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。)

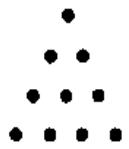
由增量计数器控制的 while 循环的模板如下所示：

```
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。）