西。然而，当处理计算机程序时，你在头脑中清楚地意识到这两者的区别是非常重要的，因为计算机可没有人那么聪明。

“缓冲区”一词来自于这个词可被用作“缓解碰撞力的软垫”之意。在早期的计算机里，缓冲区在文件和中央处理器（CPU）之间的相互作用中起缓和的作用。那时，磁鼓和磁带用于保存文件，它们和 CPU 是彼此很不相同的设备，各自以其固有的速度高速运行。缓冲区使它们能够共同高效地工作。最终，缓冲区演变成为一个中间部件，一个临时存放区，计算机的工作就是在这里进行的。这种变化就像一个小海港成长为一个大城市一样：原来它仅仅是尚未装上船的货物的临时仓库，后来以其自身条件发展成为一个商业和文化中心。

并不是所有的缓冲区都与文件联系在一起。例如，当键入 emacs 命令启动一个 Emacs 会话时，没有给出任何文件，Emacs 将在屏幕上启动一个“*scracth*”(草稿)缓冲区。这个缓冲区并没有访问任何文件。类似地，一个“*help*”(帮助)缓冲区也不与任何文件相关联。

如果切换到“*scratch*”缓冲区，键入（buffer-name），将光标置于列表之后，并键入 C-x C-e 对这个表达式求值，“*scratch*”这个名字将显示在回显区中。“*scratch*”就是这个缓冲区的名字。然而，如果输入（buffer-file-name）并对它求值，nil 将显示在回显区中。nil 一词来自于拉丁语，意指“什么都没有”（空）。在这种情况下，它是指“*scratch*”缓冲区没有与任何文件关联。（在Lisp 中，nil 也用于指“假”，或者用作空列表（）的同义语。）

顺便提一下，如果你在“*scratch*”缓冲区中并希望由一个表达式返回的值出现在缓冲区中而不是出现在回显区中，键入 C-u C-x C-e 而不是键入 C-x C-e。这将使返回值显示在表达式的后面。缓冲区看起来如下所示：

```
(buffer-name)"*scratch*"
```

你无法在 Info 中完成上面的工作，因为 Info 是只读的，它不允许你改变缓冲区中的内容。但是你可以在任意能够进行编辑的缓冲区中这样做；当编写代码或者文档时（例如我写作本书时），这个特性是非常有用的。

2.2 获得缓冲区

buffer-name函数返回缓冲区的名字。为了获得缓冲区本身，需要另外一个函数：

current-buffer。如果在代码中使用这个函数，得到的将是这个缓冲区本身。

一个名字与名字所指的对象或实体是互不相同的。你不是你的名字。你是一个用名字指向的人。如果你要求与 George 讲话，有人给你一张印有“G”“e”“o”“r”“g”“e”这几个字母的名片，你可能被搞糊涂，你不会满意的。你不是要对这个名字讲话，而是要与这个名字所指的那个人讲话。缓冲区就与此类似：草稿缓冲区的名字是“*scratch*”，但是这个名字本身不是缓冲区。为了得到缓冲区本身，你需要使用一个函数，如 current-buffer。

然而，这里带有一点复杂性：如果在缓冲区中的一个表达式内对 current-buffer 求值，就像我们将要做的那样，你所看到的是打印出来的这个缓冲区对应的名字，而没有缓冲区的内容。Emacs 这样做有两个理由：缓冲区可能有数千行长——显示起来太长了，不方便；而且，另外一个缓冲区可能有同样的内容，只是名字不一样而已，将它们之间区分开来是很重要的。

下面是包含这个current-buffer函数的一个表达式：

```
(current-buffer)
```

如果用常规的办法对这个表达式求值，“#<buffer *info*>”将显示在回显区中。这个特殊的格式表明这个缓冲区本身被返回了，而不仅仅是其名字。

顺便说一下，可以在一个程序中输入一个数或者符号，但却不能用这种方法得到缓冲区的打印表示：得到缓冲区本身的唯一方法是用一个函数，如 current-buffer函数。

一个相关的函数是 other-buffer。这个函数返回最近使用过的缓冲区，而不是当前使用的那个缓冲区。如果最近经常对“*scratch*”缓冲区不停地来回切换，那么 other-buffer 函数将返回那个缓冲区。

对下面的表达式求值就可以看到这一点：

```
(other-buffer)
```

应该看到“#<buffer *scratch*>”或者最近从其中切换回来的缓冲区的名字显示在回显区中。

2.3 切换缓冲区

当other-buffer函数被一个函数用作参量时，这个other-buffer 函数实际上提供了一个缓冲区。通过使用 other-buffer函数和 switcb-to-buffer 函数来切换到另外一个缓冲区，我们将看到这一点。

但是，先简单介绍一下 switch-to-buffer 函数。当在Info和草稿缓冲区“*scratch*”之间来回切换来对 (buffer-name) 表达式求值时，很可能要键入键序列C-x b，并当在小缓冲区中提示要求你输入希望切换到的缓冲区的名字时输入“*scratch*”。键序列 C-x b，使 Lisp 解释器对交互性的 Emacs Lisp 函数 switch-to-buffer 求值。正像我们在前面讲的，这就是 Emacs 的工作方式：不同的键序列调用和运行不同的函数。例如，C-f 调用forward-char 函数，M-e 调用 forward-sentence 函数，等等。

在一个表达式中写入 switch-to-buffer 函数，并给它一个要切换到的缓冲区，就可以像 C-x b 那样切换缓冲区了。

以下就是完成这个任务的 Lisp 表达式：

```
(switch-to-buffer (other-buffer))
```

符号 switch-to-buffer 是这个列表的第一个元素，因此 Lisp 解释器将它视作成一个函数，并执行这个函数的指令。但在这样做之前，解释器将注意到 other-buffer 在一个括号内，因此先处理这个列表。other-buffer 是这个列表的第一个元素（在这种情况下也是仅有的一个元素），因此 Lisp 解释器调用和运行这个函数。它返回另外一个缓冲区。下一步，解释器运行 switch-to-buffer 函数，将另外这个缓冲区作为一个参量传送给它，这后面一个缓冲区就是 Emacs 要切换到的缓冲区。如果你在 Info 中阅读这本教程，现在就可试一下，求这个表达式的值。（要返回的话，键入 C-x b RET。）

在这本教程的后续的编程例子中，将更多地看到 set-buffer 函数而不是switch-to-buffer 函数。这是因为人和计算机程序之间的一个差别：人有眼睛，并希望在他们工作的计算机终端上看到缓冲区。这是如此的直观，几乎不言自明。然而，计算机程序没有眼睛，当计算机程序工作在一个缓冲区时，缓冲区无需在屏幕上显示出来。

switch-to-buffer 函数是为人设计的，它完成两件不同的事情：一是切换到 Emacs 关注的缓冲区；一是从当前显示在窗口中的缓冲区切换到一个新的缓冲区。另一方面，set-buffer 函数只做一件事：它将计算机的注意力切换到另外一个不同的缓冲区。屏幕上显示的缓冲区并不改变（当然，直到命令运行完之前一般这不会发生任何事情）。

这里，我们已经接触了另外一个术语：调用（*call*）。当对第一个元素是一个函数的列表求值时，就是在调用那个函数。这个词的使用来自这样的概念，函数作为一个实体，如果“呼叫”它，它可以为你做某些事情——就像水管工人在你呼叫他时能帮你补漏一样。

2.4 缓冲区大小和位点的定位

最后，来看看几个相当简单的函数：buffer-size、point、point-min 和 point-max。这些函数给出缓冲区大小以及其中的位点的位置等信息。

buffer-size 函数给出当前缓冲区的大小，也就是，这个函数返回关于这个缓冲区中字符数的计数。

```
(buffer-size)
```

可以用通常的办法，即将光标置于这个表达式后面并键入 C-x C-e 来对这个表达式求值。

在 Emacs 中，光标所在的当前位置被称为“位点”（*point*）。表达式（point）返回一个数字，这个数字给出光标所处的位置，即从这个缓冲区首字符开始到光标所在位置之间的字符数。

用通常的办法对下面的表达式求值，你可以看看光标在这个缓冲区中当前位点的字符计数：

```
(point)
```

当我写到这里时，point 函数的返回值是 65724。point 函数将经常出现在本书后面的例子中。

当然，位点的值依赖于它在缓冲区中的位置。如果在这个点对 point 函数求值，返回数值会更大些：

```
(point)
```

对我而言，在这个位置中的位点的值是 66043，这意味着在这两个表达式之间有 319个字符（包括空格）。

point-min函数与 point 函数有点类似，但是它返回在当前缓冲区中位点的最小可能值。除非设置了变窄（*narrowing*），这个值一般就是 1。（变窄是一种自我限制的机制，限制用户或者一个程序只能对缓冲区的一部分进行操作。参见第 6 章，“变窄和增宽”。）与此类似，函数 point-max 返回在当前缓冲区中位点的最大可能值。

2.5 练习

找一个文件，对它进行操作，将光标移动到缓冲区的中间部分。找出它的缓冲区名、文件名、长度、和你在文件（其实是缓冲区）中的位置。

第3章 如何编写函数定义

当 Lisp 解释器对一个列表求值时，它查看列表中的第一个符号是否有一个与之联系在一起的函数定义，或者用另外一种说法，就是第一个符号是否指向一个函数定义。如果它确实有一个函数定义，计算机执行函数定义中的指令。有函数定义的符号被简单地称作一个函数（虽然正确的说法是这个函数的定义，函数符号指向这个定义）。

除了一些基本函数是用 C 语言编写的之外，其他所有函数都是用别的函数来定义的。你将在 Emacs Lisp 中编写函数的定义，并用其他函数作为你的基本构件。你将要用到的一些函数本身就是用 Emacs Lisp 编写的（可能就是你编写的），而另一些基本函数可能是用 C 语言编写的。这些基本函数的用法与用 Emacs Lisp 编写的函数的用法完全一样，表现也很相似。由于它们是用 C 语言编写的，因此我们可以容易地在任何运算能力足够强、能运行 C 程序的计算机上运行 GNU Emacs。

再重申一次：当你在 Emacs Lisp 中编写代码时，你无法分清在 C 语言中编写的函数和在 Emacs Lisp中编写的函数。它们之间的区别是不相关的。之所以提到它们的区别是因为知道这一点很有趣。实际上，除非你深入研究，否则你将不知道已经编写好的函数是用 Emacs Lisp 编写的还是用 C 语言编写的。

3.1 defun 特殊表

在 Lisp 中，一个类似于 mark-whole-buffer 这样的符号已经有代码与之联系了，这告诉计算机当函数被调用时要做些什么。该代码被称作*函数定义*，它是通过对一个以符号 defun（defun 是“*define function*”的缩写）开头的 Lisp 表达式求值而被建立的。因为defun不以通常的方式对它的参量求值，因此它被称为*特殊表*。

在后续的章节中，我们将从 Emacs 源代码（如 mark-whole-buffer）来看看函数定义的问题。在这一节中，我们将描述一个简单的函数定义，因此你可以看看它是什么样子。这个函数定义用到了算术，因为它是一个简单的例子。有些人不喜欢使用算术的例子，然而，如果你是这样的人，不要绝望。这本教程中我们将要学到的例子中很少有这类用到算术和数学的例子。例子绝大多数都是以这种或者那种方式与文本有关的。

一个函数定义在 defun 一词之后最多有下列五个部分：

1) 符号名，这是函数定义将要依附的符号。

2) 将要传送给函数的参量列表。如果没有任何参量传送给函数，那它就是一个空列表()。

3) 描述这个函数的文档。（技术上说，这部分是可选的，但是我强烈推荐你使用。）

4) 一个使函数成为交互函数的表达式，这是可选的。因此，可以通过键入 M-x和函数名来使用它，或者键入一个适当的键或者键序列来使用它。

5) 指导计算机如何运行的代码，这是函数定义的主体。

将函数定义的五个部分有序地组织成一个模板是很有用的，每个部分各有其自己的位置：

```
(defun function-name (arguments...)
  "optional-documentation..."
  (interactive argument-passing-info) ;optional
  body...)
```

作为一个例子，下面是一个函数的代码，这个函数将它的参量乘以 7。(这个例子不是交互式的。参见3.3节 "使函数成为交互函数"。)

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

这个定义以括号和符号 defun 开始，后接函数名。

函数名后接一个列表，这个列表包含将要传送给这个函数的参量。这个列表被称作参量列表。在目前的情况下，这个参量列表仅有一个元素，即符号 number。当这个函数被使用时，这个符号将被绑定到一个值上，就像参量被绑定到函数上一样。

若不选择 number 这个词作为这个参量的名字，也可以用其他任何名字。例如，可以选取 multiplicand 这个词。我之所以选择 "number" 这个词是因为它告诉我在这个位置应是什么类型的值。但是也可以用 "multiplicand" 这个词来表示放在这个位置的这个值将在函数中扮演的角色。甚至可以将这个参量名取为 foogle，但是这是一个很糟糕的选择，因为它无法告诉人们它的含义。名字的选择是程序员的事情，应该适当地选择以使函数的意义明白无误。

你确实可以选择你希望的任何名字作为参量列表中的符号，甚至可以是在其他函数中使用过的名字：你在一个参量列表中使用的名字对这个特定的函数定义而言是私有的。在那个函数定义中，这个名字是指向一个不同的实体，而不是指向函数之外其他同名的使用。假设在家里你有一个绰号 "Shorty"，当你的家人提到 "Shorty" 的时候，他们是指你。但是在你的家庭之外，比如一部电影中，"Shorty" 这个名字指其他人。因为参量列表中的名字是这个函数定义私有的，因此可以在一个函数的主体中改变这个符号的值而不改变它在函数之外的值。这种效果与 let 表达式产生的效果相似。(参见3.6节 "let函数"。)

参量列表之后跟随着描述函数的文档字符串。这就是当键入 C-h f 并输入函数名时看到的内容。顺便说一下，当你编写类似这样一个文档字符串时，应当使其第一行是一个完整的句子，因为有些命令(如 apropos)仅仅打印多行文档字符串的第一行。同样，如果有第二行，不要在第二行缩进文档字符串，因为当你用 C-h f (describe-function) 时，看起来会很奇怪。文档字符串是可选的，但是它很有用，几乎所有你编写的函数都应当包含它。

例子的第三行由函数定义的主体组成。(当然，绝大多数函数的定义都比这个函数定义长)。在这个例子中，函数定义的主体是一个列表 (* 7 number)，它是将 number的值乘以 7。(在 Emacs Lisp 中，* 代表乘法函数，就像 + 代表加法函数一样)。

当你使用 multiply-by-seven 函数时，参量 number 给出你要使用的实际值。下面的例子演示如何使用 multiply-by-seven 函数，但是现在不要试图对它求值！

```
(multiply-by-seven 3)
```

对于在下一节定义的符号number，在实际使用这个函数时，它被赋给或者绑定到值 3。注意，虽然 number 在函数定义的括号内，但是传送给 multiply-by-seven 函数的这个参量并不在括号中。括号写在函数定义中，因此计算机能够分清参量列表在何处结束以及函数定义的其他部分从何处开始。

如果对这个例子求值，你很可能得到一个错误消息。（试一试吧！）。这是因为我们已经编写了函数定义，但是还没有告诉计算机你的函数定义，即我们在Emacs 中还没有安装（或者加载）这个函数定义。安装一个函数是告诉 Lisp 解释器这个函数定义的过程。下一节将讲述函数定义的安装。

3.2 安装函数定义

如果你在 Emacs 的 Info 中阅读这本教程，可以先对 multiply-by-seven 求值，然后对（multiply-by-seven 3）求值。这个函数定义的一个拷贝列在下面。将光标置于函数定义的最后一个括号之后，并键入 C-x C-e。当你这样做时，multiply-by-seven 将出现在回显区中。（这意味着当一个函数定义被求值时，它的返回值是已定义的函数的名字。）同时，这个动作安装了函数定义。

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

通过对这个函数定义求值，你已经在Emacs中将 multiply-by-seven 安装好了。这个函数就像 forward-word 或者你使用的其他编辑函数一样成了 Emacs 的一部分。（multiply-by-seven 将一直安装在 Emacs 中，直到你退出 Emacs 。关于在启动Emacs时自动加载代码，可以参见3.5节“永久地安装代码”。）

通过对下面的例子求值就可以看到安装 multiply-by-seven 函数的效果了。将光标置于下面的表达式后并键入 C-x C-e。数字 21 将显示在回显区中。

```
(multiply-by-seven 3)
```

如果你愿意，可以键入 C-h f（describe-function）以及函数名multiply-by-seven 来阅读相应函数的文档。当你这样做时，“*Help*”窗口将显示在你的屏幕上，并说：

```
multiply-by-seven:
Multiply NUMBER by seven.
```

(键入 C-x 1就可以返回在你的屏幕上只显示一个窗口。)

改变函数定义

如果要改变multiply-by-seven函数中的代码，只需重写它即可。为了安装这个新版本的函数定义而不是旧的那个函数定义，只需对函数定义再求值一次即可。这就是在 Emacs 中改变代码的方法。它是非常简单的。

作为一个例子，可以这样改变这个multiply-by-seven函数：将 number 加 7 次，而不是用 7 乘以number。这将产生同样的结果，只是用的方法不同。同时，我们对代码作一个注释，

注释部分是文本，Lisp 解释器将忽略它，但是当人们阅读源代码时就会发现注释部分是很有用的，它可以起到提醒的作用。在这个例子中，注释部分是："second version"。

```
(defun multiply-by-seven (number)       ; Second version.
  "Multiply NUMBER by seven."
  (+ number number number number number number number))
```

注释部分以分号"；"开始。在 Lisp 代码行的任何地方，分号后面的内容都是注释。行的结束就是注释的结束。为了使注释延伸到两行或者更多行，可以在每一行前都加一个分号。

关于注释，可以参见16.3节，"开始改变".emacs"文件"，以及《*GNU Emacs Lisp*技术手册》中的"注释"一节。

可以用同样的办法对 multiply-by-seven 函数定义求值并安装它：将光标置于最后一个括号之后并键入 C-x C-e。

总之，在 Emacs Lisp 中就是这样编写代码的：编写一个函数；安装它；测试它；然后修改或者增强它的功能并重新安装。

3.3 使函数成为交互函数

使一个函数成为交互函数可以这样实现：在函数文档后面增加一个以特殊表interactive开始的列表。用户键入 M-x 和函数名就可以激活一个交互函数，或者键入绑定的键序列也可以激活它，例如键入 C-n 可以激活 next-line 函数；键入C-x h 可以激活 mark-whole-buffer 函数。

有趣的是，当用交互的方法调用一个交互函数时，函数的返回值不会自动显示在回显区中。这是因为你一般总是喜欢得到调用交互函数的附带效果。例如，向前移动一个字或者一行，而不在乎返回值。如果每键入一个键，返回值都显示在回显区中，这很分散注意力。

使用特殊表 interactive 和在回显区中显示一个值这两种办法都能通过创建一个交互形式的 multiply-by-seven 函数面得到验证。

代码如下：

```
(defun multiply-by-seven (number)       ; Interactive version.
  "Multiply NUMBER by seven."
  (interactive "p")
  (message "The result is %d" (* 7 number)))
```

通过将光标置于上面的函数定义之后并键入 C-x C-e 对其求值，就可以将这个函数定义安装。函数名将显示在回显区中。然后，通过键入 C-u 和一个数字并键入 M-x multiply-by-seven 和按下回车键，就可以使用这个函数了。加上了结果的句子"The result is..."将显示在回显区中。

更一般地说，可以用下列两种方法之一激活一个函数：

1) 键入一个包含了传送始函数的数字的前缀参量和 M-x 以及函数名，如下所示：C-u 3 M-x forward-sentence；或者

2) 键入函数绑定键或者键序列，如下所示：C-u 3 M-e。

这两种方法结果都是一样的，都将位点向前移动了三个句子。(因为 multiply-by-seven 没有绑定键，它不能被用作键绑定的例子。)

(有关如何将命令绑定到键，请参见16.7节“一些绑定键”。)

可以通过键入 META 键和后接一个数字(如 M-3 M-e)来将一个前缀参量传递给一个交互函数；也可以通过键入 C-u 和后接一个数字(如 C-u 3 M-e)来将一个前缀参量传递给一个交互函数(如果键入了 C-u 而没有后接一个数字，就使用默认的数值 4)。

交互的 multiply-by-seven 函数

让我们看看将殊表 interactive 以及这个交互的 multiply-by-seven 函数中的 message 函数的使用方法。你会记得交互的 multiply-by-seven 函数定义如下所示：

```
(defun multiply-by-seven (number)          ; Interactive version.
  "Multiply NUMBER by seven."
  (interactive "p")
  (message "The result is %d" (* 7 number)))
```

在这个函数中，表达式 (interactive "p") 是由两个元素组成的一个列表。其中的“p”告诉 Emacs 要传送一个前缀参量给这个函数，并将它的值用于函数的参量。

这个参量将是一个数。这意味着符号 number 将在这行绑定到一个数字上：

```
(message "The result is %d" (* 7 number)))
```

例如，如果前缀参量是5，则 Lisp 解释器对这一行求值时就好像是对以下行 (message “The result is %d” (* 7 5) 求值一样。

(如果你是在 GNU Emacs 中阅读这份文档的话，可以对这个表达式求值试试。) 首先，解释器将对内层的列表求值，即 (* 7 5)。返回值是 35。然后，解释器对外层表达式求值，将外层列表的第二个元素和后续的元素传送给 message 函数。

就像你所看到的那样，message 是一个 Emacs Lisp 函数，这个函数的作用就是将一行消息传递给用户 (参见1.8.5节的“message 函数”)。总之，message函数将其第一个元素中除“%d”、“%s”、“%c” 之外的内容打印在回显区中。当它看到这些控制序列时，函数查看第二个及后续的参量，将它们的值取代这些控制序列，打印在回显区中。

在交互的 multiply-by-seven 函数中，控制符就是“%d”。它要求一个数字来替代，也就是用 (* 7 5) 的返回值 35 来替代。之后，数字 35 替代了“%d”，因此显示的消息就是“The result is 35”。

(注意，当你调用函数 multiply-by-seven 时，回显区的消息是不带引号的。但是，当你调用 message 函数时，打印出来的文本是带引号的。这是因为由message 函数返回的值将显示在回显区，而将其嵌入到一个函数中时，message 打印出来的文本是作为一个附带效果出现的，因此不带引号。)

3.4 interactive函数的不同选项

在上面的例子中，multiply-by-seven 函数使用“p”作为交互命令interactive 的

参量。这个参量告诉 Emacs 将你正在键入的 C-u 加上一个数字或 META 加上一个数字解释为一个命令，用来将这个数字作为参量传送给函数。Emacs 有多于20个为 interactive 预先定义好的字符。在几乎每一种情况下，一个或者多个这种选项将使你能将正确的信息交互地选送给函数。(参见《*GNU Emacs Lisp 技术手册*》中的“interactive 的控制符”一节。)

例如，字符“r”使 Emacs 将位点所在区域的开始值和结束值作为函数的两个参量。用法如下：

```
(interactive "r")
```

在另一方面，“B”告诉 Emacs 用缓冲区的名字作为函数的参量。在这种情况下，Emacs会在小缓冲区提示用户输入缓冲区的名字，并使用跟在“B”后面的字符串表示这种要求(如“BAppend to buffer:”)。Emacs 不仅提示输入函数名，而且如果用户给出了足够的信息并按下 TAB 键，Emacs 会自动补齐函数名。

对于有两个或者更多参量的函数，对其参量可以各有各的值，在 interactive 中相应地增加一些内容就行了。当你这样做时，这些信息以其在 interactive 中定义的顺序被传送给每一个参量。在字符串中，两个部分之间用“\n”分隔开，这代表一个新的行。例如，你可以在“BAppend to buffer:”后面加上一个“\n”和一个“\r”。这将使 Emacs 将位点和标记的值传送给函数并提示你输入缓冲区名字——一共是三个参量。

在这个例子中，函数定义看起来就像下面的例子一样，其中 buffer、start 和 end 是 interactive 绑定的当前缓冲区以及当前区域的起始值和结束值的符号：

```
(defun name-of-function (buffer start end)
  "documentation..."
  (interactive "BAppend to buffer: \nr")
  body-of-function...)
```

(冒号后面的空格使提示输入内容时更好看一些。append-to-buffer 函数与这个函数看起来很像。参见4.4节“append-to-buffer 函数的定义”。)

如果一个函数没有参量，interactive就不需要任何东西。这样的函数只有一个简单的表达式：(interactive)。mark-whole-buffer 函数就是这样的。

作为选择，如果这些特殊控制符都无法满足你的应用需要，你可以将自己的参量传送给nteractive作为一个列表。有关这种高级技术的更多信息，参见《*GNU Emacs Lisp 技术手册*》中的“使用interactive”一节。

3.5 永久地安装代码

当你对一个函数定义求值来安装它时，它将一直保留在 Emacs 之中直到你退出Emacs 为止。你下次再次启动一个新的 Emacs 会话时，除非你再一次对这个函数定义求值，否则这个函数将不会被安装。

在有些时候，你可能要求当你启动一个新的 Emacs 会话时将函数定义自动安装。为了达到这个目的，可以使用下面几种方法：

- 如果这个要自动安装的代码仅仅供你个人使用的，你可以将这个函数定义的代码放到你的

".emacs" 初始化文件中。当你启动 Emacs 时，你的 ".emacs" 文件被自动求值，其中的所有函数定义都会被安装。参见第16章 "配置你的 '.emacs' 文件"。

- 你可以将需要自动安装的函数定义放在一个或者多个文件中，然后使用 load 函数让Emacs对它们求值，从而安装这些文件中的所有函数。参见16.8节 "加载文件"。
- 在另一方面，如果有些函数定义是该计算机的所有用户都要使用的，这种情况下经常将它放到一个叫做 "site-init.el" 的文件中，这个文件在Emacs启动时被加载。这使得所有使用你的计算机的用户都可以使用这些函数。(参见随 Emacs 一起发行的 "INSTALL" 文件)。

最后，如果你编写的某些函数是所有使用 Emacs 的用户都要使用的，你可以将它放到一个计算机网络中，或者给自由软件基金会发送一份拷贝。(当你这样做时，请在后面附加一份 copyleft 声明。) 如果你给自由软件基金会发送了一份拷贝，它将可能被加到 Emacs 的下一个发行版本中。很大程度上说，这就是 Emacs 在过去的年代里成长的道路——奉献。

3.6 let 函数

let 表达式是 Lisp中的一个特殊表，用户在绝大多数函数定义中都需要使用它。因为let 表达式是如此的通用，所以这一节将专门介绍它。

let 用于将一个符号附着到或者绑定到一个值上，对于这样绑定的变量，Lisp 解释器就不会将其与函数之外的同名变量混淆了。理解为什么这是一个特殊表是很必要的。考虑一下这种情况：你拥有一个家，在句子中你一般将其称为 "房子"，"房子需要粉刷"。如果你在拜访你的朋友时，他提到 "房子"，他是指他的房子而不是你的房子，即一幢不同的房子。如果他指他的房子，而你认为他是在指你的房子，你们可能就弄糊涂了。如果一个函数中的一个变量与另外一个函数中的某个变量同名，而且它们本来就不是指同一件事，类似的混淆事情也可能发生在 Emacs 中。

let 特殊表就避免了这种情况的发生。let 创建的*局部变量(local variable)*屏蔽了任何在这个let 表达式之外同名的变量。这就像每当你的朋友提到 "房子" 时就是指他自己的房子而不是你的房子一样。(在参量列表中的符号与此类似。参见3.1节 "defun特殊表"。)

由 let 表达式创建的局部变量只是在 let 表这式中保留它们的值(当然也可在 let 表达式调用的表达式中保留局部参量的值)；局部变量不会影响 let 表达式之外的东西。

let 表达式一次可以创建多个变量。同样，let 表达式给每一个变量赋由你创建的一个初始值，或者赋由你给定的一个值，或者赋nil (用术语来说，这是将变量绑定到值上)。let 表达式创建并绑定变量之后，它执行 let 表达式主体本身(即对 let 表达式求值)，并返回表达式主体中最后一个表达式的值，这作为整个let 表达式的返回值。("执行" 是一个术语，表示对一个列表求值；它的词义来自于 "给予实际的效果" (牛津英语词典)。由于你对一个表达式求值是为了完成某个动作，因此 "执行" 一词被演化为 "求值" 一词的同义词。)

3.6.1 let 表达式的各个部分

let 表达式是一个具有三个部分的列表。let 表达式的第一个部分就是 let 符号。第二部分是一个列表，称为变量列表(*varlist*)，这个列表的每一个元素是一个符号或者一个两元素的

列表，而它的第一个元素一定是一个符号。let 表达式的第三个部分是 let 表达式主体，这个主体由一个或者多个列表组成。

let 表达式的模板看起来如下所示：

(let *varlist body...*)

变量列表中的符号是由 let 特殊表赋初始值的变量。符号本身的初始值是 nil 。作为两元素列表的首元素的每一个符号将被绑定到对第二个元素求值后的返回值。

因而，变量列表就是这个样子：(thread (needles 3))。在这个例子中，在一个 let 表达式中，Emacs 将符号 thread 绑定到初始值 nil，并将符号needles 绑定到初始值 3 上。

当你编写一个 let 表达式时，你所要做的就是将适当的表达式添入 let 表达式模板的适当位置。

如果变量列表是由两元素列表组成的（这是常见的情况），就可以采用下面的let 表达式模板：

```
(let ((variable value)
      (variable value)
      ...)
      body...)
```

3.6.2 let 表达式例子

下面的表达式创建两个变量 zebra 和 tiger，并给它们赋初值。这个 let 表达式的主体是一个使用 message 函数的列表。

```
(let ((zebra 'stripes)
      (tiger 'fierce))
  (message "One kind of animal has %s and another is %s."
           zebra tiger))
```

在这个例子中，变量列表是：((zebra 'stripes)(tiger 'fierce))。

例子中的两个变量是：zebra 和 tiger。每一个变量都是各自所在的两元素列表的第一个元素，它们的值分别是两元素列表的第二个元素。在变量列表中，Emacs 将变量 zebra 绑定到值 stripes，将 tiger 绑定到值fierce。在这个例子中，这两个值都是标引号的符号。当然，绑定到变量上的值也可以是其他列表或者字符串。let 表达式的主体跟在变量列表之后。在这个例子中，let 表达式主体是一个列表，这个列表使用 message 函数往回显区中打印一个字符串。

可以使用通用的方法对上面的例子求值，将光标置于最后一个括号之后，键入C-x C-e。当键入这些命令时，下面的字符串将显示在回显区中：

```
"One kind of animal has stripes and another is fierce."
```

就像我们前面看到的那样，message 函数将它的第一个参量（不含“%s”）打印到回显区

中。在这个例子中，变量 zebra 的内容打印到第一个“%s”的位置，而变量 tiger 的内容打印到第二个“%s”位置。

3.6.3 let 语句中的未初始化变量

在 let 语句中，如果没有将变量绑定到用户指定的一个特定的初始值上，则它们将被自动地绑定到 nil 这个初始值上。下面就是这样的一个例子：

```
(let ((birch 3)
      pine
      fir
      (oak 'some))
  (message
   "Here are %d variables with %s, %s, and %s value."
   birch pine fir oak))
```

这里，变量列表是((birch 3) pine fir (oak ' some))。

如果用通常的办法对这个表达式求值，下列信息将显示在回显区中：

```
"Here are 3 variables with nil, nil, and some value."
```

在这个例子中，Emacs 将符号 birch 绑定到数值3，将符号 pine 和 fie 绑定到 nil，将符号 oak 绑定到 some。

注意，在 let 表达式的第一个部分，变量 pine 和 fir 是作为独立的原子出现的，它们没有用括号括起来。这是因为它们将绑定到空列表 nil。但是 oak 被绑定到 some，因此要作为列表（oak 'some）的一部分。类似的，birch 绑定到 3，因此要放在有这个数字的列表中。(因为数字的本身就是它的值，因此无需用引号标出。同样，数字将取代“%d”面不是“%s”打印到回显区中。)这四个变量作为一个整体放在一个列表中，以区别于 let 表达式的主体。

3.7 if 特殊表

除了 defun 和 let 特殊表之外，第三个特殊表就是 if 条件特殊表。这个表用于指导计算机做出判断。可以不用 if 特殊表编写函数定义，但是它使用得如此频繁，因而我们在这里要特别加以讲述。例如，它用在函数 beginning-of-buffer的代码中。

if 特殊表背后的基本含义是：如果一个测试是正确的，则对后续的表达式求值；如果这个测试不正确，则不对这个表达式求值。例如，你可能要对“如果天气晴朗暖和，就去海滩”这样的情况做出决定。

在 Lisp 中，if 表达式并没有使用“then”这样的字眼，但是，测试和执行代码就必然是第一个元素为 if 的这个列表的第二和第三个元素。然而，测试部分常被称为“if 部”（if-part），而第二个参量常被称为“then 部”（then-part）。

同样，当编写一个 if 表达式时，真假测试经常写在 if 符号这一行，但是“then部”，即如果测试为“真”后的执行部分，则写在这一行下面的一行及其后。这种写法使 if 表达式易于阅读。

```
(if true-or-false-test
    action-to-carry-out-if-test-is-true)
```

真假测试是一个由 Lisp 解释器求值的表达式。

这是一个可以用通常的办法求值的例子。这个例子的测试是“5是否大于4”。因为5大于4，因此消息“5 is greater than 4!”将被打印出来。

```
(if (> 5 4)                              ; if-part
    (message "5 is greater than 4!"))    ; then-part
```

(>函数测试它的第一个参量是否大于第二个参量，如果真的如此，则返回“真”。)

当然，在实际使用中，if 表达式中的测试并不是像表达式（> 5 4）这样是固定不变的。相反，至少测试表达式中的一个变量是被绑定到一个预先不确定的值上的。(如果预先已经知道这个值，就无需执行这个测试了。)

例如，变量可能被绑定到一个函数定义的一个参量上。在下面的函数定义中，动物的特性是作为值传送给函数的。如果绑定到符号 characteristic 的值是 fierce，则打印消息“It's a tiger!”。否则，返回 nil。

```
(defun type-of-animal (characteristic)
  "Print message in echo area depending on CHARACTERISTIC.
If the CHARACTERISTIC is the symbol 'fierce',
then warn of a tiger."
  (if (equal characteristic 'fierce)
      (message "It's a tiger!")))
```

如果你在 GNU Emacs 中阅读这份文档，可以用通常的办法对这个函数定义求值以将它安装到Emacs中。然后，可以对下面两个表达式求值：

```
(type-of-animal 'fierce)

(type-of-animal 'zebra)
```

当你对（type-of-animal 'fierce）求值时，将看到这样的内容显示在回显区中：“It's a tiger”。如果你对（type-of-animal 'zebra）求值，将看到 nil 显示在回显区中。

type-of-animal 函数详解

让我们仔细看看 type-of-animal 这个函数。

type-of-animal函数的函数定义是根据两个模板写就的，一个模板就是函数定义模板，另一个模板就是 if 表达式模板。

如前所述，非交互的函数定义模板如下所示：

```
(defun name-of-function (argument-list)
  "documentation..."
  body...)
```

type-of-animal 函数中与此模板类似的部分如下所示：

```
(defun type-of-animal (characteristic)
  "Print message in echo area depending on CHARACTERISTIC.
If the CHARACTERISTIC is the symbol 'fierce',
then warn of a tiger."
  body: the if expression)
```

在这个例子中，函数名是type-of-animal，它被传递一个参量的值。参量列表后跟着一个多行文档字符串。例子中包含了文档字符串，因为为每一个函数书写文档是一个好习惯。函数定义的主体由 if 表达式组成。

if 表达式模板如下所示：

```
(if true-or-false-test
    action-to-carry-out-if-the-test-returns-true)
```

在 type-of-animal 函数中，if 表达式的实际代码如下所示：

```
(if (equal characteristic 'fierce)
    (message "It's a tiger!")))
```

在这里，真假测试是这样一个表达式：

```
(equal characteristic 'fierce)
```

在Lisp中，equal 是一个判定它的第一个参量是否等于第二个参量的函数。例子中，第二个参量是带引号的符号' fierce。测试表达式的第一个参量就是符号characteristic的值，换句话说，就是传送到这个函数的参量。

在type-of-animal的第一个求值练习中，参量 fierce被传送给函数 type-of-animal。因为fierce 等于 fierce，所以(equal characteristic ' fierce)表达式返回值为“真”。这时，对if 表达式的第二个参量或then部(即(message “It' s a tiger!”))求值。

在另一方面，在type-of-animal的第二个求值练习中，参量 zebra 被传送给函数。zebra 不等于 fierce，函数的 then 部不被求值，故返回 nil。

3.8 if-then-else 表达式

if 表达式可以有第三个参量，称为 else 部。这是为真假测试返回“假”时使用的。当真值测试返回“假”时，if 表达式的第二个参量(即 then 部)不被求值，但是其第三部分(即 else 部)被求值。可以将这理解为“如果天气晴朗暖和，就去海滨吧，否则就读书“这种选择中作为多云天气时的一种选择。

“else”一词并不被写在 Lisp 代码中，if 表达式的 else 部紧接在 then 部的后面。在 Lisp 中，else 部经常在一个新行中书写，并且缩进得比 then部少：

```
(if true-or-false-test
    action-to-carry-out-if-the-test-returns-true)
  action-to-carry-out-if-the-test-returns-false)
```

例如，如果用通常的办法求值，则下面的 if 表达式将消息“4 is not greater than 5!”打印出来。

```
(if (> 4 5)                              ; if-part
```

```
    (message "5 is greater than 4!")     ; then-part
  (message "4 is not greater than 5!")) ; else-part
```

注意，不同的缩进尺度使 then 部与 else 部区别开来。(GNU Emacs 有几个自动缩排 if 表达式的命令，参见1.1.3节，"GNU Emacs 帮助你输入列表"。)

我们仅仅将一个附加部分加入if表达式使之包含 else 部，就可以扩展 type-of-animal 函数。

如果对下面的这个版本的 type-of-animal 函数定义求值以安装它，然后对后续的两个表达式求值以传送不同的参量给函数，这样你就可以看到 else 部的作用了。

```
(defun type-of-animal (characteristic)  ; Second version.
  "Print message in echo area depending on CHARACTERISTIC.
If the CHARACTERISTIC is the symbol 'fierce',
then warn of a tiger;
else say it's not fierce."
  (if (equal characteristic 'fierce)
      (message "It's a tiger!")
    (message "It's not fierce!")))
(type-of-animal 'fierce)

(type-of-animal 'zebra)
```

当你对 (type-of-animal' fierce) 求值时，下面的消息将显示在回显区中："It's a tiger!"；但是当你对 (type-of-animal, zebra) 求值时，将看到"It's not fierce!"显示在回显区中。

(当然，如果 *characteristic*的值是 ferocious，则消息"It's not fierce!"将会显示出来。这会误导人的！当编写代码时，需要考虑所有可能性，对这样的参量将用 if 表达式进行测试并因此编写程序。

3.9 Lisp 中的真与假

if 表达式中的真假测试有一个重要的特性。迄今为止，我们已经说到过"真"与"假"作为测试的可能值，就好像它们是 Lisp 的新对象一样。事实上，"假"只不过是我们前面提到的 nil 的另一种形式。其他所有东西，都是"真"。

如果求值的结果是一个非nil的值时，则测试真假的表达式被解释为"真"(*true*)。换句话说，如果真假测试返回的结果是一个数字 (如47)、一个字符串(如"hello")、一个符号(除nil外，如flowers)、一个列表、甚至一个缓冲区时，则测试结果为"真"。

在演示这些内容之前，需要先解释一下 nil。

在 Lisp 中，nil 这个符号有两种意思[⊖]。第一，它表示一个空列表。第二，它表示"假"，并是真假测试为"假"时的返回值。可以将nil写作一个空列表()或nil。只要是关于Lisp解释器，()和nil都是相同的。但是人类却倾向于用nil代表"假"，用()代表空列表。

在 Lisp 中，除 nil外的任何值，只要不是一个空列表，都被认为是"真"。这意味着，

⊖ 实际上，nil还有第三种意思，表示符号"nil"。

如果一个求值过程返回的是除空列表外的任何内容，if 表达式将视其为“真”。例如，如果测试中是一个数字，它将被求值并将返回它本身，因为对数字求值时就是返回其本身。在这种情况下，if 表达式测试结果为“真”。只有当对表达式求值所返回的值是nil（一个空列表）时，则表达式测试结果为“假”。

对下面两个表达式求值就可以看清这一点。

在第一个例子中，if 表达式中的测试对数字4求值（这返回数字 4 本身），因此if 表达式的 then 部被求值并返回：“true”显示在回显区中。在第二个例子中，nil 意味着“假”，因此 if 表达式的 else 部被求值并返回：“false”显示在回显区中。

```
(if 4
    'true
  'false)
(if nil
    'true
  'false)
```

顺便说一说，如果当测试返回“真”而又无法使用那些适当的值时，Lisp 解释器将返回符号 t 作为“真”。例如，对表达式（> 5 4）求值时返回 t，可以用通常的办法对这个表达式求值并在回显区中看到结果：。

```
(> 5 4)
```

另一方面，如果测试为“假”，函数将返回 nil。

```
(> 4 5)
```

3.10 save-excursion 函数

save-excursion 函数是第四个特殊表，也是在这一章讨论的最后一个特殊表。

在 Emacs 中，Lisp 程序常用作编辑文档，save-excursion 函数在这些程序中很常用。这个函数将当前的位点和标记保存起来，执行函数体，然后，如果位点和标记发生改变就将位点和标记恢复成原来的值。这个特殊表的主要目的是使用户避免位点和标记的不必要移动。

然而，在讨论 save-excursion 函数之前，回顾一下 Emacs 中的位点和标记可能是很有用的。位点（*point*）就是光标所处的当前位意。不管光标在什么地方，那里就是位点。更准确地说，光标在终端上显示在什么字符上，位点就在这个字符前面。在 Emacs Lisp 中，位点是一个整数。缓冲区的第一个字符对应数字 1，第二个字符对应数字 2，以此类推。point函数返回光标的当前位置，其值是一个数。每一个缓冲区都有它自己的位点。

标记（*mark*）是缓冲区中的另外一个位置。它的值可以用一个命令(如C-SPC(set-mark-command))来设置。如果设置了一个标记，可以用命令C-x C-x(exehange-point-and-mark)使光标从位点跳到标记处，并将光标当初所处的位置设置成一个标记。另外，如果设置了另外一个标记，原来标记的位置就被保存在标记环中，用这种方法可以保存许多标记位置。可以一次或者多次键入C-u C-SPC命令来使光标跳到被保存的标记处。

位点和标记之间的缓冲区被称作现域（*region*），或简称域或区域。许多命令是对域操作的，这些命令包括：center-region、count-lines-region、kill-region 和 print-region。

save-excursion 特殊表将位点和标记的当前位置保存起来，并当特殊表主体代码由Lisp解释器执行完毕之后恢复原来的位点和标记的位置。因而，如果位点在一块文本的开头，而某些代码将位点移动到缓冲区末尾，save-excursion 将使位点在函数体的表达式求值完毕后返回到它当初的位置。

在 Emacs 中，有些函数经常移动位点，并将这当做其内部工作的一部分，尽管用户并不希望如此。例如，count-lines-region 函数就移动位点。为了使用户避免非预期的、不必要的位点的改变，save-excursion 函数因此常被用来保持用户期望的位置上的位点和标记。save-excursion 的使用起到了很好的管家作用。

为保持整洁，即使其中的代码出错时（或者更精确地用术语讲，就是在不正常退出时），save-excursion 仍然恢复位点和标记的值。这个特性是非常有帮助的。

除了记录位点和标记的值外，save-excursion 函数也跟踪当前的缓冲区，并恢复它。这意味着，可以编写一些将改变缓冲区的代码，并用 save-excursion切换到原来的缓冲区中。这就是将save-excursion 特殊表用于append-to-buffer 函数的用法。（参见4.4节，“append-to-buffer 函数的定义”。）

save-excursion 表达式模板

使用 save-excursion 特殊表的代码的模板很简单：

```
(save-excursion
  body...)
```

函数体是一个或者多个表达式，这些表达式将被 Lisp 解释器依次求值。如果函数体中有多于一个的表达式，最后一个表达式的值将作为这个 save-excursion 函数的返回值。函数体的其他表达式也被求值，但仅仅产生附带效果。save-excursion 特殊表本身也是仅仅使用它的附带效果（即恢复位点和标记的位置）。

详细地说，save-excursion 表达式的模板如下所示：

```
(save-excursion
  first-expression-in-body
  second-expression-in-body
  third-expression-in-body
  ...
  last-expression-in-body)
```

当然，一个表达式可以是一个符号，也可以是一个列表。

在 Emacs Lisp 代码中，一个 save-excursion 表达式经常出现在一个 let 表达式主体中。它看起来就像这样：

```
(let varlist
  (save-excursion
```

body...))

3.11 回顾

在这几章中，已经介绍了好几个函数和特殊表。在此，简单描述一下已经介绍过的函数和特殊表，并给出没有提到过的一些类似的函数。

• `eval-last-sexp`

对光标所处的位点前的最后一个符号表达式求值。如果这个函数被激活时没有带参量，返回值输出在回显区中。如果这个函数被激活时带有参量，其输出打印在当前缓冲区中。这个命令一般被绑定到`C-x C-e`。

• `defun`

定义函数。这个特殊表最多可以有五个部分：函数名、传送给函数的参量的模板、文档、一个可选的交互函数声明以及函数体。

例如，

```
(defun back-to-indentation ()
  "Point to first visible character on line."
  (interactive)
  (beginning-of-line 1)
  (skip-chars-forward " \t"))
```

• `interactive`

向 Lisp 解释器声明这个函数可以被交互地使用。这个特殊表可以用一个字符串，分成一个部分或者几个部分，依次传送信息到这个函数的参数。这些部分也可以告诉 Lisp 解释器提示这些信息。字符串的每一个部分用换行符“\n”分开。

其中常用到的控制字符是：

`b` 一个已经存在的缓冲区的名字。

`f` 一个已经存在的文件的名字。

`p` 数字前缀参量。（注意，这个字符是小写“p”。）

`r` 位点和标记，作为两个数字参量，小的在前面。这是唯一定义两个连续参量而不是一个参量的控制符。

有关控制符的完整列表，参见《*GNU Emacs Lisp* 技术手册》的“`interactive` 的控制符”一节。

• `let`

声明在 `let` 表达式主体中使用的变量列表并给它们赋初始值，初始值要么是`nil`，要么是一个指定的依；然后对 `let` 表达式主体的其他表达式求值并返回最后一个表达式的值。在 `let` 表达式主体中，Lisp 解释器看不到被绑定在 `let` 表达式之外的同名变量的值。

例如，

```
(let ((foo (buffer-name))
      (bar (buffer-size)))
  (message
   "This buffer is %s and has %d characters."
```

```
foo bar))
```

• save-excursion

在对这个特殊表主体求值前，记录位点和标记的值以及当前缓冲区。求值之后恢复原来位点和标记的值以及缓冲区。

例如，

```
(message "We are %d characters into this buffer.
         (- (point)
            (save-excursion
              (goto-char (point-min)) (point))))
```

• if

对函数的第一个参量求值；如果这个值是“真”，则对第二个参量求值；否则，如果有第三个参量的话就对第三个参量求值。

if 特殊表被称作一个条件表达式。在 Emacs Lisp 中还有其他条件表达式，但是 if 条件表达式可能是其中最经常使用的。

例如，

```
(if (string= (int-to-string 19)
             (substring (emacs-version) 10 12))
    (message "This is version 19 Emacs")
  (message "This is not version 19 Emacs"))
```

• equal、eq

测试两个对象是否相同。如果两个对象有相似的结构和内容，equal则返回“真”。如果两个参量确实是完全相同的对象，则另一个函数 eq 返回“真”。

• < 、 > 、 <= 、 >=

< 函数测试其第一个参量是否小于第二个参量。与之对应的> 函数 则测试其第一个参量是否大于第二个参量。同样地，<= 函数测试其第一个函数是否小于或者等于第二个参量，>= 函数则测试第一个参量是否大于或者等于第二个参量。所有这些函数使用的参量都是数字。

• message

这个函数往回显区中打印一条消息。打印的消息只可以有一行。这个函数的第一个参量是一个字符串，这个字符串中能够包含 “%s”、“%d” 或者 “%c”，以打印字符串后面的参量的值。用来替代“%s”的参量必须是一个字符串或者一个符号；用来替代“%d”的参量必须是一个数字。面用来替代“%c”的参量必须是一个数字，它将打印出具有相应数值的ASCII字符。

• setq、set

setq 函数将其第一个参量的值设置为第二个参量的值。第一个参量由这个setq函数自动地加上引号。这个函数对后续的成对参量执行同样的赋值操作。另外一个 set函数只能接受两个参量，并在将其第一个参量返回的值设置为其第二个参量返回的值之前对它们求值。

• buffer-name

这个函数不需要参量，它将缓冲区的名字以一个字符串的形式返回。

• buffer-file-name

这个函数不需要参量，它返回缓冲区正在访问的文件的名字。

• current-buffer

返回 Emacs 中当前缓冲区的名字，这个当前缓冲区可能并不是屏幕上看到的缓冲区。

• other-buffer

返回最近选择过的缓冲区（既不是作为参量传送给 other-buffer 函数的缓冲区，也不是当前缓冲区。）

• switch-to-buffer

这个函数为 Emacs 选择一个活动的缓冲区，并将它显示在当前的窗口，以使用户能够看到它。这个函数经常被绑定到 C-x b 键序列。

• set-buffer

将Emacs的注意力切换到另外一个运行程序的缓冲区。不要改变当前窗口正在显示的内容。

• buffer-size

返回当前缓冲区中的字符数。

• point

返回当前光标位置对应的值，这个值是从缓冲区的开始处直到光标所在位置所占的总的字符数。

• point-min

返回当前缓冲区中位点的最小可能值。如果变窄没有开启，这个值就是 1。

• point-max

返回当前缓冲区中位点的最大可能值。如果变窄没有开启，这个值就是缓冲区末尾对应的值。

3.12 练习

- 编写一个非交互的函数，这个函数将其第一个参量（是一个数）的值翻倍。然后使这个函数成为交互函数。
- 编写一个函数，测试 fill-column 的当前值是否大于传送给函数的参量的值，如果是，则打印适当的消息。

第4章 与缓冲区有关的函数

在这一章，我们将详细学习几个在 GNU Emacs 中使用的函数。我们称之为一次“浏览”。这些函数是作为 Lisp 代码的例子使用的，但是这些并不是很具创造性的例子。除了其中第一个简化了的函数定义之外，这些函数显示了在 GNU Emacs 中实际使用的代码。你可以从这些函数定义中学到很多东西。这里描述的这些函数都与缓冲区有关。在后面，我们将学习其他一些函数。

4.1 查找更多的信息

在这个浏览中，我将对要遇到的每一个函数进行或者详细或者简单的描述。如果你感兴趣的话，可以键入 C-h f 以及函数名（当然还要加上回车键RET），随时得到任何一个 Emacs Lisp 函数的全部文档。类似地，可以键入 C-h v 和变量名（以及回车键RET）得到任何变量的全部文档。

同样，如果要在一个原始的源代码文件中查看一个函数定义，可以使用find-tags 函数跳到相应的位置。键入 M-.（即同时键入 META 键和句点，或者键入ESC键后再键入句点），然后在提示符下输入要查看源代码的函数的函数名，例如mark-whole-buffer，然后键入回车键RET。Emacs 将切换缓冲区并在屏幕上显示函数的源代码。要切换回原来的缓冲区，键入 C-x b RET。

根据你的 Emacs 拷贝的初始缺省设置值的不同，你可能还需要定义一个“标记表”（tags table），这是一个名为“TAGS”的文件。这个文件最可能在 “emacs/src”目录中，因此应使用 M-x visit-tages-table 命令并指定一个类似“/usr/local/ lib/emacs/19.23/src/TAGS”这样的路径名。关于如何创建自己的“TAGS”文件，可以参见《*GNU Emacs* 技术手册》的“标记表”一节，也可以参见本书的12.5节，“创建自己的‘TAGS’文件”。

当你对 Emacs Lisp 更加熟悉后，你会频繁地使用 find-lags 去浏览源代码，并且可以创建自己的“TAGS”标记表。

顺便说一说，包含 Lisp 代码的文件习惯上称为*库*（*library*）。这种隐喻说法来自于特别定义的库，例如法律库、工程库面非一般的图书馆。每一个库，或者一个文件，都是与某个主题或者某项活动有关的函数。例如，“abbrev.el”用于处理缩写和其他输入快捷键，面“help.el”用于在线帮助。（有时为了某个任务需要几个库，例如各种“rmail...”文件为阅读电子邮件提供代码。）在《*GNU Emacs* 技术手册》中，将看到这样的句子：“C-h p 命令让你用主题关键字搜索 Emacs Lisp 标准库”。

4.2 简化的 beginning-of-buffer 函数定义

beginning-of-buffer 命令是一个很好的开始，因为你可能已经对它有所熟悉，并且

它易于理解。作为一个交互命令，beginning-of-buffer 函数将光标移动到缓冲区的开始位置，在原来的位置设置一个标记。这个函数一般绑定到 M-<。

在这一节中，我们将讨论这个函数的一个简化版本，这个简化版本展示如何经常使用这个函数。这个简化的函数像所编写的那样运作，但是它没有包含处理复杂选项的代码。在另一节，我们将描述这个函数的全部代码（参见5.3节，“beginning-of-buffer 函数的完整定义”）。

在考虑这个函数的代码之前，让我们看看这个函数定义应当包含哪些部分：它必须包含使之成为交互命令的 表达式，这样它才能在键入 M-x beginning-of-buffer 或键入C-<这样的键序列时被调用；它也必须包含在缓冲区中的原始位置保留标记的代码，以及将光标移动到缓冲区开始处的代码。

下面就是这个简化的 beginning-of-buffer 函数的完整代码：

```
(defun simplified-beginning-of-buffer ()
  "Move point to the beginning of the buffer;
leave mark at previous position."
  (interactive)
  (push-mark)
  (goto-char (point-min)))
```

就像所有的函数定义一样，这个定义在特殊表 defun 之后有五个部分：

1) 函数名：在这个例子中，就是 simplified-beginning-of-buffer。

2) 参量列表：在这个例子，是一个空列表（）。

3) 文档字符串。

4) 交互表达式。

5) 函数体。

在这个函数定义中，参量列表是空的。这意味着这个函数无需任何参量。（当讨论这个函数的完整定义时，将看到它可能被传递一个可选参量。）

交互表达式（interactive）告诉 Emacs 这个函数可以被交互地使用。在这个例子中，interactive 没有参量，因为 simplified-beginning-of-buffer 不需要参量。

函数体由两行组成：

```
(push-mark)
(goto-char (point-min))
```

上述两行中的第一行是一个表达式（push-mark）。当 Lisp 解释器对这个表达式求值时，它在光标的当前位置（无论是在哪个位置）设置一个标记。这个标记的位置被保存到标记环中。

第二行是表达式（goto-char (point-min)）。这个表达式将光标跳到本缓冲区的最小可能位点处，也就是缓冲区的开始处（如果变窄开启，就是这个缓冲区中可访问部分的开始处。有关内容参见第6章“变窄和增宽”。

push-mark 命令将一个标记设置在光标移动前所处的位置。光标将根据(goto-char (point-min))表达式的要求移动到缓冲区的开始处。接下来，如果你愿意的话，你可以返回到原来所处的位置，只需键入 C-x C-x 即可。

这就是这个简化函数的完整定义！

当在阅读代码（如这个例子）的过程中遇到不熟悉的函数(如goto-char 函数)时，可以用describe-function 命令找到关于这些函数功能的说明。键入 C-h f 并随后输入函数名和回车键RET，就可以使用这个命令。describe-function命令将在一个“*Help*”窗口中打印函数的文档字符串。例如，goto-char 函数的文档如下所示：

```
One arg, a number.  Set point to that number.
Beginning of buffer is position (point-min),
end is (point-max).
```

(describe-function 命令的提示符将自动提供光标前的那个符号，因此你能够将光标移动到那个不熟悉的函数之后并随后键入 C-h f RET即可得到相应函数的说明，从而可以节省输入的时间)。

end-of-buffer 函数定义就是用类似于 beginning-of-buffer 函数定义的方式编写的，不同之处在于函数体用 (goto-char (point-max)) 表达式代替了（goto-char (point-min)）表达式。

4.3 mark-whole-buffer 函数的定义

mark-whole-buffer 函数并不比 simplified-beginning-of-buffer函数更难理解。然而，在这个例子中，将讨论这个函数的完整代码而不是一个简化的版本。

虽然 mark-whole-buffer 函数并不像 beginning-of-buffer 函数那样被经常使用，但是它依然很有用：它将整个缓冲区作为一个域来标记，方法是将位点置于缓冲区开始的位置，在缓冲区的末尾位置放一个标记。这个命令一般绑定到 C-x h。

这个函数的完整定义代码如下所示：

```
(defun mark-whole-buffer ()
  "Put point at beginning and mark at end of buffer."
  (interactive)
  (push-mark (point))
  (push-mark (point-max))
  (goto-char (point-min)))
```

就像其他函数一样，mark-whole-buffer 函数适合于用作函数定义的模板。模板如下所示：

```
(defun name-of-function (argument-list)
  "documentation..."
  (interactive-expression...)
  body...)
```

这个函数是这样工作的：函数名是 mark-whole-buffer；函数名后面跟着一个空列表“()”，这意味着这个函数无需参量。函数文档就跟在后面。

再下面一行是（interactive）表达式，这个表达式告诉 Emacs 该函数可以被交互地使用。这些细节都与前一节的 simplified-beginning-of-buffer 函数类似。

mark-whole-buffer 的函数体

mark-whole-buffer 函数的函数体由三行组成：

```
(push-mark (point))
(push-mark (point-max))
(goto-char (point-min))
```

其中第一行是表达式（(push-mark (point)）。

这一行的作用与 simplified-beginning-of-buffer 函数的函数体的第一行（push-mark）的作用完全一样。这两个例子中，Lisp 解释器都在当前光标所在的位置设置一个标记。

我不知道为什么在 mark-whole-buffer 函数中这个表达式写成(push-mark (point))这种形式，而在 beginning-of-buffer 函数中这个表达式却写成（push-mark）形式。可能是编写这个代码的人不知道 push-mark 函数的参量是可选的，而且如果没有传送参量给 push-mark，这个函数自动地在当前位点的位置设置标记。或者，这可能是为了与下一行结构一致。无论如何，这一行是使 Emacs 决定位点的位置并在此设置一个标记。

mark-whole-buffer 函数的下一行是（push-mark (point-max)）。这个表达式在缓冲区中数值最大的位点处设置一个标记。这个位点将是缓冲区的末尾（或者，如果变窄 narrowing 开启，就是缓冲区的可访问域的末尾，有关变窄的详细内容，参见第6章“变窄和增宽”）。设置好这个标记之后，原来的标记(即设置在位点上的标记)就不再是标记了，但是 Emacs 记住了它的位置，就像其他最近的标记被记住一样。这就是说，如果你乐意的话，可以通过键入C-u C-SPC 两次来返回到原来的位点处。

最后，这个函数体的最后一行是（goto-char (point-min)）。这种编写方法与 heginning-of-buffer 函数中的编写方法完全类似。这个表达式将光标移动到缓冲区中位点的最小值处，也就是缓冲区的开始处（或者是该缓冲区中可访问域的开始处）。这个函数求值的结果，就是位点被置于缓冲区的开始，标记被设置在缓冲区的末尾。因此，整个缓冲区就是一个域。

4.4 append-to-buffer函数的定义

append-to-buffer 命令几乎就像 mark-whole-buffer 命令一样简单。这个命令的功能就是从当前缓冲区中拷贝一个域（即缓冲区中介于位点和标记之间的区域）到一个指定的缓冲区。

append-to-buffer 命令使用 insert-buffer-substring 函数来拷贝一个域。由 insert-buffer-substring 函数名就可以看出这个函数的功能是：它从一个缓冲区提取一部分作为一个字符串，即“子字符串”（substring），并将这个字符串插入到另外一个缓冲区中。append-to-buffer 函数的绝大部分工作就是为 insert-buffer-substring 函数创建适当的条件：即它的代码必须指定字符串的来源缓冲区和目的缓冲区。下面就是这个函数定义的全部内容：

```
(defun append-to-buffer (buffer start end)
  "Append to specified buffer the text of the region.
```

```
It is inserted into that buffer before its point.

When calling from a program, give three arguments:
a buffer or the name of one, and two character numbers
specifying the portion of the current buffer to be copied."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

通过观察一系列已经填充好的模板，可以逐一理解这个函数。

最外一层模板就是函数定义模板。在这个例子中，它看起来如下所示：

```
(defun append-to-buffer (buffer start end)
  "documentation..."
  (interactive "BAppend to buffer: \nr")
  body...)
```

函数定义的第一行包含了函数名以及它的三个参量。这些参量中，buffer 参量是指拷贝文本的目的缓冲区，start和 end 参量是指将要被拷贝的当前缓冲区中指定域的起始和终止位点。

函数的下一个部分就是函数文档，它是非常清楚和完整的。

4.4.1 append-to-buffer 函数的交互表达式

因为append-to-buffer函数将被交互地使用，所以函数必须有一个 interactive表达式。（有关 interactive 的评述，参见3.3节，“使函数成为交互函数”。）函数中的这个交互表达式读作：

```
(interactive "BAppend to buffer: \nr")
```

这个表达式有一个位于双引号中的参量，这个参量有两部分，其间由“\n”分隔开来。

参量的第一个部分是“BAppend to buffer:”。这里，“B”控制符告诉Emacs要求输入缓冲区名并将这个名字传送给函数。Emacs 将在小缓冲区中打印出“B”字符后面的字符串“Append to buffer:”来提示用户输入这个缓冲区名。然后，Emacs 将函数参量列表中的参量 buffer绑定到指定的缓冲区。

换行符“\n”将参量的两个部分分隔开，参量的第二部分就是“r”。它告诉 Emacs 将函数参量列表中将号“buffer”之后的两个参量（即 start 和 end）绑定到位点和标记的值上。

4.4.2 append-to-buffer 函数体

append-to-buffer 函数的函数体从 let 表达式开始。

就像我们在前面已经看到的那样（参见3.6节，“let参数”），let 表达式的目的是创建一个或者多个在 let 表达式主体中使用的变量，并对它们赋初使。这意味着，这样的变量将不会与 let 表达式之外的同名变量混淆。

通过下面显示的用let表达式的 append-to-buffer 函数的模板，我们将看到let 表达式是如何在整体上符合函数定义的要求。

```
(defun append-to-buffer (buffer start end)
  "documentation..."
  (interactive "BAppend to buffer: \nr")
  (let ((variable value))
       body...)
```

let 表达式有三个元素：

1) 符号let；

2) 一个变量列表，在这个例子中，这个变量列表包含一个两元素列表（*variable value*）

3) let 表达式主体。

在append-to-buffer函数中，变量列表如下所示：

```
(oldbuf (current-buffer))
```

在 let 表达式的这个部分，一个变量（即 oldbuf）被绑定到由（current-buffer）表达式返回的值上。这个oldbuf变量用于跟踪当前的缓冲区。

变量列表的元素是由一组括号包围起来的，因此 Lisp 解释器能够将变量列表从let 表达式主体中区分出来。结果，变量列表中的这个两元素列表被一组限制性的括号包围起来。这一行如下所示：

```
(let ((oldbuf (current-buffer)))
  ... )
```

oldbuf 前面的第一个括号标明的是变量列表的界限，而第二个括号标明的是这个两元素列表（oldbuf (current-buffer)）的开始，如果你没有意识到这一点，oldbuf 前面的这两个括号可能会使你大吃一惊。

4.4.3 append-to-buffer函数中的 save-excursion

append-to-buffer 函数中 let 表达式的主体由一个 save-excursion 表达式组成。

这个 save-excursion 函数保存位点和标记的位置，并当这个函数体中的其他表达式都被求值之后恢复位点和标记到相应位置。另外，save-excursion 保存原始的缓冲区并恢复它。这就是在 append-to-buffer 函数中如何使用 save-excursion 函数的方法。

顺便提一下，值得注意的是：一个 Lisp 函数一般都具有很好的格式，因此在多行展开中包含的所有内容比第一个符号缩进得更多。在下面这个函数定义中，let 表达式就比 defun 缩进得更多，而 save-excursion 表达式比 let 表达式又缩进更多：

```
(defun ...
  ...
  ...
  (let...
    (save-excursion
      ...
```

从这种格式化编排约定可以很容易看出 save-excursion 表达式主体中的两行由与 save-excursion 联系在一起的括号包围，就像 save-excursion 表达式本身由与 let 表达式联系在一起的括号包围一样：

```
(let ((oldbuf (current-buffer)))
  (save-excursion
    (set-buffer (get-buffer-create buffer))
    (insert-buffer-substring oldbuf start end))))
```

save-excursion 函数的使用可以被看做是填充下面这个模板的一个过程：

```
(save-excursion
  first-expression-in-body
  second-expression-in-body
  ...
  last-expression-in-body)
```

在这个函数中，save-excursion 表达式的主体仅仅包含两个表达式。这个表达式如下所示：

```
(set-buffer (get-buffer-create buffer))
(insert-buffer-substring oldbuf start end)
```

当对 append-to-buffer 函数求值时，save-excursion 主体中的两个表达式依次被求值。后一个表达式的值作为 save-excursion 函数的值被返回，而第一个表达式被求值仅仅是一个附带效果。

save-excursion 函数主体的第一行使用 set-buffer 函数来将当前缓冲区变换到另外一个指定的缓冲区。这个指定的缓冲区就是用 append-to-buffer 函数的第一个参量指出的另一个缓冲区。（改变缓冲区仅仅是它的附带效果；就像我们在前面说到的那样，在 Lisp 中，附带效果经常是我们所需要的。）第二行完成这个函数的主要工作。

set-buffer 函数将 Emacs 的注意力转移到文本将要拷贝到的目的缓冲区，面save-excursion 函数将从这个缓冲区返回。完成这一工作的一行表达式如下所示：

```
(set-buffer (get-buffer-create buffer))
```

这个列表中最内层的表达式是（get-buffer-create buffer）。这个表达式使用了 get-buffer-create 函数，get-buffer-create 函数要么获得已经命名的缓冲区，要么用给定的名字创建一个缓冲区（如果给定名字没有对应的缓冲区）。这意味着可以使用 append-to-buffer 函数将一段文本放到一个原本不存在的缓冲区中。

get-buffer-create 函数同时使 set-buffer 避免了一个不必要的错误：set-buffer 函数需要一个缓冲区；如果你指定一个实际上不存在的缓冲区给它，Emacs 将阻止你这样做。因为如果缓冲区不存在，get-buffer-create 将创建一个缓冲区，因此 set-buffer 函数总会得到一个存在的缓冲区。

append-to-buffer 的最后一行所做的工作就是增添文本：

```
(insert-buffer-substring oldbuf start end)
```

insert-buffer- substring函数从作为其第一个参量指定的缓冲区中拷贝一个字符串，并将其插入到当前的缓冲区中。在这个例子中，传送给 insert-buffer-substring 的参量是由 let 创建并绑定的变量的值，也就是 oldbuf 的值，它是当发出 append-to-buffer 命令时的当前缓冲区。

insert-buffer-substring 函数执行完之后，save-excursion 将恢复对原来缓冲区的操作，并且append-to-buffer 将完成其工作。

用一个粗略的框架来描述这个函数，其函数体的工作如下所示：

```
(let (bind-oldbuf-to-value-of-current-buffer)
  (save-excursion                          ; Keep track of buffer.
    change-buffer
    insert-substring-from-oldbuf-into-buffer)

  change-back-to-original-buffer-when-finished
let-the-local-meaning-of-oldbuf-disappear-when-finished
```

总之，append-to-buffer 函数是这样工作的：它在变量 oldbuf 中保存当前缓冲区的值；并获得一个新的缓冲区(如果需要的话就创建一个新的缓冲区)，然后使 Emacs 切换到这个缓冲区。使用 oldbuf 的值，这个函数将来自原来缓冲区的文本域插入到新的缓冲区中，然后用 save-excursion 函数返回到原来的缓冲区。

在考查这个 append-to-buffer 函数时，你已经深入接触了一个相当复杂的函数。它演示了如何使用 let 和 save-excursion，以及如何变换到其他缓冲区并返回原来的缓冲区。许多函数定义中都是这样使用 let、save-excursion 和 set-buffer的。

4.5 回顾

下面简单地小结一下本章讨论过的函数。

• describe-function、describe-variable

打印一个函数或者一个变量的文档。习惯上将它绑定到 C-h f 和 C-h v。

• find-tag

找到存放某个函数或者变量的源代码的文件，并切换到这个缓冲区，将位点（光标）置于相应函数或者变量的开始处。习惯上将它绑定到 M-。

• save-excursion

保存位点和标记的位置，并在 对save-excursion 参量求值之后恢复这些值。它也保存当前缓冲区并返回到该缓冲区。

• push-mark

在指定位置设置一个标记，并在标记环中记录原来标记的值。标记是缓冲区中的一个位置，即使有一些文本被从缓冲区删除或者增加到缓冲区，标记仍将保持它的相对位置。

• goto-char

将位点设置为由参量值指定的位置。参量值可以是一个数，也可以是一个标记，甚至可以是一个返回一个位置的数字的表达式，如（point-min）。

• insert-buffer-substring

将来自一个缓冲区（这是被作为一个参量而传递给函数的）的文本域拷贝到当前缓冲区。

• mark-whole-buffer

将整个缓冲区标记为一个域。一般将这个函数绑定到 C-x h。

• set-buffer

将 Emacs 的注意力转移到另一个缓冲区，但是不改变显示的窗口。这通常是由另外的人在不同的缓冲区中执行程序时使用。

• get-buffer-create、get-buffer

寻找一个已指定名字的缓冲区，或当指定名字的缓冲区不存在时就创建它。如果指定名字的缓冲区不存在，get-buffer 函数就返回 nil。

4.6 练习

- 编写自己的 simplified-end-of-buffer 函数定义，然后测试它是否能工作。
- 用 if 和 get-buffer 编写一个函数，这个函数要打印一个说明某个缓冲区是否存在的消息。
- 用find-tag找到 copy-to-buffer 函数的源代码。

第5章 更复杂的函数

在这一章中，在前面几章已经学习过的内容的基础上考察几个更复杂的函数。copy-to-buffer 函数展示了在一个函数定义中使用两次save-excursion表达式的情况，而 insert-buffer 函数展示了在一个 interactive 表达式中使用 *以及使用 or 函数，并且还将展示一个名字和这个名字所指的对象两者之间的重要区别。

5.1 copy-to-buffer 函数的定义

理解了 append-to-buffer 函数是如何工作的之后，copy-to-buffer 函数就很容易理解了。这个函数将文本拷贝进一个缓冲区，它不是追加到原有缓冲区，而是替换了原有缓冲区中的文本。copy-to-buffer 函数的代码几乎与append-to-buffer 函数的代码完全一样，不同之处仅在于它使用了 erase-buffer 函数，并两次使用了 save-excursion 函数。(关于 append-to-buffer 的描述参见4.4节，“append-to-buffer 函数的定义”。)

copy-to-buffer 函数体如下所示：

```
...
(interactive "BCopy to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (set-buffer (get-buffer-create buffer))
      (erase-buffer)
      (save-excursion
        (insert-buffer-substring oldbuf start end)))))
```

这些代码与 append-to-buffer 函数定义中的代码类似：仅在切换到要拷贝文本的缓冲区之后的那部分代码与 append-buffer 函数代码开始不同—— copy-to-buffer 函数删除了缓冲区中原有的内容。(这就是“取代”一词的含义；取代文本，对 Emacs 而言就是删除原有的内容后插入新的文本。)在删除缓冲区的原有内容之后，save-excursion 第二次被使用，新的文本被插入进来。

为什么要两次使用 save-excursion 函数呢？再一次考查这个函数做些什么。

在结构上说，copy-to-buffer 函数体如下所示：

```
(let (bind-oldbuf-to-value-of-current-buffer)
  (save-excursion              ; First use of save-excursion.
    change-buffer
      (erase-buffer)
      (save-excursion          ; Second use of save-excursion.
        insert-substring-from-oldbuf-into-buffer)))
```

第一次使用 save-excursion 函数使 Emacs 返回到将要拷贝文本的缓冲区。这是清楚的，

就像在 append-to-buffer 函数中使用的那样。为什么要第二次使用save-excursion函数呢? 原因在于 insert-buffer-substring 函数总是在被插入内容的缓冲区的域的末尾设置位点。函数定义中的第二个 save-excursion 函数使Emacs 在被插入内容的缓冲区的域的开始设置位点。在绝大多数情况下，用户更希望位点在被插入文本的开始处。(当然，copy-to-buffer 函数返回到原来的缓冲区——但是如果用户切换到文本插入的缓冲区，位点将处于文本的开始处。因而，第二次使用这个 save-excursion 函数是不错的。)

5.2 insert-buffer 函数的定义

insert-buffer 函数是另外一个与缓冲区有关的函数。这个命令将另外一个缓冲区内容拷贝到当前缓冲区中。它与 append-to-buffer 或者 copy-to-buffer 函数正好相反，因为它们是从当前缓冲区中拷贝一个域内的文本到另外一个缓冲区。

除此之外，这个函数的代码显示了如何在 interactive 表达式中使用一个缓冲区。这个缓冲区可能是只读的。同时这个函数展示了一个对象的名字与这个对象本身实际所指的内容之间的重要区别。下面就是这个函数的代码：

```
(defun insert-buffer (buffer)
  "Insert after point the contents of BUFFER.
Puts mark after the inserted text.
BUFFER may be a buffer or a buffer name."
  (interactive "*bInsert buffer: ")
  (or (bufferp buffer)
      (setq buffer (get-buffer buffer)))
  (let (start end newmark)
    (save-excursion
      (save-excursion
        (set-buffer buffer)
        (setq start (point-min) end (point-max)))
      (insert-buffer-substring buffer start end)
      (setq newmark (point)))
    (push-mark newmark)))
```

就像其他的函数定义一样，可以使用函数定义的模板来分析这个函数的框架：

```
(defun insert-buffer (buffer)
  "documentation..."
  (interactive "*bInsert buffer: ")
  body...)
```

5.2.1 insert-buffer 函数中的交互表达式

在insert-buffer函数中，给 interactive 表达式说明的参量有两个部分：一部分是一个星号“*”，另一部分是 “bInsert buffer:”。

1. 只读缓冲区

星号“*” 用于缓冲区是一个只读缓冲区的情况，只读缓冲区就是指一个不能改变内容的缓

冲区。如果 insert-buffer 被一个只读缓冲区调用，一条消息将打印在回显区，终端将发出蜂鸣或者闪亮一下，警告不允许往这个缓冲区插入任何东西。星号这个控制符无需后接一个换行符来分隔不同的参量。

2. 交互表达式中的 "b"

在交互表达式中的下一个参量以 "b" 字符开始。(这一点与 append-to-buffer 函数的代码不同，在那个函数定义中使用一个大写的 "B"。参见4.4节，"append-to-buffer 函数的定义"。)小写的 "b" 告诉 Lisp 解释器传送给 insert-buffer 函数的参量应是一个存在的缓冲区或者这个缓冲区的名字。(大写的 "B" 可以允许参量传送不存在的缓冲区)。Emacs 将提示你输入缓冲区的名字，并为你提供一个默认的缓冲区。如果这个缓冲区不存在，你将收到一条消息告之 "找不到该缓冲区"，你的终端也将对你发生蜂鸣叫声。

5.2.2 insert-buffer 函数体

insert-buffer 函数体有两个主要的部分：一个 or 表达式和一个 let 表达式。or 表达式的目的是为了确保 buffer 参量真正与一个缓冲区绑定在一起，而不是绑定到缓冲区的名字。let 表达式包含了将另外一个缓冲区的内容拷贝到当前缓冲区所需的代码。

从结构上说，适合insert-buffer 函数的两个表达式如下所示：

```
(defun insert-buffer (buffer)
  "documentation..."
  (interactive "*bInsert buffer: ")
  (or ...
      ...
  (let (varlist)
       body-of-let... )
```

为了理解 or 表达式如何确保参量 buffer 确实是绑定到一个缓冲区，而不是绑定到一个缓冲区的名字，首先需要理解 or 函数本身。

在讲述 or 函数之前，让我用 if 表达式重新改写函数的这一部分，这样你就可以看到这是如何以另外一种方式完成类似的工作的。

5.2.3 用 if 表达式(而不是or表达式)编写的 insert-buffer 函数

用 if 表达式来改写这个部分代码，是为了确保 buffer 的值是缓冲区本身而不是缓冲区的名字。如果 buffer 的值是缓冲区的名字，则一定要设法得到缓冲区本身。

可以想象，好比在一个会议上，一个引座员手拿一本印有你大名的花名册寻找你：引座员绑定到你的名字，而不是你，但是当引座员找到你并拉住你的手的时候，引座员就绑定到你了。

在 Lisp 中，可以这样描述这种情况：

```
(if (not (holding-on-to-guest))
    (find-and-take-arm-of-guest))
```

我们希望对一个缓冲区做同样的事情——如果没有获得缓冲区本身，就要设法得到它。

一个名为 bufferp 的谓词会告诉我们是否得到一个缓冲区本身(而不是缓冲区的名字)，

因此可以这样编写代码：

```
(if (not (bufferp buffer))              ; if-part
    (setq buffer (get-buffer buffer)))  ; then-part
```

这里，这个 if 表达式中的真假测试是 (not (bufferp buffer))，它的then部是表达式 (setq buffer (get-buffer buffer))。

在这个测试中，如果其参量是一个缓冲区， bufferp 函数返回值为“真”，但是如果其参量是一个缓冲区的名字，则函数 bufferp 的返回值为“假”。(bufferp 函数名的最后一个字符是“p”；就像前面看到的，这样使用“p”是一种习惯，它表示这个函数是一个谓词，也就是这个函数将判断一些特性的真假。参见1.8.4节，“用一个错误类型的数据对象作为参量”。)

not 函数在表达式 (bufferp buffer) 之前，因此真假测试就是：

```
(not (bufferp buffer))
```

not 函数的功能是：如果其参量为“假”，则其返回“真”；如果其参量值为“真”，则其返回“假”。因此，如果(bufferp buffer)表达式返回“真”，则 not 表达式返回“假”；反之亦然：“不真”就是“假”，“不假”就是“真”。

采用了这个真假测试后，if 表达式便这样工作：当变量 buffer 的值确实是一个缓冲区而不是它的名字时，这个真假测试返回“假”，if 表达式不对 then 部求值。这正是我们需要的，因为如果变量 buffer 的值真的是一个缓冲区，那么确实无需对它再做什么。

另一方面，当 buffer 的值不是缓冲区本身而是缓冲区的名字时，真假测试返回“真”，因而then部被求值。在这个例子中，if 表达式的then部是 (setq buffer (get-buffer buffer))。这个表达式用 get-buffer 函数通过缓冲区的名字来获得这个实际的缓冲区。setq 则将缓冲区本身的值设置给变量 buffer，以取代它的原先值(即这个缓冲区的名字)。

5.2.4 函数体中的 or 表达式

insert-buffer 函数中的 or 表达式的目的是确保 buffer 参量绑定到一个缓冲区而不仅仅是一个缓冲区的名字。前面一节介绍了这可以用 if 表达式完成。然而，insert-buffer 函数实际上使用了 or 表达式。为了理解这一点，有必要弄懂 or 表达式是如何工作的。

一个 or 函数可以有很多参量。它逐一对每一个参量求值并返回第一个其值不是nil的参量的值。同样，这是 or 表达式的一个重要特性，一旦遇到其值不是 nil 的参量之后，or 表达式就不再对后续的参量求值。

or 表达式如下所示：

```
(or (bufferp buffer)
    (setq buffer (get-buffer buffer)))
```

or 表达式的第一个参量是(bufferp buffer)。如果变量 buffer确实对应着一个缓冲区，则这个表达式返回“真”(一个非空值，not-nil)；如果 buffer 仅仅是一个缓冲区的名字，则这个表达式返回“假”。在这个 or 表达式中，如果确实是这么一回事，则or 表达式返回这个“真”值，而且不再对其下一个表达式求值——这正是我们需要的，如果 buffer 确实是一个缓

冲区，无需对 buffer 的值做任何事情。

另一个方面，如果（bufferp buffer）的值是nil，Lisp 解释器则对 or表达式的下一个元素求值。这种情况当buffer的值是一个缓冲区的名字时就会发生。下一个元素就是表达式（setq buffer (get-buffer buffer)）。这个表达式返回一个非空值，其返回值就是变量 buffer 设置的值——也就是缓冲区本身，而不是缓冲区的名字。

所有这些代码的执行结果是：符号 buffer 总是绑定到一个缓冲区本身面不是缓冲区的名字。之所以需要这些代码，是因为后面的 set-buffer 函数仅仅对一个缓冲区起作用，面不对缓冲区的名字起作用。

顺便提一下，用 or 函数，那么引座员的情况可以写成下面的形式：

```
(or (holding-on-to-guest) (find-and-take-arm-of-guest))
```

5.2.5 insert-buffer 函数中的let表达式

在确保变量 buffer 是指向一个缓冲区而不是一个缓冲区的名字之后，insert-buffer 函数继续使用 let 表达式。这定义了三个变量：start、end 和 newmark，并将它们绑定到初始值 nil。这些变量在 let 表达式内部使用，并暂时屏蔽 Emacs中所有同名的变量直到 let 表达式结束。

let 表达式的主体包含了两个 save-excursion 表达式。首先，将详细分析内层的那个 save-excursion 表达式。这个表达式如下所示：

```
(save-excursion
  (set-buffer buffer)
  (setq start (point-min) end (point-max)))
```

表达式 (set-buffer buffer) 将 Emacs 的注意力从当前的缓冲区切换到要从中拷贝文本的缓冲区。在那个缓冲区中，用 point-min 和 point-max 函数将变量 start 和 end 设置成该缓冲区的开始处和结束处。注意，这里已经演示了 setq 如何在一个表达式中给两个变量赋值。setq 的第二个参量的值被赋给第一个参量，第四个参量的值被赋给第三个参量。

内层的 save-excursion 表达式被求值之后，它恢复原来的缓冲区，但是 start 和 end 变量仍然被设置成从中拷贝文本的缓冲区的开始处和结束处。

外层的 save-excursion表达式如下所示：

```
(save-excursion
  (inner-save-excursion-expression
     (go-to-new-buffer-and-set-start-and-end)
  (insert-buffer-substring buffer start end)
  (setq newmark (point)))
```

insert-buffer-substring 函数将文本从由 buffer 中的 start 和 end 变量界定的区域拷贝到当前缓冲区。因为第二个缓冲区的全部内容都在 start 和 end 之间，因此第二个缓冲区的所有内容都被拷贝到你正在编辑的当前缓冲区。接下来，位于插入文本末尾的位点的值记录在变量 newmark 中。

外层的 save-excursion 被求值之后，位点和标记被重新定位到它们原来的位置。

然而，习惯上，标记的位置一般在新插入的文本之后，而位点的位置在新插入的文本的开始处。newmark 变量记录新插入的文本的末尾。在 let 表达式的最后一行，(push-mark newmark)表达式在这个位置设置了一个标记。(上一个标记的位置仍然可以找到，它记录在标记环中，可以键入 C-u C-SPC 返回到原来标记处。)同时，位点设置在插入的文本的开始处，也就是位点设置在你调用插入函数之前所处的位置。

整个 let 表达式如下所示：

```
(let (start end newmark)
  (save-excursion
    (save-excursion
      (set-buffer buffer)
      (setq start (point-min) end (point-max)))
    (insert-buffer-substring buffer start end)
    (setq newmark (point)))
  (push-mark newmark))
```

就像 append-to-buffer 函数那样，insert-buffer 函数使用了 let、save-excursion 和 set-buffer 函数。除此之外，这个函数展示了使用 or 函数的一个方法。在后面将一次又一次看到这些函数，它们都是构建 Lisp 程序的基本函数。

5.3 beginning-of-buffer 函数的完整定义

在前面已经讨论过beginning-of-buffer 函数的基本结构。(参见4.2节，“简化的 beginning-of-buffer 函数定义”。)本节描述这个函数定义的复杂部分。

就像前面描述的，不带参量激活 beginning-of-buffer 函数时，它将光标移动到缓冲区的开始处，并在原来光标的位置设置一个标记。然而，当带参量激活 beginning-of-buffer 函数时，如果参量是介于 1 和 10 之间的一个数，则该函数认为那个数是指缓冲区长度的十分之几，而且 Emacs 将光标移动到从缓冲区开始到这个分数值所指示的位置。因此，可以用键序列 M-< 调用这个函数，这是指将光标移动到缓冲区的开始处，也可以用这样的命令调用这个函数：C-u 7 M-<，这是指将光标移动到从缓冲区开始的这个缓冲区的70%处。如果这个作为参量的数大于 10，函数则将光标移动到缓冲区的末尾。

beginning-of-buffer 函数可以带参量调用，也可以不带参量调用，参量的使用是可选的。

5.3.1 可选参量

除非已经声明，否则 Lisp 总是希望一个函数定义中带参量的函数在被调用时要传递一个值给该参量。如果没有传递相应的值，函数就会出错，并得到这样一个错误消息：“Wrong number of arguments”。

然而，可选参量是 Lisp 的一个特性：有一个关键词可以用于告诉 Lisp 解释器某个参量是可选的。这个关键词是 &optional（在单词“optinal”之前的符号“&”是关键词的一部分）。

在一个函数定义中，如果一个参量跟在 &optinal 这个关键词后面，则当调用这个函数时就不一定要传送一个值给这个参量。

因此，beginning-of-buffer 函数定义的第一行就变成如下所示的形式：

```
(defun beginning-of-buffer (&optional arg)
```

从结构上说，整个函数如下所示：

```
(defun beginning-of-buffer (&optional arg)
  "documentation..."
  (interactive "P")
  (push-mark)
  (goto-char
    (if-there-is-an-argument
        figure-out-where-to-go
      else-go-to
      (point-min))))
```

除了这个函数在 interactive 表达式中使用了“P”参量以及将 goto-char 函数用在一个条件表达式中以判断将光标移动到何处之外，上面这个函数与simplified-beginning-of-buffer 函数很相似。

interactive 表达式中的“P”参量告诉 Emacs，如果有参量的话，就传递一个前缀参量给这个函数。一个前缀参量由键入 META 键以及后接的一个数组成的，或者由键入 C-u 和一个后接的数组成（如果你没有键入一个数，C-u默认为4）。

上面的函数定义中，if 条件表达式的真假测试很简单：它只是参量 arg 而已。如果参量 arg 有一个非空（nil）值，即当 beginning-of-buffer 函数带参量调用时，真假测试返回“真”，并且 if 表达式中的 then 部被求值；另一方面，如果不带参量调用 beginning-of-buffer 函数，这个 arg 参量为 nil，并且if 表达式的 else 部被求值。if 表达式的 else 部只是 point-min，并且当这就是 if 表达式的结果时，整个 goto-char 表达式就是 (goto-char (point-min))，这就是我们在前面看到的 beginning-of-buffer 函数的简化形式。

5.3.2 带参量的 beginning-of-buffer 函数

当带参量调用 beginning-of-buffer 函数时，就要有一个表达式计算应该传递什么值给 goto-char 表达式。这个表达式初看起来似乎相当复杂，它包含一个 if 表达式和许多算术计算。这个表达式如下所示：

```
(if (> (buffer-size) 10000)
    ;; Avoid overflow for large buffer sizes!
    (* (prefix-numeric-value arg) (/ (buffer-size) 10))
  (/
   (+ 10
      (*
       (buffer-size) (prefix-numeric-value arg))) 10))
```

就像其他复杂的表达式一样，这个表达式也可以用模板来一一揭开其中的奥秘。在这个例子中，模板就是 if 表达式模板。当用结构框架来看这个表达式时，这个表达式如下所示：

```
(if (buffer-is-large
     divide-buffer-size-by-10-and-multiply-by-arg
  else-use-alternate-calculation
```

在这个内层的 if表达式中，真假测试用于检查缓冲区的大小。之所以要检查缓冲区的大小是因为第18版的Emacs Lisp使用了不大于 8 000 000 的数字来描述缓冲区的大小（更大的数就不需要了），并当在后续的计算中遇到很大的缓冲区时，Emacs 就试图使用超大的数来描述它。在注释中提到的术语“overflow”(溢出)是指所用的数太大了。

这里有两种情况：缓冲区很大或者并不大。

1. 大缓冲区的情况

在 beginning-of-buffer 函数中，内层的 if 表达式判断缓冲区是否大于10 000 个字符。它使用 > 函数和 buffer-size 函数来完成这一工作：

```
(if (> (buffer-size) 10000)
```

当缓冲区大于 10 000 时，if 表达式的 then 部被求值。其中 then 部如下所示：

```
(*
  (prefix-numeric-value arg)
  (/ (buffer-size) 10))
```

这个表达式是一个乘法，*函数有两个参量。

其中的第一个参量是 (prefix-numeric-value arg)。当在 interactive 表达式中使用“P”参量时，作为函数参量传给函数的值是以一个“未加工的前缀参量”(raw prefix argument) 的形式传递的。(它是在一个列表中的一个数。）为了执行算术运算，有必要对它进行变换，prefix-numeric-value 函数就是完成这一工作的。

其中的第二个参量是 (/ (buffer-size) 10)。这个表达式将缓冲区的大小（数字）除以 10。这个表达式产生一个数，这个数就是指缓冲区大小的十分之一有多少字符。（在 Lisp 中，/用于除法，就像 * 用于乘法一样）。

在乘法表达式中，这个数作为一个整体乘以前缀参量的值，乘法的结构如下所示：

```
(* numeric-value-of-prefix-arg
   number-of-characters-in-one-tenth-of-the-buffer)
```

例如，如果前缀参量是“7”，则缓冲区的十分之一的值乘以 7 得到缓冲区的 70%。

如果是大缓冲区，则所有这些代码的最后结果就使 goto-char 表达式变成这样：

```
(goto-char (* (prefix-numeric-value arg)
              (/ (buffer-size) 10)))
```

这个表达式的功能是将光标置于我们需要的地方。

2. 小缓冲区的情况

如果缓冲区中包含的字符数少于 10 000 个，就要执行一个稍微不同的计算。你可能认为这不必要，因为前面的计算可以完成这个工作。然而，在一个小缓冲区中，第一种方法无法精确地将光标置于所需的那一行。这第二种方法可以更好地做到这一点。

这部分代码是：

```
(/ (+ 10 (* (buffer-size) (prefix-numeric-value arg))) 10))
```

这个函数代码看似有些复杂，但是通过逐一分析函数是如何嵌入到括号中，就可以清楚地分析出最后的结果。如果以缩进的方式重写每一个表达式，就可以更容易地阅读它。

```
(/
 (+ 10
    (*
     (buffer-size)
     (prefix-numeric-value arg)))
 10))
```

检查这些括号，会发现最内层的操作是 (prefix-numeric-value arg)，即把未加工的前缀参量转换成一个数。这个数在下面的表达式中乘以缓冲区的大小:

```
(* (buffer-size) (prefix-numeric-value arg)
```

这个乘法的结果是产生一个数，这个数可能大于缓冲区的大小——如果参量是 7，就是缓冲区的 7 倍。然后这个数再加 10，最后用这个结果除以 10，这样产生的数比缓冲区中相应比例仅仅多一个字符。

所有这些代码执行后产生的最终的一个数被传递到 goto-char 函数，并且光标就移动到那个位点。

5.3.3 完整的 beginning-of-buffer 函数

下面是 beginning-of-buffer 函数的完整形式:

```
(defun beginning-of-buffer (&optional arg)
  "Move point to the beginning of the buffer;
leave mark at previous position.
With arg N, put point N/10 of the way
from the true beginning.
Don't use this in Lisp programs!
\(goto-char (point-min)) is faster
and does not set the mark."
  (interactive "P")
  (push-mark)
  (goto-char
   (if arg
       (if (> (buffer-size) 10000)
           ;; Avoid overflow for large buffer sizes!
           (* (prefix-numeric-value arg)
              (/ (buffer-size) 10))
         (/ (+ 10 (* (buffer-size)
                     (prefix-numeric-value arg)))
            10))
     (point-min)))
  (if arg (forward-line 1)))
```

除了两个小点外，前面的讨论展示了这个函数是如何工作的。第一点处理文档字符串中的细节，第二点关于函数的最后一行。

在文档字符串中，提到了这样一个表达式：

```
\(goto-char (point-min))
```

其中的第一个括号之前有一个“\”符号。这个符号告诉 Lisp 解释器将这个表达式作为文档打印出来，而不作为一个符号表达式对它求值。

最后，当这个函数带参量调用时，beginning-of-buffer 函数的最后一行是让光标移动到后续一行的开始处：

```
(if arg (forward-line 1))
```

这个命令将光标置于缓冲区中相应于前缀参量值的位置的后续第一行的行首。这是一个好主意，它意味着光标总是置于缓冲区中至少是需要的位置。我们当然希望光标精确地置于需要到达的位置，但是这并不是必须的。如果没有精确地置于需要到达的位置，也不要抱怨太多。

5.4 回顾

下面是这一章中讨论的部分主题的一个简要小结：

- or

逐一对每一个参量求值，并返回第一个非空值（不是 nil）。如果所有参量的值都是 nil，就返回 nil。简要地说，它返回参量的第一个“真”值；如果一个参量或者其他任何参量的值为“真”时，则返回“真”值。

- and

逐一对每一个参量求值，如果有任何一个参量的值为 nil，则返回 nil。如果没有 nil 值，则返回最后一个参量的值。简要地说，仅当所有参量都是“真”值时，它才返回一个“真”值；如果一个参量和其他所有参量的值都是“真”值时，则返回“真”值。

- &optional

在函数定义中用于指出一个参量是可选参量。这意味着这个函数可以带参量调用，也可以不带参量调用。

- prefix-numeric-value

将一个由 (interactive "P") 产生的未加工的前缀参量转换成一个数值。

- forward-line

将光标移动到下一行的行首，如果其参量大于 1，则移动多行。如果无法移动所需的行数，forward-line 就移动尽可能多的行数，并返回它实际少移动的行数。

- erase-buffer

删除当前缓冲区的全部内容。

- bufferp

如果其参量是一个缓冲区则返回“真”，否则返回“假”(nil)。

5.5 &optional 参量练习

编写一个带可选参量的交互函数，这个函数要测试函数被调用时是否有参量（其值是一个数），这个数是否大于或小于 fill-column 的值，并将结果以一个消息的形式给出。然而，如果不带参量调用这个函数时，则使用 56 作为默认值。

第6章 变窄和增宽

“变窄”(narrowing)是Emacs的一个特性，这个特性允许你让 Emacs 关注于一个缓冲区的特定部分，而不会在无意中更改缓冲区的其他部分。变窄一般都是没有开启的，因为它会将新手弄得不知所措。

采用变窄技术之后，缓冲区的其余部分就变成不可见的了，就像它们并不存在一样。这样做有一个好处，例如，要在缓冲区的某个部分而不是在别的部分替换一个单词：首先将那个部分隔离出来，替换工作就在缓冲区的这个变窄的部分进行，而不在缓冲区的其余部分进行。查询也是在缓冲区的一个变窄的部分进行，而不是在缓冲区的其他部分进行。因此，如果你正在修改一个文档，你可以使自己严格限制在要修改的那个部分，而不要跑到其他部分去。只要用变窄技术就可以实现这一点。

然而，变窄确实使缓冲区的其余部分不可见，这会使那些无意中设置了变窄功能的人惊恐不安，并认为他们已经删除了文档的一部分。更有甚者，undo 命令（这个命令经常绑定到 C-x u）无法取消变窄开启（或者不应当），因此，如果人们不知道可以用 widen 命令使其余部分意新变成可见的，他们就会非常绝望。(在 Emacs 第18版中，widen 命令一般绑定到C-x w；而在第19版中，则绑定到 C-x n w。)

变窄技术不仅对人有用，而且对 Lisp 解释器也同样有用。通常，一个 Emacs Lisp 函数被设计成针对缓冲区的一个区域操作，或者反过来说，Emacs Lisp 函数需要在一个变窄开启的缓冲区中执行。例如，如果一个缓冲区设置了变窄开启，what-line 函数从缓冲区中取消变窄开启，然后完成它本身的工作，随后再恢复缓冲区中原来的变窄开启。另一方面，由 what-line 调用的 count-line 函数则用变窄技术来将这个函数的执行范围限制到缓冲区中你感兴趣的那个部分，并随后恢复原来的状态。

6.1 save-restriction特殊表

在 Emacs Lisp 中，能用save-restriction特殊表来跟踪变窄开启的部分。当 Lisp 解释器遇到 save-restriction 特殊表时，它执行这个表达式中的代码，并恢复这些代码导致的变窄开启的变更。例如，如果缓冲区本来是变窄开启的，而 save-restriction表达式中的代码取消了变窄开启，save-restriction 特殊表就返回变窄开启的缓冲区部分。在 what-line 命令中，缓冲区中可能包含的变窄开启都可以用紧跟在 save-restriction 特殊表后而的widen 命令取消。在这个函数执行完之前，所有的变窄开启都被恢复了。

save-restriction 表达式的模板很简单：

```
(save-restriction
  body... )
```

save-restriction 特殊表的主体是一个或多个表达式，这些表达式将被Lisp 解释器逐一求值。

最后，有一点值得提醒一下：当你同时使用 save-excursion和 save-restriction 时（并且是一个紧接着另一个使用时），应当在外层使用 save-excursion。如果采用了相反的次序，就会在调用 save-excursion 之后无法记录缓冲区中变窄开启的标记。因而，当同时使用这两个特殊表时，应当采用类似下面的顺序：

```
(save-excursion
  (save-restriction
    body...))
```

6.2 what-line函数

what-line 命令告诉你光标所在的行数。这个函数展示了如何使用save-excursion 和 save-restriction 命令。下面是这个函数的全部代码：

```
(defun what-line ()
  "Print the current line number (in the buffer) of point."
  (interactive)
  (save-restriction
    (widen)
    (save-excursion
      (beginning-of-line)
      (message "Line %d"
               (1+ (count-lines 1 (point)))))))
```

这个函数有一个文档说明行，并且该函数就像你希望的那样，也是一个交互函数。函数定义中接下来的两行使用了 save-restriction 和 widen 函数。

save-restriction 特殊表判断当前缓冲区是否设置了变窄开启，如果设置了，就在 save-restriction 特殊表中的所有代码执行完之后恢复变窄开启。

上面的代码中，save-restriction 特殊表之后紧跟了一个 widen 命令。当what-line 被调用时，这个函数取消当前缓冲区中可能有的所有的变窄开启标记。（其中的变窄开启标记由 save-restriction 特殊表记录下来。）这个增宽操作使对行的计数从缓冲区的开始处进行。否则，计数就被局限在缓冲区的可见部分。所有原有的变窄开启在 save-restriction 特殊表执行完时被恢复。

在 widen 命令之后紧跟着 save-excursion 特殊表，它保存光标的位置（即位点），并在此作一个标记，当 save-excursion 特殊表中的代码使用beginning-of-line 函数来移动位点之后再恢复它们。

（注意，(widen) 表达式夹在 save-restriction 和 save-excursion 之间。当你编写连续含有两个 save-…的表达式时，总是要将 save-excursion 写在最外层。）

what-line 函数的最后两行代码对缓冲区中行数进行计数，然后在回显区中打印这个数。

```
(message "Line %d"
         (1+ (count-lines 1 (point)))))))
```

这个 message 函数在 Emacs 屏幕底部输出一行消息。该函数的第一个参量是引号中的字

符串。然而，这个字符串可以包含如“%d”、“%s”或者“%c”这样的控制符，以打印跟在字符串后面的参量。“%d”将后续的参量作为十进制数输出。因此这个消息将输出如“Line 243”这类的消息。

代替“%d”而输出的数是由函数的最后一行计算得到的：

```
(1+ (count-lines 1 (point)))
```

这个表达式所做的工作，就是从缓冲区的开始位置(由1表示)计数，直到位点处（point），并对最后的计数加 1。（1+ 函数就是对其参量加 1。）我们之所以加 1，是因为第 2 行只是在第 1 行前面 1 行。而且 count-lines 只对当前行前面的行数计数。）

在 count-lines 求值完成时，消息输出在回显区，save-excursion 恢复位点和标记到它们原来的位置；而如果有变窄开启，save-restriction 则恢复变窄开启的原有标记。

6.3 练习：变窄

编写一个函数，这个函数在即使设置了变窄开启而使缓冲区的前一半不可见的情况下也能显示出当前缓冲区的头 60 个字符。要在显示完成之后恢复位点、标记和变窄开启等相关设置。对这个练习题，要使用 save-restriction、widen、goto-char、point-min、buffer-substring、message 和其他函数，真可以算得上是一个大杂烩！

第7章 基本函数：car、cdr、cons

在 Lisp 中，car、cdr 和 cons 都是基本函数。cons 函数用于构造列表，car 和 cdr 函数则用于拆分列表。

在详细分析 copy-region-as-kill 函数的过程中，将看到 cons函数以及 cdr函数的两个变种：setcdr 和 nthcdr。（参见8.5节“copy-region-as-kill函数”。）

cons 函数的函数名不是没有理由的：它是“construct”(构造)一词的缩写。相比之下，函数名 car 和 cdr 的来历则很深奥：car是“Contents of the Address part of the Register”（寄存器地址部分的内容）短语的首字母缩写；而 cdr (读作“could-er”)是“Contents of the Decrement part of the Register”（寄存器后部内容）短语的首字母缩写。这些短语是指在开发 Lisp 的早期使用的计算机上的特定硬件部分。除了有点陈腐外，这些短语与 25 年后接触 Lisp 的任何人都完全无关。尽量如此，仍然有一些勇敢的学者已经开始对这些函数采用其他似乎更有道理的名字，但是这些术语仍旧被使用。特别是，因为这些术语常用在Emacs Lisp 源代码中，所以在这本书中继续使用它们。

7.1 car和cdr 函数

一个列表的 car，简单地说，就是返回这个列表的第一个元素。因而列表 (rose violet daisy buttercup)的 car 就是 rose。

如果你在 GNU Emacs 的 Info 中阅读这份文档，对下面的表达式求值就可以看到上面说的这个例子：

```
(car '(rose violet daisy buttercup))
```

对这个表达式求值之后，rose 一词将出现在回显区中。

很明显，car函数的一个更为合适的名字可能是first。人们也的确经常这样建议。

car不将第一个元素从列表中移走，它仅仅报告列表的第一个元素是什么。car应用到一个列表之后，列表依旧是它自己。用术语来说，car 是“非破坏性”的。这种特性是非常重要的。

一个列表的 cdr 就是这个列表的其余部分（除第一个元素外的其余部分），也就是说，cdr 函数返回列表中第一个元素后的所有内容。因而，列表 '(rose violet daisy buttercup) 的 car 是 rose，而这个列表的其余部分就是 (violet daisy buttercup)，这正是 cdr 返回的值。

用通常的方法对下面的表达式求值就可以看到这一点：

```
(cdr '(rose violet daisy buttercup))
```

当对它求值时，(violet daisy buttercup) 将出现在回显区中。

像 car 一样，cdr 也不从列表中移走任何元素——它仅仅返回包含列表的第二个和随后的所有元素列表。

顺便提一下，在这个例子中，花的列表带有单引号。如果没有这个单引号，Lisp 解释器将试图通过调用 rose 作为一个函数来对这个列表求值。在这个例子中，我们并不希望这样做。

很明显，cdr 函数的一个更为合适的名字可能是 rest。

（这里有一个教训：今后当你给一个新函数取名字时，要仔细考虑你所做的一切，因为在你无法预见的将来，你可能被这个名字弄得痛苦不堪。这份文档中继续使用这些名字的原因是 Emacs Lisp 的源代码中使用了它们，而且如果我不使用它们，读者在阅读这些代码时将很困难；但是你自己确实要避免使用这些术语。那样的话，今后的人们会感激你的。）

当对一个由符号组成的列表使用 car 和 cdr 时，例如对列表 (pine fir oak maple)使用car和cdr时，由 car 返回的列表的元素是符号 pine，没有任何括号在它两边。pine是这个列表的第一个元素。然而，列表的 cdr 是一个列表本身，即 (fir oak maple)。用通常的办法对下面的例子求值就可以看到这种不同：

```
(car '(pine fir oak maple))

(cdr '(pine fir oak maple))
```

另一方面，在一个列表的列表中，其第一个元素本身就是一个列表。car 返回作为一个列表的这第一个元素。例如，下面的列表包含三个子列表，一个是猛兽的列表，一个是食草动物的列表，一个是海洋动物的列表：

```
(car '((lion tiger cheetah)
       (gazelle antelope zebra)
       (whale dolphin seal)))
```

在这个例子中，列表的第一个元素（或者列表的 car）就是猛兽的列表，(lion tiger cheetah)。其余部分就是 ((gazelle antelope zebra) (whale dolphin seal))。

```
(cdr '((lion tiger cheetah)
       (gazelle antelope zebra)
       (whale dolphin seal)))
```

值得在此说明的是，car 和 cdr 函数都是“非破坏性”的——也就是说，它们不改变它们所作用的列表。这一点对于如何使用这两个函数非常重要。

而且，在第1章中讨论原子时，曾经说过，在 Lisp 中，某些类型的原子(例如数组)能够被分割成几个部分；但是这种分割的机制与分割一个列表的机制是不同的。只要是论及 Lisp，列表中的原子就是不可分的。（参见1.1.1节，“ Lisp 原子”。）car 和 cdr 函数用于分割一个列表，并且是 Lisp 的基本操作。因为它们不能分割一个数组或者对数组的一部分操作，所以一个数组被认为是一个原子。相反，另外一些基本的函数(例如 cons)，能够组成或构建一个列表，但是不能构建一个数组。（数组是由与数组相关的函数来处理的。参见《*GNU Emacs Lisp 技术手册*》中的“数组”一节。）

7.2 cons函数

cons 函数可以构造列表，它的作用与 car 和 cdr 函数正好相反。例如，cons 函数能够被用于将三个元素的列表(fir oak maple)变成四个元素的列表：

```
(cons 'pine '(fir oak maple))
```

对这个列表求值后，你将看到

```
(pine fir oak maple)
```

出现在回显区中。cons 函数将一个新元素放到一个列表的开始处，它往列表中插入元素。

cons 必须有一个待插入元素的列表[⊖]。绝对不能从一无所有开始。如果正在创建一个列表，首先至少需要提供一个空列表。下面是一系列 cons函数，它们构建了一个花的列表。如果你在GNU Emacs的Info 中阅读这份文档，可以用通常的方法对下面的每一个表达式求值，表达式的值是打印在“⇒”之后的文本，你可以将其读作“求值得”：

```
(cons 'buttercup ())
     ⇒ (buttercup)

(cons 'daisy '(buttercup))
     ⇒ (daisy buttercup)

(cons 'violet '(daisy buttercup))
     ⇒ (violet daisy buttercup)

(cons 'rose '(violet daisy buttercup))
     ⇒ (rose violet daisy buttercup)
```

在第一个例子中，空列表是()，由 buttercup 组成的列表因此构造出来。就像你能够看到的，空列表不在构造后的列表中出现。你所看到的是(buttercup) 列表。空列表不作为列表的一个元素，因为空列表中什么也没有。一般来说，一个空列表是不可见的。

第二个例子中，(cons ' daisy ' (buttercup)) 通过将 daisy 放到buttercup 之前构建了一个新的、两元素的列表；而第三个例子则将 violet插入到 daisy 和 buttercup 之前构建了一个三元素列表。

查询列表的长度：length函数

通过使用 Lisp的 length函数，你能够找出一个列表中有多少元素。例如，

```
(length '(buttercup))
     ⇒ 1

(length '(daisy buttercup))
     ⇒ 2

(length (cons 'violet '(daisy buttercup)))
     ⇒ 3
```

⊖ 实际上，可以将一个元素 cons 进一个原子来生成一个带点的偶对。带点偶对的内容超过了本书的范围，详情请参考《*GNU Emacs Lisp* 技术手册》。

在上面的第三个例子中，cons 函数被用于构建一个三元素列表，这个列表随后被传递给 length 函数作为其参量。

也可以用 length 函数来计算空列表中元素的个数：

```
(length ())
     ⇒ 0
```

就像你会想到的，空列表中没有任何元素。

一个有趣的试验是，如果你试图找出一个非列表对象的长度，将会发生什么？也就是说不给 length 函数传递参量，哪怕是一个空列表，那会发生什么？

```
(length )
```

如果你对这个表达式求值，你看到的将是一个错误消息：

```
Wrong number of arguments: #<subr length>, 0
```

这意味着这个函数接收了错误数量的参量（0），而它希望得到其他数目的参量。在这个例子中，length 函数希望得到一个参量，这个参量是一个函数正欲求其长度的列表。（注意，一个列表是一个参量，即使这个列表有许多元素也是如此。）

错误消息中的“#<subr length>”是函数名。它是以一种特殊形式写出来的，“#subr”是指 length 函数是一个用 C 语言编写的、而不是用 Emacs Lisp 编写的基本函数。（“subr”是 “subroutine”（子例程）一词的缩写。）更多关于子例程的资料，参见《*GNU Emacs Lisp* 技术手册》中的“什么是函数？”一节。

7.3 nthcdr函数

nthcdr 函数与 cdr 函数联系在一起。它所做的就是重复地取列表的 cdr。

如果你取列表 (pine fir oak maple) 的cdr，你将得到列表 (fir oak maple)。如果你对返回值重复取 cdr，你将得到 (oak maple) 列表。（当然，由于函数不改变列表，因此对一个原始列表重复取 cdr 将得到原始的cdr。还可以对列表的 cdr 取 cdr，等等。）如果你继续这样做，最后你将返回一个空列表，在这个例子中，最后显示的不是一个空列表 ( )，而是显示 nil。

总的来说，下面是一系列重复的 cdr，“⇒”后面的文本显示的是返回值：

```
(cdr '(pine fir oak maple))
     ⇒(fir oak maple)

(cdr '(fir oak maple))
     ⇒ (oak maple)

(cdr '(oak maple))
     ⇒(maple)

(cdr '(maple))
     ⇒ nil

(cdr 'nil)
     ⇒ nil
```

```
(cdr ())
     ⇒ nil
```

你也可以无需输出值而连续使用 cdr，就像这样：

```
(cdr (cdr '(pine fir oak maple)))
     ⇒ (oak maple)
```

在这个例子中，Lisp 解释器首先对最内层的列表求值。最内层的列表带有引号，因此它仅仅将这个列表本身传递给 cdr。这个 cdr 将一个由原来列表的第二个元素以及其后的其他元素组成的列表传递给外部的 cdr。这个 cdr 则将产生由原始列表的第三个元素以及其后的其他元素组成的列表。在这个例子中，cdr 函数被重复使用，并返回一个由原始列表的除头两个元素之外的元素组成的列表。

nthcdr 函数的功能就像重复调用 cdr 函数一样。在后续的例子中，参量 2 以及一个列表被传递给nthcdr函数，返回值是由原始列表中除头两个元素之外的元素组成的列表，这与重复两次使用 cdr 函数得到的结果完全一样：

```
(nthcdr 2 '(pine fir oak maple))
     ⇒ (oak maple)
```

使用 4 个元素的原始列表，可以看到当给 nthcdr 函数传递不同的数字参量时会发生什么情况，例子中使用的参量包括0、1 和 5：

```
;;留下列表全部。
(nthcdr 0 '(pine fir oak maple))
     ⇒ (pine fir oak maple)

; ; 返回移去第一个元素的列表。
(nthcdr 1 '(pine fir oak maple))
     ⇒ (fir oak maple)

; ; 返回移去 3 个元素的列表。
(nthcdr 3 '(pine fir oak maple))
     ⇒ (maple)

; ; 返回移去 4 个元素的列表。
(nthcdr 4 '(pine fir oak maple))
     ⇒ nil

; ; 返回一个移去所有元素的列表。
(nthcdr 5 '(pine fir oak maple))
     ⇒ nil
```

值得一提的是，就像 cdr 函数一样，nthcdr 函数也不改变原始列表——这个函数是非破坏性的。这一点与 setcar 和 setcdr 函数截然不同。

7.4 setcar函数

从函数的名字，你可能已经猜测到，setcar 和 setcdr 函数将一个列表的 car和 cdr 设置为一个新的值。不像 car 和 cdr 函数不改变原始列表，setcar 和 setcdr 这两个函数实

际上改变了原始列表。了解它们如何工作的一个途径就是不断尝试。我们将从 setcar 函数开始。

首先，构造一个列表并用 setq 函数将这个列表赋值给一个变量。下面是一个动物的列表：

```
(setq animals '(giraffe antelope tiger lion))
```

如果你是在 GNU Emacs的Info 中阅读这份文档，就可以用通常的方法对上面的表达式求值，将光标置于表达式末尾，键入 C-x C-e。(我写到这里的时候就是这么做的。在计算环境中安装解释器的好处之一就在于此。)

当对变量 animals 求值时，我们看到它被绑定到列表 (giraffe antelope tiger lion)上：

```
animals
     ⇒ (giraffe antelope tiger lion)
```

这也就是说，变量animals 指向了列表(giraffer antelope tiger lion)。

接下来，要给 setcar 函数设置两个参量，一个是变量 animals，一个是带引号的符号 hippopotamus；这是通过编写一个三元素列表 (setcar animals 'hippopotamus) 并随后用通常的方法对它求值来完成的：

```
(setcar animals 'hippopotamus)
```

对这个表达式求值之后，再对变量 animals 求值。你将会看到 animals 指向的列表已经改变了：

```
animals
     ⇒ (hippopotamus antelope tiger lion)
```

这个列表的第一个元素giraffe已经被 hippopotamus 取代了。

因此我们能够看到：setcar 函数不是像 cons 函数那样在列表中增加一个元素；它用新元素 hippopotamus 取代原来的元素 giraffe，它改变了列表。

7.5 setcdr函数

setcdr 函数与函数 setcar 相似，不同之处仅在于这个函数替换列表的第二个以及其后的所有元素，而不是列表的第一个元素。

为了了解这个函数是如何工作的，通过对下面的表达式求值来将驯养动物的列表赋值给一个变量：

```
(setq domesticated-animals '(horse cow sheep goat))
```

如果现在对这个变量求值，将返回一个列表：(horse cow sheep goat)。

```
domesticated-animals
     ⇒ (horse cow sheep goat)
```

接下来，要给 setcdr 函数设置两个参量，一个是变量名(这个变量有一个列表作为其值)，另一个参量是一个列表(它是第一个列表的 cdr 将被设置的值)：

```
(setcdr domesticated-animals '(cat dog))
```

如果对这个表达式求值，列表 (cat dog) 将出现在回显区中。这就是这个函数的返回值。但是我们感兴趣的是这个函数的附带效果，对变量 domesticated-animals 求值就可以看到这个附带效果：

```
domesticated-animals
     ⇒ (horse cat dog)
```

的确，驯养动物的列表已经从 (horse cow sheep goat) 变成了 (horse cat dog)。也就是说，列表的 cdr 已经从 (cow sheep goat) 变成了 (cat dog)。

7.6 练习

通过对几个 cons表达式求值，来构建一个四元素的、有关鸟的列表。试一试，当你对列表本身使用 cons 函数时会发生什么？用一种鱼取代这个列表的第一个元素。用其他鱼的列表取代这个列表的其余部分。

第8章　剪切和存储文本

在 GNU Emacs 中，无论你何时用“kill”命令从一个缓冲区中剪切一段文本，它总是存储在一个列表中，而且你可以用一个“yank”命令将其重新取回来。

（在 Emacs 中使用“kill”一词来表示那些并未破坏对象值的过程实在是一个不幸的历史性事故。一个好得多的词可能是“clip”（修剪），因为这正是“kill”命令所完成的工作。它们将文本从缓冲区中剪切出来，放到存储设备中，并可以从存储设备中将其取回来。我经常试图在出现“kill”命令的所有地方都用“clip”来取代它，在出现“killed”的所有地方都用“clipped”来取代它。）

当将文本从缓冲区中剪切出来时，它被储存在一个列表中，连续的一段文本在列表中也是连续地存放的，因此这个列表看起来如下所示：

```
("a piece of text" "last piece")
```

cons 函数能用于往这个列表中增加一段文本片断，就像这样：

```
(cons "another piece"
      '("a piece of text" "last piece"))
```

如果对这个表达式求值，则三个元素的列表将会出现在回显区中：

```
("another piece" "a piece of text" "last piece")
```

使用 car和nthcdr函数，能从这个列表中将任何一个文本重新提取出来。例如，在下面的代码中，nthcdr 1... 返回由参量列表的第一个元素之外的所有元素组成的列表，car 函数则返回这个中间列表的第一个元素——也就是原始列表的第二个元素：

```
(car (nthcdr 1 '("another piece"
                 "a piece of text"
                 "last piece")))
     ⇒ "a piece of text"
```

当然，Emacs 中实际的函数比这个例子更加复杂。必须编写剪切和找回 文本的代码，以便 Emacs 能计算出列表中的那个元素是你所需要的——第一个、第二个、第三个.....。除此之外，当你到达列表末尾时，Emacs 会重新回到第一个元素，而不是空元素。

保存被剪切的一段文本的列表被称为kill 环（*kill ring*）。这一章首先介绍这个 kill 环，然后用 zap-to-char 函数这个例子来了解如何使用这个列表。zap-to-char函数调用另外一个对kill环操作的函数，因而在介绍 zap-to-char 这个函数之前首先介绍那个函数，就像在攀登高峰之前，先爬小山。

随后的一章将描述如何将被剪切的文本重新找回。参见第10章，“找回文本”。

8.1 zap-to-char函数

在 GNU Emacs 第18版和 第19版中，zap-to-char函数的代码是不同的。第19版中的实现更为简单，工作方式与第18版中的实现略有不同。我们首先看看第19版中的这个函数，然后再看看第18版中的代码。

在 Emacs 第19版中，交互的 zap-to-char 函数的功能就是：将光标当前位置（即位点）与出现特定字符的下一个位置之间这一区域中的文本剪切掉。zap-to-char 函数剪切的文本放在 kill 环中，并能通过键入 C-y(yank) 命令从 kill 环中找回这些文本。如果这个命令带有参量，它就将从位点处到特定字符出现了参量所示次数的那个位置之间这一区域内的文本剪切掉。因而，如果光标在这个句子 [⊖] 的开头，而指定字符是“s”，则“Thus”一词将被剪切。如果给定的参量是2，“Thus, if the curs”将被剪切，即从位点到第二次出现指定字符“s”之间的文本（包含“cursor”中的指定字符 “s”）被剪切了。

Emacs 第18版中，这个函数将位点到指定字符区域之间的文本（但不包含指定字符）剪切。因而，在上面的例子中，字符“s”就不被剪切。

除此之外，在第18版中，如果没有找到指定字符，则将位点直到缓冲区末尾整个区域内的文本全部剪切；但是在第19版中，如果发生这种情况则仅仅产生一个错误消息（不剪切任何文本）。

为了决定究竟有多少文本被剪切，两个版本的 zap-to-char 函数都使用了一个查询函数。查询函数在操纵文本的代码中广泛使用，关注查询函数与关注删除命令一样，都是值得的。

这是第19版中 zap-to-char 函数的完整代码：

```
(defun zap-to-char (arg char)  ; version 19 implementation
  "Kill up to and including ARG'th occurrence of CHAR.
Goes backward if ARG is negative; error if CHAR not found."
  (interactive "*p\ncZap to char: ")
  (kill-region (point)
               (progn
                 (search-forward
                  (char-to-string char) nil nil arg)
                 (point))))
```

8.1.1 interactive 表达式

zap-to-char 函数中的交互(interactive)表达式如下所示：

```
(interactive "*p\ncZap to char: ")
```

括号中的部分 “*p\ncZap to char:”，指定了三件事情。第一，也是最简单的，即星号“*”，它意味着，如果缓冲区是只读的，就产生一个错误信号。这就是说，如果你试图在一个只读缓冲区中使用 zap-to-char 函数，你将无法剪切任何文本，并且你将收到一个这样的消息：

⊖ 这里是指英文原书中此处的句子：“Thus, if the cursor were at the beginning of this sentence...”。——译者注

"Buffer is read-only"，你的终端还可能会对着你鸣叫报警。

"*p\ncZap to char:"的第二部分是字符"p"。这部分以一个换行符"\n"结束。小写"p"是指传送给函数的第一个参量将是一个处理过的前缀参量的值。前缀参量用 C-u 以及其后的一个数来传送，或者用 M- 和一个数来传送。如果不带前缀参量交互地调用这个函数，默认值 1 将被传送给这个函数。

"*p\ncZap to char:"的第三部分是"cZap to char:"。在这一部分中，小写"c"是指交互表达式希望产生一个提示并且后续的参量将是一个字符。提示信息是跟在"c"之后的字符串"Zap to char:"。(冒号后面带一个空格会使提示信息更好看。)

所有这些，都是为 zap-to-char 函数准备参量。至此，这些参量都有了正确的类型，并显示给用户一个提示信息。

8.1.2 zap-to-char 函数体

zap-to-char 函数体包含了从光标的当前位置(即位点)直到(并包含)指定字符这一区域剪切文本的代码。代码的第一个部分如下所示：

```
(kill-region (point) ...
```

(point) 就是光标所处的当前位置，即位点。

代码的下一个部分是一个使用 progn的表达式。progn 表达式的主体由 search-forward 和 point 函数组成。

学习了 search-forward 函数之后，就容易理解 progn 是如何工作的了。因此我们将先学习 search-forward函数，然后再学习 progn 函数。

8.1.3 search-forward 函数

search-forward 函数是用于定位 zap-to-char 函数中被截取的字符的。如果查询成功，search-forward 函数就在目标字符串中最后一个字符处设置位点(在这个例子中，目标字符串只有一个字符)。如果查询是朝后进行的，search-forward 函数就在目标字符串的第一个字符处设量位点。同样，search-forward 函数返回 t值表示查询成功。(移动位点只是这个函数的附带效果。)

在 zap-to-char 函数中，search-forward 函数如下所示：

```
(search-forward (char-to-string char) nil nil arg)
```

search-forward函数有 4 个参量：

1)第一个参量是目标，就是所要查找的内容。这个参量必须是一个字符串，如"z"。

执行时，传送给 zap-to-char 函数的参量是一个单字符。由于计算机本身工作原理的限制，Lisp 解释器认为单个字符与一个字符串是不同的。在计算机内部，单个字符与一个仅仅包含单个字符的字符串有不同的存储格式(单个字符能用一个字节精确地记录，但是一个字符串可能很长也可能很短，计算机需要为此做准备)。因为 search-forward 函数是查询一个字符串的，所以zap-to-char 函数接收的、作为其参量的字符，必须在计算机内从一种格式转

换成另外一种格式，否则 search-forward 函数将会无法工作。char-to-string 函数就是用于完成这种转换工作。

2) 第二个参量绑定查询范围；它被指定为缓冲区中的某个位置。在这个例子中，查询能到达缓冲区末尾，因此没有设置任何绑定，第二个参量就是空（nil）。

3) 第三个参量告诉这个函数如果查询失败应该怎么办——可以发出一个出错信号(并打印一条消息)，也可以返回空值 (nil)。如果第三个参量被设置成空 (nil)，就是告诉这个函数如果查询失败就发出一个出错信号。

4) 第四个参量是重复计数值——待查找字符串出现的次数的计数。这个参量是可选的，如果在调用这个函数时没有给定计数值，就使用默认值 1。如果这个参量是一个负数，查询就朝后进行。

用一个模板形式来分析的话，search-forward 函数就是这样：

```
(search-forward "target-string"
                limit-of-search
                what-to-do-if-search-fails
                repeat-count)
```

下面我们将学习 progn 函数。

8.1.4 progn 函数

progn 函数使其每一个参量被逐一求值并返回最后一个参量的值。前面若干表达式的求值，仅仅是作为函数的附带效果，这些值被统统扔掉了。

progn 表达式的模板很简单：

```
(progn
  body...)
```

在 zap-to-char 函数中，progn表达式要完成两件事情：在正确的位置设置位点，将位点返回以使 kill-region 函数知道要剪切到什么地方。

progn 表达式的第一个参量就是 search-forward。当 search-forward函数在文本中找到目标字符串时，它就在文本中的目标字符串最后一个字符处设置位点（在这个例子中目标字符串只有单个字符）。如果查询是朝后进行的，search-forward 函数就在目标字符串的第一个字符处设置位点。位点的设置和变动也只是这个函数的附带效果。

progn 表达式的第二个也是最后一个参量是表达式 (point)。这个表达式返回位点的值，在这种情况下就是由 search-forward 函数移动过的位点的值。这个值由 progn 表达式返回，并传递给 kill-region 函数作为其第二个参量。

8.1.5 总结zap-to-char函数

现在已经了解了 search-forward 函数和 progn 函数是如何工作的，因此也就能分析 zap-to-char 函数是如何工作的了。

当 zap-to-char 函数被调用时，给 kill-region 的第一个参量是光标所在的位置，也

就是当时位点的值。在 progn 表达式内部，search-forward函数将位点移动到查找到的字符后，(point) 表达式则返回位点的值。kill-region函数将这两个值结合起来，第一个值作为要剪切部分的开始，第二个值作为要剪切部分的末尾，然后将这个区域内的文本剪切掉。

progn 函数是需要的，因为 kill-region 命令需要两个参量。如果 search-forward 函数和 (point) 表达式作为它另外两个参量，它将运行失败。progn 表达式是 kill-region 命令的一个参量而不是两个，它的返回值正是kill-region 命令需要的第二个参量。

8.1.6 第18版中 zap-to-char 函数的实现方法

在第18版中，zap-to-char 函数的实现方法与 第19版中这个函数的实现方法稍有不同：它剪切的文本不包含指定字符；并且当没找有到指定字符时就剪切到缓冲区末尾。

产生达种不同的原因在于 kill-region 命令的第二个参量。这个参量在第19版中是这样的：

```
(progn
  (search-forward (char-to-string char) nil nil arg)
  (point))
```

而在第18版中，则是下面这个样子的：

```
(if (search-forward (char-to-string char) nil t arg)
    (progn (goto-char
            (if (> arg 0) (1- (point)) (1+ (point))))
           (point))
  (if (> arg 0)
      (point-max)
    (point-min)))
```

这部分代码看起来相当复杂，但是如果将其一部分一部分分解开来分析，就容易理解了。

代码的第一部分是：

```
(if (search-forward (char-to-string char) nil t arg)
```

用 if 表达式模板来分析，就是：

```
(if able-to-locate-zapped-for-character-and-move-point-to-it
    then-move-point-to-the-exact-spot-and-return-this-location
  else-move-to-end-of-buffer-and-return-that-location)
```

对这个 if 表达式的求值，就给出了 kill-region 函数的第二个参量。因为它的第一个参量是位点，因此这个过程使 kill-region 函数可以将位点和指定字符之间的文本全部剪切掉。

我们已经描述了 search-forward 函数如何将移动位点作为它的附带效果完成的这一过程。在这个函数中，如果查找成功，search-forward 函数的返回值就是 t。否则，根据 search-forward 函数的第三个参量的不同，它要么返回 nil 值，要么产生一个错误消息。在这个例子中，t 是它的第三个参量，这使得 search-forward 函数在查找失败时返回 nil 值。就值我们将要看到的，可以容易地编写代码来处理函数返回 nil 值的达种情况。

在第18版的 zap-to-char 函数的实现中，if表达式将查询表达式作为其真假测试表达式。如果查询成功，Emacs 就对 if 表达式的 then 部求值；另一方面，如果查询失败，Emacs 就对 if 表达式的 else 部求值。

在 if 表达式中，当查询成功时，progn 表达式被执行——也就是说，它就像一个程序一样被运行。

前面已经讲过，progn是一个函数，它使其中的参量逐一被求值，并返回最后一个参量的值。前面的其他表达式仅仅作为附带效果而被求值。它们产生的值被统统扔掉了。

在这个版本的 zap-to-char 中，当查询函数 search-forward 找到它要查询的字符时，progn 表达式就被执行。这个 progn 表达式要完成两件事情：在正确的位置设置位点，返回位点的值以使 kill-region 知道要剪切到何处。

之所以需要 progn 表达式中的代码，是因为当 search-forward 函数找到指定字符串时，它就在目标字符串的最后一个字符后设置位点（在这个例子中，目标字符串只有一个字符）。如果是朝后查询的，则在目标字符串的第一个字符处前设置位点。

然而，这个版本的 zap-to-char 函数并不剪切最后匹配的字符。例如，如果zap-to-char 函数要剪切直到“z”的所有文本，实际上它并不剪切 “z”字符。因此，位点要仅仅移动到匹配字符不被剪切的位置。

8.1.7 progn表达式主体

progn 表达式的主体包含两个表达式。若要展开来详细描绘两个版本中 progn 表达式的不同之处，并加上注释，这个版本的 progn 表达式如下所示：

```
(progn
  (goto-char                    ; First expression in progn.
      (if (> arg 0)             ; If arg is positive,
          (1- (point))          ;   move back one character;
        (1+ (point))))          ;   else move forward one character.

  (point))                      ; Second expression in progn:
                                ;   return position of point.
```

这个 progn 表达式是这样工作的：当查询是朝前进行的（arg是正值），Emacs就在查找到的字符后面设置位点。通过将位点向后移动一个位置，查找到的字符就不被剪切。在这个例子中，progn 表达式应变成这样：(goto-char (1- (point)))。这个表达式将位点后移一个字符（1-函数从其参量中减1，就像1+函数往其参量中加1一样）。另一方面，如果传递给 zap-to-char 函数的参量是负数，查询就是朝后进行的。if 表达式检查到这一点，因此表达式实际上就成了：(goto-char (1+ (point)))。（1+函数往其参量中加 1。）

progn 表达式的第二个也是最后一个参量是表达式 (point)。这个表达式返回由 progn 表达式的第一个参量决定的位点的值。然后，if 表达式返回这个值。if 表达式是 kill-region 表达式的一部分，并将 if 表达式的这个返回值传递给 kill-region 表达式作为它

的第二个参量。

简要地说，这个函数的工作方式就是：kill-region 的第一个参量是当zap-to-char命令执行时光标所在的位置——也就是那个时候位点的值。然后，如果查询成功，查询函数将位点移动。progn 表达式将位点移动到匹配字符串刚好不被剪切的位置，并返回这时位点的值。最后，kill-region 函数剪切这段区域内的文本。

最后，if 表达式中的else部处理目标字符串没有被查到的情况。如果zap-to-char 函数的参量是正的（或者没有给出）而且查询失败，则当前位点到缓冲区可见区域末尾的所有文本都将被剪切。（如果没有设置变窄开启，就是从当前站点到整个缓冲区末尾的所有文本都将被剪切。）如果 arg 是负的，而又没有查找到目录字符串，则从当前位点到缓冲区可见区域开始处的文本都将被剪切。完成这些工作的代码是一个简单的 if 表达式：

```
(if (> arg 0) (point-max) (point-min))
```

这就是说，如果 arg 是一个正数，返回 point-max 的值；否则，返回 point-min 的值。

回顾起来，下面是包含 kill-region 表达式的代码(带有注释)：

```
(kill-region
 (point)                        ; beginning-of-region
 (if (search-forward
      (char-to-string char)     ; target
      nil                       ; limit-of-search: none
      t                         ; Return nil if fail.
      arg)                      ; repeat-count.
     (progn                     ; then-part
       (goto-char
        (if (> arg 0)
            (1- (point))
          (1+ (point))))
       (point))

   (if (> arg 0)                ; else-part
       (point-max)
     (point-min))))
```

通过比较你可以看到：第19版中 zap-to-char 函数的实现代码比第18版中该函数的实现代码少一些，但是更简洁。

8.2 kill-region函数

zap-to-char 函数使用了 kill-region 函数。这个kill-region函数很简单，就是删去文档字符串的一部分。其代码如下：

```
(defun kill-region (beg end)
  "Kill between point and mark.
The text is deleted but saved in the kill ring."
```

```
(interactive "*r")
(copy-region-as-kill beg end)
(delete-region beg end))
```

一个要特别注意的地方是，这个函数使用了 delete-region和 copy-region-as-kill 函数。这些函数将在接下来的章节描述。

8.3 delete-region函数：接数C

zap-to-char命令使用了 kill-region 函数，而 kill-region 函数又使用了两个其他的函数：copy-region-as-kill 和 delete-region。copy-region-as-kill函数将在随后小节中描述，它的作用是将某个区域中的文本复制一份到kill环中，因此这份文本可以重新找回来。(参见8.5节“copy-region-as-kill函数”。)

delete-region 函数删除一个区域中的内容，而且你无法找回它。

不像在这里讨论的其他函数，delete-region 函数不是用 Emacs Lisp 编写的，它是用 C 语言编写的，并且是 GNU Emacs 系统的一个基本函数。因为它非常简单，所以我就从 Lisp 中岔开来讲讲这个 C 语言函数。

就像许多其他的 Emacs 基本函数一样，delete-region是作为一个 C 语言宏的实例来被编写的，一个宏就是一个代码模板。这个宏的第一个部分如下所示：

```
DEFUN ("delete-region", Fdelete_region, Sdelete_region, 2, 2, "r",
  "Delete the text between point and mark.\n\
When called from a program, expects two arguments,\n\
character numbers specifying the stretch to be deleted.")
```

在没有深入到这个宏编写过程的细节之前，首先要指出的是这个宏是以 DEFUN 开始的。之所以选择 DEFUN 这个词，是因为它完成 Lisp 中 defun 相同的事情。DEFUN 一词后面的括号内跟着七个部分：

- 第一个部分是 Lisp 中的函数名，在这个例子中就是 delete-region。
- 第二部分是 C 语言中的函数名，即 Fdelete_region。习惯上，它以“F”开头。因为 C 语言中函数名不使用连字符，所以使用下划线。
- 第三部分是 C 常数结构名，这些常数结构在函数内部记录信息。它是 C 语言中的函数名，但是它以字符“S”开头而不是以“F”开头。
- 第四和第五部分指定了函数中允许的参量数目的最小值和最大值。在这个例子中，这个函数需要两个参量。
- 第六部分就像 Lisp 的一个函数中跟在interactive 说明之后的参量那样：要么是一个字符，要么是一个提示信息。在这个例子中，字符是“r”，它是指函数的两个参量将是一个缓冲区中某个区域的开始和结束的位置。在这段代码中，没有提示信息。
- 第七部分是文档字符串。除了每一个换行符都必须显式地写成 “\n”的形式外，它与 Emacs Lisp 中编写的函数的文档就没有别的不同之处了。

随后就是正式的参数（每个参数都有对这个参数的类型进行说明的语句），然后就是这个宏的主体部分。对 delete-region 函数而言，这个宏的主体包含了如下三行：

```
validate_region (&b, &e);
del_range (XINT (b), XINT (e));
return Qnil;
```

其中的第一个函数 validate_region 检查传递来的值的类型，判断它们作为缓冲区中一个区域的开始和结束值是否正确，是否在正确的范围之内。第二个函数 del_range 实际上真正完成删除文本的功能。如果这个函数正确地删除了文本，则第三行中的函数返回 Qnil 来表示它已经顺利完成了删除任务。

del_range 函数是一个复杂的函数，在此不再继续深入研究。它的作用是更新缓冲区并完成一些其他的事情。然而，看看传递给它的两个参量还是值得的。这两个参量是 XINT(b) 和 XINT(e)。在 C 语言中，b 和 e 是两个 32 位的整数，它们记录要删除的区域的开始和结束的位置。但是就像 Emacs Lisp 中的其他数一样，32 位中只有 24 位是用于存放实际的数值，剩下的 8 位用于跟踪这些数的类型和其他信息。(在某些机器中，只有 6 位能用于这种目的。)在这个例子中，8 位用于指出这些数是指缓冲区中的位置。当一个数中的某些位用于这样的目的时，这就被称作一个标签 (*tag*)。在 32 位数据中使用 8 位标签使得这样编写 Emacs 代码的速度比用其他方式编写代码的速度更快。另一方面，由于实际数字只占用了 24 位，因此 Emacs 缓冲区近似地限制在8MB。(通过在编译前在“emacs/src/config.h”中定义 VALBITS 和 GCTYPEBITS，你就可以增加缓冲区的最大容量。参见 Emacs 发行版本中的“emacs/etc/FAQ”文件中的注释。)

“XINT”是一个 C 语言宏，它从 32 位的 Emacs 对象中提取 24 位，用于其他目的的8位就被扔掉了。因此，del_range(XINT(b)、XINT(e)) 删除以“b”开始以“e”结束的区域中的内容。

从开发 Lisp 的人员的角度来看，Emacs 是相当简单的；但是隐藏在其中的内容却非常复杂玄妙。

8.4 用 defvar 初始化变量

不像 delete_region 函数是用 C 语言编写的，copy-region-as-kill 函数是用 Emacs Lisp 编写的。这个函数的功能就是拷贝缓冲区中的一个区域并将其保存到被称为 kill-ring 的变量中。这一节就描述这个变量如何被创建和如何被初始化。

(再一次提醒你，kill-ring这个术语确属于用词不当。从缓冲区中剪切出去的文本能够被找回来。它不是尸体之环，而是一个可以复活的文本环。)

在 Emacs Lisp 中，一个变量(如kill-ring)是通过使用 defvar 特殊表而被创建和赋初值的。这个特殊表的名字来源于“define variable”(定义变量之意)。

defvar 特殊表与给一个变量赋值的 setq 函数相似。它和 setq 有两个不同之处。第一，它只对无值的变量赋值。如果变量已经有一个值，defvar特殊表就不会覆盖已经存在的值。第二，defvar 特殊表有一个文档字符串。

可以用 describe-variable 函数查看任何一个变量的当前值，describe-variable 这个函数常常通过键入 C-h v 来激活。如果键入 C-h v，然后在提示下输入 kill-ring (并

回车），将看到当前的 kill 环中的内容——可能是相当多的。相反，如果在这个 Emacs 进程中除了阅读之外什么也没有做，你可能什么也看不到。在“*Help*”缓冲区的末尾，将看到 kill-ring 的文档：

```
Documentation:
List of killed text sequences.
```

kill环是用下面的方式由 defvar 定义的：

```
(defvar kill-ring nil
  "List of killed text sequences.")
```

在这个变量定义中，变量被初始化为 nil，这是有意义的，因为如果你什么也没有保存，当你用 yank 命令时就无需返回任何东西。文档字符串就像 defun中的文档字符串一样被编写。至于 defun 定义中的文档字符串，其第一行应当是一个完整的句子，因为有些命令(如 apropos)仅仅打印其中的第一行。后续的行不应缩排；否则当你使用 c-h v (describe-variable)时它们看起来很奇怪。

绝大多数变量是 Emacs 的内部变量，但是有一些是可以用 edit-options 命令方便地设置的。(这些设置仅仅在一个编辑过程中有效；要永久地设置一个值，可以编写一个“.emacs”文件。参见第16章，“配置你的‘.emacs’文件”。)

一个可以重新设置的变量，是用文档字符串的第一行前面加上星号“*”来使之与Emacs 中的其他变量区别开来。

例如：

```
(defvar line-number-mode nil
  "*Non-nil means display line number in mode line.")
```

这意味着你能够使用 edit-options 命令来改变 line-number-mode 变量的值。

当然，你也能够在一个 setq 表达式当中对line-number-mode 变量求值来改变这个变量的值。

```
(setq line-number-mode t)
```

参见1.9.2节，“使用setq函数”。

8.5 copy-region-as-kill函数

copy-region-as-kill函数拷贝缓冲区中的一个文本区域，并将其保存到kill-ring 变量中。

如果在调用 kill-region命令后马上调用 copy-region-as-kill函数，Emacs将会把这个新拷贝的文本追加到原来拷贝的文本后。这意味着如果要找回文本，你将得到所有的内容(包括原来拷贝的内容和新拷贝的内容)，另外，如果在调用copy-region-as-kill 函数之前有其他命令，这个函数就将文本拷贝到 kill 环的另外一个入口。

下面是第18版中的 copy-region-as-kill 函数的全部代码，在此增加了几个注释以使格式更清楚：

```
(defun copy-region-as-kill (beg end)
  "Save the region as if killed, but don't kill it."
  (interactive "r")

  (if (eq last-command 'kill-region)

      ;; then-part: Combine newly copied text
      ;;   with previously copied text.
      (kill-append (buffer-substring beg end) (< end beg))

    ;; else-part: Add newly copied text as a new element
    ;;   to the kill ring and shorten the kill ring if necessary.
    (setq kill-ring
          (cons (buffer-substring beg end) kill-ring))
    (if (> (length kill-ring) kill-ring-max)
        (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
  (setq this-command 'kill-region)
  (setq kill-ring-yank-pointer kill-ring))
```

像通常一样，这个函数能被分成几个组成部分：

```
(defun copy-region-as-kill (argument-list)
  "documentation..."
  (interactive "r")
  body...)
```

从这里可以清楚地看到，函数的参量是 beg 和 end，并且函数是交互的，带有“r”参数。因此函数的两个参量必须指向一个区域的开始和结束位置。如果你是从头阅读这份文档，理解这几部分几乎就像是例行公事一样简单。

除非你记得“kill”一词与它的原意有一定的差别，否则函数的文档会使你有些糊涂。

函数体开始于一个 if 从句。这个从句所做的工作就是判别两种不同情况：这个命令是否是在前面一个 kill-region 函数后面立即执行的。如果是，则新拷贝的文本被追加到原来拷贝的文本后。否则，就作为一个与原来的文本分开的单独文本插入 kill 环的开头。

函数的最后两行是两个 setq 表达式。其中一个表达式将变量 this-command 设置为kill-region，另外一个表达式将变量 kill-ring-yank-pointer 指向 kill 环。

copy-region-as-kill 函数体将在下面详细讨论。

copy-region-as-kill 函数体

编写copy-region-as-kill 函数是为了使在一行中两次或者多次剪切的文本最终将重新组合到 kill 环的同一个入口。如果要从 kill 环找回文本，则会得到整个文本。而且，如果是从当前光标处朝前剪切文本，则剪切掉的文本将加到原来剪切文本的末尾处；如果是从当前光标处朝后剪切文本，则剪切掉的文本将加到原来剪切文本的开始处。这就是说，kill 环中的文字仍然是以正常的次序存放的。

copy-region-as-kill 函数使用了两个变量存放当前和之前的一个 Emacs 命令。这两个变量是 this-command 和 last-command。

正常情况下，每当一个函数执行时，Emacs 将 this-command变量设置为正被执行的函数（在这个例子中就是 copy-region-as-kill）。同时，Emacs 将变量 last-command 设置为变量 this-command 原来的值。然而，copy-region-as-kill 命令就不同，它将this-command 变量设置成 kill-region，这是调用copy-region-as-kill的函数名。

在 copy-region-as-kill 函数体的第一部分，if 表达式判定 last-command 变量的值是否是 kill-region。如果是，这个表达式的 then 部被求值，它使用kill-append 函数将拷贝的文本追加到 kill 环中第一个元素的文本之后。另一方面，如果变量 last-command 的值不是 kill-region，copy-region-as-kill 函数在kill环中加入一个新的元素。

这个if 表达式如下所示，它使用了一个我们未曾看到过的 eq 函数：

```
(if (eq last-command 'kill-region)
    ;; then-part
    (kill-append (buffer-substring beg end) (< end beg))
```

eq 函数测试其第一个参量是否与其第二个参量是同一个 Lisp 对象。eq 函数与equal 函数类似，它们在用于测试是否相等方面是一样的；但是，它们在判定不同名的两种表示所对应的 Lisp 对象是否是计算机中的同一个对象时是不同的。equal 函数判定两个表达式的结构和内容是否完全等同。

1. kill-append 函数

kill-append 函数如下所示：

```
(defun kill-append (string before-p)
  (setcar kill-ring
          (if before-p
              (concat string (car kill-ring))
              (concat (car kill-ring) string))))
```

可以一部分一部分地分析这个函数。其中，setcar 函数使用 concat 函数将新的文本追加到 kill环的 car 中（即第一个元素）。它是否追加或者前插文本依赖于 if 表达式的结果：

```
(if before-p                            ; if-part
    (concat string (car kill-ring))     ; then-part
  (concat (car kill-ring) string))      ; else-part
```

如果被剪切的文本是在最近一个命令剪切的文本之前，则这些文本应被插入到原来 kill 环中保存的内容之前。反过来，如果被剪切的文本是在最近被剪切的文本之后，则它应当被追加到原来那些文本之后。if 表达式依赖于 before-p 的判断来决定是否应将新保存的文本放在原来保存的文本之前还是之后。

符号 before-p 是 kill-append 函数的另外一个参量的名字。当 kill-append 函数被求值时，这个符号绑定到实际参量被求值后返回的值。在这个例子中，这就是表达式 (< end beg)。这个表达式并不直接决定剪切的文本是插入到前一个命令剪切的文本之前，还是追加到该文本之后。这个表达式所做的工作就是判定变量 end 的值是否小于变量 beg 的值。如果是，

则说明用户希望朝缓冲区开头剪切。同样地，如果对这个表达式求值的结果为“真”，则被剪切的文本将被插入到原来文本之前。另一方面，如果变量 end 的值大于变量 beg 的值，被剪切的文本就被追加到原来文本之后。

当新保存的文本要被插入到原有文本之前时，带新文本的字符串将被连接到老文本之前：

```
(concat string (car kill-ring))
```

但是，如果文本是被追加，则它将被连接到老文本之后：

```
(concat (car kill-ring) string))
```

为了理解这是如何实现的，首先需要回顾一下 concat 函数。concat 函数将两个文本字符串连接在一起，其结果是一个字符串。例如：

```
(concat "abc" "def")
     ⇒ "abcdef"

(concat "new "
        (car '("first element" "second element")))
     ⇒ "new first element"


(concat (car
        '("first element" "second element")) " modified")
     ⇒ "first element modified"
```

现在，我们能够来关注 kill-append 函数：它改变了 kill 环中的内容。kill 环是一个列表，其中的每一个元素都是保存的文本。setcar 函数实际上改变这个列表的第一个元素。它是通过使用 concat 函数将最新保存的文本连接到kill环的第一个元素来取代原来的第一个元素来实现的。最新保存的文本放在老文本之前或者之后，这依赖于在缓冲区中它是在原有文本之前还是之后。连接后的元素，成为 kill 环新的第一个元素。

顺便说一说，下面就是我的 kill 环开始处的内容：

```
("concatenating together" "saved text" "element" ...
```

2. copy-region-as-kill 函数中的 else 部

现在，回到 copy-region-as-kill 函数的解释：

如果最后一个命令不是 kill-region，则函数不是调用 kill-append，而是调用下面代码中的 else 部：

```
(if true-or-false-test
    what-is-done-if-test-returns-true
  ;; else-part
  (setq kill-ring
        (cons (buffer-substring beg end) kill-ring))
  (if (> (length kill-ring) kill-ring-max)
      (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
```

else 部中的 setq 这一行，将在 kill 环上追加被剪切的字符串，并将这个新值赋给 kill 环。用一个小例子就可以看到它是如何运作的：

```
(setq example-list '("here is a clause" "another clause"))
```

键入 C-x C-e 对这个表达式求值后，可以对 example-list 求值并查看它所返回的值：

```
example-list
     ⇒ ("here is a clause" "another clause")
```

现在，通过对下面的表达式求值，就能够往这个列表中增加一个新的元素：

```
(setq example-list (cons "a third clause" example-list))
```

当对 example-list 求值时，将发现它的值是：

```
example-list
     ⇒ ("a third clause" "here is a clause" "another clause")
```

因而，通过 cons 函数，第三个元素增加到了 example-list 列表中。

除了用 buffer-substring 截取剪切区域中的文本之外，这个例子与在 copy-region-as-kill 函数中使用 setq 和 cons 函数的情况非常相似。这个语句重新写在下面：

```
(setq kill-ring (cons (buffer-substring beg end) kill-ring))
```

copy-region-as-kill 函数中 else 部的下一段是另外一个if 表达式。这个 if 表达式使 kill 环不致于过长。它是这样的：

```
(if (> (length kill-ring) kill-ring-max)
    (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil))
```

这部分代码检查 kill 环的长度是否大于最大允许的长度。最大允许长度就是kill-ring-max 的值（默认的情况下是30）。如果 kill 环的长度太长，这部分代码将 kill 环的最后一个元素设置为 nil。这是通过使用 setcdr 和nthcdr 函数来实现的。

我们首先来看看 setcdr 函数（参见7.5节“setcdr函数”）。这个函数设置一个列表的 cdr，就像 setcar 函数设置一个列表的 car 一样。然而在这个例子中，setcdr 并不是设置整个 kill 环列表的 cdr；nthcdr函数用于设置 kill 环的第二个元素到最后一个元素这个列表的 cdr——这意味着，如果第二个元素到最后一个元素的 cdr 是kill环的最后一个元素，它将设置 kill环的最后一个元素。

nthcdr 函数反复地取一个列表的 cdr——即它取一个列表的cdr的 cdr 的 cdr..... 这样重复 *N* 次，并返回最后的结果。

因此，如果现有一个4元素的列表，而假设只能有3个元素，则需要设置第二个元素到最后一个元素的 cdr为最后一个元素并赋 nil 值，从而缩短列表的长度。

依次对下面三个表达式求值就可以看到这一点。首先将变量 trees的值设置为(maple oak pine birch)，然后设置其第二次 cdr 的 cdr 为 nil，然后求 trees变量的值：

```
(setq trees '(maple oak pine birch))
     ⇒ (maple oak pine birch)
(setcdr (nthcdr 2 trees) nil)
     ⇒ nil

trees
     ⇒ (maple oak pine)
```

（由 setcdr 表达式返回的值是 nil，因为这就是 cdr 设置的值。）

再重复一下，在 copy-region-as-kill 函数中，nthcdr 函数重复取若干次的 cdr，其次数是kill环的最大允许长度减 1，并将那个元素的 cdr（其实这就是列表的最后一个元素）设置为 nil。这就可以避免 kill 环无限制地增长。

copy-region-as-kill 函数的倒数第二行是：

```
(setq this-command 'kill-region)
```

这一行代码既不属于内层 if 表达式，也不属于外层 if表达式，因此每当 copy-region-as-kill 函数被调用一次，这个表达式就被求值一次。在这里，我们发现此处this-command 变量被赋值为 kill-region。就像前面看到的，当执行下一个命令时，变量last-command将被赋为这个值。

最后，copy-region-as-kill 函数的最末一行是：

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

变量 kill-ring-yank-pointer 是一个全局变量，它被设置为 kill-ring。

虽然 kill-ring-yank-pointer 变量被称为“pointer”(指针)，但是它仅仅是一个像 kill 环这样的列表变量。然而，选择这样的名字是为了帮助人们理解如何使用这个变量。像 yank 和 yank-pop 这样的函数常使用这个变量（参见第10章，“找回文本”）。

yank 这些函数使我们将已经剪切的文本重新找回来。然而，在讨论 yank 命令之前，最好先学习列表是如何在计算机中实现的。这可以使像术语“指针”的使用这样神秘的内容变得清楚易懂。

8.6 回顾

下面是本章已经介绍过的一些函数的简单小结。

- cdr、car

car返回一个列表的第一个元素，cdr 则返回列表的第二个元素直到最后一个元素的列表。

例如，

```
(car '(1 2 3 4 5 6 7))
     ⇒ 1
(cdr '(1 2 3 4 5 6 7))
     ⇒ (2 3 4 5 6 7)
```

- cons

这个函数通过将它的第一个参量插入到它的第二个参量中来构造一个列表。

例如，

```
(cons 1 '(2 3 4))
     ⇒ (1 2 3 4)
```

- nthcdr

这个函数返回对一个列表求 *N*次 cdr 的值，也就是“剩余的剩余部分”。

例如，

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

• setcdr、setcar

setcar改变一个列表的第一个元素，而setcdr则改变一个列表的第二个到最后一个元素。

例如：

```
(setq triple '(1 2 3))

(setcar triple '37)

triple
     ⇒ (37 2 3)

(setcdr triple '("foo" "bar"))

triple
     ⇒ (37 "foo" "bar")
```

• progn

这个函数依次对其每一个参量求值，并返回最后一个参量的值。

例如：

```
(progn 1 2 3 4)
     ⇒ 4
```

• save-restriction

这个函数记录当前缓冲区中变窄开启是否设置，如果已经设置，就在对后续的参量求值之后恢复变窄开启。

• search-forward

这个函数查找一个字符串，并且如果找到这个字符串就移动位点。

这个函数有4个参量：

1) 要查找的字符串。

2) 查找的限制范围（可选）。

3) 如果查找失败应如何处理，是返回nil还是返回一个错误消息（可选）。

4) 重复查找多少次，如果这个参量的值是负的，就是朝后查找（可选）。

• kill-region、delete-region、copy-region-as-kill

kill-region函数将一个缓冲区中位点和标记之间的文本剪切掉，并将这些文本保存在kill环中，因此能够将它们重新找回来。

delete-region函数将缓冲区中位点和标记之间的文本移走并扔掉，不能够将它们再重新找回来。

copy-region-as-kill函数将缓冲区中位点和标记之间的文本拷贝到kill环中，从kill环中可以将它们重新找回来。这个函数不将缓冲区中的文本剪切掉。

8.7 查找练习

• 编写一个查找字符串的交互函数。如果找到需要的字符串，在其后设置位点并显示这样

一条消息："Found!"。（不要使用 search-forward 作为这个函数的函数名；如果使用了这样一个函数名，将覆盖 Emacs 的 search-forward 函数本身。可以使用如 test-search 这样的函数名。）

- 编写一个函数，这个函数在回显区打印 kill 环的第三个元素。如果 kill 环没有第三个元素，则打印一条适当的消息。
- 在第19.29版中，copy-region-as-kill 函数不再设置 this-command 变量。这种变化的后果是什么？要采取什么样的相应变化，才能达到同样的效果？

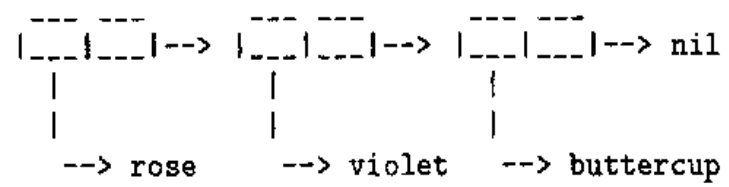
第9章 列表是如何实现的

在 Lisp 中，原子是以一种直观的方式记录在计算机中的；如果说它在实际的实现上不那么简单，那么它至少在理论上是简单的。例如，原子“rose”是由紧挨着的四个字符“r”、“o”、“s”、“e”记录下来的。在另一方面，列表是用一种不同的方式保存的。列表的保存机制同样简单，但是要理解这个思想需要花一点时间。列表是用一系列成对的指针保存的。在这个成对的指针系列中，每一对指针的第一个指针要么指向一个原子，要么指向另外一个列表；而其第二个指针要么指向下一个指针对，要么指向符号 nil，这个符号标记一个列表的结束。

指针本身相当简单，就是它指向的电子地址。因此，一个列表实际上就是被保存为一系列电子地址。

例如，列表 (rose violet buttercup) 有三个元素：“rose”、“violet”和“buttercup”。在计算机中，“rose”的电子地址存储在计算机内存片段中，其后紧跟着给出原子“violet”存放位置的地址。这个地址又与给出“buttercup”存放位置的地址连接在一起。

这听起来似乎很复杂，用一个图来说明就很简单了：

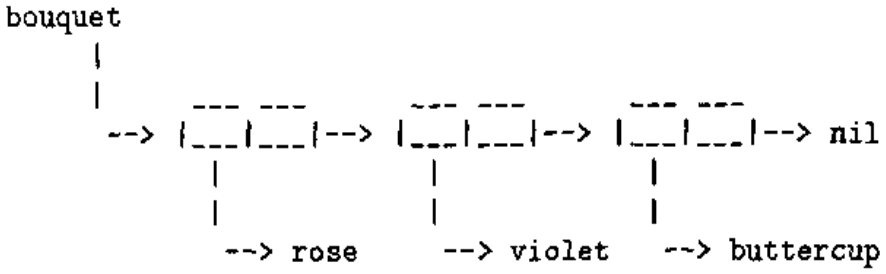


在这个图中，每一个方框代表计算机内存中的一个“字”（word），它通常是以内存地址的方式存放一个 Lisp 对象。这些方框，也就是这些地址，是成对的。图中的箭头要么是指向一个原子的地址，要么是指向另外一对地址的地址。第一个方框是“rose”的地址，其箭头指向“rose”；第二个方框是下一对方框的地址。这第二对方框中的第一个方框是“violet”的地址，而其第二个方框指向下一对方框的地址。最后一对方框的第二个方框（也就是最后一个方框）指向符号 nil。这个符号标记一个列表的结束。

当用一个函数（如 setq）将一个列表赋给一个变量时，实际上就是将列表的第一个方框的地址赋给那个变量。因此，对下面这个表达式求值

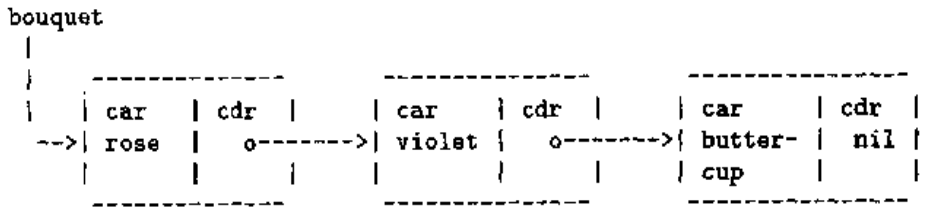
```
(setq bouquet '(rose violet buttercup))
```

产生如下图展示出来的情况：



在这个例子中，符号 bouquet 中保存第一个方框对的地址。确实，这个符号由一组地址框组成，其中第一个地址框就是“bouquet”一词的地址；如果有同名的函数定义加到这个符号上，其中第二个地址框是这个函数定义的地址；其中第三个地址框就是列表 (rose violet buttercup) 的成对地址框系列中第一对的地址；等等。

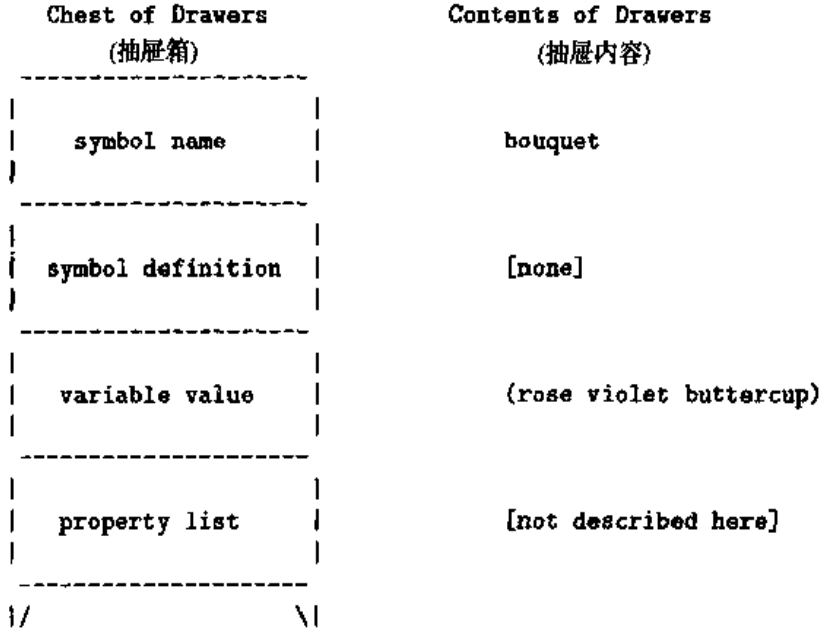
这个列表同样可以用下面这种不同的方式来表示：



在前面的一节中，我曾经建议将一个符号想象成为一个抽屉箱。函数定义被放在其中一个抽屉中，符号的值放在另外一个抽屉中，等等。保存符号值的抽屉中的内容的改变，不影响保存函数定义的抽屉中的内容。反之亦然。实际上，放在每一个抽屉中的都是符号值的地址或者函数定义的地址。这就像你在阁楼中找到一个旧抽屉箱，在其中一个抽屉中发现了一张地图，这张地图告诉你财宝存放的位置。

（除了符号名、符号定义和变量的值之外，符号还有一个“抽屉”保存其属性列表。这个属性列表能用于记录其他信息。属性列表不在这里介绍，有关它的内容可以参见《*GNU Emacs Lisp* 技术手册》的“属性列表”一节。）

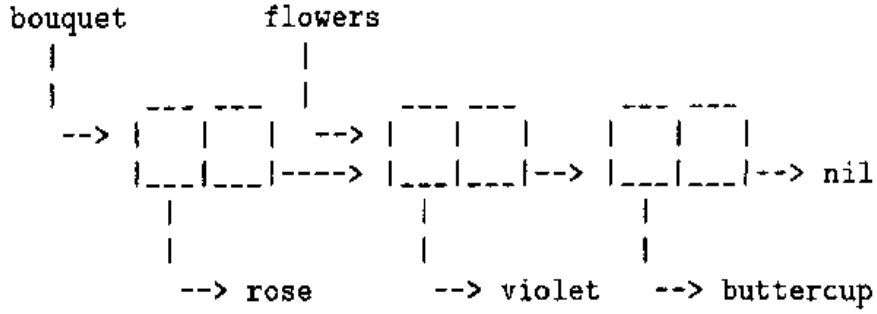
下面是一个充满想象力的表示：



如果一个符号被设置为一个列表的 cdr，这个列表本身没有改变；符号仅仅只有列表的地址。（用术语来说，car和 cdr 是非破坏性的。）因此，对下面的表达式求值

```
(setq flowers (cdr bouquet))
```

将产生这样的结果：



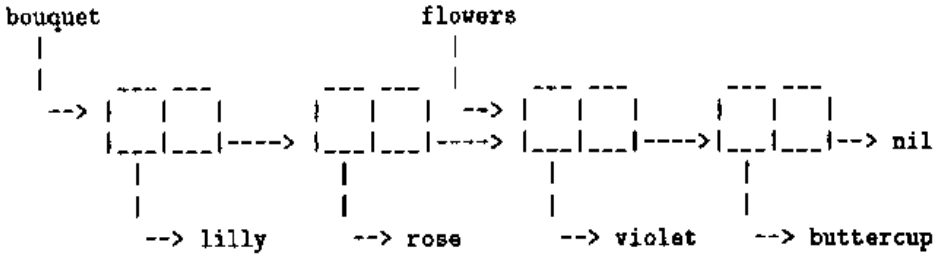
flowers 的值是（violet buttercup），这就是说，符号 flowers 存放成对的地址框的地址，成对的地址框的第一个地址对中存放violet 的地址，第二个地址对中存放 buttercup 的地址。

一个成对的地址框被称为一个“cons 原胞”（*cons cell*）或者一个“带点偶对”（*dotted pair*）。具体内容参见《*GNU Emacs Lisp*技术手册》中的“Lisp 类型”一节以及“带点偶对注解”一节。

就像前面说的那样，cons函数在一系列地址对前面插入一个新的地址对。例如，对下面的表达式求值

```
(setq bouquet (cons 'lilly bouquet))
```

会得到：



然而，这并不改变符号 flowers 的值，不信可以通过对下面的表达式求值来看一看：

```
(eq (cdr (cdr bouquet)) flowers)
```

这个表达式的返回值为“真”（t）。

除非被重新赋值，否则符号 flowers 的值仍旧是 (violet buttercup)。这就是说，它拥有首地址为 violet 的地址的cons 原胞的地址。同样，这并不改变任何已经存在的 cons 原胞，它们还是原封不动地存在那里。

因面，在 Lisp 中，要得到一个列表的cdr，只要得到地址系列中下一个cons原胞的地址即可；要得到一个列表的 car，就是得到这个列表的第一个元素的地址；要用 cons 函数在一个列表中插入一个新元素，其作用就是往列表中插入了一个新的 cons 原胞。这就是列表实现的方式。Lisp 底层的结构就是这样不可思议的简单。

cons 原胞系列中最后一个地址指向什么呢？就是指向空列表(即符号 nil)的地址。

总之，当为一个 Lisp 变量赋值时，它提供的是列表的地址，变量就指向这个列表的地址。

练习

将符号 flowers 设置为 violet 和 buttercup 两个元素组成的列表。往这个列表中增加两种新的花名，并将这个新的列表赋值给 more-flowers 变量。将 flowers 的car设置为一种鱼的名字。看一看 more-flowers 列表现在的内容是什么？

第10章 找回文本

在 GNU Emacs 中，无论你何时用“kill”命令从缓冲区中剪切了文本，你都能用一个“yank”命令将其重新找回。从缓冲区中剪切的文本被放在 kill 环中，yank 命令则将 kill 环中适当的内容重新插入到缓冲区中（不一定非得是原来的缓冲区）。

一个简单的 C-y (yank) 命令，就从 kill 环中取出第一个元素插入到当前的缓冲区中。如果 C-y 命令后面紧跟一个 M-y 命令，则不是第一个元素而是第二个元素被插入到当前缓冲区中。连续的 M-y 命令则将使第三个元素或第四个元素等代替第二个元素而被插入到当前缓冲区中。当这样不断键入 M-y 而到达 kill 环的最后一个元素时，它就循环地将第一个元素插入到当前缓冲区中。（因此 kill 环被称为“环”，而不仅仅是列表。然而，保存文本的实际数据结构是一个列表。关于将一个列表作为环来处理的详细内容，参见附录B，“处理 kill 环”。）

10.1 kill 环总览

kill环是文本字符串的一个列表。它类似于下面的列表：

```
("some text" "a different piece of text" "yet more text")
```

如果 kill 环的内容就是这样的一个列表，当键入 C-y 时，字符串“some text”将插入到当前缓冲区中光标当前所处的位置（即位点处）。

yank 命令也用于复制文本，这通过拷贝文本实现。被拷贝的文本不是从缓冲区中删除掉，但是这部分文本的一个拷贝被放到 kill 环中，并可以用 yank 命令将其找回来插入到当前缓冲区中。

能够将文本从 kill 环中找回的函数有三个：yank 函数，通常绑定到 C-y；yank-pop 函数，通常绑定到 M-y；rotate-yank-pointer 函数，这个函数被前面两个函数使用。

这些函数通过一个被称为 kill-ring-yank-pointer 的变量指向 kill 环。事实上，yank 函数和 yank-pop 函数中插入文本的代码都是下面这个表达式：

```
(insert (car kill-ring-yank-pointer))
```

在开始理解 yank 函数和 yank-pop 函数如何工作之前，分析一下 kill-ring-yank-pointer 变量和 rotate-yank-pointer 函数是必要的。

10.2 kill-ring-yank-pointer变量

就像 kill-ring 是一个变量一样，kill-ring-yank-pointer 也是一个变量。它通过被绑定到相应的值来指向某些东西，这一点与其他 Lisp 变量没有什么两样。

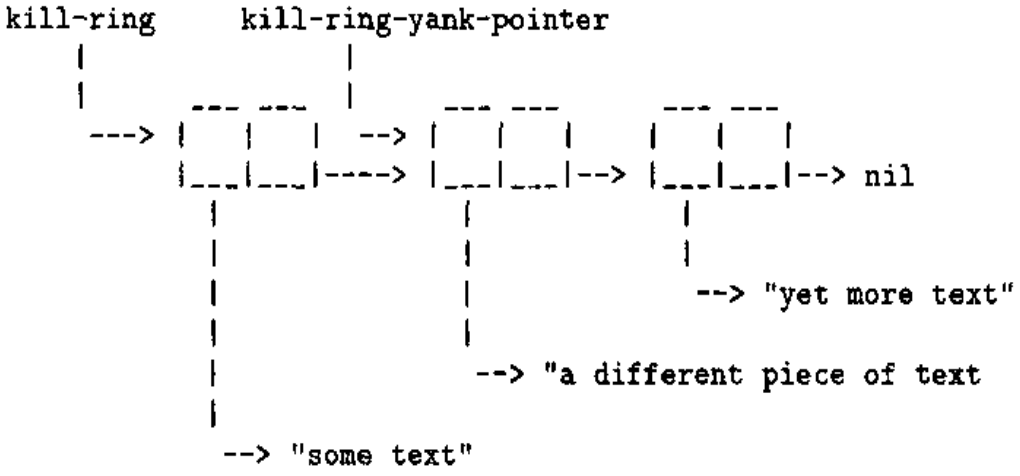
因而，如果 kill 环的值是：

```
("some text" "a different piece of text" "yet more text")
```

并且 kill-ring-yank-pointer 变量指向其中的第二个元素，则 kill-ring-yank-pointer 变量的值就是：

```
("a different piece of text" "yet more text")
```

就像在前面章节中解释的那样（参见第 9 章，“列表是如何实现的”），计算机并不保存同时被 kill-ring 变量和 kill-ring-yank-pointer 变量指向的内容的两个拷贝。“a different piece of text”和“yet more text”两个字符串都没有复制的副本。事实上，两个 Lisp 变量指向同样的一个文本块。用一个图来表示就是：



变量 kill-ring 和变量 kill-ring-yank-pointer 都是指针。但是在提到 kill 环本身时，常常就像是指其组成部分。在提到变量 kill-ring 时，就像它是一个列表而不是一个指向这个列表的指针。相反，在提到变量 kill-ring-yank-pointer 时，它确实是指向一个列表的指针。

这样两种讨论同样一件事情的方法，使人初看之下不免迷惑；但是仔细分析，就会发现这是有意义的。kill环一般被认为是保存从当前缓冲区中剪切的文本信息的完整结构。另一方面，变量 kill-ring-yank-pointer 则用于指示(即“指向”)kill 环的某一部分，kill 环的这一部分的第一个元素将被插入到缓冲区中。

rotate-yank-pointer 函数改变 kill-ring-yank-pointer 变量指向的 kill 环中的元素。当这个指针指向的下一个元素超出了 kill 环的末尾，它自动将这个指针指向 kill 环的第一个元素。这就是将一个列表自动地转化为一个环的方法。rotate-yank-pointer 函数本身并不难理解，但是它包含了许多细节。在附录B“处现 kill 环”中，将描述这个函数以及更简单的 yank 函数和 yank-pop 函数。

10.3 练习：使用 yank函数和 nthcdr 函数

- 使用 C-h v(describe-variable) 命令，看一看你的 kill 环的值。为你的kill环增加几个元素，然后再看一看它的值。使用 M-y(yank-pop) 命令在 kill环中移动。你的 kill 环中有多少个元素？用 kill-ring-max 得到这个值。你的 kill 环是否已经满了，或者你能否可以继续往其中保存更多的文本块？
- 使用 nthcdr 和 car 函数构建一系列表达式分别来返回一个列表的第一个元素、第二个元素、第三个元素、第四个元素，等等。

第11章 循环和递归

Emacs Lisp 有两种方式使一个表达式或者一系列表达式不断被求值：一是使用while 循环，一是使用“递归”（*recursion*）。

循环操作是很有价值的。例如，要向前移动四句，只需编写一个程序，这个程序不断地重复一个过程——向前移动一句的过程，重复四次就行了。因为计算机对循环不会厌烦也不会疲劳，所以这样的重复劳动不会对计算机造成有害的影响，而对人而言这种过度的错误循环是有害的。

11.1 while

while 特殊表对其第一个参量求值，并测试这个返回值的真假。这与 Lisp 解释器处理 if 表达式的情况相似。然而，接下来就不一样了。

在 while 表达式中，如果对其第一个参量的求值结果是“假”，则 Lisp 解释器跳过这个表达式的其余部分（也就是这个表达式的主体）而不对它求值。但是，如果第一个参量的返回值为“真”，则 Lisp 解释器就继续对这个表达式的主体求值，然后再次测试 while 的第一个参量是否为“真”。如果第一个参量的返回值为“真”，则Lisp 解释器再一次对表达式主体求值。。

while 表达式模板如下所示：

```
(while true-or-false-test
  body...)
```

只要 while 表达式中的测试结果是“真”值，这个表达式主体就被重复求值。这个过程之所以被称为循环，就是因为 Lisp 解释器一次又一次地反复做同样一件事情，就像一架飞机飞一个环形轨道一样。当测试参量的返回值为“假”时，Lisp 解释器不对表达式的其余部分求值，并“退出这个循环”。

很清楚，如果对第一个参量求值的结果总是返回“真”，这个表达式的主体就会被一次又一次地不断重复求值，一直持续下去。相反，如果第一个参量的值永远也不可能是“真”，则表达式主体也将永不被求值。要编写一个完整的while 循环，意味着要选择一个正确地返回“真”和“假”值的机制，这个测试机制只在需要的次数内返回“真”值，然后测试返回值就为“假”。这里的次数就是表达式主体将被求值的次数。

while 循环的返回值就是真假测试的返回值。它的一个有趣的结果就是一个无错的 while 循环，将总是返回 nil 或者“假”，而不管它循环求值了 1 次或者 100 次，或者根本一次也没有。一个求值成功的 while 表达式从不返回“真”值！这意味着，while 表达式总是为了它的附带效果，也就是在 while 循环主体中的表达式的结果，而被求值。这一点很有意义。循环本身不是目的，但是在循环中表达式被求值才是重要的。

11.1.1 while 循环和列表

控制 while 循环的一个通用方法就是测试一个列表中是否还有元素。如果有，循环就重复下去；如果没有，循环就结束。因为这是一种重要的技术，因此在此将创建一个简短的例子来演示它。

测试一个列表中是否有元素的简单方法就是对列表求值：如果列表中没有元素，它就是一个空列表，因此这种测试返回一个空列表 ( )，也就是 nil 或者“假”。另一方面，如果列表中有元素，对列表的求值就返回这些元素。因为 Lisp 认为非空的任何元素都是“真”，所以任何有元素的列表在测试时都返回“真”。

例如，通过对下面的 setq表达式求值，能够将变量 empty-list 设置为 nil。

```
(setq empty-list ())
```

对这个setq表达式求值后，能够用通常的办法——将光标置于符号之后并键入 C-x C-e，来对变量 empty-list 求值。nil 将出现在回显区中：

```
empty-list
```

另一方面，如果将一个有元素的列表赋值给一个变量，当对这个变量求值时，它指向的列表中的元素将显示在回显区中。对下面的两个表达式求值就可以看到这一点：

```
(setq animals '(giraffe gazelle lion tiger))

animals
```

因此，要创建一个 while 循环来测试在列表 animals 中是否有元素，while 循环的第一个部分将是这个样子：

```
(while animals
       ...
```

当 while 测试它的第一个参量时，animals 列表被求值。这返回一个列表。只要这个列表中还有元素，while 就认为测试为“真”。但是当列表空了时，则测试结果为“假”。

为了避免 while 循环永无止境地循环下去，需要一些机制来最终提供空的列表。一个经常使用的技术，就是在 while表达式主体中用一个表达式将这个列表的值设置为这个列表的 cdr。每执行一次 cdr 求值，列表就更短，元素就更少，直到最终只剩下一个空列表。这时，while循环的测试将返回“假”，就不会再对循环主体求值了。

例如，绑定到变量 animals 的关于动物的列表，能够用下面的表达式将其设置为原来列表的 cdr。

```
(setq animals (cdr animals))
```

如果你已经对前面的表达式求值并随后再对这个表达式求值，你将看到(gazelle lion tiger) 显示在回显区中。如果再一次对这个表达式求值，(lion tiger) 将显示在回显区中。如果继续对这个表达式求值，(tiger) 将显示在回显区。再继续求值下去最后就是一个空列表，显示为 nil。

下面是一个 while 循环模板，它重复地使用 cdr函数以使 while循环的真假测试最终返

回“假”值：

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

对列表长度的测试以及使用 cdr 来逐一减少列表中的元素，可以放在一个函数中，用来遍历一个列表，并将列表的每一个元素以一个元素独占一行的方式打印出来。

11.1.2 一个例子：print-elements-of-list

print-elements-of-list 函数演示了使用一个列表作为测试部分的 while 循环的结构。

这个函数需要几行的空间来输出。因为回显区只有一行，我们不像前面演示其他函数那样在 Info 中对它求值来显示它是如何工作的，而是需要将必要的表达式拷贝到“*scrach*”(草稿)缓冲区，并在草稿缓冲区中对它们求值。你可以用 C-SPC(set-mark-command) 命令标记待拷贝区域的开始处，将光标移动到这个区域的末尾并键入 M-w(copy-region-as-kill) 命令来拷贝这个区域。在草稿缓冲区中，能够通过键入 C-y(yank) 命令来插入这些表达式。

当你已经将这些表达式拷贝到草稿缓冲区之后，逐一对这些表达式求值。一定要对最后一个表达式——(print-elements-of-list animals)求值，求值是通过键入 C-u C-x C-e 完成的，也就是给 eval-last-sexp命令传送一个参量。这将值对表达式的求值结果被显示在草稿缓冲区中而不是被显示在回显区中。(否则你将在回显区中看到诸如：^Jgiraffe ^J^Jgazelle^J^Jlion^J^Jtiger^Jnil 这样的东西，其中，“^J”代表一个新行，在草稿缓冲区中，每一个元素都放在单独的一行中。如果你高兴的话，只管在 Info 中对这些表达式求值好了，看看会得到什么样的结果。)

```
(setq animals '(giraffe gazelle lion tiger))

(defun print-elements-of-list (list)
  "Print each element of LIST on a line of its own."
  (while list
    (print (car list))
    (setq list (cdr list))))

(print-elements-of-list animals)
```

当你在草稿缓冲区中依次对上述三个表达式求值时，下面的内容将显示在草稿缓冲区中：

```
giraffe

gazelle

lion

tiger

nil
```

列表的每一个元素都分别占一行打印出来（这是由 print 函数完成的），然后这个函数的返回值 nil 被打印出来。因为这个函数的最后一个表达式是while 循环，又因为 while 循环总是返回 nil 的，所以打印完列表的最后一个元素之后，就打印函数的返回值nil。

11.1.3 使用增量计数器的循环

只有当该停止时就停止下来，这个循环才是有用的。除了用列表来控制循环之外，一个通用的终止一个循环的方法是：当所需数量的循环次数执行完毕时，其作为测试内容的第一个参量变成“假”。这意味着循环必须要有一个计数器——一个记录循环次数的表达式。

这种测试可以是这样的表达式 (< count desired-number)。这个表达式在变量 count 的值小于 desired-number 变量的值时返回“真”，而在 count的值等于或者大于 desired-number 的值时返回“假”。对变量 count 进行增量运算的表达式可以是简单的 setq 表达式，如(setq count (1+ count))，这里 1+ 函数是 Emacs Lisp 的内置函数，它对其参量加 1。(表达式（1+ count）与表达式(+ count 1)功能完全等价，但是更易于人们阅读。)

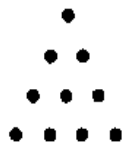
由增量计数器控制的 while 循环的模板如下所示：

```
set-count-to-initial-value
(while (< count desired-number)          ; true-or-false-test
  body...
  (setq count (1+ count)))               ; incrementer
```

注意，你需要为变量count 设置初始值，通常将它设置为 1。

1. 使用增量计数器的例子

假设你正在海滩上玩耍，并决定用鹅卵石排出一个三角形。在第一行放置一块鹅卵石，在第二行放置两块鹅卵石，在第三行放置三块鹅卵石，如此等等。如下所示：



（大约在2500年前，毕达哥拉斯和其他人通过考虑类似的问题发展了早期的数论。）

假设你想知道，要排出一个 7 行的三角形，需要多少块鹅卵石？

很清楚，你所要做的就是从 1 加到 7。有两种方法完成它，一是从小的数开始，依次往上加；一是从最大的7开始，依次减到1。由于在编写 while循环时，这两种方式都是通用的，因此我们将创建两个例子，一个从 1 往上加到 7，一个从 7 往下减到1。在第一个例子中，我们从 1 开始往上加 2、3，等等。

如果你仅仅对一个短的数字列表求和，最简单的办法就是直截将这些数字全部加起来。但是，如果你事先不知道列表中有多少数字，或者要对一个相当长的列表操作，就需要设计求和函数，以使你只需多次重复一个简单的过程，而不是单次执行一个复杂的过程来完成求和工作。

例如，不要一次将所有的鹅卵石块加起来，你可以将第一行的鹅卵石数 1 加上第二行的鹅卵石数 2，然后将这个结果加上第三行的鹅卵石数 3。之后，加上第四行的鹅卵石数 4……如此

等等。

这个过程的关键特征就在于重复执行的每一个过程都很简单。在这个例子中，每一步我们只要将两个数相加，即当前一行的鹅卵石数加上先前已经加好的前几行的鹅卵石总数。将两个数相加的过程，一次又一次地重复执行，直到最后一行加到前面各行的总数之中。在更加复杂的循环中，重复执行的每一步可能不像这样简单，但它总比一次完成所有的事情要简单。

2. 函数定义部分

前面的分析告诉了我们这个函数定义的骨架：首先，需要一个称为total 的变量，这个变量记录鹅卵石的总数。这个值最终由函数返回。

其次，这个函数需要一个参量：即三角形的总行数。它叫做 number-of-rows。

最后，需要一个作为计数器的变量。可以将这个变量称为 counter，但是一个更好的名字是 row-number。这是因为，这个计数器是对行数计数的，而且一个程序应当尽可能使人们易于理解。

当 Lisp 解释器对这个函数的表达式求值时，total 变量的值应当被设置为 0，因为此时尚未往它中加任何数字。然后，这个函数应当往 total 变量中加上第一行的鹅卵石数 1，继而加上第二行的鹅卵石数 2.....直到没有可加的为止。

total 变量和 row-number 变量都只用在函数内部，因此可以用 let命令将它们声明为局部变量并赋初始值。很清楚，total 变量的初始值应当是 0。row-number 变量的初始值应当是1，原因是要从第一行开始计数。这意味着let语句应该这样：

```
(let ((total 0)
      (row-number 1))
  body...)
```

声明了内部变量并绑定到它们的初始值之后，就可以开始 while 循环了。用作测试的表达式应当在row-number变量的值小于或者等于 number-of-rows 变量的值时返回“真”。（如果测试表达式只在 row-number 的值小于 number-of-rows 的值时返回“真”，三角形的最后一行将不被计算到总数中，因此行数应当小于或者等于总行数）。

Lisp 提供了 <=函数，这个函数在其第一个参量小于或者等于其第二个参量的值时返回“真”；否则返回“假”。因此这个 while 表达式的第一个参量应当如下所示：

```
(<= row-number number-of-rows)
```

鹅卵石的总数是这一行的鹅卵石数加上前面所有各行的鹅卵石数目的总和。因为每一行的鹅卵石的数目就等于行数，因此可以直接将 row-number 变量的值加到鹅卵石总数上。（很清楚，在更复杂的情况下，某一行的鹅卵石的数目以一种更复杂的方式与它所在的行相关。如果是这种情况，就要用适当的表达式取代下面表达式中的 row-number 变量。）

```
(setq total (+ total row-number))
```

这个表达式所做的工作就是将 total 变量重新赋值为原有值加上当前一行的鹅卵石数目。

设置了 total 变量的值之后，如果有的话，就应当为下一次循环建立条件了。这就是将 row-number 变量用作一个计数器，将它递增1。row-number 变量被递增后，while 循环的测试表达式重新测试这个计数器的值是否依然小于或者等于 number-of-rows 的值。如果是，

在循环中将 row-number变量的新值加到 total变量中。

Emacs Lisp 的内置函数 1+ 的功能，就是往一个数增加1，因此 row-number 变量可以用下面的表达式递增：

```
(setq row-number (1+ row-number))
```

3. 组装完成函数定义

我们已经创建了这个函数定义的各个部分；现在需要将这些部分组装起来。

首先，while 表达式的内容如下所示：

```
(while (<= row-number number-of-rows)    ; true-or-false-test
  (setq total (+ total row-number))
  (setq row-number (1+ row-number)))     ; incrementer
```

这个表达式再加上 let 表达式的变量列表，就已经非常接近于整个函数定义的函数体了。但是，还需要一个最后的元素，这有一点微妙。

这最后一个元素就是将变量 total 单独作为一行放在 while 表达式之后。否则，整个函数的返回值总是空 (nil)。因为，在这种情况下，整个函数的返回值就是 let 表达式主体中最后一个表达式的值，也就是 while 表达式的返回值，而这个返回值总是 nil。因此为了返回求和的总数，需要增加单独的这一行。

这一点初看起来似乎并不明显。看起来，函数的最后一个表达式好像是那个增量表达式。但是这个增量表达式是 while 表达式主体的一部分。它是以符号while 开始的列表的最后一个元素。而且，整个 while 循环是 let 表达式主体中的一个列表。

从结构上看，这个函数将是如下所示：

```
(defun name-of-function (argument-list)
  "documentation..."
  (let (varlist)
    (while (true-or-false-test)
      body-of-while... )
    ... )                              ; Need final expression here.
```

对 let 表达式求值所返回的结果就是 defun函数的返回值。这是因为 let 表达式没有嵌入到其他列表中，而只是嵌入在函数定义（defun）之中。然而，如果while 表达式是 let 表达式的最后一个元素，这个函数将总是返回一个空值 nil。这并不是我们所需要的！确实，我们所需要的是变量 total 的值。只要简单将这个变量作为 let 表达式的最后一个元素放在其中就行了。在前面的元素求值之后，这个元素被求值，这意味着它被正确地赋值后再被求值。

将所有这些元素放在一行中就更容易看明白。这种格式使人明显地看到 *varlist* 和 while 表达式是 let 表达式的第二个和第三个元素，而变量 total 则是 let 表达式的最后一个元素。

```
(let (varlist) (while (true-or-false-test) body-of-while... ) total)
```

将所有的表达式组合起来，这个 triangle 函数定义就是如下所示：

```
(defun triangle (number-of-rows)       ; Version with
                                       ;   incrementing counter.
```

```
  "Add up the number of pebbles in a triangle.
The first row has one pebble, the second row two pebbles,
the third row three pebbles, and so on.
The argument is NUMBER-OF-ROWS."
  (let ((total 0)
        (row-number 1))
    (while (<= row-number number-of-rows)
      (setq total (+ total row-number))
      (setq row-number (1+ row-number)))
    total))
```

在你通过对上面的这个函数求值而安装了triangle函数之后，你就可以试验这个函数了。这里又两个例子：

```
(triangle 4)

(triangle 7)
```

求值的结果是：前4个数的和是10，前7个数的和是28。

11.1.4 使用减量计数器的循环

编写 while 循环的另外一个通用的方法是在编写测试表达式时根据一个计数器是否大于零来决定“真假”值。只有当计数器大于零时，循环才继续下去。当计数器等于或者小于零，循环就中止了。为了使这种方法能够工作，这个计数器一定要从大于零开始计数，并通过一个不断重复求值的表达式一步一步地变得越来越小。

测试将是这样的一个表达式：(> counter 0)，如果 counter 变量的值大于零，则测试返回“真”；如果 counter 变量的值等于或者小于零，测试就返回“假”。值计数器变量的值越来越小的表达式可以是一个简单的 setq 表达式，如(setq counter (1- counter))，这里 1- 函数是 Emacs Lisp中的一个内置函数，它将其参量的值减 1。

使用减量计数器的 while 循环的模板如下所示：

```
(while (> counter 0)                    ; true-or-false-test
  body...
  (setq counter (1- counter)))          ; decrementer
```

1. 使用减量计数器的例子

为了演示使用减量计数器的 while 循环，我们将重新编写 triangle 函数，使计数器递减到零来控制循环的执行。

这是前面那个函数的对应版本。在这种情况下，为了得到 3 行的三角形是由多少块鹅卵石排成的，就要将第三行的鹅卵石数 3 加上第二行的鹅卵石数 2，再加上第一行的鹅卵石数 1。

同样，为了得到 7 行的三角形由多少块鹅卵石排成的，就要将第 7 行的鹅卵石数 7加上它前面一行的鹅卵石数 6，再将这两个数之和加上更前面一行的鹅卵石数5，如此等等。就像前面那个例子一样，每一次加法都只是涉及到两个数的相加。已经加过的行，其中的鹅卵石数也已经加到总的鹅卵石数中了。将两个数相加的过程一次又一次地重复执行，直到所有行都加完。

我们知道从多少块鹅卵石开始：因为最后一行的鹅卵石数就等于它所在的行数。如果三角形有 7 行，最后一行的鹅卵石就有 7 块。同样，我们知道前面一行有多少块鹅卵石：它比当前行的鹅卵石数少一块。

2. 函数定义的各部分

我们以三个变量开始，它们是：三角形中总的行数、每一行中的鹅卵石数以及要计算的总的鹅卵石数。这些变量可以分别取名为: number-of-rows，number-of-pebbles-in-row 和 total。

total变量和 number-of-pebbles-in-row 变量只用在函数内部，它们是用 let表达式声明的。当然，total变量的初始值应当是零。然而，number-of-pebbles-in-row 变量的初始值应当等于三角形中的行数，因为函数是从最长的一行开始相加的。

这意味着 let 表达式的开头部分是：

```
(let ((total 0)
      (number-of-pebbles-in-row number-of-rows))
  body...)
```

鹅卵石的总数能够通过反复地将当前行的鹅卵石数加上已经求和的各行的鹅卵石数得到，也就是反复地对下面的表达式求值：

```
(setq total (+ total number-of-pebbles-in-row))
```

在将 number-of-pebbles-in-row 与 total 相加后，number-of-pebbles-in-row 的值应该减1，因为下一次循环时，下一行的鹅卵石数将被加到鹅卵石的总数中。

下一行的鹅卵石数比当前行的鹅卵石数少1，因此在计算下一行的鹅卵石数时可以值用 Emacs Lisp 的内置函数1-。这可以用下面的表达式完成：

```
(setq number-of-pebbles-in-row
      (1- number-of-pebbles-in-row))
```

最后，我们知道，当一行中没有鹅卵石时，while 循环应当停止下来。因此while 循环中的测试表达式很简单：

```
(while (> number-of-pebbles-in-row 0)
```

3. 组装完成函数定义

将这些表达式组装起来，就有了一个这样的函数定义：

```
;;; First subtractive version.
(defun triangle (number-of-rows)
  "Add up the number of pebbles in a triangle."
  (let ((total 0)
        (number-of-pebbles-in-row number-of-rows))
    (while (> number-of-pebbles-in-row 0)
      (setq total (+ total number-of-pebbles-in-row))
      (setq number-of-pebbles-in-row
            (1- number-of-pebbles-in-row)))
    total))
```

就像编写的那样，这个函数可以正常工作了。

然而，其中的一个局部变量 number-of-pebbles-in-row 并不是必需的。

当 triangle 函数被求值时，符号 number-of-pebbles-in-row 被绑定到一个数字上，给它赋初始值。在函数体中，这个数可以被当做一个局部变量一样被改变，而不用担心这种改变会影响函数之外的变量的值。这是 Lisp 的一个非常有用的特性；它意味着变量 number-of-rows 可以用于函数中变量 number-of-pebbles-in-row 使用的任何地方。

下面就是这个函数的第二个版本，它写得更清楚一点：

```
(defun triangle (number)                    ; Second version.
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (setq number (1- number)))
    total))
```

简要地说，一个正确的 while 循环将包含三个部分：

1) 一个真假测试表达式，它在循环体执行正确的循环次数之后返回“假”。

2) 一个求值表达式，它在求值完成之后返回用户需要的值。

3) 一个表达式，它用来改变传送给真假测试表达式的值，这样在循环体执行正确的循环次数之后才能返回“假”值给真假测试表达式。

11.2 递归

递归函数，就是自己调用自己的函数。当函数对其自身求值时，它找到要求对自己求值的代码，因此这个函数对它本身一次又一次地求值。除非递归函数提供了一个停止自我调用的机制，否则递归函数将永不停止地反复对自身求值。

一个递归函数通常包含一个条件表达式，这个条件表达式有三个部分：

1) 一个真假测试，它决定函数是否继续调用自身，这里称之为*do-again-test*。

2) 函数名。

3) 一个表达式，它在函数被重复求值正确的次数之后使条件表达式返回“假”值，称为*next-step-expression*。

递归函数能够比任何其他函数都简单。实际上，当人们第一次使用递归函数时，它总是显得如此神秘、简单，以致无法理解。就像学骑自行车一样，阅读一个递归函数定义需要一个诀窍，这个诀窍初看很难，其实很简单。

递归函数的模板如下所示：

```
(defun name-of-recursive-function (argument-list)
  "documentation..."
  body...
  (if do-again-test
    (name-of-recursive-function
          next-step-expression)))
```

递归函数每被求值一次，一个参量就被绑定到 next-step-expression 的值上。这个值又用于 do-again-test。设计next-step-expression 表达式的目的是当函数不再需要被重复求值时 do-again-test 测试表达式能够返回“假”。

这个真假测试有时被称为停止条件（*stop condition*），因为当它的测试结果为“假”时，就停止循环调用。

11.2.1 使用列表的递归函数

将一个数字列表中的元素打印出来的 while 循环的例子，也能够用递归的方法写出来。下面就是它的代码，其中包含一个将变量 animals 的值赋给一个列表的表达式。

这个例子一定要拷贝到草稿缓冲区，并且要在草稿缓冲区中对每一个表达式都求值。可以使用 C-u C-x C-e 对 (print-elements-recursively animals) 表达式求值，以使结果打印在缓冲区中；否则 Lisp 解释器将试图把结果压缩到一行中打印到回显区中。

同样，要将光标紧紧置于 print-elements-recursively 函数的最后一个括号之后（注释之前）。否则，Lisp 解释器将试图对注释求值。

```
(setq animals '(giraffe gazelle lion tiger))

(defun print-elements-recursively (list)
  "Print each element of LIST on a line of its own.
Uses recursion."
  (print (car list))                    ; body
  (if list                              ; do-again-test
      (print-elements-recursively       ; recursive call
       (cdr list))))                    ; next-step-expression

(print-elements-recursively animals)
```

这个 print-elements-recursively 函数首先打印列表的第一个元素，即列表的 car。然后，如果列表不是空，这个函数调用它自己，但是传递给函数本身作为其参量的不是整个这个列表，而是由列表的第二个元素到最后一个元素组成的列表，即原来列表的 cdr。

继续求值时，这个函数首先打印出它接收到的列表的第一个元素（也就是初始列表的第二个元素）。之后，if 表达式被求值，当它为“真”时，函数又用其列表的 cdr 作为参量继续调用自身，这次传送的列表是初始列表的 cdr 的 cdr 了。（这是第二次循环）

这个函数每调用自身一次，它都用一个更短的列表作为其参量。最后，这个函数在调用自身时使用一个空列表。print 函数将空列表打印为 nil。然后，条件表达式测试列表的值。因为这时列表的值是空（nil），if 表达式返回“假”，因而 if 表达式的 then 部不被求值。这个函数最终返回一个 nil。因此，当对这个函数求值时，你将看到两个 nil。

当你在草稿缓冲区中对表达式 (print-elements-recursively animals)求值时，你将看到这样的结果：

```
giraffe
```

```
gazelle

lion

tiger

nil

nil
```

(其中第一个nil是打印的最后一个列表的值，第二个nil是整个函数的返回值。)

11.2.2 用递归算法代替计数器

在前一节中描述的triangle函数，可以用递归算法改写。它看起来就是：

```
(defun triangle-recursively (number)
  "Return the sum of the numbers 1 through NUMBER inclusive.
Uses recursion."
  (if (= number 1)                      ; do-again-test
      1                                 ; then-part
    (+ number                           ; else-part
       (triangle-recursively            ; recursive call
        (1- number)))))                 ; next-step-expression

(triangle-recursively 7)
```

对这个函数求值，就能够将它安装到 Emacs 中，并且随后可以尝试对表达式(triangle-recursively 7)求值。(记住，将光标紧紧置于函数定义的最后一个括号之后、注释内容之前，才能正确安装这个函数。)

为了理解这个函数是如何工作的，让我们考虑当函数被传递了 1、2、3、4 作为其参量的值时，在这些不同的情况下将发生什么？

首先，如果参量的值是 1，将发生什么？

这个函数定义中有一个 if 表达式，它紧接在文档字符串后。它测试变量 number的值是否等于 1。如果是等于 1，Emacs 就对 if 表达式的 then 部求值，它将返回 1。1 就是函数的值。(只有一行的三角形，只有一块鹅卵石)

然而，假设参量的值是 2。在这种情况下，Emacs 计算 if 表达式的 else 部。

if 表达式的 else 部由一个加法、对triangle-recursively 函数的递归调用以及一个递减表达式组成。就是：

```
(+ number (triangle-recursively (1- number)))
```

当 Emacs 对这个表达式求值时，最内部的表达式首先被求值，然后是其他部分。具体步骤如下：

第一步：对最内层的表达式求值。

最内层的表达式是(1- number)，因此 Emacs 将变量 number 的值从 2 递减到 1。

第二步：计算triangle-recursively函数。

这个函数包含在其自身这一点并不要紧。Emacs 将第一步的结果作为参量传递给 triangle-recursively 函数的这个实例。

在这种情况下，Emacs 就用 1 作为参量的值对 triangle-recursively 函数求值。这意味着，这次求值的结果返回 1。

第三步：计算变量 number 的值。

变量 number 是以 + 开始的列表的第二个元素，它的值为 2。

第四步：计算 + 表达式。

这个 + 表达式接受两个参量，第一个参量的值来自对变量 number 的求值结果（即第三步的结果），第二个参量的值来自对函数 triangle-recursively 求值的结果（即第二步的结果）。

这个表达式的结果就是 2 加 1 的和—— 3，并且3这个值将作为函数值返回。即有两行的三角形只有 3块鹅卵石。

参量值为 3 的情况

假设用参量值为3调用 triangle-recursively 函数，此时的情况是：

第一步：计算条件测试表达式。

if 表达式首先被求值。这个条件表达式被求值时返回“假”，因此 if 表达式的else 部被求值。（注意，在这个例子中，这个条件测试表达式在结果为“假”时，使函数调用它自己，而不是在结果为“真”时调用它自己。）

第二步：对 else 部中最内层的表达式求值。

当 else 部最内层的表达式被求值时，将 3 递减为 2。这就是next-step-expression表达式（1- number）的作用。

第三步：计算triangle-recursively函数 。

数 2 被传递给函数 triangle-recursively。通过前面的介绍，我们已经知道这种情况下 Emacs 对 triangle-recursively 函数(参量值为2)求值时将得到什么结果。通过执行前面讲述的那些过程，函数返回3。这就是这一步。

第四步：计算加法表达式。

数 3 将作为参量被传递给加法表达式，并且它将与函数调用时的参量的值3相加。结果就是 6。

整个函数的返回值，就是 6。

现在，我们知道用 3 作为参量调用 triangle-recursively 函数时将得到什么结果以及如何得到的。因此，用 4 作为参量调用这个函数的情况也就很清楚了：

在这个递归调用中，对表达式 (triangle-recursively(1- 4)) 的求值将得到表达式 (triangle-recursively 3) 的值。这个值是 6。这个值与数值4在加法表达式的第三行相加。因此整个函数的返回值就是 10。

triangle-recursively 函数每一次被求值，它就用比当前参量值小 1 的值调用它本身，直到参量值足够小、不能再调用本身为止。

11.2.3 使用 cond 的递归例子

前面讲述的 triangle-recursively 函数，是用 if 特殊表编写的。这个函数也可以用另外一个被称为 cond 的特殊表编写。这个特殊表的名字 cond 是单词“conditinal”（条件）的缩写。

虽然 cond 特殊表并不像 if 表达式那样在 Emacs Lisp 源代码中频繁使用，但它用得也相当多，解释它的使用也是很正当的。

cond 表达式的模板如下所示：

```
(cond
 body...)
```

其中，*body* 是一系列的列表。

写得更详细些，cond 表达式模板应当如下所示：

```
(cond
 ((first-true-or-false-test  first-consequent)
  (second-true-or-false-test  second-consequent)
  (third-true-or-false-test  third-consequent)
  ...)
```

当 Lisp 解释器对 cond 表达式求值时，它先计算 cond 表达式主体当中的一系列表达式中的第一个表达式的第一个元素（也就是这个表达式的 car 或者真假测试表达式）。

如果这个真假测试表达式返回 nil，则这个表达式的其余部分(结果部分)就被忽略，而下一个表达式中的真假测试被求值。如果有一个表达式的真假测试结果不是 nil，则那个表达式的后续部分就被求值。后续部分可以是一个表达式也可以是多个表达式。如果后续部分是多个表达式组成的，则这些表达式被依次求值，并且最后一个表达式的值被返回。如果这个表达式只有真假测试表达式而没有后续表达式，真假测试表达式的值就作为结果被返回。

如果所有真假测试表达式的值都是“假”，则 cond 函数返回 nil。

用 cond 特殊表来重写 triangle 函数，这个函数就是下面这个样子：

```
(defun triangle-using-cond (number)
  (cond ((<= number 0) 0)
        ((= number 1) 1)
        ((> number 1)
         (+ number (triangle-using-cond (1- number))))))
```

在这个例子中，如果变量 number 的值小于或者等于零，cond 表达式就返回 0。如果 number 的值等于 1，则返回 1。如果 number 大于 1，则计算 (+ number (triangle-using-cond (1- number))) 表达式的值。

11.3 有关循环表达式的练习

- 编写一个与 triangle 函数相似的函数，在这函数中，每一行的值等于所在行数的平方。使用 while 循环来编写这个函数。

- 编写一个与 `triangle` 函数相似的函数，求这些数的积而不是和。
- 用递归的方法重新编写上面这两个函数。然后用 `cond` 表达式重新编写这两个函数。
- 为 Texinfo 模式编写一个函数，这个函数在每一个以 “`@dfn`” 开始的段落创建一个索引入口。（在一个 Texinfo 文件中，“`@dfn`” 标记一个函数定义。关于这方面的详细资料，参见《*Texinfo: GNU 文档格式*》中关于“标记函数和命令等”的一节。）

第12章 正则表达式查询

正则表达式查询在 GNU Emacs 中使用很广泛。forward-sentence 和 forward-paragraph 这两个函数充分展示了如何使用正则表达式进行查询。

在《*GNU Emacs*技术手册》中的“正则表达式查询”一节和《*GNU Emacs Lisp* 技术手册》中的“正则表达式”一节都有正则表达式查询的有关内容。在编写这一章时，假定读者对正则表达式已经有所了解。要记住的最重要的一点是，正则表达式允许按一定的模式查询，也允许完全按照字符串字面意义进行查询。例如，forward-sentence 函数查询能够标记句子结束的可能的字符模式，并将位点移动到句子末尾。

在展示 forward-sentence 函数代码之前，分析什么肯定是标记句子结束的模式是值得的。这个模式在下一节讨论，随后的一节是对这个正则表达式查询函数re-search-forward的描述。再后面一节描述 forward-sentence 函数。最后，在这一章的最后一节将讲解整个 forward-paragraph 函数。forward-paragraph-end函数是一个复杂的函数，其中引入了 Emacs Lisp 函数的几个新特征。

12.1 查询sentence-end的正则表达式

符号sentence-end被绑定到标记句子结束的模式上。这个正则表达式应是怎样的呢?

很清楚，一个句子可以以一个句点、一个问号或者一个感叹号结束。确实，只有当一个从句以上面三个符号之一结束才应被认为是一个句子的结束。这意味着，这个模式应当包括下面这个字符集：

```
[.?!]
```

然而，我们不希望 forward-sentence 函数仅仅移动到句点、问号或者感叹号之后，因为这样的字符可能出现在句子当中。例如，句点被用于缩写之后。因此，这种查询还需要其他信息。

根据惯例，在每个句子之后空上两个空格，在句子当中的句点、问号和感叹号之后仅空一个空格。因此，句点、问号和感叹号后有两个空格的话，这就是标记一个句子结束的一个好的指示器。然而，在一个文件中，两个空格可能用 TAB 键或者换行符代替了。这意味着，这个正则表达式应当包含这三种情况。这组可选方案就是：

```
\\($\\| \\|  \\)
       ^  ^^
       TAB  SPC
```

其中，“$” 符号表示一行的结束。在上面的例子中已经指出 TAB 键和两个空格插入到表达式中的情形。这两种情况都是将实际的字符插入到表达式中。

括号和垂直线之前的两个反斜线“\\”是需要的：在 Emacs 中，第一个反斜线引用第二个反斜线；第二个反斜线指出后续的字符（即括号或垂直线）是特殊字符。

而且，一个句子可以后接一个或者多个回车符，就像这样：

```
[
]*
```

就像制表符（TAB键）和空格一样，回车符(RET键)也是直接插入到正则表达式中的。上面这个星号表示回车符重复了零次或者多次。

但是一个句子结束并不仅仅是由句点、问号、感叹号及其后接的适当的空格组成的：空格之前可能还有一个右引号或者一个大右括号。确实，在空格之前可以有多于一个这样的引号或者大括号。这种情况需要下面的表达式：

```
[]\"')}]*
```

在这个表达式中，第一个“]”是这个表达式的第一个字符。第二个字符是“"”，它之前有一个“\”字符以告诉 Emacs 这个“"”字符不是特殊字符。最后三个字符是“'”、“)”和“}”。

所有这些表示与一个句子结束相匹配的正则表达式应当是什么样的。确实，如果对 sentence-end 函数求值的话，将发现它返回下面这样的值：

```
sentence-end
     ⇒ "[.?!][]\"')}]*\\($\\|      \\|  \\)[
]*"
```

12.2 re-search-forward 函数

re-search-forward 函数与 search-forward 函数非常相似。（参见8.1.3节，“search-forward 函数”）。

re-search-forward 函数查询一个正则表达式。如果查询成功，它就紧接在最后的目标字符后面设置位点。如果查询是朝后进行的，就在第一个目标字符之前设置位点。可以告诉 re-search-forward 函数在查询成功时返回“真”(t)。（移动位点只是这个函数的一个附带效果。）

就像 search-forward 函数，这个 re-search-forward 函数也有四个参量：

1) 第一个参量是函数要查询的正则表达式。这个正则表达式是一对引号之间的字符串。

2) 第二个参量是可选的，这个参量限制函数查询的范围；它是在缓冲区中指定一个位置的约束值。

3) 第三个参量是可选的，它告诉函数如何对查询失败作出响应：如果第三个参量是nil，函数在查询失败时产生一个错误消息；如果第三个参量是其他值，函数在查询失败时返回 nil，查询成功时返回t。

4) 第四个参量也是可选的，它是一个重复计数。一个负的重复计数使 re-search-forward 函数向后查询。

re-search-forward 函数的模板是：

```
(re-search-forward "regular-expression"
                   limit-of-search
                   what-to-do-if-search-fails
                   repeat-count)
```

第二、第三和第四个参量是可选的。然而，如果要给最后两个参量或者最后两个参量的任意一个参量传递一个值，必须要给其他前面的参量传递相应的值。否则，Lisp 解释器将不知道要将值传递给哪一个参量。

在 forward-sentence 函数中，正则表达式将是变量 sentence-end 的值，也就是：

```
"[.?!][]\"')}]*\\($\\|   \\|  \\)[
]*"
```

查询的范围至段落的末尾（因为一个句子不会超过一段）。如果查询失败，这个函数将返回 nil，重复计数将由函数 forward-sentence 的参量提供。

12.3 forward-sentence 函数

将光标移动到一个句子之前的命令，是展示如何在 Emacs Lisp 中使用正则表达式查询的简单例子。确实，这个函数看起来比它实际的更长、更复杂，这是因为这个函数设计成既可以朝前也可以朝后移动，作为可选项，也可以移过多个句子。这个函数通常绑定到命令键 M-e 上。

下面是 forward-sentence 函数的代码：

```
(defun forward-sentence (&optional arg)
  "Move forward to next sentence-end.  With argument, repeat.
With negative argument, move backward repeatedly to sentence-beginning.
Sentence ends are identified by the value of sentence-end
treated as a regular expression.  Also, every paragraph boundary
terminates sentences as well."
  (interactive "p")
  (or arg (setq arg 1))
  (while (< arg 0)
    (let ((par-bag
           (save-excursion (start-of-paragraph-text) (point))))
      (if (re-search-backward
           (concat sentence-end "[^ \t\n]") par-beg t)
          (goto-char (1- (match-end 0)))
        (goto-char par-beg)))
    (setq arg (1+ arg)))
  (while (> arg 0)
    (let ((par-end
           (save-excursion (end-of-paragraph-text) (point))))
      (if (re-search-forward sentence-end par-end t)
          (skip-chars-backward " \t\n")
        (goto-char par-end)))
    (setq arg (1- arg))))
```

这个函数初看起来很长，因此最好先来看看这个函数的骨架结构，然后再分析其“肌肉”——具体的内容。这个函数的骨架结构由最靠左边的几个表达式组成：

```
(defun forward-sentence (&optional arg)
  "documentation..."
  (interactive "p")
```

```
(or arg (setq arg 1))
(while (< arg 0)
  body-of-while-loop
(while (> arg 0)
  body-of-while-loop
```

这看起来简单多了。这个函数定义包含了文档、一个 interactive表达式、一个 or 表达式和两个 while 循环。

让我们依次来分析这几个部分。

文档部分是详尽的，也很容易理解。

这个函数有一个交互的声明：interactive "p"。这意味着，如果有前缀参量，就将其作为这个函数的参量传递给函数（这个参量将是一个数）。如果没有传递参量（参量是可选的）给这个函数，参量 arg 就被绑定到数值 1。当函数forward-sentence 是无参量、非交互调用时，arg 绑定到 nil。

or 表达式处理前缀参量。它所做的，要么是保持 arg的原值（仅当arg被绑定到一个值时），要么将arg的值设定为 1（在 arg 被绑定到 nil 的情况下）。

1. while 循环

在 or 表达式之后跟着两个 while 循环。第一个 while 循环有一个真假测试表达式，当传递给 forward-sentence 函数的前缀参量是一个负数时，这个测试表达式结果为“真”。这是为朝后查询设置的。这个循环的主体与第二个while从句的循环体很相似，但是并不完全一样。我们将跳过这个 while 循环而首先关注第二个 while 循环。

第二个 while 循环完成将位点前移的工作。它的骨架结构是这样的：

```
(while (> arg 0)                ; true-or-false-test
  (let varlist
    (if (true-or-false-test)
        then-part
      else-part
  (setq arg (1- arg))))         ; while loop decrementer
```

这个 while 循环是使用减量计数器的那一种循环（参见11.1.4节，“使用减量计数器的循环”。）它有一个真假测试表达式，只要计数器（在这里是变量arg）的值大于零，测试的结果就为“真”。它同时有一个递减器，在每一次循环中将计数器的值减 1。

如果没有前缀参量被传递给 forward-sentence 函数(这也是这个函数最常使用的方式)，这个 while 循环只运行一次，因为变量 arg 的值在这种情况下默认为 1。

while 循环体包含一个 let 表达式，这个 let 表达式创建并绑定一个局部变量。let 表达式中又有一个作为 let 表这式主体的 if 表达式。

这个 while 循环体如下所示：

```
(let ((par-end
       (save-excursion (end-of-paragraph-text) (point))))
  (if (re-search-forward sentence-end par-end t)
      (skip-chars-backward " \t\n")
```

```
    (goto-char par-end)))
```

其中，let表达式创建并绑定局部变量 par-end。就像将要看到的，这个局部变量是为了给正则表达式查询提供一个边界和限制而设计的。如果查询没能在段落结束时找到正确的句子，它将在段落末尾停止查询工作。

但是首先，要检查一下变量 par-end 是如何绑定到段落结束处的。这是let表达式在 Lisp 解释器完成对下面的表达式求值之后将其返回值赋给变量 par-end来完成的。

```
(save-excursion (end-of-paragraph-text) (point))
```

在这个表达式中，(end-of-paragraph-text) 将位点移动到段落末尾，(point) 返回位点的值，然后 save-excursion 恢复位点当初的值。因而，let表达式将 save-excursion的返回值（也就是段落结束的位置）赋给变量par-end。((end-of-paragraph-text) 函数使用了forward-paragraph函数，在后面将对这个函数作简短的介绍。)

Emacs 下一步计算 let 表达式主体，也就是下面的 if 表达式：

```
(if (re-search-forward sentence-end par-end t) ; if-part
    (skip-chars-backward " \t\n")              ; then-part
  (goto-char par-end)))                        ; else-part
```

这个 if 表达式测试它的第一个参量是否为“真”。如果为“真”，就对它的 then 部求值，否则 Emacs Lisp 解释器就对 else 部求值。if 表达式中的真假测试是一个正则表达式查询。

看起来也许奇怪，这个正则表达式查询像forward-sentedce函数的“实际工作”，但是这是这种操作在 Lisp 中实现的常用方法。

2. 正则表达式查询

re-search-forward 函数查询句子的结束，也就是查询由正则表达式sentence-end定义的模式。如果找到了这个模式——找到了句子的结束标志——re-search-forward 函数将完成两件事情：

1) re-search-forward 函数完成一个附带效果，将位点移动到当前找到的句子结束处。

2) re-search-forward 函数返回一个“真”值。这是一个由 if 函数接收的值，它意味着查询成功。

这个函数的附带效果——移动位点——是在 if 函数被递交由查询的成功结束所返回的值之前被完成的。

当 if 函数从对 re-search-forward的成功调用中接收到返回的“真”值时，就对 then 部，也就是表达式(skip-chars-backward “\t\n”)求值。这个表达式朝后移动并忽略所有空格、制表符（tab键）以及回车符，直到找到一个印刷字符为止，并将位点设置在这个字符之后。因为位点已经移动到标记句子结束的正则表达式模式末尾，这个动作就是将位点紧紧置于句子的结束打印字符之后，它通常就是一个句点。

另一方面，如果 re-search-forward 函数没能找到表示句子结束的相应模式，则函数返回“假”。查询失败使 if 函数对它的第三个参量，也就是对表达式(goto-char par-end)求值，即将位点移动到段落的结尾。

正则表达式查询特别有用，由re-search-forward 说明的查询模式也随处可见。在re-search-forward 中，查询就是通过if表达式的真假测试完成的。你将经常看到这个正则表达式，或编写包括这个正则表达式模式的代码。

12.4 forward-paragraph: 函数的金矿

forward-paragraph 函数将位点朝前移动到段落末尾。它一般绑定到M-}上。这个函数使用了大量很重要的函数，包括 let*、match-beginning 和looking-at 函数。

forward-paragraph 函数的定义比 forward-sentence 的函数定义长得多，因为它是查询一个段落。段落的每一行都可能以一个填充前缀开始。

一个填充前缀由一个字符串组成，这个字符串在每一行开始处重复出现。例如，在 Lisp 代码中，对于一个成段的注释，习惯上用“；；；”开始。在文本模式中，4个空格组成另外一个通用的填充前缀，用于创建一个缩排段落。（详细情况参见《*GNU Emacs* 手册》中的“填充前缀”一节。）

填充前缀的存在，意味着除了能够找到从最左边一列开始的段落的结束处之外，forward-paragraph 函数还一定能够找到缓冲区中所有或许多行都是以填充前缀开始的段落的结束处。

而且，有时实际上要忽略已经存在的填充前缀，特别是当空白行分割不同段落时。这是一个额外的复杂性。

在此不将 forward-paragraph 函数的所有代码打印出来，只是打印其中的一些部分。如果读代码前未做任何准备，这个函数将使你望而生畏。

从骨架结构上看，这个函数是这样的：

```
(defun forward-paragraph (&optional arg)
  "documentation..."
  (interactive "p")
  (or arg (setq arg 1))
  (let*
      varlist
    (while (< arg 0)          ; backward-moving-code
      ...
      (setq arg (1+ arg)))
    (while (> arg 0)          ; forward-moving-code
      ...
      (setq arg (1- arg)))))
```

这个函数的前面部分是通常的样子：函数的参量列表由一个可选参量组成，后面跟着函数文档。

interactive 中小写的“p”意味着，如果有前缀参量，就将其传递给这个函数。这个前缀参量是一个数，是关于要移动多少段落位点的重复计数。后面一行中的 or 表达式处理没有前缀参量传递给函数的情况，这种情况可能发生在这个函数被其他代码调用而不是交互地调用之中。这种情况在前面已经讲到过（参见12.3节，“forward-sentence函数”），这里就不再重

复。好了，到现在为止，这个函数中熟悉的部分都已讲完了。下面介绍其中的一些新东西。

1. let* 表达式

forward-paragraph 函数的后续一行开始于一个 let* 表达式。这是一种与我们前面接触过的表达式不同的表达式。符号 let* 不是let!

除了 Emacs 将变量依次赋值之外，let* 特殊表与 let 相似。let* 表达式中，变量列表中后面的变量可以使用前面的变量已经由 Emacs 设置的值。

在这个函数的 let* 表达式中，Emacs绑定了两个变量：fill-prefix-regexp和paragraph-separate。其中变量 paragraph-separate 的值依赖于变量 fill-prefix-regexp的值。

让我们逐个来看一看。符号 fill-prefix-regexp被设置为对下面的列表求值所返回的值：

```
(and fill-prefix
     (not (equal fill-prefix ""))
     (not paragraph-ignore-fill-prefix)
     (regexp-quote fill-prefix))
```

这个表达式的第一个元素是 and 函数。

and 函数不停地对它的参量求值，直到遇到一个返回值为 nil的参量为止。这时and 表达式的返回值就是 nil。然而，如果没有一个参量的值是 nil，则最后一个参量的值作为表达式的值被返回。(因为这个值不是 nil，它在 Lisp 中被认为是“真”。)换句话说，一个 and 表达式只有当它的所有参量的值都是“真”的时候，才返回“真”值。

在这个例子中，只有当下面四个表达式都产生一个“真”值(如非空值)时，变量 fill-prefix-regexp 才被绑定到一个非空值上；否则这个变量就被绑定到 nil。

- fill-prefix

当这个变量被求值时，填充前缀的值被返回。如果没有填充前缀，这个变量返回 nil。

- (not (equal fill-prefix “”))

这个表达式检查填充前缀是否是一个空白字符串（即没有字符的字符串）。空白字符串是没有什么用处的填充前缀。

- (not paragraph-ignore-fill-prefix)

当变量 paragraph-ignore-fill-prefix 已经被赋值为一个“真值”(如t)之后，这个表达式返回 nil。

- (regexp-quote fill-prefix)

这是 and 函数的最后一个参量。如果所有的参量都是“真”值，对这个表达式求值的结果将作为 and 表达式的返回值返回，这个返回值被绑定到变量 fill-prefix-regexp。

对 and 表达式的成功求值，就使变量 fill-prefix-regexp 绑定到 fill-prefix 的值上。这个值由 regexp-quote 函数修改。regexp-quote 函数所做的就是读入一个字符串并返回一个精确匹配这个字符串的正则表达式。这意味着，如果填充前缀存在，fill-prefix-regexp 将被设置为与填充前缀精确匹配的值。否则，这个变量被设置为 nil。

let* 表达式中的第二个局部变量是 paragraph-separete。它被绑定到对下面的表达式求值所返回的值上：

```
(if fill-prefix-regexp
    (concat paragraph-separate
            "\\|^" fill-prefix-regexp "[ \t]*$")
  paragraph-separate)))
```

这个表达式解释了为什么值用 let* 表达式而不是值用 let表达式。if 表达式的真假测试依赖于变量 fill-prefix-regexp 的值是 nil 还是其他值。

如果变量 fill-prefix-regexp 没有值(即它为nil)，Emacs 对 if 表达式的else部求值，并将变量paragraph-separate 绑定到它的局部值上。(paragraph-separate是一个正则表达式，是用于匹配分离的段落的正则表达式模式。)

但是如果 fill-prefix-regexp 变量确实有一个值(非空值)，Emacs 就计算if 表达式的 then 部，并将 paragraph-separate 绑定到一个正则表达式上，这个正则表达式包含fill-prefix-regexp作为模式的一部分。

特别地，paragraph-separate 被设置成匹配分离的段落的正则表达式的初始值，并在其后追加一个可供选择的表达式，这个追加的表达式由 fill-prefix-regexp 加上一个空行组成。其中的“^”符号表示 fill-prefix-regexp 必须是一行的开始，行末的可选空格由“[\t]*$”定义。“\\|”则定义了分离的段落的另外一种匹配方式。

现在进入 let* 表达式的主体。这个 let* 表达式主体的第一部分处理当这个函数被赋一个负参量并因此值位点朝后移动时的情况。我们的讨论将跳过这一部分。

2. 朝前查询的 while 循环

let* 表达式主体的第二部分处理位点的朝前移动。它是一个 while 循环。只要变量 arg 的值大于零，这个循环就不停地重复下去。在值用这个函数的绝大多数情况下，arg 变量的值是1，因此这个 while 循环体正好被求值 1 次，光标向前移动一个段落。

这部分代码处理三种情况：当位点处于两个段落之间时；当位点处于一个有填充前缀的段落当中时；当位点处于一个没有填充前缀的段落当中时。

while 循环看起来是这样的：

```
(while (> arg 0)
  (beginning-of-line)

  ;; between paragraphs
  (while (prog1 (and (not (eobp))
                     (looking-at paragraph-separate))
           (forward-line 1)))

  ;; within paragraphs, with a fill prefix
  (if fill-prefix-regexp
      ;; There is a fill prefix; it overrides paragraph-start.
      (while (and (not (eobp))
                  (not (looking-at paragraph-separate))
```

```
                    (looking-at fill-prefix-regexp))
            (forward-line 1))

        ;; within paragraphs, no fill prefix
        (if (re-search-forward paragraph-start nil t)
            (goto-char (match-beginning 0))
          (goto-char (point-max))))

      (setq arg (1- arg)))
```

我们能够立即看出，这是一个使用减量计数器的 while 循环，它使用表达式(setq arg (1- arg))作递减计数操作。循环体由三个表达式组成：

```
;; between paragraphs
(beginning-of-line)
(while
    body-of-while)

;; within paragraphs, with fill prefix
(if true-or-false-test
    then-part

;; within paragraphs, no fill prefix
  else-part
```

当 Emacs Lisp 解释器对 while循环体求值时，第一件事情就是计算表达式(beginning-of-line)的值，并将位点移动到这一行的开始。随后，函数中有一个内层的 while 循环。这个 while 循环是为将光标移出段落之间的空白处而设计的，如果光标碰巧正好在段落之间的空白处，就需要用到这个 while循环。最后，有一个 if 表达式，这个表达式完成真正将位点移动到段落末尾的操作。

3. 在段落之间的情况

首先，看看内层的 while 循环。这个循环处理位点处于段落之间的情况。它使用三个新函数：prog1、eobp 和 looking-at。

- prog1 函数与 progn 函数类似。不同之处在于 prog1 函数依次对它的参量求值并将其第一个参量的值作为整个表达式的值返回（progn 函数将它最后一个参量的值作为这个表达式的值返回）。prog1 的第二个和第三个参量只是作为附带效果被求值的。
- eobp 是“End Of Buffer P”（缓冲区的末尾）的缩写。当位点处于这个缓冲区末尾时，这个函数返回“真”。
- 当紧跟在位点之后的文本与传递给 looking-at 函数作为其参量的正则表达式匹配时，looking-at 函数返回“真”。

现在学习的这个 while 循环看起来是这样的：

```
(while (prog1 (and (not (eobp))
                   (looking-at paragraph-separate))
              (forward-line 1)))
```

这是一个没有循环体的 while 循环。循环的真假测试就是待计算的表达式：

```
(prog1 (and (not (eobp))
            (looking-at paragraph-separate))
       (forward-line 1)))
```

prog1函数的第一个参量是 and 表达式。在其内部有一个关于位点是否在缓冲区末尾的测试表达式，以及关于位点后的模式是否与分隔段落的正则表达式匹配的测试。

如果光标不在缓冲区的末尾，并且如果光标后面的字符表示两个分离的段落，则and 表达式为“真”。对 and 表达式求值之后，Lisp 解释器对 prog1 表达式的第二个参量 forward-line 求值。这个表达式使位点向前移动一行。然而，prog1 函数的返回值是其第一个参量的值，因此只要位点不在缓冲区的末尾而在两个段落之间时，while 循环将继续执行下去。最后，当位点被移动到一个段落时，and 表达式的值为“假”。然而要注意，forward-line 命令仍然要执行，这意味着，当位点从两个段落之间移动到一个段落当中时，它处于这个段落第二行的开头位置。

4. 在段落当中的情况

外层 while 循环的下一个表达式就是 if 表达式。当 fill-prefix-regexp 变量是一个非空值时，Lisp 解释器执行 if 表达式的 then 部。当fill-prefix-regexp 变量的值是 nil 时（也就是没有填充前缀时），Lisp 解释器执行 if 表达式的 else 部。

5. 没有填充前缀的情况

在没有填充前缀的情况下，代码是最简单的。这部分代码也由另一个内层 if 表达式组成，就像下面这样：

```
(if (re-search-forward paragraph-start nil t)
    (goto-char (match-beginning 0))
  (goto-char (point-max)))
```

这个表达式完成的工作实际上是 forward-paragraph 命令应当完成的主要工作：正则表达式查询，这个查询一直持续到下一段落的开始处。如果查询成功，则将位点移动到那里。但是，如果没有找到下一个段落的开始位置，它将位点移动到缓冲区中可访问区域的末尾。

这部分代码中你唯一不熟悉的部分就是 match-beginnning 的使用。这是另外一个新函数。match-beginning 函数最终返回一个数，这个数指定与最后一个正则表达式查询匹配的文本的开始处的位置。

因为这是一个具有代表性的查询过程，因此在这里使用了 match-beginning 函数。一个成功的朝前查询，不论它是一个普通的查询还是一个正则表达式查询，都将位点移动到找到的文本末尾。在这个例子中，一个成功的查询，会将位点移动到 paragraph-start 正则表达式模式末尾。这是下一段落的开始而不是当前段落的结束。

然而，我们的目的是要将位点置于当前段落的末尾，而不是将位点置子下一段落的开始。这两个位置是不同的，因为可能有几个空行在段落之间。

当给 match-beginning 函数传递参量 0 时，这个函数返回最近与正则表达式匹配的下一个段落的开始位置。在这个例子中，最近的正则表达式查询就是查询paragraph-start，因此 match-beginning 函数返回这个正则表达式模式的开始位置，而不是这个模式的末尾位置。开始的位置就是段落的末尾。

（顺便提一下，当给 match-beginning 传递一个正数作为参量时，这个函数将使位点置于

最后一个正则表达式中带括号的表达式处。这是一个很有用的函数。）

6. 有填充前缀的情况

刚才讨论的内层 if 表达式是外层 if 表达式的 else 部。这个 if 表达式判断是否存在填充前缀。如果有填充前缀，这个if表达式的 then 部被求值。就像下面这样：

```
(while (and (not (eobp))
            (not (looking-at paragraph-separate))
            (looking-at fill-prefix-regexp))
  (forward-line 1))
```

只要下面三种条件都为真，这个表达式就将位点向前一行一行地移动：

1) 位点不在缓冲区的末尾。

2) 位点后面的文本不分隔段落。

3) 位点之后的模式是有填充前缀的正则表达式。

除非你记得在 forward-paragraph 函数中位点被移动到一行的开始位置，否则最后这个条件可能使人疑惑。这意味着如果文本有填充前缀，looking-at 函数就将发现它。

7. 小结

总的来说，当朝前移动时，forward-paragraph 函数完成下面的工作：

- 将位点移动到一行的开始位置。
- 跳过段落之间的空行。
- 检查是否有填充前缀，如果有：

  ——只要不是分隔段落的空行，就要一行一行地向前移动。
- 但是，如果没有填充前缀：

  ——查询下一个段落开始的模式。

  ——移动到下一个段落开始模式处，这将是前一个段落的末尾。

  ——否则移动到缓冲区中可访问区域的末尾。

为复习方便，下面列出的是刚才讨论过的代码。为清楚起见，用缩进的方式将它们排列起来：

```
(interactive "p")
(or arg (setq arg 1))
(let* (
       (fill-prefix-regexp
        (and fill-prefix (not (equal fill-prefix ""))
             (not paragraph-ignore-fill-prefix)
             (regexp-quote fill-prefix)))

       (paragraph-separate
        (if fill-prefix-regexp
            (concat paragraph-separate
                    "\\|^"
                    fill-prefix-regexp
                    "[ \t]*$")
```

```
          paragraph-separate)))
  backward-moving-code (omitted) ...

  (while (> arg 0)                    ; forward-moving-code
    (beginning-of-line)

    (while (prog1 (and (not (eobp))
                       (looking-at paragraph-separate))
                  (forward-line 1)))
  (if fill-prefix-regexp
      (while (and (not (eobp))  ; then-part
                  (not (looking-at paragraph-separate))
                  (looking-at fill-prefix-regexp))
        (forward-line 1))
                                ; else-part: the inner-if
    (if (re-search-forward paragraph-start nil t)
        (goto-char (match-beginning 0))
      (goto-char (point-max))))

  (setq arg (1- arg)))))          ; decrementer
```

forward-paragraph 函数的完整定义，不仅包含这些朝前移动的代码，而且包含朝后移动的代码。

如果你在 GNU Emacs 中阅读这份文档，并且你想要看一看 forward-paragraph 函数的完整代码，可以键入 M-.(find-tag) 并在提示符下输入函数名来查看。如果find-tag函数首先问你“TAGS”表的文件名，就输入你的“emacs/src”目录中“TAGS”文件的文件名，这是一个类似“/usr/local/lib/emacs/19.23/src/TAGS”的路径名。(“emacs/src”目录的确切路径由你的Emacs拷贝安装的方式和位置决定。如果不知道路径，有时能用 C-h I 命令进入 Info，然后键入 C-x x-f 来查看“emacs/info”的实际路径。“TAGS”文件的路径对应着“emacs/src”的路径。然而，有时 Info 文件存放在其他地方。)

如果尚没有“TAGS”(标签文件)，你也能够创建自己的“TAGS”文件。

12.5 创建自己的“TAGS”文件

你能够创建自己的“TAGS”文件来帮助你访问源代码。例如，如果你有大量的文件在你的“~/emacs”目录中(就像我这样，我有 127个 .el 文件，而我希望载入17个)，你将发现如果创建了自己的“TAGS”文件，即使你使用 grep 命令或者别的命令来查找特定的函数，也能很容易地找到特定的函数。

你能够通过调用 etags 程序来创建自己的“TAGS”文件。这个程序是作为 Emacs 发行版本的一部分发行的。通常，etags 程序在 Emacs 安装时就会被编译和安装。(etags 不是 Emacs Lisp 的一个函数或者 Emacs 的一部分，它是一个 C 语言函数。)

要创建一个“TAGS”文件，首先要切换到需要创建这个“TAGS”文件的目录。在 Emacs 中，可以用 M-x cd 命令来切换到指定的目录，或者通过访问该目录下的一个文件，或者通过

用C-x d（dired）命令列出这个目录下的文件，来达到切换目录的目的。然后输入：

```
M-! etags *.el
```

来创建一个“TAGS”文件。etags 程序接收所有常用的 shell 的通配符。例如，如果需要为两个不同的目录创建一个共同的“TAGS”文件，输入以下这样一个命令就可以了，其中“.../elisp/”是第二个目录：

```
M-! etags  *.el ../elisp/*.el
```

输入

```
M-! etags --help
```

则可以列出 etags 程序能够接受的所有选项的列表。

这个 etags 程序能处理 Emacs Lisp、Common Lisp、Scheme、C、Fortran，Pascal、LaTeX以及大部分汇编语言。这个程序对不同的语言没有什么特殊的开关选项，它根据文件名以及文件的内容来识别文件所属的语言类型。

同样，当你自己编写函数代码并且希望参考自己已经编写好的函数时，etags程序就很有用。只要你在编写新的函数时不时地运行 etags 程序，就可以将这些新编写的函数变成“TAGS”文件的一部分。

12.6 回顾

以下是最近介绍的函数的简要总结。

• while

只要表达式主体的第一个元素的测试为“真”，这个表达式的主体就被不断地重复求值。最后返回 nil。（其中的表达式只是作为它的附带效果而被求值的。）

例如：

```
(let ((foo 2))
  (while (> foo 0)
    (insert (format "foo is %d.\n" foo))
    (setq foo (1- foo))))

     ⇒       foo is 2.
             foo is 1.
             nil
```

(insert 函数的工作就是在位点处插入它的参量。而 format 函数的作用就是以其参量的格式返回一个字符串，就像 message 函数一样，\n 产生一个新行。)

• re-search-forward

查询一种模式，并且如果找到这种模式，就将位点移动到那个位置。

同 search-forward 函数一样，这个函数也接受 四个参量：

1) 一个指定要查找的模式的正则表达式。

2) 可选的参量，即查询限制范围。

3) 可选参量，如果查询失败，返回 nil 值或者产生错误消息。

4) 可选参量，重复查询的次数；如果这个参量的值为负，表示查询朝后进行。

• let*

将局部变量绑定到指定的值上，然后对剩余的变量求值，它的返回值是最后一个参量的值。在绑定局部变量时，如果有的话就使用已经绑定的局部变量的值。

例如:

```
(let* ((foo 7)
       (bar (* 3 foo)))
  (message "'bar' is %d." bar))
    ⇒ 'bar' is 21.
```

• match-beginning

返回由最后一个正则表达式查询所找到的文本的开始位置。

• looking-at

如果位点后的文本与正则表达式匹配，就返回“真”(t)。

• eobp

如果位点是缓冲区的可访问区域的末尾，就返回“真”(t)。如果变窄没有开启，缓冲区中可访问区域的末尾就是缓冲区的末尾；如果变窄开启，它就是变窄部分的末尾。

• prog1

依次对其每一个参量求值，然后返回第一个参量的值。

例如:

```
(prog1 1 2 3 4)
     ⇒ 1
```

12.7 练习：使用 re-search-forward

- 编写一个函数，这个函数通过一个正则表达式来查询两个或者更多的连续空行。
- 编写一个函数，用来查询重复的单词，如“the-the”。关于如何编写一个正则表达式来匹配由两个相同部分组成的字符串，可以参见《*GNU Emacs* 手册》中的“正则表达式句法”一节。你能为此设计出几个正则表达式，一些比另一些更好。我使用的这个函数以及几个正则表达式附在附录A“the-the 重复单词函数”中。

第13章 计数：重复和正则表达式

重复和正则表达式查询是你在 Emacs Lisp 中编写代码时经常使用的功能强大的工具。这一章介绍在用 while 循环和递归方法构造的单词计数命令中正则表达式查询的使用情况。

在 Emacs 的标准发行版本中，包含了对一个区域内所有行进行计数的函数。然而，却没有对其中的单词进行计数的相应函数。

某些写作需要对单词进行计数。因而，如果你撰写一篇随笔，篇幅可以限制在800 字之内；如果你撰写一篇小说，可以规定自己一天撰写 1000字。Emacs 竟然缺少一个单词计数命令，这令我很奇怪。也许，人们使用 Emacs 的目的主要是编写代码或者那些不需要进行单词计数的文档，或者也许他们只是使用操作系统的单词计数命令。这个命令将文档中的字符数除以 5 得到单词数。无论如何，这里有一些单词计数命令。

13.1 count-words-region 函数

一个单词计数命令可以对一行、一个段落、一个区域或者一个缓冲区进行计数。这个命令应当包含哪些内容呢？你应当设计一个命令来对整个缓冲区中的单词进行计数。然而，Emacs 的传统鼓励程序富有弹性——即可以只对一节中的单词计数，而不是对一个缓冲区中的单词计数。因此，编写一个对一个区域计数的命令更有意义。一旦有一个 count-words-region 命令，如果愿意的话，也能够用 C-x h(mark-whole-buffer) 键序列对整个缓冲区中的单词计数。

很清楚，对单词计数是一个重复的动作：从这个区域的头部开始，对第一个单词开始计数，然后是第二个、第三个……等等，直到这个区域的末尾。这意味着单词计数使用递归的方法或者 while 循环是很理想的。

首先，将用 while 循环实现这个单词计数命令，然后用递归的方法实现这个命令。当然，这个命令将是交互的。

一个交互函数定义的模板总是这样的：

```
(defun name-of-function (argument-list)
  "documentation..."
  (interactive-expression...)
  body...)
```

现在需要做的就是往其中填入适当的内容。

函数名应当直接表明自己的用途而无需另加说明，并且应当与已有的 count-lines-region 函数名相似。这可以使函数名易于记住。因此 count-words-region 是单词计数函数名的一个好选择。

这个函数对一个区域中的单词计数。这意味着参量列表一定要包含绑定到两个位置的符号，

这两个位置就是该区域的开始位置和结束位置。这两个位置可以分别被叫做“beginning”和“end”。文档的第一行应当是一个完整的句子，因为像 apropos 这样的命令将只打印函数说明文档的第一行。交互表达式的形式将是“(interactive "r")”，因为这将使 Emacs 将计数区域的开始位置和结束位置作为参量传递给这个函数。所有这些都是按部就班的。

需要编写函数体以使其完成这样三件任务：首先建立 while 循环的条件，其次是 while 循环，最后是发送一个消息给用户。

当用户调用 count-words-region 函数时，位点可能是在指定区域的开始，也可能是在末尾。然而，计数过程必须从区域的开始位置开始。这意味着，如果位点原来并不在指定区域的开始处，就要使位点移动到那里。执行（goto-char beginning）可以达到这个目的。当然，当函数执行完之后，要将位点返回到调用这个函数时的位置。为了这个原因，函数体必须包含在一个 save-excursion 表达式中。

函数体的中心部分由一个 while 循环组成，在这个循环中，一个表达式使位点一个单词一个单词地往前移动，另外一个表达式对移动的次数计数。只要位点还要继续往前移动，while循环的真假测试结果应当为“真”，当位点到达指定区域的末尾时，真假测试结果为“假”。

将使用 (forward-word 1) 作为表示向前一个单词一个单词地移动位点的表达式，但是如果使用一个正则表达式查询，就更容易看到 Emacs 是如何区分一个单词的。

正则表达式查询不仅查找一个模式，而且在找到之后将位点移动到其后。这意味着，一系列成功的正则表达式查询将使位点一个单词一个单词地向前移动。

实际的问题是，我们想要正则表达式查询直接跳过单词之间的空格和标点符号，就像跳过单词本身一样。一个拒绝跳过单词之间空格的正则表达式永远不会跳过多于一个的单词。这就是说，如果有空格和标点符号，正则表达式应当包含单词之后的这些空格和标点符号，把它们当做单词本身一样处理。（一个可能的情况是，一个单词可能结束一个缓冲区，而没有任何空格或者标点符号在其后，因此正则表达式的这个部分是可选的。）

因而，我们需要的这个正则表达式是一个模式，它定义一个或多个单词构词要素以及其后（可选的）的一个或多个不是单词构词要素的其他字符。这个正则表达式就是：

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

缓冲区语法表决定了哪些字符是单词的构词要素而那些字符不是单词的构词要素。（有关语法的更多内容参见本书的14.2节，“单词或者符号是由什么构成的？”。同时，还可参见《*GNU Emacs手册*》的“语法表”一节以及《*GNU Emacs Lisp技术手册*》的“语法表”一节）。

查询表达式是这样：

```
(re-search-forward "\\w+\\W*")
```

（注意在“w”和“W”之前的成对的反斜线。对 Emacs Lisp 解释器来说，单个反斜线是有特殊意义的。它指后续的字符以不同于平常的方式被解释。例如，字符 “\n” 代表“newline”(即换行)，而不是一个反斜线和一个字母“n”。一行中两个连续的反斜线则代表一个通常的没有特殊意义的反斜线。）

需要一个计数器来对单词个数进行计数。这个变量必须首先设置为 0，然后Emacs 每循环一次，这个计数器就加 1。这个递增表达式很简单：

```
(setq count (1+ count))
```

最后，想要告诉用户在这个区域中有多少单词。message 函数就适合于用来向用户发出这种消息。这个消息必须是一个短语的形式，这样不管其中究竟有多少单词，它读起来总是正确的。我们不要这样的句子 "there are 1 words in the region"（有英文语法错误）。单数和复数之间的冲突是不符合英文语法规则的。用一个条件表达式根据这个区域中单词数目的不同给英文出不同的消息就能够解决这个问题。这里有三种可能：没有单词，一个单词，多于一个单词。这意味着在这里使用 cond 特殊表很合适。

所有这些综合起来就是下面这个函数定义：

```
;;; First version; has bugs!
(defun count-words-region (beginning end)
  "Print number of words in the region.
Words are defined as at least one word-constituent
character followed by at least one character that
is not a word-constituent.  The buffer's syntax
table determines which characters these are."
  (interactive "r")
  (message "Counting words in region ... ")

;;; 1. Set up appropriate conditions.
  (save-excursion
    (goto-char beginning)
    (let ((count 0))
;;; 2. Run the while loop.
      (while (< (point) end)
        (re-search-forward "\\w+\\W*")
        (setq count (1+ count)))

;;; 3. Send a message to the user.
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

就像写着的那样，这个函数可以正常工作，但是它并没有涵盖所有的情况。

count-words-region 中的空格bug

前一节描述的 count-words-region 函数命令中有两个bug，或一个bug的两种表现。第一，如果你标记的那个区域只有在某些文本当中包含空格，那个函数就会告诉你这个区域只包含一个单词。第二，如果你标记的区域只有在缓冲区的末尾或者在变窄的缓冲区的可以访问区域的末尾处包含一些空格，这个命令会显示一条这样的错误消息：

```
Search failed: "\\w+\\W*"
```

如果你是在 GNU Emacs 的 Info 中阅读这份文档，你可以自己测试这些 bug。

首先，以通常方式对上面那个函数定义求值以安装它。

如果愿意的话，你也可以通过对下面这个表达式求值来安装这个绑定键：

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

为了测试第一个 bug，在下面这一行的开始和末尾设置标记和位点，然后键入 C-c= (如果你没有绑定 C-c=，就使用 M-x count-words-region 命令)：

```
one   two  three
```

Emacs将正确地告诉你这个区域有三个单词。

重复这个测试，这次将标记置于这一行的开始，而将位点紧紧置于单词“one” 之前。再一次键入C-c= (或者 M-x count-words-region)。这时 Emacs 应当告诉你这个区域没有任何单词，因为它仅仅是由这一行的开始部分的空格组成的。但是，Emacs 会告诉你这里有一个单词！

再来进行第三次测试，将这一行拷贝到“*scratch*”缓冲区的末尾，然后在这一行的末尾输入几个空格。将标记置于单词“three”之后，并将位点置于这一行的末尾（也就是缓冲区的末尾）。再像前面那样键入 C-c= (或者 M-x count-words-region)，这时 Emacs 也应当告诉你这个区域没有任何单词，因为它只有这一行末尾的几个空格而已。但是，Emacs 显示一个错误消息：“Search failed”。

这两个bug源自同样一个问题。

首先，考虑这个 bug 的第一个表现。在这里，命令告诉你位于一行开始处的空格包含了一个单词。这是因为，M-x count-words-region 命令将位点移动到了区域的开始处。while 循环测试位点的这个值是否小于 end 的值，这个测试为“真”。相应地，正则表达式查找并找到第一个单词。Emacs 将位点置于这个单词之后，count 变量被设置为1。while 循环继续循环，但是这一次位点的值大于 end变量的值，退出循环。简单地说，正则表达式查询查找并找到一个被标记的区域之外的单词。

在bug的第二种表现中，指定的区域是缓冲区末尾的空格。Emacs 发出 “Search failed”的消息。这是因为 while 循环的真假测试值为“真”，因此查询表达式被求值，但是由于这个区域没有单词，因此查询失败。

在这个 bug 的两种表现当中，前一种是查询范围扩大了，后一种是试图到指定区域之外查询。

解决办法是限制查询的区域——这个动作相当简单，但是就像你将要看到的，它的实现方式实际上并非你想象的那么简单。

就像我们已经看到的，re-search-forward 函数接受一个查询模式作为它的第一个参量。但是除了这个强制性的参量之外，它还接受三个可选参量。可选的第二个参量绑定这个查询。可选的第三个参量，如果是 t，就使函数在查询失败时返回 nil 而不是发出一个错误消息。可选的第四个参量是一个重复计数器。（在 Emacs 中，你能够用 C-h f 加上函数名和回车符(RET键)得到一个函数的说明文档。）

在 count-word-region 函数定义中，指定区域的末尾的值是由变量 end 存放的，这个值作为一个参量传递给函数。因而，能够将 end 作为一个参量加入到正则表达式的查询表达式中：

```
(re-search-forward "\\w+\\W*" end)
```

然而，如果你仅在 count-words-region 的函数定义中做了这样的改变，并在一组空格上继续测试这个新版函数定义，你将得到一个错误消息声明查询失败："Search failed"。

出现这样问题的原因在于：查询被限制在这个区域中，而这其中没有组成单词的字符。因为查询失败，我们得到一个错误消息。但是我们不希望在这种情况下得到这样的一个错误消息。我们希望得到的消息是："The region does NOT have any words"。

这个问题的解决方法是为 re-search-forward 提供第三个参量 t，这使得函数当查询失败时返回 nil，而不是一个错误消息。

然而，如果你做出这样的改动并测试它，你将看到信息 " Counting words in region..."，并将一直看到这个消息，直到键入 C-g(keyboard-quit)为止。

这是因为，就像前面一样，查询被限制在指定区域，由于这个区域没有单词构词要素字符，因此查询就像希望的那样失败了。相应地，re-search-forward表达式返回 nil，什么也没有做。特别是，它没有移动位点，这原本是作为找到查询目标的一个附带效果实现的。在 re-search-forward表达式返回nil之后，while循环中的下一个表达式被求值，这个表达式使计数器递增1。然后循环继续下去。while循环中的真假测试结果一直为"真"，因为位点的值一直小于end的值（因为位点没有移动），循环一直进行下去。这就是你所看到的结果。

count-words-region函数需要另外一种改进，使 while 循环中的真假测试在查询失败时测试值为假。这时可以采用另外的方法，即要同时满足两种条件再对计数器做递增计算：位点必须还在指定区域中，并且查询表达式必须找到要计数的一个单词。

因为第一种和第二种条件必须同时为"真"，所以这两个表达式——区域测试表达式和查询表达式——能够用一个 and 函数结合成单独一个表达式，并嵌入到 while循环作为其真假测试表达式，就像这样：

```
(and (< (point) end) (re-search-forward "\\w+\\W*" end t))
```

(关于 and 函数的有关信息，参见12.4节"forward-paragraph:函数的金矿"。)

如果查询成功，re-search-forward 表达式返回t，作为其一个附带效果，位点也移动到相应的位置。因此，在找到单词的同时，位点从指定区域中一一移过。当查询表达式没能找到下一个单词，或者位点已经是指定区域的末尾时，真假测试结果为"假"，随即退出 while 循环，count-words-region 函数显示其消息。

做了这些最终的修改之后，count-words-region 函数就没有 bug 了（或者至少我没有发现它的 bug 了）。下面就是这个函数定义的完整代码：

```
;;; Final version: while
(defun count-words-region (beginning end)
  "Print number of words in the region."
  (interactive "r")
```

```
  (message "Counting words in region ... ")

;;; 1. Set up appropriate conditions.
  (save-excursion
    (let ((count 0))
      (goto-char beginning)

;;; 2. Run the while loop.
      (while (and (< (point) end)
                  (re-search-forward "\\w+\\W*" end t))
        (setq count (1+ count)))

;;; 3. Send a message to the user.
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

13.2 用递归的方法实现单词计数

与前面用 while 循环的函数类似，可以用递归的方法编写对单词计数的函数。下面，来看一看这是如何实现的。

首先，需要认识到 count-words-region 函数有三个任务：建立适当的计数条件，在指定的区域对单词计数，向用户发送消息告诉他们指定区域内有多少单词。

如果编写单独一个递归函数来完成所有的事情，那么将在每一次递归调用时都得到一个消息。如果这个区域有 13 个单词，就将得到 13 条消息，一个接着一个。实际上，我们不需要这样。相反，我们必须编写两个函数来完成这个工作，其中一个函数（递归函数）将在另外一个函数内部使用。一个函数将建立测试条件以及显示消息，另一个函数将返回单词计数。

让我们先从显示消息的函数入手。同时将继续称这个函数为 count-words-region。

这是用户将要调用的一个交互函数。确实，除了它将调用 recursive-count-words 函数来确定指定区域中有多少单词之外，它与这个函数的前面一个版本非常相似。

基于这个函数前面的版本，能够容易地构造出这个函数的模板：

```
;; Recursive version; uses regular expression search
(defun count-words-region (beginning end)
  "documentation..."
  (interactive-expression...)

;;; 1. Set up appropriate conditions.
  (explanatory message)
  (set-up functions...
```

```
;;; 2. Count the words.
    recursive call

;;; 3. Send a message to the user.
    message providing word count))
```

除了由递归调用返回的计数必须被传递给显示单词计数的消息这一点之外，这个定义看起来很直接。稍微思考一下就可以知道，这可以用 let 表达式实现：能够将 let 表达式的变量列表中的一个变量绑定到指定区域的单词数上，这个数值是由递归调用返回的；然后 cond 表达式就可以根据绑定的值向用户显示特定的消息。

人们常常认为 let 表达式中的这种绑定作用是这个函数的“第二位”的工作。但是，在这个例子中，你所认为的函数的“第一位”的工作——对单词计数，就是在 let 表达式中完成的。

使用 let 表达式，这个函数定义如下所示：

```
(defun count-words-region (beginning end)
  "Print number of words in the region."
  (interactive "r")

;;; 1. Set up appropriate conditions.
  (message "Counting words in region ... ")
  (save-excursion
    (goto-char beginning)

;;; 2. Count the words.
    (let ((count (recursive-count-words end)))

;;; 3. Send a message to the user.
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

下面，需要编写递归计数函数。

一个递归函数至少有三个部分：一个测试表达式、一个 next-step 表达式和递归调用。

测试表达式决定函数是否要再调用。因为是在一个指定区域中对单词计数，并且可以使用一个单词接一个单词地朝前移动位点的函数，所以测试表达式能够检查位点是否还在指定区域中。测试表达式应当找到位点的值并决定位点是在指定区域的末尾之前、正在末尾或者在末尾之后。能够用 point 函数来确定位点的值。很清楚，必须将指定区域的末尾的值作为一个参量传递给递归计数函数。

除此之外，测试表达式应当也能测试是否找到一个单词。如果没有找到单词，这个函数就不应当再调用它本身了。

next-step 表达式要改变一个值，以值当递归函数应当停止调用它本身时就可以停下来。更准确地说，next-step 表达式改变一个值以使之能在正确的时候使测试表达式让递归表达式停止调用函数本身。在这个例子中，next-step 表达式可以是一个单词接一个单词地朝前移动位点的表达式。

递归函数的第三部分就是递归调用。

在某处，我们也需要一个真正完成函数“工作”的那部分代码，也就是完成计数的代码。一个真正有用的部分！

但是，我们已经有了一个递归计数函数的骨架：

现在需要填满充实这个骨架。让我们首先从最简单的情况开始：如果位点在指定区域的末

```
(defun recursive-count-words (region-end)
  "documentation..."
   do-again-test
   next-step-expression
   recursive call)
```

尾或者超出了指定区域的末尾，就没有任何单词了。因此这时函数应当返回零。同样，如果查询失败，也没有任何单词计数，因此函数也应当返回零。

另一个方面，如果位点在指定的区域之内，查询也是成功的，函数应当继续调用本身。因而，测试表达式应当是：

```
(and (< (point) region-end)
     (re-search-forward "\\w+\\W*" region-end t))
```

注意，查询表达式是测试表达式的一部分——如果查询成功，函数返回t；如果查询失败，则返回 nil。（有关re-search-forward函数如何工作的解释，参见13.1节中的“count-words-region中的空格bug”小节。）

这个测试表达式是 if 函数的一个真假测试表达式。很清楚，如果测试表达式为“真”，if 从句的 then 部应当再一次调用函数；但是如果它为“假”，if 从句的 else 部应当返回零，因为不管位点是在指定区域之外或者查询失败，总之是没有别的单词了。

但是在考虑递归调用之前，需要先考虑 next-step 表达式。它是什么？有趣的是，它是测试表达式的查询部分。

除了为测试表达式返回 t或者 nil 之外，re-search-forward 函数在查询成功时还将位点向前移动，这是这个函数的一个附带效果。这个动作改变了位点的值，以使递归函数在位点完全移动到指定区域之外时，停止调用自身。因而，这个 re-search-forward 表达式就是 next-step 表达式。

从结构骨架上说，recursive-count-words 函数体如下所示：

```
(if do-again-test-and-next-step-combined
    ;; then
    recursive-call-returning-count
  ;; else
  return-zero)
```

那么它如何与计数的部分协作呢？

如果你不习惯编写递归函数，类似这样的问题可能会困扰你。但是它能够、也应当被系统

地解决。

我们知道，计数的方法应当通过某种途径与递归调用结合起来。确实，因为next-step 表达式将位点朝前移动一个单词，并且因为一个递归调用是对每一个单词都进行的，所以计数的方法必须是一个表达式，它将由recursive-count-words 调用返回的值增1。

考虑下面几种情况：

- 如果在指定区域中有两个单词，当函数对第一个单词计数时，函数应当返回的值是1 加上这个函数对区域中剩余的单词计数的结果（在这种情况下就是1）。
- 如果指定区域中只有一个单词，当函数对那个单词计数时，函数应当返回的值是 1 加上这个函数对区域中剩余的单词计数的结果（在这种情况下就是 0）。
- 如果指定区域中没有单词，则函数应当返回零。

从这个草案中，我们可以看到：在没有单词的情况下if 表达式的 else 部将返回零。这就是说，if 表达式的 then 部必须返回一个值，这个值是这个函数对剩余的单词计数的结果加1得到的。

这个表达式写在下面，其中 1+ 是一个对其参量加 1 的函数。

```
(1+ (recursive-count-words region-end))
```

而其中的 recursive-count-words 函数如下所示：

```
(defun recursive-count-words (region-end)
  "documentation..."

;;; 1. do-again-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))

;;; 2. then-part: the recursive call
      (1+ (recursive-count-words region-end))

;;; 3. else-part
    0))
```

让我们检查一下这个函数是如何工作的：

如果在指定的区域内没有单词，if 表达式的 else 部被求值，从而这个函数返回零。

如果指定区域中有一个单词，位点的值小于 region-end 变量的值，查询也是成功的。在这种情况下，if 表达式的测试结果为“真”，因此 if 表达式的 then 部被求值，也就是计数表达式被求值。这个表达式返回一个值（这也将是整个函数的返回值），这个值是 1 与递归函数调用返回的值之和。

同时，next-step 表达式已经使位点跳过第一个单词（并且在这种情况下也只有一个单词）。这意味着，当 (recursive-count-words region-end) 表达式第二次被求值时，作为递归调用的返回值，这时位点的值将等于或者大于region-end 变量的值。因此，这时 recursive-count-words 函数将返回零。零将被增大到1，并且前一个递归调用 recursive-count-words函数将返回 1加0，也就是 1，这就是正确的答案。

很明显，如果在指定区域中有两个单词，第一次调用recursive-count-words函数返回 1，加上对剩余区域作递归调用的 recursive-count-words 函数所返回的值——也就是 1加

1，即 2，这就是正确的答案。

类似地，如果在指定区域中有3个单词，recursive-count-words 函数的第一次调用返回 1 加上对剩余的两个单词的区域作递归调用的recursive-count-words 函数所返回的值——如此重复下去。

在此基础上加上完整的函数文档，这两个函数代码就完整地列于下面：

```
(defun recursive-count-words (region-end)
  "Number of words between point and REGION-END."
;;; 1. do-again-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))
;;; 2. then-part: the recursive call
      (1+ (recursive-count-words region-end))

;;; 3. else-part
    0))
```

调用这个递归函数的函数是：

```
;;; Recursive version
(defun count-words-region (beginning end)
  "Print number of words in the region.
Words are defined as at least one word-constituent
character followed by at least one character that is
not a word-constituent.  The buffer's syntax table
determines which characters these are."
  (interactive "r")
  (message "Counting words in region ... ")
  (save-excursion
    (goto-char beginning)
    (let ((count (recursive-count-words end)))
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

13.3 练习：统计标点符号的数量

用 while 循环编写一个函数，它对一个指定区域中的标点符号进行计数。标点符号包括句点、逗号、分号、冒号、感叹号、问号。另外，再用递归的方法编写一个这样的函数。

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。

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。

第16章 配置你的“.emacs”文件

“不要只为了喜欢 Emacs 而喜欢它”——这句看似悖论的话其实就是 GNU Emacs 的秘密所在。通常的 Emacs 是一个通用的工具。大多数使用它的人都根据自己的要求来定制适合自己的 Emacs。

GNU Emacs 的大部分是用 Emacs Lisp 编写的，这意味着通过在Emacs Lisp中编写表达式，你能够改变或者扩充 Emacs。

也有些人喜欢 Emacs 的默认配置。毕竟，当你编辑一个 C 语言文件时，Emacs就进入 C 语言模式；当你编辑一个 Fortran 语言文件时，Emacs 就进入 Fortran 模式；而当你编辑一个无格式的文件时，Emacs 就进入基本模式。如果你不知道谁将使用 Emacs，这种适应性很有意义。有谁知道哪种用户希望用 Emacs 编辑无格式的文件呢？Emacs 的基本模式就是针对编辑这种文件的，就像 C 模式是默认针对 C 语言文件的一样。但是当你确实知道将使用 Emacs 的人就是你自己时，那么你自己定制 Emacs 环境就有意义了。

例如，当我编写一个没有特定目的普通文件时，我很少希望进入基本模式，我希望进入文本模式。这就是为什么我要定制 Emacs 的原因：使它适合我。

通过编写或者改编“~/.emacs”文件，你就能够定制并扩充 Emacs。这是你个人的初始化文件，它的内容是由 Emacs Lisp 编写的，其作用是告诉 Emacs 做些什么。

这一章描述一个简单的“.emacs”文件，关于这个文件的更多信息参见《*GNU Emacs* 技术手册》以及《*GNU Emacs Lisp* 技术手册》的有关章节。

16.1 全站点的初始化文件

除了你个人的初始化文件之外，如果存在全站点初始化文件，Emacs 将自动加载各种不同的全站点初始化文件。这些文件的格式与你个人的初始化文件的格式是一样的。但是这些文件供所有人加载。

最常见的情况是，“site-load.el”和“site-init.el”两个全站点初始化文件被加载到 Emacs 中，然后如果创建了一个转储版本，随后就转储这两个文件。(Emacs 的转储版本可以加载得更快，然而，一旦一个文件被加载和转储了，对这个文件的改变就不会改变 Emacs，除非你自己来加载这个文件或者重新转储它。参见《*GNU Emacs Lisp* 技术手册》中“建立 Emacs”一节和“INSTALL”文件。)

其他三个全站点初始化文件在你每一次使用 Emacs 时就会被自动地加载(如果这三个文件存在的话)。这些文件中，“site-start.el”文件在用户个人的初始化文件“.emacs”加载之前被加载，“default.el”以及终端类型文件都是在用户个人的初始化文件“.emacs”加载之后被加载。

用户个人的初始化文件“.emacs”中的设置以及定义将覆盖“site-start.el”文件

（如果它存在的话）中与之有冲突的设置和定义。但是“default.el”以及终端类型文件中的设置和定义将覆盖用户个人的初始化文件中有冲突的设置和定义。（你将终端类型文件中的term-file-prefix项设置成nil，就可以避免终端类型文件的干扰。参见16.10节，“一个简单的功能扩充”。）

Emacs的发行版本中的“INSTALL”文件描述了“site-init.el”和“site-load.el”这两个文件。

“loadup.el”、“startup.el”以及“loaddefs.el”文件控制Emacs初始化文件的加载过程。这些文件都位于Emacs发行版本的“lisp”目录中，都值得好好研究。

“loaddefs.el”文件包含了许多建议，如你个人的初始化文件中应当放些什么内容，或者在一个全站点初始化文件中应当放些什么内容，等等。

16.2 为一项任务设置变量

我的Emacs第19.23版中有392个可选项，可以用edit-options命令来设置它们。这些“可选项”无外乎就是一些变量，与我们前面已经看到的用defvar定义的变量没有什么两样。

Emacs判断一个变量是否是可以设置的，是通过观看这个变量的说明文档字符串中的第一个字符来决定的；如果说明文档字符串中的第一个字符是一个星号“*”，这个变量就是用户可以自行设置的选项。（参见8.4节，“用defvar初始化变量”。）

edit-options命令列出了当人们在编写Emacs Lisp函数库时Emacs中所有可以重新设置的变量。它提供了一个易于使用的界面来重新设置这些变量。

另一方面，使用edit-options命令来设置的可选项只在你的编辑会话中有效。这些新的值并不永久保存供其他会话使用。Emacs每一次启动，它都读入其源代码中的初始定义的变量。要实现在不同会话之间永久地保存一个变化的设置，你需要在“.emacs”或者其他初始化文件中使用setq表达式来设置。

对我来说，edit-options命令的主要用法是用来建议哪些变量应当由我自己在初始化文件“.emacs”中设置。我强烈要求你仔细浏览一遍这个变量列表。

关于这个问题的更多信息，参见《*GNU Emacs技术手册*》中“编辑变量的值”一节。

16.3 开始改变“.emacs”文件

当你启动Emacs时，它加载你个人的初始化文件“.emacs”，除非你在命令行用“-q”参数告诉它不要加载这个文件。（emacs -q命令使你进入普通模式）。

“.emacs”文件包含了Lisp表达式。通常，这无非是一些设置变量的表达式，有时也有函数定义表达式。

关于初始化文件的简短描述，参见《*GNU Emacs技术手册*》中的“初始化文件‘~/.emacs’”一节。

这一章仔细讲解一些共同的东西，但是会对一个完整的、我长期使用的“.emacs”文件作一个通盘介绍。

这个文件的第一个部分由注释组成：用于提醒自己。现在，当然，我记得这些东西，但是

当我刚接触 Emacs 时，并不记得这么多。

```
;;;; Bob's .emacs file
; Robert J. Chassell
; 26 September 1985
```

瞧这个日期！我使用这个文件已经有相当长的历史了。我从那时起不断增加其内容。

```
; Each section in this file is introduced by a
; line beginning with four semicolons; and each
; entry is introduced by a line beginning with
; three semicolons.
```

这一段描述 Emacs Lisp 中注释的通常惯例。每一个以一个分号开始的行都是一个注释行，两个、三个或者四个分号分别是节和小节的标记。（参见《*GNU Emacs Lisp* 技术手册》中的“注释”一节。）

```
;;;; The Help Key
; Control-h is the help key;
; after typing control-h, type a letter to
; indicate the subject about which you want help.
; For an explanation of the help facility,
; type control-h three times in a row.
```

记住，连续键入 C-h 三次就可得到帮助信息。

```
; To find out about any mode, type control-h m
; while in that mode.  For example, to find out
; about mail mode, enter mail mode and then type
; control-h m.
```

这是“Mode help”（模式帮助）提示，就像我自己称呼的这样，是非常有用的。通常，它告诉你所有你需要知道的东西。

当然，你无需在你自己的初始化文件“.emacs”中加入这样的注释。我将这些注释加入到我的初始化文件中是因为我总是忘记模式帮助以及注释的惯例——但是我可以记得来这里看一看让自己得到提醒。

16.4 文本和自动填充模式

现在开始学习开启“Text mode”(文本模式)和“Auto Fill”（自动填充）模式的部分。

```
;;; Text mode and Auto Fill mode
; The next two lines put Emacs into Text mode
; and Auto Fill mode, and are for writers who
; want to start writing prose rather than code.

(setq default-major-mode 'text-mode)
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

这是“.emacs”文件的第一部分，这一部分没有别的功能，无非是为了提醒一个健忘的

人而已。

括号中的这两行告诉 Emacs，当找到一个文件时就开启文本模式，除非那个文件应当进入别的模式，如 C 模式等。

当 Emacs 读入一个文件时，它查看这个文件名的后缀（如果有后缀的话）。（文件名后缀是以“.”开始的那个部分）。如果文件是以“.c”结尾的，或者以“.h”结尾，那么 Emacs 就进入 C 模式。同样，Emacs 查看这个文件的第一个不是空白的行，如果这一行是“-*-C-*-”，那么 Emacs 也进入 C 模式。Emacs 拥有一个后缀列表并自动地使用它。另外，如果缓冲区的最后一页中有一个局部变量列表，Emacs 就查看缓冲区最后一页中的这个局部变量列表，根据它的内容来决定进入何种模式。

有关更多的信息，请参见《*GNU Emacs* 技术手册》中的“如何选择主要的模式”一节和“文件中的局部变量”一节。

现在回到初始化文件“.emacs”中。

后面的一行就列在下面，它是如何工作的呢?

```
(setq default-major-mode 'text-mode)
```

这一行很短，但是它确实是一个 Emacs Lisp 表达式。

我们已经熟悉了setq 函数，它将后续的变量 default-major-mode 设置为text-mode。在 text-mode 之前的单引号是告诉 Emacs 直接处理 text-mode 变量，而不是其内容。参见 1.9节“给一个变量赋值”可以得到关于 setq的资料。值得重提的主要的一点是，在“emacs”文件中设置一个变量的过程与在 Emacs 其他地方设置变量的过程没有任何区别。

第二行是：

```
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

在这一行，add-hook 命令将 turn-on-auto-fill 加到变量 text-mode-hook之后。而 turn-on-auto-fill 是一个程序的名字，它启动自动填充模式。

Emacs 每次启动文本模式，它都运行这些命令改变文本模式的某些属性。因此，Emacs 每次启动文本模式，Emacs 同时也启动了自动填充模式。

简而言之，当你编辑一个文件时，上面的第一行使 Emacs 进入文本模式，除非这个文件的后缀名、文件的第一个非空的行或者局部变量告诉 Emacs 进入其他模式。

其他动作当中的文本模式为用户设置语法表以方便用户。在文本模式中，Emacs 认为省略号是单词的一个部分就像构词要素字符一样，但是 Emacs 并不认为句点或者一个空格是单词的一个部分。因而，M-f 使你移过“it's”。另一方面，在 C模式下，M-f 则停在“it's”的“t”字母之后。

第二行使 Emacs 在开启文本模式的时候，同时开启自动填充模式。在自动填充模式中，Emacs 自动地分行使超出屏幕的过多的文本自动转移到下一行。Emacs在单词之间分行，而不是在单词中分行。

当自动填充模式被关闭时，如果你不断输入字符，它们就不断地接在这行的后面，而不换行。根据你设置的 truncate-lines 的值，你输入的单词要么显示在你的屏幕之外，要么被显示出来，当然这些单词是以相当难看和难以阅读的格式在一个连续行显示的。

16.5 邮件别名

下面是一个“开启”邮件别名的 setq 表达式，当然还包括不少提示信息：

```
;;; Mail mode
; To enter mail mode, type `C-x m'
; To enter RMAIL (for reading mail),
; type `M-x rmail'

(setq mail-aliases t)
```

这个 setq 表达式设置变量 mail-aliases的值为 t。因为t就是指“真”，这一行意思就是说，“对，使用邮件别名。”

使用邮件别名可以很方便地用简短的名字代替很长的邮件地址或者邮件地址列表。存放你的邮件别名的文件是“~/.mailrc”。使用别名的方法如下所示：

```
alias geo george@foobar.wiz.edu
```

当你给 George 一个消息时，只要指出地址 “geo” 即可；邮件程序会自动将 “geo” 别名扩展成它完整的邮件地址。

16.6 缩排模式

默认的情况下，如果要格式化一个区域，Emacs 在多个空格的地方插入制表符来代替。（例如，可能用 indent-region 命令一次性地缩排多行文本。）在终端屏幕上或者普通打印时，使用制表符将是很好的。但是当你使用 TEX或者 Texinfo 时，由于TEX忽略制表符，因此使用制表符的话将得到错误缩排的输出。

下面的表达式关闭制表符缩排模式：

```
;;; Prevent Extraneous Tabs
(setq-default indent-tabs-mode nil)
```

注意，上面的表达式值用 setq-default 函数而不是前面看到的 setq 函数。setq-default 命令只是在缓冲区中设置值而不是为变量设置它自己的局部值。

参见《*GNU Emacs* 技术手册》中的“制表符和空格”一节以及“文件中的局部变量一节”。

16.7 一些绑定键

现在该讲讲一些个人的绑定键了：

```
;;; Compare windows
(global-set-key "\C-cw" 'compare-windows)
```

compare-windows 是一个极有用的命令，这个命令将你当前缓冲区中的文本与下一个窗口中的文本进行比较。它将位点置于每一个窗口的开始进行比较，只要能够匹配就尽可能使位点移过整个缓冲区。我总是使用这个命令。

这个表达式同样显示了如何设置一个全局性的绑定键。这种设置对所有模式都是有效的。

设置全局性的绑定键的命令就是 global-set-key。这个命令之后就是绑定键。在一个

“.emacs”文件中，键的绑定是这样完成的：\C-c 代表“control-c”，这是指“同时按下 control 键和 c 键。其中的 w 是指“按下w键”。绑定键是由双引号包围的。在文档中，可以这样写：C-c w。（如果绑定 meta 键(如 M-c)而不是 control 键，应该这样写：\M-c。）关于这个问题的详细资料，可以参见《*GNU Emacs* 技术手册》中的“在用户初始化文件中重新绑定键”一节。

由这些键激活的命令是 compare-windows。注意，compare-windows 是由一个单引号开始的，否则 Emacs 将首先试图对这个符号求值来决定它的值。

双引号、C 字符前的反斜线以及单引号，都是键绑定过程的三个必要的部分，而这往往是我经常忘记的。幸运的是，我已经开始记得应当看一看自己的初始化文件，并采纳其中的提示。这就是注释的妙处。

对于这个绑定键本身：C-c w，它是由前缀键和一个单字符组成的。在这个例子中，前缀键就是 C-c，单字符就是w。这个键的集合，C-c 和其后的 w 键，是保留给用户使用的。如果你编写一个表达式来扩充 Emacs 的功能，请避免使用这些键。你可以创建一个类似 C-c C-w 的绑定键。否则，我们将会“用光”留给自己使用的键。

下面是另外一个键绑定及其注释：

```
;;; Keybinding for 'occur'
; I use occur a lot, so let's bind it to a key:
(global-set-key "\C-co" 'occur)
```

其中的 occur 命令显示当前缓冲区中包含了与某个正则表达式匹配内容的所有行。匹配的行显示在一个称为“*Occur*”的缓冲区中。这个缓冲区是作为一个菜单来显示结果的。

要取消一个键的绑定，下面的表达式将告诉你如何做。取消键绑定之后，这些键就不再起作用了。

```
;;; Unbind 'C-x f'
(global-unset-key "\C-xf")
```

之所以要取消键绑定，是因为我发现当我要键入 C-x C-f 的时候不可避免地键入了 C-x f。也就是说，当我要查找一个文件的时候，总是不小心设置了不是我所需要的文本宽度。因为我几乎从不重新设置默认的文本宽度，因此我简单地取消了这个键绑定。

下面的表达式重新绑定一个已经存在的键绑定：

```
;;; Rebind 'C-x C-b' for 'buffer-menu'
(global-set-key "\C-x\C-b" 'buffer-menu)
```

默认的情况是，C-x C-b 运行了 list-buffer 命令。这个命令在另外一个窗口中列出所有的缓冲区。因为我几乎总是要在这个窗口中做点什么，因此我更喜欢 buffer-menu 命令。这个命令不仅列出所有缓冲区，而且将位点移动到那个缓冲区中。

16.8 加载文件

GNU Emacs 社团中的许多人已经为 Emacs 编写了大量的扩展功能。随着时间的流逝，这些扩展功能经常被包含在新发行的版本之中，例如，Calendar 和 Diary 包现在都是标准的 Emacs 第19版的一部分了。但是它们曾经并不是标准的 Emacs第18版中的一部分。

（calc 是一个计算器，我认为这是 Emacs 的一个重要部分，应该成为 Emacs 的一个标准发行版的部分，只是它太大了而成为了一个独立的软件包。）

能够使用 load 命令对一个完整的文件求值，并因此将这个文件中所有函数和变量安装到 Emacs 中。例如，

```
(load "~/emacs/kfill")
```

对这个表达式求值，即从用户个人目录的“ emacs”子目录中加载“kfill.el”文件。（或者如果存在的话，加载这个文件的字节编译文件“kfill.elc”速度会更快。）

（“kfill.el”是 Bob Weiner 从 Kyle E. Jones 的“filladapt.el”包中整理出来的，并且它提供从新闻或者邮件消息，以及从Lisp C++和 PostScript 甚至 shell 注释中获得的缩排的、清晰文本的功能。我经常使用它们，也希望它能被打包进入标准的发行版本中。）

如果你要加载许多扩充功能包，就像我这样，无需精确指定扩充文件的准确路径，只要像上面那样指定它们作为 Emacs 的load-path 一部分的目录即可。然后，当 Emacs 加载一个文件时，它将查询这个目录以及它的默认的目录列表。（默认的目录列表是 Emacs 安装时在“paths.h”文件中指定的。）

下面的命令将你的“~/emacs”目录增加到已经存在的加载目录中：

```
;;; Emacs Load Path
(setq load-path (cons "~/emacs" load-path))
```

顺便说一下，load-library 是 load函数的交互接口，这个函数的完整形式如下所示：

```
(defun load-library (library)
  "Load the library named LIBRARY.
This is an interface to the function 'load'."
  (interactive "sLoad library: ")
  (load library))
```

这个函数的函数名，load-library，来自于人们使用“library”（库）来作为“file”(文件)的同义语。load-library 命令的源代码在“files.el”库中。

有另外一个交互命令与 load-file 命令完成相似的工作。参见《*GNU Emacs技术手册*》中的“Emacs 的 Lisp 代码库”，可以得到关于 load-library 和 load-file 之间差别的信息。

16.9 自动加载

除了通过加载包含指定函数的文件来实现函数的加载和安装，以及通过对函数定义求值来实现函数的安装之外，你还能够在不真正安装函数代码的情况下使用这个函数。这个函数是在它第一次被调用的时候安装的。这称为*自动加载*。

当你执行一个自动加载函数时，Emacs 自动地对包含这个函数的文件求值，然后调用这个函数。

Emacs 使用自动加载函数时执行得更快，因为自动加载函数的函数库不是直接被加载的，你在第一次使用这个函数的时候需要稍微等待一会儿，这是因为包含这个函数的文件正在被求值。

不常用的函数经常是自动加载的函数。“loaddefs.el”库包含了几百个自动加载的函数，从bookmark-set 到 workstar-mode。当然，你可能经常使用一个“不常用”的函数。在这种情况下，你应当在自己的“.emacs”初始化文件中用一个 load 表达式加载包含这个函数的文件。

在我的 Emacs 第19.23版的“.emacs”初始化文件中，我一共加载了17个库，这些库包含了原本要被自动加载的函数。(实际上，当我创建我的 Emacs 时，我应当将它包含在我的“转储”Emacs 中。但是我忘了。关于“转储”的更详细的介绍，参见《*GNU Emacs Lisp 技术手册*》中的“创建 Emacs”一节。)

你也可以将自动加载函数的相关文件包含在你个人的“.emacs”初始化文件的自动加载的表达式中。autoload 是一个内置的函数，这个函数接收五个参量。其中的最后三个是可选的。第一个参量是被自动加载的函数名，第二个参量是被加载的文件名。第三个参量是为这个函数编写的文档。而第四个参量是告之这个函数是否能被交互地调用。最后一个参量，也就是第五个参量，告诉对象是什么类型的—— autoload (自动加载) 函数可以处理函数也可以处理键图和宏。(默认情况下是函数)

下面是一个典型的例子：

```
(autoload 'html-helper-mode
  "html-helper-mode" "Edit HTML documents" t)
```

这个表达式从 html-helper-mode.el 文件中(或者如果存在的话就从 html-helper-mode.elc 文件中加载)自动加载 html-helper-mode 函数。这个文件必须是在由 load-path 定义的一个目录中。函数的文档说，这是一个帮助你用 HTML 语言编辑文档的模式。键入 M-x html-helper-mode，你能够交互地调用这个模式。(你需要复制自动加载表达式中的函数文档，因为这个函数还没有加载，因此它的文档也没有。)

参见《*GNU Emacs Lisp 技术手册*》的“自动加载”一节，可以得到更多的信息。

16.10 一个简单的功能扩充：line-to-top-of-window

这是一个对 Emacs 的简单的功能扩充，这个扩充使某一行的位置移动到窗口的顶端。我总是使用这个函数以使文本易于阅读。

你可以将下面的代码放进一个独立的文件中，并从你个人的“.emacs”初始化文件中加载它，或者你可以将这个函数包含在你个人的“.emacs”初始化文件中。

下面就是这个函数的定义：

```
;;; Line to top of window;
;;; replace three keystroke sequence  C-u 0 C-l
(defun line-to-top-of-window ()
  "Move the line point is on to top of window."
  (interactive)
  (recenter 0))
```

现在就要绑定组合键了。

虽然大部分 Emacs第18版的“.emacs”文件在第19版下工作得很好，但是它们之间还是有

区别的（当然，在第19版中有一些新的特性）。

在第19版的 Emacs 中，你可以编写这样的函数："[f6]"。在第18版中，你必须由键盘指定键序列。例如，对于一个 Zenith 29 键盘而言，当我按下它的第六个功能键的时候，键盘发送 ESC P 键序；对于一个 Ann Arbor Ambassador 型的键盘，则发送一个 ESC O F 键序列。这些键序列应被分别写成"\eP"和"\eOF"。

在我的第18版的".emacs"初始化文件中，我将 line-to-top-of-window 绑定到一个键上，这个键依赖于终端的类型：

```
(defun z29-key-bindings ()
  "Function keybindings for Z29 terminal."
  ;; ...
  (global-set-key "\eP" 'line-to-top-of-window))
(defun aaa-key-bindings ()
  "Function keybindings for Ann Arbor Ambassador"
  ;; ...
  (global-set-key "\eOF" 'line-to-top-of-window))
```

(你可以通过键入功能键来发现各个功能键的键值，然后键入 C-h l (view-lossage) 来显示最后 100 个键序列的值。)

在定义了绑定键之后，我对一个表达式求值，这个求值是根据我所使用的终端的类型从各种键序列中选择出来的。然而，在这样做之前，我关闭了预先定义的、与终端类型相关的默认绑定键，因为如果它们冲突的话就会覆盖在".emacs"中定义的键序列。

```
;;; Turn Off Predefined Terminal Keybindings

; The following turns off the predefined
; terminal-specific keybindings such as tbe
; vt100 keybindings in lisp/term/vt100.el.
; If there are no predefined terminal
; keybindings, or if you like them,
; comment this out.

(setq term-file-prefix nil)
```

下面是选择表达式本身：

```
(let ((term (getenv "TERM")))
  (cond
   ((equal term "z29") (z29-key-bindings))
   ((equal term "aaa") (aaa-key-bindings))
   (t (message
       "No binding for terminal type %s."
       term))))
```

在第19版中，功能键（包括鼠标事件和非 ASCII 字符）都是写在方括号中的，而不要引号。将 line-to-top-of-window 绑定到 F6 功能键上：

```
(global-set-key [f6] 'line-to-top-of-window)
```

简单多了！

关于绑定键的更详细的信息，参见《*GNU Emacs* 技术手册》中的“在你的初始化文件中重新绑定键”一节。

如果同时运行 Emacs 的第18版和第19版两个版本，可以用下面的条件表达式选择其中之一求值：

```
(if (string=
     (int-to-string 18)
     (substring (emacs-version) 10 12))
    ;; evaluate version 18 code
    (progn
      ... )
  ;; else evaluate version 19 code
  ...
```

16.11 键图

Emacs 使用键图(*keymaps*)来记录什么键调用什么命令。特定的模式，如 C 模式或者文本模式，都有它们自己的键图。与模式有关的键图将覆盖由所有缓冲区共享的全局键图。

`global-set-key` 函数的功能是绑定、或者重新绑定全局键图。例如，下面的键绑定表达式将 `C-c C-l` 绑定到 `line-to-top-of-window` 函数：

```
(global-set-key "\C-c\C-l" 'line-to-top-of-window))
```

与模式有关的键图是用 `define-key` 函数绑定的，它接受一个指定的键图、键以及命令作为其参量。例如，我的“`.emacs`”初始化文件包含下面的表达式，这些表达式将命令 `texinfo-insert-@group` 绑定到 `C-c C-c g`：

```
(define-key texinfo-mode-map "\C-c\C-cg"
  'texinfo-insert-@group)
```

`texinfo-insert-@group` 函数本身是对 Texinfo 模式的一个小的功能扩充，这个表达式将“`@group`”插入到一个 Texinfo 文件中。我总是用这个命令，而且喜欢键入 `C-c C-c g` 而不喜欢键入 `@group`。(“`@group`”和它的对应物“`@end group`”是将一页中的文本组织起来的两个命令，在这本书中许多的多行例子都是由这两个命令“`@group…@end group`”组织起来的。)

下面是 `texinfo-insert-@group` 函数的定义：

```
(defun texinfo-insert-@group ()
  "Insert the string @group in a Texinfo buffer."
  (interactive)
  (beginning-of-line)
  (insert "@group\n"))
```

(当然，我可以使用缩写模式来节省打字输入，而不是编写一个函数来插入一个单词，但是

我更喜欢与 Texinfo 模式中的绑定键一致的键序列。）

你将在“loaddefs.el”中看到大量的 define-key 表达式，也将在不同的模式库（如“c-mode.el”和“lisp-mode.el”中看到这些表达式）。

关于这方面的更多的信息，参见《*GNU Emacs* 技术手册》中的“定制键绑定”一节和《*GNU Emacs Lisp* 技术手册》中的“键图”一节。

16.12 X11的颜色

当你在 MIT X Window 系统上使用 Emacs 第19版及以上的版本时，你就能够定义屏幕显示的颜色。（所有前面的例子既能在第19版上运行，也能在第18版上运行，但是这一节讲述的内容只能在 第19版上起作用。）

我讨厌自己系统中的默认颜色，因此我要自己指定颜色。

在我的系统中，大部分对颜色的定义放在不同的 X 初始化文件之中。同时我在自己的“.emacs”初始化文件中做了注释，提醒我自己所做的修改：

```
;; I use TWM for window manager;
;; my ~/.xsession file specifies:
;    xsetroot -solid navyblue -fg white
```

实际上，X Window 系统的根目录根本不是 Emacs 的一部分，但是我喜欢这种提示。

```
;; My ~/.Xresources file specifies:
;     XTerm*Background:    sky blue
;     XTerm*Foreground:    white
;     emacs*geometry:      =80x40+100+0
;     emacs*background:    blue
;     emacs*foreground:    grey97
;     emacs*cursorColor:   white
;     emacs*pointerColor:  white
```

下面是在我的“./emacs”初始化文件中的设置这些变量的表达式：

```
;;; Set highlighting colors for isearch and drag
(set-face-foreground 'highlight "white" )
(set-face-background 'highlight "slate blue")
(set-face-background 'region    "slate blue")
(set-face-background
 'secondary-selection "turquoise")

;; Set calendar highlighting colors
(setq calendar-load-hook
      '(lambda ()
         (set-face-foreground 'diary-face   "skyblue")
         (set-face-background 'holiday-face "slate blue")
         (set-face-foreground 'holiday-face "white")))
```

不同的蓝色阴影使我的眼睛感觉很柔和，并使我避免看到屏幕的闪烁。

16.13 V19中的小技巧

这里列出的是第19版中设置的一些小技巧：

1) 自动根据需要改变小缓冲区的大小：

```
(resize-minibuffer-mode 1)
(setq resize-minibuffer-mode t)
```

2)开启对查询字符串的高亮显示：

```
(setq search-highlight t)
```

3)将每一个框架都设置为显示一个菜单条，并在你的鼠标移动到它上面时弹出菜单：

```
(setq default-frame-alist
      '((menu-bar-lines . 1)
        (auto-lower . t)
        (auto-raise . t)))
```

4) 设置鼠标的形状和颜色：

```
; Cursor shapes are defined in
; '/usr/include/X11/cursorfont.h';
; for example, the 'target' cursor is number 128;
; the 'top_left_arrow' cursor is number 132.
(let ((mpointer (x-get-resource "*mpointer"
                                "*emacs*mpointer")))
  ;; If you have not set your mouse pointer
  ;;     then sent it, otherwise leave as is:
  (if (eq mpointer nil)
      (setq mpointer "132")) ; top_left_arrow
  (setq x-pointer-shape (string-to-int mpointer))
  (set-mouse-color "white"))
```

16.14 修改模式行

最后，还有一个特性是我喜欢的：修改模式行。

因为我有时在一个网络系统上工作，因此我经常将平时正常显示在模式行的左边部分的“Emacs:”用我当时的系统名替代——否则的话，我会忘记自己究竟在使用哪一台计算机。另外，我总是列出默认的目录以免不知道自己处于什么位置，并且我开启行位点开关，使其显示“Line”值。我的“.emacs”初始化文件如下所示：

```
(setq mode-line-system-identification
  (substring (system-name) 0
             (string-match "\\..+" (system-name))))

(setq default-mode-line-format
      (list ""
            'mode-line-modified
            "<"
            'mode-line-system-identification
```

```
"> "
"%14b"
" "
'default-directory
" "
"%[("
'mode-name
'minor-mode-alist
"%n"
'mode-line-process
")%]--"
 "Line %l--"
'(-3 . "%P")
"-%-"))

;; Start with new default.
(setq mode-line-format default-mode-line-format)
```

我设置默认的模式行格式以允许不同的模式(如 Info)正常覆盖它。这个列表的许多元素的意义都是不言自明的：mode-line-modified 变量告诉缓冲区是否已被修改，mode-name 变量告诉模式的名称，等等。

其中的“%14b”显示当前缓冲区的名称（使用我们熟悉的 buffer-name 函数），其中的“14”定义最大显示的字符数是 14。当一个缓冲区的名称的字符数低于 14 个字符时，就用空格添满。“%[”和“%]”符号使得在每一个递归的编辑层次中使用一对方括号。“%n”则在变窄开启时显示“Narrow”。“%P”告诉你在窗口底部之上的部分占整个缓冲区的百分比，或者告诉你在“TOP”（顶端）、“Bottom”（底端）或“All”（全部）。（小写的“p”字符告诉你，在窗口顶部之上的部分所占的百分比）。“%-”则插入足够的破折号来填满一行。

对于 Emacs第19版及其后的版本，可以使用 frame-title-format 来设置一个Emacs 框架的标题。这个变量与 mode-line-format 变量有同样的结构。

模式行的格式在《*GNU Emacs Lisp 技术手册*》中的“模式行格式”一节中有详细的描述。

记住，“不要只为了喜欢 Emacs 而喜欢它”——你自己的 Emacs 可以与默认的Emacs拥有不同的颜色、不同的命令以及不同的绑定键。

另一方面，如果要进入一个没有任何定制的普通 Emacs 环境，键入：

```
emacs -q
```

就行了。这个命令将使 Emacs 不加载你个人的“~/.emacs”初始化文件。仅提供一个普通的默认的 Emacs，其他什么也没有。

第17章 调 试

GNU Emacs 有两个调试器：debug 和 edebug。第一个调试器 debug 内置在Emacs之中，并且总是可以供你使用；第二个调试器 edebug 则是 Emacs 的一个扩充，这个调试器已经成为 Emacs 第19版的标准发行版本中的一个部分。

两个调试器都在《*GNU Emacs Lisp* 技术手册》中的“调试 Lisp 程序”一节中有详尽的描述。在这一章，将分别结合一个简单的例子简要地介绍这两个调试器。

17.1 debug

假设你已经编写了一个函数定义，这个函数将计算数字 1 到一个给定的数字之和，并返回这个结果。（这就是前面讲到的 triangle 函数。参见11.1.4节中“减量计数器的例子”小节的有关讨论。）

然而，你的函数定义有一个bug。你在输入“1-”的时候错误地输入了“1=”。下面是包含了这个错误的函数定义：

```
(defun triangle-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (setq number (1= number)))        ; Error here.
    total))
```

如果你在 Info 中阅读这份文档，可以用通常的方法对这个函数定义求值。你将看到 triangle-bugged 出现在回显区中。

现在用参量4传送给triangle-bugged函数，并对它求值：

```
(triangle-bugged 4)
```

你将得到下面的错误消息：

```
Symbol's function definition is void: 1=
```

实际上，对于这样的一个简单的 bug，传递这个错误消息的目的是告诉你改正这个函数定义所应当知道的内容。然而，假设你对此不确定，那怎么办呢？

你可以通过将 debug-on-error的值设置成 t 来开始调试：

```
(setq debug-on-error t)
```

这个表达式使 Emacs 在它下一次遇到一个错误的时候进入调试器。

通过将这个变量的值设置为 nil 就可以关闭它：

```
(setq debug-on-error nil)
```

将 debug-on-error 的值设置为 t，并对下面的表达式求值：

```
(triangle-bugged 4)
```

这一次，Emacs 将创建一个称为“*Backtrace*”缓冲区：

```
---------- Buffer: *Backtrace* ----------
Signalling: (void-function 1=)
  (1= number))
  (setq number (1= number)))
  (while (> number 0) (setq total (+ total number))
       (setq number (1= number))))
  (let ((total 0)) (while (> number 0) (setq total ...)
       (setq number ...)) total))
  triangle-bugged(4)
  eval((triangle-bugged 4))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(我已经将这个例子的格式稍微改动过，调试器不会将长的行折叠过来。)

你可以从这个缓冲区的底部开始往上读，它将告诉你Emacs 是如何一步一步地遇到错误。在这个例子中，Emacs 所做的就是交互地调用了 C-x C-e(eval-last-sexp)，这个命令导致对 triangle-bugged 表达式的求值。上面的每一行都告诉你后面 Lisp 解释器是如何求值的。

从缓冲区顶部开始的第三行是：

```
(setq number (1= number))
```

Emacs 试图对这个表达式求值。为了对这个表达式求值，它先对内部的表达式(1= number)求值，这个显示在从缓冲区顶部开始的第二行上。

```
(1= number)
```

这就是错误发生的地方，就像缓冲区最顶端的一行所示：

```
Signalling: (void-function 1=)
```

现在你能够改正这个错误，然后再对这个函数定义求值，并再一次运行你的测试表达式。

如果你正在 Info 中阅读这部分，现在就可以通过将其值设置为 nil来关闭 debug-on-error了：

```
(setq debug-on-error nil)
```

17.2 debug-on-entry

第二种启动 debug 的方法，是当你调用一个函数的时候进入调试器。调用debug-on-entry 就能够实现这一点。

键入：

```
M-x debug-on-entry RET triangle-bugged RET
```

现在，对下面的表达式求值：

```
(triangle-bugged 5)
```

Emacs 将创建名为“*Backtrace*”的缓冲区，并告诉你它将开始对triangle-bugged 函数求值：

```
---------- Buffer: *Backtrace* ----------
Entering:
* triangle-bugged(5)
  eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

在“*Backtrace*”缓冲区中，键入 d。Emacs 将对 triangle-bugged 函数的第一个表达式求值，缓冲区内容将是：

```
---------- Buffer: *Backtrace* ----------
Beginning evaluation of function call form:
* (let ((total 0)) (while (> number 0) (setq total ...)
        (setq number ...)) total))
  triangle-bugged(5)
* eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

现在，一次一次地慢慢键入 d，一共键入 8 次。每当你键入一次 d 键，Emacs 将对函数定义中的下一个表达式求值。最终，缓冲区内容将是：

```
---------- Buffer: *Backtrace* ----------
Beginning evaluation of function call form:
* (setq number (1= number)))
* (while (> number 0) (setq total (+ total number))
        (setq number (1= number))))
* (let ((total 0)) (while (> number 0)
        (setq total ...) (setq number ...)) total))
  triangle-bugged(5)
* eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

最后，你再键入两次 d 之后，Emacs 将遇到错误，这时“*Backtrace*”缓冲区顶部的两行是：

```
---------- Buffer: *Backtrace* ----------
Signalling: (void-function 1=)
* (1= number))
...
---------- Buffer: *Backtrace* ----------
```

总之，通过键入 d 键，你可以一步一步地测试函数的每一个表达式。

通过键入 q 就可以退出“*Backtrace*”缓冲区，这个命令退出函数调试，但是并不取消 debug-on-entry。

要想取消 debug-on-entry，就要调用 cancel-debug-on-entry：

M-x cancel-debug-on-entry RET triangle-debugged RET

（如果你在 Info 中阅读这份文档，现在就可以取消 debug-on-entry。）

17.3 debug-on-quit 和 (debug)

除了设置 debug-on-error 和调用 debug-on-entry 之外，还有另外两种方法可以启动 debug。

通过将变量 debug-on-quit 设置为 t，可以使你无论何时键入 C-g(keyboard-quit)都能够启动 debug。这对于调试无限的循环很有用。

或者，你能够在你的代码中需要调试的一行中插入“(debug)”，就像这样：

```
(defun triangle-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (debug)                          ; Start debugger.
      (setq number (1= number)))       ; Error here.
    total))
```

有关debug 函数的详细描述，参见《*GNU Emacs Lisp* 技术手册》中的“Lisp 调试器”一节。

17.4 源代码级调试器 edebug

edebug 通常显示你所调试的函数的源代码。对于你当前执行的那一行，edebug 用一个箭头在左边进行提示。

你可以一行一行地跟踪整个函数的执行，或者快速运行直到到达一个断点处(在这个断点处 Emacs 执行停止)。

edebug 在《*GNU Emacs Lisp* 技术手册》中的“edebug”一节中有详细介绍。

这里是用于triangle-recursively 的一个带有 bug 的函数定义。对于triangle-recursively 函数定义本身，可以参见 11.2.2节“ 用递归算法代替计数器”。下面显示的这个例子没有对 defun 这一行进行通常的缩排。

```
(defun triangle-recursively-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive.
Uses recursion."
  (if (= number 1)
      1
    (+ number
       (triangle-recursively-bugged
        (1= number)))))                ; Error here.
```

通常，将光标置于函数定义的最后一个括号之后并键入 C-x C-e(eval-last-sexp)，或者将光标置于函数定义之中并键入 C-M-x(eval-defun)，你都能够安装这个函数。(默认情况下，eval-defun 命令仅仅工作在 Emacs Lisp 模式或者 Lisp 交互模式中。)

然而，要为 edebug 准备这个函数定义，你必须首先要用另外的命令将这个函数定义配置好。在 Emacs第19版中，将光标置于函数定义中并键入下面的命令就可以实现这个目的：

```
M-x edebug-defun RET
```

这个命令在没有加载 edebug 的情况下使 Emacs 自动地加载 edebug，然后正确地配置这个函数。(在加载 edebug 之后，你可以使用它的标准的绑定键，如C-u C-M-x (eval-defun以及一个前缀参量)，来调用 edebug-defun。)

在 Emacs 第18版中，需要自己来加载 edebug。将适当的 load 命令增加到你的“.emacs”初始化文件中就行了。

如果你在 Info 中阅读这份文档，你能够配置上面显示的这个 triangle-recursively-bugged 函数。对于函数定义行是缩进的那个函数定义，edebug-defun 无法正确定位其边界，因此这个例子没有按通常的缩排方式显示。

配置完函数之后，将光标置于下面的表达式后并键入C-x C-e(eval-last-sexp)：

```
(triangle-recursively-bugged 3)
```

光标将回退到 triangle-recursively-bugged 函数源代码中，而且光标位于这个函数的 if 表达式那一行的开始。同样，你将看到，一个箭头在这一行的左边，就像这样：“⇒”。这个箭头表示这一行是函数当前运行的那一行。

```
=>*(if (= number 1)
```

在这个例子中，位点的位置是用一个星号“*”显示(在 Info 中，位点则是以“-!-”显示)。

如果你现在键入空格键(SPC)，位点将移动到下一个执行的表达式，显示是这样的：

```
=>(if *(= number 1)
```

当你继续按下空格键的时候，位点将从一个表达式移动到下一个表达式。同时，每当表达式返回一个值时，这个值将显示在回显区中。例如，当你将位点移过 number 之后，你将看到下面的内容：

```
Result: 3 = C-c
```

这意味着 number 变量的值是 3，它对应着 ASCII码 的 C-c(CTL-c)。

你可以继续移过代码直到你到达有错误的那一行。在求值之前，这一行是：

```
=>          *(1= number)))))                    ; Error here.
```

当你按下空格键时，你将产生这样一个错误消息：

```
Symbol's function definition is void: 1=
```

这是程序的一个 bug。

按“q”键就可以退出 edebug。

要去除一个函数定义的 edebug，只要对它重新求值而不安装即可。例如，你可以将光标置于函数定义的最后一个括号之后并键入 C-x C-e 即可。

edebug 完成的工作比仅仅遍历整个函数代码多得多。你可以将它设置成自行运行，直到到达有错误的那一行或者在指定的断点才停下来，你可以使它显示表达式值的改变；你可以找出一个函数被调用了多少次，等等。

edebug 在《*GNU Emacs Lisp* 技术手册》中的“edebug”中有详细介绍。

17.5 调试练习

- 安装 count-words-region 函数，然后使之在你调用它的时候进入一个内置的调试器。在一个包含两个单词的区域运行这个命令。你将需要按下 d 键很多次。在你的系统中，在这个命令执行完之后，是否有一个“hook”调用？（关于 hook 的更多信息，参见《*GNU Emacs Lisp* 技术手册》中“命令循环概述”一节。）
- 将 count-words-region 函数拷贝到草稿缓冲区，如果需要，将defun一行前面的空格去掉，为 edebug 配置好这个函数，并跟踪它的执行。无需为这个函数制造一个 bug，虽然你完全可以做到这一点。如果一个函数没有 bug，遍历整个函数也是没有任何问题的。
- 在运行 edebug 的时候，键入“？”，看一看 edebug 的命令列表。（global-edebug-prefix 通常绑定到 C-x X，也就是 CTL-x 后接一个大写的 X，使用这个命令前缀可以暂时离开 edebug 的调试缓冲区。）
- 在 edebug 的调试缓冲区中，用 p (edebug-bounce-point) 命令看一看 count-words-region 函数在区域中的什么位置执行。
- 将位点移动到函数之后，然后键入 h (edebug-goto-here)命令，以使之跳到这个位置。
- 使用 t (edebug-trace-mode) 命令以使 edebug 自行跟踪这个函数的执行；使用大写的 T 用于 edebug-Trace-fast-mode，试一试，看结果会是什么？
- 设置一个断点，然后在跟踪模式中运行 edebug 直到它到达停止点。

第18章 结 论

现在到了这本入门教程的末尾了。你已经学习了足够的关于用 Emacs Lisp 中编程的知识，如给变量赋值、编写简单的“.emacs”文件，编写简单的功能扩充程序等。

现在是该告一段落的时候了。如果你愿意，你现在可以直接独自朝前走了——自学吧！

你已经学习了编程的一些基本知识。但是这仅仅是一部分而已。还有大量的易于使用的工具我们还没有接触到。

学习更多知识的一个途径，是阅读 GNU Emacs 源代码以及《*GNU Emacs Lisp*技术手册》[⊖]中提到的代码。

阅读 Emacs Lisp 源代码是一次冒险。当你阅读这些源代码并遇到一个不熟悉的函数或者表达式时，你需要弄清它的功能是什么。

去看看那本《*GNU Emacs Lisp*技术手册》吧。它对 Emacs Lisp 作了完全的、且易于阅读的描述。它不仅是为专家编写的，同时也是为像你这样的读者编写的（这本技术手册随着 GNU Emacs 的标准发行版本发行，并且是一个 Texinfo 源文件，因此你能够在线阅读它，或者打印出来阅读）。[⊖]

GNU Emacs 的另外一个在线帮助系统是所有函数的在线帮助文档以及帮助你阅读这些源代码的命令——`find-tags`。

下面是我阅读源代码的一个例子。很久以前，我总是根据文件的名字首先查看“`simple.el`”文件。当我阅读这个文件中的函数定义时，碰巧在这个文件中有一些函数很复杂，或者至少初看起来很复杂。例如，第一个函数看起来就很复杂。这个函数是 `open-line` 函数。

你可以慢慢地阅读这个函数，就像我们处理 `forward-sentence` 函数那样（参见12.3节）。或者你希望跳过这个函数阅读下一个函数，比如 `split-line`。其实，你无需阅读所有这些函数。根据 `count-words-in-defun` 函数，`split-line` 函数包含 27 个单词和符号。

虽然这个函数很短，但是它包含四个我们没有学到的函数：`skip-chars-forward`、`indent-to`、`insert`和“`?\n`”。

先来考虑 `insert`函数（这个函数曾经出现在第12章的回顾部分）。在 Emacs 中，你能够通过键入 `C-h f (describe-function)` 以及这个函数名，来找到关于这个函数的更多信息——打印出函数文档。你也可以用 `find-tag` 来阅读函数定义的源代码，这个命令绑定到 `M-.`(在这个例子中，这个命令并不很有用，因为这是一个用 C 语言编写的基本函数)。最后，你还可以找到技术手册中关于这个函数的说明。你可以在 Info 中键入 `i (Info-index)` 以及函数名来访问

⊖ 《*GNU Emacs Lisp* 技术手册》中文版由自由软件基金会中国研究院翻译。

⊖ 《*GNU Emacs* 技术手册》中文版由自由软件基金会中国研究院翻译。

技术手册中关于这个函数的内容，或者在一份打印的手册中根据索引来查阅 insert 这个函数。

类似地，你可以找到“?\n”的含义。你可以尝试用 Info-index 来查找相关信息。对于这个函数这样做毫无帮助，但是不要放弃。如果在索引中仅仅查询“\n”而不查找“?”，你将直接到达手册中的相关章节（参见《*GNU Emacs* 技术手册》中的“字符类型”一节。“?\n”代表换行符）。

你可能已经猜测到 skip-chars-forward 和 indent-to 函数的功能了，否则你可以仔细看一看。（顺便提一下，describe-function函数本身在“help.el” 文件中，它是那些较长但是好理解的函数中的一种。它的函数定义说明了如何不用标准的控制字符来定制 interactive 表达式，并且显示了如何建立一个临时缓冲区。）

其他有趣的源代码包含在“paragraphs.el”、“loaddefs.el”以及“loadup.el”文件中。其中“paragraph.el”文件中包含了短的、易于理解的函数，也有个别长的。“loaddefs.el”文件中包含许多标准的自动加载函数以及许多键图。我从来没有全部阅读完这个文件中的所有函数，仅仅阅读了一部分。“loadup.el”文件加载 Emacs 的标准部分，它告诉你如何构建 Emacs 的许多知识（详细情况参见《*GNU Emacs* 技术手册》）。

就像我所说的，你已经学习了一些简单的、但是却是非常重要的知识。我们几乎没有触及编程的主要部分，除了使用预先定义的 sort 函数之外，我没有说过任何关于信息分类的事情。我没有提到如何编写用来“编写程序”的程序。这是其他层次较深的书的主题。那是另外一种学习进阶。

阅读完本书，你所做的是学习了足够的使用 Emacs 的实际知识，你已经入门了。这本入门教程到此结束。

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