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。