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。