Lisp 入门教程

作者：Geoffrey J. Gordon <ggordon@cs.cmu.edu> 1993/02/05 星期五
修订：Bruno Haible <haible@ma2s2.mathematik.uni-karlsruhe.de>
翻译： 刘鑫 <March.Liu@gmail.com>
整理：张泽鹏 <redraiment@gmail.com> 2011/06/24 星期五

注意：这份 Common Lisp 入门教程是针对 CMU 环境编写，所以在其它环境运行 Lisp 时可能会有细节上的区别。
附：
据我所知最好的 Lisp 书籍是：Guy L. Steele Jr. 《Common LISP: the Language》 Digital Press. 1984.
第一版很容易阅读，第二版介绍了更新的标准。（两个标准的差异很小，对于粗心的程序员没有什么区别。）
我还记得 Dave Touretsky 写了一本，不过我从来没读过，所以不能对那本书发表评论。

Symbols

符号仅仅是字符串。你可以在符号中包含字母、数字、连接符等等，唯一的限制就是要以字母开头。（如果你只输入数字，最多再以一个连接符开头的话，LISP 会认为你输入了一个整数而不是符号。）例如：

```
a
b
c1
foo
bar
baaz-quux-garply
```

接下来我们可以做些事情。（">"标记表示你向 LISP 输入的东西，其它的是 LISP 打印返回给你的。";"是 LISP 的注释符：";"后面的整行都会被忽略。）

```
> (setq a 5)                    ;store a number as the value of a symbol
5
> a                             ;take the value of a symbol
5
> (let ((a 6)) a)               ;bind the value of a symbol temporarily to 6
6
> a                             ;the value returns to 5 once the let is finished
5
> (+ a 6)                       ;use the value of a symbol as an argument to a function
11
> b                             ;try to take the value of a symbol which has no value
Error: Attempt to take the value of the unbound symbol B
```

有两个特殊的符号， t 和 nil 。 t 的值总是定义为 t，nil 的值总是定义为 nil 。LISP 用 t 和 nil 代表 true 和 false。以下是使用这个功能的 if 语句，后面再做详细说明：

```
> (if t 5 6)
5
> (if nil 5 6)
6
> (if 4 5 6)
5
```

最后一个例子看起来很怪，但是没有错：nil 代表 false ，其它任意值代表 true。（为了代码清晰，在没有什么特别原因的情况下，我们用 t 代表 true。）

t 和 nil 这样的符号被称为自解析符号，因为他们解析为自身。自解析符号称为关键字；任一以冒号开头的符号都是关键字。（下面是一些关键字的应用）如下所示：

```
> :this-is-a-keyword
:THIS-IS-A-KEYWORD
> :so-is-this
:SO-IS-THIS
> :me-too
:ME-TOO
```

Numbers

数值类型是数字文本，可能会以 + 或 - 开头。实数和整数很相像，但是它带有小数点，还可能写成科学计数法。有理数就像是两个整数之间带有一个/。LISP 支持复数，写为#c(r i)（r 表示实部，i 表示虚部）。以上统称为数值。下面是一 些数值：

```
5
17
-34
+6
3.1415
1.722e-15
#c(1.722e-15 0.75)
```

标准的计算函数包括：+, -, *, /, floor, ceiling, mod, sin, cos, tan, sqrt, exp, expt 等等。所有这些函数都可以接受任意数值类型参数。+、-、* 和 / 返回尽可能大的类型：一个整数加一个有理数返回有理数，一个有理数加一个实数是一个实数，一个实数加一个复数是一个复数。如下所示：

```
> (+ 3 3/4)                    ;type contagion
15/4
> (exp 1)                      ;e
2.7182817
> (exp 3)                      ;e*e*e
20.085537
> (expt 3 4.2)                 ;exponent with a base other than e
100.90418
> (+ 5 6 7 (* 8 9 10))         ;the fns +-*/ all accept multiple arguments
```

对于整数来说，唯一的大小限制就是机器的内存。当然大数值运算（这会调用大整数）可能会很慢。（因此我们可以计算有理数，尤其是小整数和浮点数的比较运算）

Conses

cons 就是一个包含两个字段的记录。出于历史原因，两个字段分别被称为 "car"和"cdr"。（在第一台实现 LISP 的机器上，用 CAR 和 CDR 代表"地址寄存器的内容"和"指令寄存器的内容"。Conses 的实现主要依靠这两个寄存器。）

Conses 很容易使用：

```
> (cons 4 5)                   ;Allocate a cons. Set the car to 4 and the cdr to 5.
(4 . 5)
> (cons (cons 4 5) 6)
((4 . 5) . 6)
> (car (cons 4 5))
4
> (cdr (cons 4 5))
5
```

Lists

你可以构造 conses 之外的结构。可能最简单的是链表：每一个 cons 的 car 指向链表 的一个元素，cdr 指向另一个 cons 或者 nil。我们可以使用 list 函数构造链表。

```
> (list 4 5 6)
(4 5 6)
```

需要注意的是 LISP 用一种特殊的方式打印链表：它忽略掉某些分隔和括号。规则如下：如果某个 cons 的 cdr 是 nil ，LISP 不打印 nil 和段标记，如果 cons A 的 cdr 是 cons B，LISP 不打印 cons B 的括号和 cons A 的分隔符。如下：

```
> (cons 4 nil)
(4)
> (cons 4 (cons 5 6))
(4 5 . 6)
> (cons 4 (cons 5 (cons 6 nil)))
(4 5 6)
```

最后一个例子相当于调用(list 4 5 6)。要注意的是这里 nil 表示没有元素的空链表：包含两个元素的链表(a b)中，cdr 是(b)，一个含有单个元素的链表；包含一个元素的链表(b)，cdr 是 nil，故此这里必然是一个没有元素的链表。

如果你把链表存储在变量中，可以将它当作堆栈来使用：

```
> (setq a nil)
NIL
> (push 4 a)
(4)
> (push 5 a)
(5 4)
> (pop a)
5
> a
(4)
> (pop a)
4
> (pop a)
NIL
> a
NIL
```

Functions

前面我们讨论过一些函数的例子，这里还有更多：

```
> (+ 3 4 5 6)                   ;this function takes any number of arguments
18
> (+ (+ 3 4) (+ (+ 4 5) 6))     ;isn't prefix notation fun?
22
> (defun foo (x y) (+ x y 5))   ;defining a function
FOO
> (foo 5 0)                     ;calling a function
10
> (defun fact (x)               ;a recursive function
    (if (> x 0)
      (* x (fact (- x 1)))
      1))
FACT
> (fact 5)
```

```
120
> (defun a (x) (if (= x 0) t (b (- x)))) ;mutually recursive functions
A
> (defun b (x) (if (> x 0) (a (- x 1)) (a (+ x 1))))
B
> (a 5)
T
> (defun bar (x)                 ;a function with multiple statements in
    (setq x (* x 3))             ;its body -- it will return the value
    (setq x (/ x 2))             ;returned by its final statement
    (+ x 4))
BAR
> (bar 6)
13
```

当我们定义函数的时候，设定了两个参数，x 和 y。现在当我们调用 foo，需要给出两个参数：第一个在 foo 函数调用时成为 x 的值，第二个成为 y 的值。在 LISP 中，大部分的变量都是局部的，如果 foo 调用了 bar，bar 中虽然使用了名字为 x 的引用，但 bar 得不到 foo 中的 x。

在调用过程中给一个符号赋值的操作被称为绑定。

我们可以给函数指定可选参数，在符号 &optional 之后的参数是可选参数：

```
> (defun bar (x &optional y) (if y x 0))
BAR
> (defun baaz (&optional (x 3) (z 10)) (+ x z))
BAAZ
> (bar 5)
0
> (bar 5 t)
5
> (baaz 5)
15
> (baaz 5 6)
11
> (baaz)
13
```

bar 函数的调用规则是要给出一个或两个参数。如果它用一个参数调用，x 将会绑定到这个参数值上，而 y 就是 nil；如果用两个参数调用它，x 和 y 会分别绑定到第一和第二个值上。

baaz 函数有两个可选参数。它为它们分别提供了默认值：如果调用者只给出了一个参数，z 会绑定为 10 而不是 nil，如果调用者没有给出参数，x 会绑定为 3，而 z 绑定为 10。

在参数列表的最后设置一个 &rest 参数，可以使我们的函数接受任意数目的参数。LISP 把所有的附加参数都放进一个链表并绑定到 &rest 参数。如下：

```
> (defun foo (x &rest y) y)
FOO
> (foo 3)
NIL
> (foo 4 5 6)
(5 6)
```

最后，我们可以为函数指定一种被称为关键字参数的可选参数。调用者可以用任意顺序调用这些参数，因为他们已经通过关键字标示出来。

```
> (defun foo (&key x y) (cons x y))
FOO
> (foo :x 5 :y 3)
(5 . 3)
> (foo :y 3 :x 5)
```

```
(5 . 3)
> (foo :y 3)
(NIL . 3)
> (foo)
(NIL)
```

关键字参数也可以有默认值：

```
> (defun foo (&key (x 5)) x)
FOO
> (foo :x 7)
7
> (foo)
5
```

Printing

某些函数可以用来输出。最简单的一个是 print，它可以打印参数并且返回它们。

```
> (print 3)
3
3
```

首先打印 3，然后返回它。

如果你需要更复杂的输出，可能会用到 format，这里有个例子：

```
> (format t "An atom: ~S~%and a list: ~S~%and an integer: ~D~%" nil (list 5) 6)
An atom: NIL
and a list: (5)
and an integer: 6
```

第一个参数可以是 t，nil 或者一个流。t 意味着输出到终端；nil 意味着不打印任何东西，而是把它返回。流是用于输出的通用方式：它可以是一个指定的文件，或者一个终端，或者另一个程序。这里不再详细描述流的更多细节。

第二个参数是个格式化模版，即一个包含格式化设定的字符串。

所有其它的参数由格式化设定引用。LISP 会根据标示所引用的参数，将其替换 为合适的字符，并返回结果字符串。

如果 format 的第一个参数是 nil，它返回一个字符串，什么也不打印，否则它总是返回 nil。

前面的例子中有三种不同的标示：~S，~D 和~%。第一个接受任意 LISP 对象并且将其替换为这个对象的打印描述（与使用 print 打印出的描述信息相同）。第二个接受一个整数。第三个总是替换为一个回车符。

另一个常用的标示是~~，它替换为单个~。

LISP 手册中介绍了其它（很多，很多）的格式化标示。

我们输入到 LISP 解释器的东西被称为语句；LISP 解释器逐条循环读取每条语句，进行解析，将结果打印出来。这个过程被称为读取-解析-打印循环。

某些语句会发生错误，LISP 会引领我们进入调试器，以便我们找出错误原因。LISP 的各种调试器有很多差异，不过使用"help"或":help"命令就会给出一些语句帮助。

通常，一个语句是一个原子（例如，一个符号或者整数，或者字符串）或者一个列表，如果换某个语句是原子，LISP 立即解析它。符号解析为它们的值；整数和字符串解析为它们自身。如果语句是一个列表，LISP 视它的第一个元素为函数名；它递归的解析其余的元素，然后将它们的值作为参数来调用这个函数。

例如，如果 LISP 遇到语句 (+ 3 4)，它尝试将 + 作为函数名。然后将 3 解析为 3，4 解析为 4；最后用 3 和 4 作为参数调用 +。LISP 打印出 + 函数的返回值 7。

顶级循环还提供了一些其它的便利；一个特别方便的地方就是获取以前输入的语句的结果。LISP 总会保存最近三个结果；它将它们保存在 *，** 和 *** 三个符号的值中，例如：

```
> 3
3
> 4
4
> 5
5
> ***
3
> ***
4
> ***
5
> **
4
> *
4
```

Special forms

有几个特殊语句看起来像函数调用，但其实不是。这里面包括流程控制语句，例如 if 语句和 do loops；赋值语句，例如 setq，setf，push 和 pop；定义语句，例如 defun 和 defstruct；以及绑定构造，如 let。（这里没有提及所有的特殊语句。我们继续。）

一个很有用的特殊语句是 quote：quote 取消其参数的绑定状态。例如：

```
> (setq a 3)
3
> a
3
> (quote a)
A
> 'a                              ;'a is an abbreviation for (quote a)
A
```

另一个类似的语句是 fuction：function 使得解释器将其参数视为一个函数而不是解析值，例如：

```
> (setq + 3)
3
> +
3
> '+
+
> (function +)
#<Function + @ #x-fbef9de>
> #'+                             ;#'+ is an abbreviation for (function +)
#<Function + @ #x-fbef9de>
```

当我们需要将一个函数作为参数传递给另一个函数时会用到 function 语句。后面有些示例函数将函数作为参数。

Binding

绑定是词汇作用域赋值（汗，怎么读怎么别扭--译者）。每当函数调用时，它就发生于函数的参数列变量中：形式参数被取代为调用函数时的实际参数。你可以在程序中随处绑定变量，就像下面这样：

```
(let ((var1 val1)
      (var2 val2)
      ...)
  body)
```

let 把 val1 绑定到 var1，把 val2 绑定到 var2，依次类推；然后在它的程序体中执行语句。let 的程序体与函数体的执行规则完全相同。例如：

```
> (let ((a 3)) (+ a 1))
4
> (let ((a 2)
        (b 3)
        (c 0))
    (setq c (+ a b))
    c)
5
> (setq c 4)
4
> (let ((c 5)) c)
5
> c
4
```

你可以用 (let (a b) ...) 代替 (let ((a nil) (b nil)) ...) 。

val1，val2 等等。在 let 内部不能引用 var1，var2 等等 let 正在绑定的成员。例如（简而言之，在参数表中，形式参数之间不能互相引用--译者）：

```
> (let ((x 1)
        (y (+ x 1)))
    y)
```

Error: Attempt to take the value of the unbound symbol X

如果符号 x 已经有了一个全局值，会产生一些奇怪的结果：

```
> (setq x 7)
7
> (let ((x 1)
        (y (+ x 1)))
    y)
8
```

let* 语句类似于 let，但它允许引用之前在 let* 中定义的变量的值。例如：

```
> (setq x 7)
7
> (let* ((x 1)
         (y (+ x 1)))
    y)
2
```

语句

```
(let* ((x a)
       (y b))
```

```
   ...)
```

等价于

```
(let ((x a))
  (let ((y b))
    ...))
```

Dynamic Scoping

与我们在 C 或 Pascal 中编写程序不同，let 和 let* 语句提供了词汇作用域。动态作用域是我们在 BASIC 里用的那种：如果我们给一个动态作用域变量赋了值，那么所有对这个变量的访问都会取得这个值，直到给同一个变量赋了另一个值为止。

在 LISP 中，动态作用域变量被称为特化变量 。你可以用 special 语句 defvar 定义一个特化变量。这里有一些词汇化和动态作用域变量的示例。

在以下示例中，check-regular 函数引用一个 regular （也就是一个词汇作用域）变量。因为 check-regular 在绑定 regular 的 let 外部词汇化，check-regular 返回变量的全局值。

```
> (setq regular 5)
5
> (defun check-regular () regular)
CHECK-REGULAR
> (check-regular)
5
> (let ((regular 6)) (check-regular))
5
```

在这个例子中，函数 check-special 引用了一个特化（动态作用域）变量。因此 check-special 调用临时发生于特化绑定的 let 内部，check-special 返回变量的局部值。

```
> (defvar *special* 5)
*SPECIAL*
> (defun check-special () *special*)
CHECK-SPECIAL
> (check-special)
5
> (let ((*special* 6)) (check-special))
6
```

方便起见，特化变量以一个 * 开始和结束。特化变量主要用于全局变量，因为程序员通常期望局部变量使用词汇作用域，全局变量使用动态作用域。

词汇和动态作用域的更多差异参见《Common LISP: the Language》 。

Arrays

make-array 函数定义一个数组。aref 函数访问它的元素。所有元素初始化为 nil。例如：

```
> (make-array '(3 3))
#2a((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
> (aref * 1 1)
NIL
> (make-array 4)                    ;1D arrays don't need the extra parens
#(NIL NIL NIL NIL)
```

数组索引从 0 开始。

以后讨论如何设置一个数组的元素。

Strings

字符串是包含在双引号之间的字符串。LISP 用字符串代表一个变长字符数组。我们可以用一个反斜杠加一个双引号来表示字符串中的双引号，两个反斜杠表示一个单独的反斜杠。例如：

```
"abcd" has 4 characters
"\"" has 1 character
"\\" has 1 character
```

有一些函数可以用于操作字符串：

```
> (concatenate 'string "abcd" "efg")
"abcdefg"
> (char "abc" 1)
#\b                              ;LISP writes characters preceded by #\
> (aref "abc" 1)
#\b                              ;remember, strings are really arrays
```

concatenate 函数可以用于连接任何类型的序列：

```
> (concatenate 'string '(#\a #\b) '(#\c))
"abc"
> (concatenate 'list "abc" "de")
(#\a #\b #\c #\d #\e)
> (concatenate 'vector '#(3 3 3) '#(3 3 3))
#(3 3 3 3 3 3)
```

Structures

LISP 结构类似于 C 结构或 Pasacal 记录。如下所示：

```
> (defstruct foo
    bar
    baaz
    quux)
FOO
```

这个示例定义了一个名为 foo 的数据类型，它是一个带有三个字段的结构。它同时定义了 4 个函数用于操作这个数据类型：make-foo，foo-ba，foo-baaz 和 foo-quux。第一个函数构造了一个 foo 类型的新对象，另外三个用于访问一个 foo 类型的对象。以下是这些函数的用法：

```
> (make-foo)
#s(FOO :BAR NIL :BAAZ NIL :QUUX NIL)
> (make-foo :baaz 3)
#s(FOO :BAR NIL :BAAZ 3 :QUUX NIL)
> (foo-bar *)
NIL
> (foo-baaz **)
3
```

make-foo 函数可以使用 foo 结构类型的字段作为关键字参数。字段访问函数以 foo 类型的结构为参数，返回对应的字段。

以后讨论如何设置一个结构的字段。

Setf

LISP 确认语句的时间，自然就会定义一个内存区域。例如，如果 x 的值是 foo 类型的一个结构，(foo-bar x)定义 x 中 bar 字段的值。或者，如果 y 是一个一维数组，(aref y 2)定义 y 的第三个元素。

setf 语句用于在内存中将它的第一个参数定位到第二个参数上，并且返回内存区域的结果值。例如：

```
> (setq a (make-array 3))
#(NIL NIL NIL)
> (aref a 1)
NIL
> (setf (aref a 1) 3)
3
> a
#(NIL 3 NIL)
> (aref a 1)
3
> (defstruct foo bar)
FOO
> (setq a (make-foo))
#s(FOO :BAR NIL)
> (foo-bar a)
NIL
> (setf (foo-bar a) 3)
3
> a
#s(FOO :BAR 3)
> (foo-bar a)
3
```

setf 是给结构的字段或者数组的元素赋值的唯一方法。

这里还有一些关于 setf 和相关函数的示例。

```
> (setf a (make-array 1))        ;setf on a variable is equivalent to setq
#(NIL)
> (push 5 (aref a 1))            ;push can act like setf
(5)
> (pop (aref a 1))               ;so can pop
5
> (setf (aref a 1) 5)
5
> (incf (aref a 1))              ;incf reads from a place, increments,
6                                ;and writes back
> (aref a 1)
6
```

Booleans and Conditionals

LISP 用自解析符号 nil 来代表 false。任何 nil 之外的其它东西都代表 true。除非有什么特别的原因，我们总是使用 true 的标准自解析符号 t。

LISP 提供了一个逻辑运算的标准函数集，例如，与、或、非。与和或支持短路算法：在遇到第一个 nil 后不再解析右面的参数，而与在遇到第一个 t 后也不再解析右面的 参数。

LISP 还为条件控制提供了几个特殊语句。最简单的是 if。if 语句的第一个参数决定了执行第二或第三个参数中的哪一个：

```
> (if t 5 6)
5
> (if nil 5 6)
6
```

```
> (if 4 5 6)
5
```

如果你需要在 then 或 else 块中放置超过一行的语句，可以使用特殊语句 progn。Progn 在它的程序体内依次执行每一条语句，返回最后一个结果。

```
> (setq a 7)
7
> (setq b 0)
0
> (setq c 5)
5
> (if (> a 5)
    (progn
      (setq a (+ b 7))
      (setq b (+ c 8)))
    (setq b 4))
13
```

一个没有 else 或 when 的语句可以写为 when 或 unless 语句。

```
> (when t 3)
3
> (when nil 3)
NIL
> (unless t 3)
NIL
> (unless nil 3)
3
```

与 if 不同，when 和 unless 允许在程序体内放置任意多的语句。（例如，(when x a b c) 等价于 (if x (progn a b c))）

```
> (when t
    (setq a 5)
    (+ a 6))
11
```

更复杂的条件控制可以用 cond 语句定义实现，它定价于一个 if ... else if ... fi 条件。

一个 cond 由 symbol 符号和其后的若干条件分支组成，每一个分支是一个 list。 cond 分支的第一个元素是条件；其它的元素（如果有的话）是动作。cond 语句查找第一个条件解析为 true 的分支，执行其响应动作并返回结果值。其它条件不会被解析，除了这个响应之外的分支也不会执行。例如：

```
> (setq a 3)
3
> (cond
    ((evenp a) a)               ;if a is even return a
    ((> a 7) (/ a 2))           ;else if a is bigger than 7 return a/2
    ((< a 5) (- a 1))           ;else if a is smaller than 5 return a-1
    (t 17))                     ;else return 17
2
```

如果选定的分支没有响应动作，cond 返回条件的解析结果。

```
> (cond ((+ 3 4)))
7
```

这里用 cond 巧妙的实现了一个递归函数。你可能会有兴趣证明它对于所有的整数 x 都少有个终结。（如果你成功了，请发表出来。）

```
> (defun hotpo (x steps)          ;hotpo stands for Half Or Triple Plus One
    (cond
      ((= x 1) steps)
      ((oddp x) (hotpo (+ 1 (* x 3)) (+ 1 steps)))
      (t (hotpo (/ x 2) (+ 1 steps)))))
A
> (hotpo 7 0)
16
```

LISP 的 case 语句类似于 C 的 switch 语句：

```
> (setq x 'b)
B
> (case x
    (a 5)
    ((d e) 7)
    ((b f) 3)
    (otherwise 9))
3
```

末尾的 otherwise 语句意味着 x 不是 a、b、c、d、e、f，case 语句将会返回 9。

Iteration

LISP 中最简单的迭代结构是 loop：loop 结构反复执行它的程序体直到到达一个返回语句，例如：

```
> (setq a 4)
4
> (loop
    (setq a (+ a 1))
    (when (> a 7) (return a)))
8
> (loop
    (setq a (- a 1))
    (when (< a 3) (return)))
NIL
```

dolist 是下一个最简单的：dolist 把一个变量依次绑定到一个列表的各个元素上，在到达列表结尾时结束。

```
> (dolist (x '(a b c)) (print x))
A
B
C
NIL
```

Dolist 总是返回 nil。注意上例中的 x 永远不会为 nil：C 后面的 NIL 是 dolist 返回的，它被读取-解析-打印循环所打印。

更复杂的迭代称为 do。do 语句的示例如下：

```
> (do ((x 1 (+ x 1))
       (y 1 (* y 2)))
      ((> x 5) y)
    (print y)
    (print 'working))
1
WORKING
2
```

```
WORKING
4
WORKING
8
WORKING
16
WORKING
32
```

do 的第一步是指定绑定的变量，它们的初始值，以及如何更新。第二步是指定一个终止条件和返回值。最后是程序体。do 语句像 let 一样把它的变量绑定到初始值，然后校验终止条件。条件为 false 时，它重复执行程序体；当条件为 true，它返回 return-value 语句的值。

do* 语句之于 do 如同 let* 之于 let。

Non-local Exits

前一节中迭代示例里的 return 语句是一个无定位返回的示例，另一个是 return-from，它从包围它的函数中返回指定值。

```
> (defun foo (x)
    (return-from foo 3) x)
FOO
> (foo 17)
3
```

实际上，return-from 语句可以从任何已命名的语句块中退出--只是默认情况下函数是唯一的命名语句块而已。我们可以用 block 语句自己定义一个命名语句块。

```
> (block foo
    (return-from foo 7)
    3)
7
```

return 语句可以从任何 nil 命名的语句块中返回。默认情况下循环是 nil 命名，而我们可以创建自己的 nil 标记语句块。

```
> (block nil
    (return 7)
    3)
7
```

另外一个无定位退出语句是 error 语句：

```
> (error "This is an error")
Error: This is an error
```

error 语句格式化它的参数，然后进入调试器。

早先我承诺介绍一些可以将函数作为参数调用函数，它们在这里：

```
> (funcall #'+ 3 4)
7
> (apply #'+ 3 4 '(3 4))
14
> (mapcar #'not '(t nil t nil t nil))
(NIL T NIL T NIL T)
```

funcall 用它的其它参数调用它的第一个参数。

Apply 和 funcall 很相像，不过它的最后一个参数可以是一个列表；这个列表被看作是 funcall 的附加参数。

mapcar 的第一个参数必须是一个单参数的函数；mapcar 在列表上逐个元素应用该函数，并将返回值包含如另一个链表。

Funcall 和 apply 主要用于第一个参数是变量的场合。例如，搜索引擎把一个启发式函数作为参数，在一个状态描述上应用 funcall 或者 apply。

Mapcar 配合匿名函数（后面介绍），可以取代很多循环。

Lambda

如果你只是想创建一个临时函数，并不想给它一个命名，lambda 如你所愿。

```
> #'(lambda (x) (+ x 3))
(LAMBDA (X) (+ X 3))
> (funcall * 5)
8
```

lambda 和 mapcar 的组合可以取代很多循环，例如，如下的两个语句是等价的：

```
> (do ((x '(1 2 3 4 5) (cdr x))
       (y nil))
      ((null x) (reverse y))
    (push (+ (car x) 2) y))
(3 4 5 6 7)
> (mapcar #'(lambda (x) (+ x 2)) '(1 2 3 4 5))
(3 4 5 6 7)
```

Sorting

LISP 提供两种主要的排序：排序和稳定排序。

```
> (sort '(2 1 5 4 6) #'<)
(1 2 4 5 6)
> (sort '(2 1 5 4 6) #'>)
(6 5 4 2 1)
```

sort 的第一个参数是一个列表，第二个是一个比较函数。sort 函数不保证稳定性：如果这里有 a 和 b 两个元素，(and (not (< a b)) (not (< b a)))，sort 可能会改变它们的位置。stable-sort（稳定排序）与 sort 非常像，不过它确保两个相等的元素在排序后的列表中的顺序与排序前列表中的顺序相同。

注意：sort 可能会破坏它的参数，如果原始序列对我们很重要，最好使用 copy-list 或 copy-seq 函数创建一个副本。

Equality

关于相等，LISP 有很多不同的观点。数值相等意味着=。两个符号当且仅当他们完全一样时相同。同一个列表的两个副本不相同，但是它们相等。

```
> (eq 'a 'a)
T
> (eq 'a 'b)
NIL
> (= 3 4)
NIL
> (eq '(a b c) '(a b c))
```

```
NIL
> (equal '(a b c) '(a b c))
T
> (eql 'a 'a)
T
> (eql 3 3)
T
```

谓词 eql 代表相等，它对于符号表示相同，对于数值表示=。

```
> (eql 2.0 2)
NIL
> (= 2.0 2)
T
> (eq 12345678901234567890 12345678901234567890)
NIL
> (= 12345678901234567890 12345678901234567890)
T
> (eql 12345678901234567890 12345678901234567890)
T
```

equal 谓词对于符号和数值是相等。当且仅当两个 conse 的 car 和 cdr 都相等时它们才是相等的。当且仅当两个结构是同类型而且各字段都相等时它们相等。

Some Useful List Functions

这些函数都用来操作列表。

```
> (append '(1 2 3) '(4 5 6))      ;concatenate lists
(1 2 3 4 5 6)
> (reverse '(1 2 3))              ;reverse the elements of a list
(3 2 1)
> (member 'a '(b d a c))          ;set membership -- returns the first tail
(A C)                             ;whose car is the desired element
> (find 'a '(b d a c))            ;another way to do set membership
A
> (find '(a b) '((a d) (a d e) (a b d e) ()) :test #'subsetp)
(A B D E)                         ;find is more flexible though
> (subsetp '(a b) '(a d e))       ;set containment
NIL
> (intersection '(a b c) '(b))    ;set intersection
(B)
> (union '(a) '(b))               ;set union
(A B)
> (set-difference '(a b) '(a))    ;set difference
(B)
```

Subsetp，intersection，union 和 set-difference 都允许各个参数包含不匹配的元素 --例如，(subsetp '(a a) '(a b b))可以为 fail。

Find，subsetp，intersection，union 和 set-difference 都可以接受一个 :test 关键字参数；默认情况下，它们是等价的。

Getting Started with Emacs

你可以使用 Emacs 编辑 LISP 代码：Emaces 在打开.lisp 文件时总会自动进入 LISP 模式，不过如果我们的 Emacs 没有成功进入这个状态，可以通过输入 M-x lisp-mode 做到。

我们也可以在 Emacs 下运行 LISP：先确保在我们的私人路径下可以运行一个叫 "LISP" 的命令。例如，我们可以输入：

```
ln -s /usr/local/bin/clisp ~/bin/lisp
```

然后在 Emacs 中输入 M-x run-lip。我们可以向 LISP 发送先前的 LISP 代码，做其它很酷的事情；在 LISP 模式下的任何缓冲输入 C-h m 可以得到进一步的信息。

实际上，我们甚至不需要建立链接。Emacs 有一个变量叫 inferior-lisp-program；所以我们可以把下面这行

```
(setq inferior-lisp-program "/usr/local/bin/clisp")
```

输入到自己的 .emacs 文件中，Emacs 就会知道在你输入 M-x run-lisp 时去哪里寻找 CLISP。