Elisp 入门

叶文彬 (wenbinye@gmail.com)

2007 年 7 月 17 日

前言

emacs 的高手不能不会 elisp。但是对于很多人来说 elisp 学习是一个痛苦的历程，至少我是有这样一段经历。现在，我的 elisp 也算有小成了，所以斗胆写这样文章为后来者提供一点捷径。

目 录

第一章　一个 Hello World 例子

自从 K&R 以来，hello world 程序历来都是程序语言教程的第一个例子。我也用一个 hello world 的例子来演示 emacs 里执行 elisp 的环境。下面就是这个语句：

```
(message "hello world")
```

前面我没有说这个一个程序，这是因为，elisp 不好作为可执行方式来运行（当然也不是不可能），所有的 elisp 都是运行在 emacs 这个环境下。

首先切换到 *scratch* 缓冲区里，如果当前模式不是 lisp-interaction-mode，用 M-x lisp-interaction-mode 先转换到 lisp-interaction-mode。然后输入前面这一行语句。在行尾右括号后，按 C-j 键。如果 Minibuffer 里显示 hello world，光标前一行也显示“hello world”，那说明你的操作没有问题。我们就可以开始 elisp 学习之旅了。

注：elisp 里的一个完整表达式，除了简单数据类型（如数字，向量），都是用括号括起来，称为一个 S-表达式。让 elisp 解释器执行一个 S-表达式除了前一种方法之外，还可以用 C-x C-e。它们的区别是，C-x C-e 是一个全局按键绑定，几乎可以在所有地方都能用。它会将运行返回值显示在 Minibuffer 里。这里需要强调一个概念是返回值和作用是不同的。比如前面 message 函数它的作用是在 Minibuffer 里显示一个字符串，但是它的返回值是“hello world” 字符串。

第二章　基础知识

这一节介绍一下 elisp 编程中一些最基本的概念，比如如何定义函数，程序的控制结构，变量的使用和作用域等等。

2.1　函数和变量

elisp 中定义一个函数是用这样的形式：

```
(defun function-name (arguments-list)
  "document string"
  body)
```

比如：

```
(defun hello-world (name)
  "Say hello to user whose name is NAME."
  (message "Hello, %s" name))
```

其中函数的文档字符串是可以省略的。但是建议为你的函数（除了最简单，不作为接口的）都加上文档字符串。这样将来别人使用你的扩展或者别人阅读你的代码或者自己进行维护都提供很大的方便。

在 emacs 里，当光标处于一个函数名上时，可以用 C-h f 查看这个函数的文档。比如前面这个函数，在 `*Help*` 缓冲区里的文档是：

```
hello-world is a Lisp function.
(hello-world name)

Say hello to user whose name is name.
```

如果你的函数是在文件中定义的。这个文档里还会给出一个链接能跳到定义的地方。

要运行一个函数，最一般的方式是：

```
(function-name arguments-list)
```

比如前面这个函数：

```
(hello-world "Emacser")                 ; => "Hello, Emacser"
```

每个函数都有一个返回值。这个返回值一般是函数定义里的最后一个表达式的值。

elisp 里的变量使用无需象 C 语言那样需要声明，你可以用 setq 直接对一个变量赋值。

```
(setq foo "I'm foo")                    ; => "I'm foo"
(message foo)                           ; => "I'm foo"
```

和函数一样，你可以用 C-h v 查看一个变量的文档。比如当光标在 foo 上时用 C-h v 时，文档是这样的：

```
foo's value is "I'm foo"


Documentation:
Not documented as a variable.
```

有一个特殊表达式（ special form）defvar，它可以声明一个变量，一般的形式是：

```
(defvar variable-name value
  "document string")
```

它与 setq 所不同的是，如果变量在声明之前，这个变量已经有一个值的话，用 defvar 声明的变量值不会改变成声明的那个值。另一个区别是 defvar 可以为变量提供文档字符串，当变量是在文件中定义的话，C-h v 后能给出变量定义的位置。比如：

```
(defvar foo "Did I have a value?"
  "A demo variable")                    ; => foo
foo                                     ; => "I'm foo"
(defvar bar "I'm bar"
  "A demo variable named \"bar\"")      ; => bar
bar                                     ; => "I'm bar"
```

用 C-h v 查看 foo 的文档，可以看到它已经变成：

```
foo's value is "I'm foo"


Documentation:
A demo variable
```

由于 elisp 中函数是全局的，变量也很容易成为全局变量（因为全局变量和局部变量的赋值都是使用 setq 函数），名字不互相冲突是很关键的。所以除了为你的函数和变量选择一个合适的前缀之外，用 C-h f 和 C-h v 查看一下函数名和变量名有没有已经被使用过是很关键的。

2.2 局部作用域的变量

如果没有局部作用域的变量，都使用全局变量，函数会相当难写。elisp 里可以用 let 和`let*`进行局部变量的绑定。let 使用的形式是：

```
(let (bindings)
  body)
```

bingdings 可以是 (var value) 这样对 var 赋初始值的形式，或者用 var 声明一个初始值为 nil 的变量。比如：

```
(defun circle-area (radix)
  (let ((pi 3.1415926)
        area)
```

```
    (setq area (* pi radix radix))
    (message "直径为 %.2f 的圆面积是 %.2f" radix area)))
(circle-area 3)
```

C-h v 查看 area 和 pi 应该没有这两个变量。

let* 和 let 的使用形式完全相同，唯一的区别是在let*声明中就能使用前面声明的变量，比如：

```
(defun circle-area (radix)
  (let* ((pi 3.1415926)
         (area (* pi radix radix)))
    (message "直径为 %.2f 的圆面积是 %.2f" radix area)))
```

2.3 lambda 表达式

可能你久闻 lambda 表达式的大名了。其实依我的理解，lambda 表达式相当于其它语言中的匿名函数。比如 perl 里的匿名函数。它的形式和 defun 是完全一样的：

```
(lambda (arguments-list)
  "documentation string"
  body)
```

调用 lambda 方法如下：

```
(funcall (lambda (name)
           (message "Hello, %s!" name)) "Emacser")
```

你也可以把 lambda 表达式赋值给一个变量，然后用 funcall 调用

```
(setq foo (lambda (name)
            (message "Hello, %s!" name)))
(funcall foo "Emacser")                    ; => "Hello, Emacser!"
```

lambda 表达式最常用的是作为参数传递给其它函数，比如 mapc。

2.4 控制结构

2.4.1 顺序执行

一般来说程序都是按表达式顺序依次执行的。这在 defun 等特殊环境中是自动进行的。但是一般情况下都不是这样的。比如你无法用 eval-last-sexp 同时执行两个表达式，在 if 表达式中的条件为真时执行的部分也只能运行一个表达式。这时就需要用 progn 这个特殊表达式。它的使用形式如下：

```
(progn A B C ...)
```

它的作用就是让表达式 A, B, C 顺序执行。比如：

```
(progn
  (setq foo 3)
  (message "Square of %d is %d" foo (* foo foo)))
```

2.4.2 条件判断

elisp 有两个最基本的条件判断表达式 if 和 cond。使用形式分别如下：

```
(if condition
    then
  else)

(cond (case1 do-when-case1)
      (case2 do-when-case2)
      ...
      (t do-when-none-meet))
```

使用的例子如下：

```
(defun my-max (a b)
  (if (> a b)
      a b))
(my-max 3 4)                          ; => 4

(defun fib (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2))))))
(fib 10)                              ; => 55
```

还有两个宏 when 和 unless，从它们的名字也就能知道它们是作什么用的。使用这两个宏的好处是使代码可读性提高，when 能省去 if 里的 progn 结构，unless 省去条件为真子句需要的的 nil 表达式。

2.4.3 循环

循环使用的是 while 表达式。它的形式是：

```
(while condition
  body)
```

比如：

```
(defun factorial (n)
  (let ((res 1))
    (while (> n 1)
      (setq res (* res n)
```

```
            n (- n 1)))
    res))
(factorial 10)                          ; => 3628800
```

2.5 逻辑运算

条件的逻辑运算和其它语言都是很类似的，使用 and、or、not。and 和 or 也同样具有短路性质。很多人喜欢在表达式短时，用 and 代替 when，or 代替unless。当然这时一般不关心它们的返回值，而是在于表达式其它子句的副作用。比如 or 经常用于设置函数的缺省值，而 and 常用于参数检查：

```
(defun hello-world (&optional name)
  (or name (setq name "Emacser"))
  (message "Hello, %s" name))           ; => hello-world
(hello-world)                           ; => "Hello, Emacser"
(hello-world "Ye")                      ; => "Hello, Ye"

(defun square-number-p (n)
  (and (>= n 0)
       (= (/ n (sqrt n)) (sqrt n))))
(square-number-p -1)                    ; => nil
(square-number-p 25)                    ; => t
```

2.6 函数列表

```
(defun NAME ARGLIST [DOCSTRING] BODY...)
(defvar SYMBOL &optional INITVALUE DOCSTRING)
(setq SYM VAL SYM VAL ...)
(let VARLIST BODY...)
(let* VARLIST BODY...)
(lambda ARGS [DOCSTRING] [INTERACTIVE] BODY)
(progn BODY ...)
(if COND THEN ELSE...)
(cond CLAUSES...)
(when COND BODY ...)
(unless COND BODY ...)
(when COND BODY ...)
(or CONDITIONS ...)
(and CONDITIONS ...)
(not OBJECT)
```

第三章　基本数据类型之一—数字

elisp 里的对象都是有类型的，而且每一个对象它们知道自己是什么类型。你得到一个变量名之后可以用一系列检测方法来测试这个变量是什么类型（好像没有什么方法来让它说出自己是什么类型的）。内建的 emacs 数据类型称为 primitive types，包括整数、浮点数、cons、符号（symbol）、字符串、向量（vector）、散列表（hash-table）、subr（内建函数，比如 cons, if, and 之类）、byte-code function，和其它特殊类型，例如缓冲区（ buffer）。

在开始前有必要先了解一下读入语法和输出形式。所谓读入语法是让 elisp 解释器明白输入字符所代表的对象，你不可能让 elisp 读入.#@!? 这样奇怪的东西还能好好工作吧（ perl 好像经常要受这样的折磨:)）。简单的来说，一种数据类型有（也可能没有，比如散列表）对应的规则来让解释器产生这种数据类型，比如 123 产生整数 123，(a . b) 产生一个 cons。所谓输出形式是解释器用产生一个字符串来表示一个数据对象。比如整数 123 的输出形式就是 123，cons cell (a . b) 的输出形式是(a . b)。与读入语法不同的是，数据对象都有输出形式。比如散列表的输出可能是这样的：

```
#<hash-table 'eql nil 0/65 0xa7344c8>
```

通常一个对象的数据对象的输出形式和它的读入形式都是相同的。现在就先从简单的数据类型—数字开始吧。

emacs 的数字分为整数和浮点数（和 C 比没有双精度数 double）。1，1.，+1, -1, 536870913, 0, -0 这些都是整数。整数的范围是和机器是有关的，一般来最小范围是在-268435456 to 268435455（29 位，-2**28　2**28-1）。可以从 most-positive-fixnum 和 most-negative-fixnum 两个变量得到整数的范围。

你可以用多种进制来输入一个整数。比如：

```
#b101100 => 44       ; 二进制
#o54 => 44           ; 八进制
#x2c => 44           ; 十进制
```

最神奇的是你可以用 2 到 36 之间任意一个数作为基数，比如：

```
#24r1k => 44         ; 二十四进制
```

之所以最大是 36，是因为只有 0-9 和 a-z 36 个字符来表示数字。但是我想基本上不会有人会用到 emacs 的这个特性。

1500.0, 15e2, 15.0e2, 1.5e3, 和.15e4 都可以用来表示一个浮点数 1500.。遵循 IEEE 标准，elisp 也有一个特殊类型的值称为 NaN (not-a-number)。你可以用=(/ 0.0 0.0)= 产生这个数。

3.1　测试函数

整数类型测试函数是 integerp，浮点数类型测试函数是 floatp。数字类型测试用 numberp。你可以分别运行这几个例子来试验一下：

```
(integerp 1.)                           ; => t
```

```
(integerp 1.0)                          ; => nil
(floatp 1.)                             ; => nil
(floatp -0.0e+NaN)                      ; => t
(numberp 1)                             ; => t
```

还提供一些特殊测试，比如测试是否是零的 zerop，还有非负整数测试的 wholenump。

注：elisp 测试函数一般都是用 p 来结尾，p 是 predicate 的第一个字母。如果函数名是一个单词，通常只是在这个单词后加一个 p，如果是多个单词，一般是加 -p。

3.2 数的比较

常用的比较操作符号是我们在其它言中都很熟悉的，比如<，>，>=，<=，不一样的是，由于赋值是使用 set 函数，所以 = 不再是一个赋值运算符了，而是测试数字相等符号。和其它语言类似，对于浮点数的相等测试都是不可靠的。比如：

```
(setq foo (- (+ 1.0 1.0e-3) 1.0))       ; => 0.0009999999999998899
(setq bar 1.0e-3)                       ; => 0.001
(= foo bar)                             ; => nil
```

所以一定要确定两个浮点数是否相同，是要在一定误差内进行比较。这里给出一个函数：

```
(defvar fuzz-factor 1.0e-6)
(defun approx-equal (x y)
  (or (and (= x 0) (= y 0))
      (< (/ (abs (- x y))
            (max (abs x) (abs y)))
         fuzz-factor)))
(approx-equal foo bar)                  ; => t
```

还有一个测试数字是否相等的函数 eql，这是函数不仅测试数字的值是否相等，还测试数字类型是否一致，比如：

```
(= 1.0 1)                               ; => t
(eql 1.0 1)                             ; => nil
```

elisp 没有+=，-=，/=，*= 这样的命令式语言里常见符号，如果你想实现类似功能的语句，只能用赋值函数 setq 来实现了。/= 符号被用来作为不等于的测试了。

3.3 数的转换

整数向浮点数转换是通过 float 函数进行的。而浮点数转换成整数有这样几个函数：

- truncate 转换成靠近 0 的整数
- floor 转换成最接近的不比本身大的整数

- ceiling 转换成最接近的不比本身小的整数
- round 四舍五入后的整数，换句话说和它的差绝对值最小的整数

很晕是吧。自己用 1.2, 1.7, -1.2, -1.7 对这四个函数操作一遍就知道区别了（可以直接看 info。按键顺序是 C-h i m elisp RET m Numeric Conversions RET。以后简写成 info elisp - Numeric Conversions）。

这里提一个问题，浮点数的范围是无穷大的，而整数是有范围的，如果用前面的函数转换 1e20 成一个整数会出现什么情况呢？试试就知道了。

3.4 数的运算

四则运算没有什么好说的，就是 `+ - * /`。值得注意的是，和 C 语言类似，如果参数都是整数，作除法时要记住(/ 5 6) 是会等于 0 的。如果参数中有浮点数，整数会自动转换成浮点数进行运算，所以(/ 5 6.0) 的值才会是 5/6。

没有++ 和-- 操作了，类似的两个函数是 1+ 和 1-。可以用 setq 赋值来代替++ 和--：

```
(setq foo 10)                          ; => 10
(setq foo (1+ foo))                    ; => 11
(setq foo (1- foo))                    ; => 10
```

注：可能有人看过有 incf 和 decf 两个实现 ++ 和 -- 操作。这两个宏是可以用的。这两个宏是 Common Lisp 里的，emacs 有模拟的 Common Lisp 的库 cl。但是 RMS 认为最好不要使用这个库。但是你可以在你的 elisp 包中使用这两个宏，只要在文件头写上：

```
(eval-when-compile
  (require 'cl))
```

由于 incf 和 decf 是两个宏，所以这样写不会在运行里导入 cl 库。有点离题是，总之一句话，教主说不好的东西，我们最好不要用它。其它无所谓，只可惜了两个我最常用的函数 remove-if 和 remove-if-not。不过如果你也用 emms 的话，可以在 emms-compat 里找到这两个函数的替代品。

abs 取数的绝对值。

有两个取整的函数，一个是符号 `%`，一个是函数 mod。这两个函数有什么差别呢？一是 `%` 的第个参数必须是整数，而 mod 的第一个参数可以是整数也可以是浮点数。二是即使对相同的参数，两个函数也不一定有相同的返回值：

```
(+ (% DIVIDEND DIVISOR)
   (* (/ DIVIDEND DIVISOR) DIVISOR))
```

和 DIVIDEND 是相同的。而：

```
(+ (mod DIVIDEND DIVISOR)
   (* (floor DIVIDEND DIVISOR) DIVISOR))
```

和 DIVIDEND 是相同的。

三角运算有函数：sin, cos, tan, asin, acos, atan。开方函数是 sqrt。

exp 是以 e 为底的指数运算，expt 可以指定底数的指数运算。log 默认底数是 e，但是也可以指定底数。log10 就是(log x 10)。logb 是以 2 为底数运算，但是返回的是一个整数。这个函数是用来计算数的位。

random 可以产生随机数。可以用(random t) 来产生一个新种子。虽然 emacs 每次启动后调用 random 总是产生相同的随机数，但是运行过程中，你不知道调用了多少次，所以使用时还是不需要再调用一次(random t) 来产生新的种子。

位运算这样高级的操作我就不说了，自己看 info elisp - Bitwise Operations on Integers 吧。

3.5 函数列表

```
;; 测试函数
(integerp OBJECT)
(floatp OBJECT)
(numberp OBJECT)
(zerop NUMBER)
(wholenump OBJECT)
;; 比较函数
(> NUM1 NUM2)
(< NUM1 NUM2)
(>= NUM1 NUM2)
(<= NUM1 NUM2)
(= NUM1 NUM2)
(eql OBJ1 OBJ2)
(/= NUM1 NUM2)
;; 转换函数
(float ARG)
(truncate ARG &optional DIVISOR)
(floor ARG &optional DIVISOR)
(ceiling ARG &optional DIVISOR)
(round ARG &optional DIVISOR)
;; 运算
(+ &rest NUMBERS-OR-MARKERS)
(- &optional NUMBER-OR-MARKER &rest MORE-NUMBERS-OR-MARKERS)
(* &rest NUMBERS-OR-MARKERS)
(/ DIVIDEND DIVISOR &rest DIVISORS)
(1+ NUMBER)
(1- NUMBER)
(abs ARG)
(% X Y)
(mod X Y)
(sin ARG)
(cos ARG)
```

```
(tan ARG)
(asin ARG)
(acos ARG)
(atan Y &optional X)
(sqrt ARG)
(exp ARG)
(expt ARG1 ARG2)
(log ARG &optional BASE)
(log10 ARG)
(logb ARG)
;; 随机数
(random &optional N)
```

3.6 变量列表

```
most-positive-fixnum
most-negative-fixnum
```

第四章　基本数据类型之二——字符和字符串

在 emacs 里字符串是有序的字符数组。和 c 语言的字符串数组不同，emacs 的字符串可以容纳任何字符，包括 \0:

```
(setq foo "abc\000abc")                 ; => "abc^@abc"
```

关于字符串有很多高级的属性，例如字符串的表示有单字节和多字节类型，字符串可以有文本属性（ text property）等等。但是对于刚接触字符串，还是先学一些基本操作吧。

首先构成字符串的字符其实就是一个整数。一个字符 'A' 就是一个整数 65。但是目前字符串中的字符被限制在 0-524287 之间。字符的读入语法是在字符前加上一个问号，比如 ?A 代表字符 'A'。

```
?A                                      ; => 65
?a                                      ; => 97
```

对于标点来说，也可以用同样的语法，但是最好在前面加上转义字符 '\'，因为有些标点会有岐义，比如 '?—('。'\' 必须用 '?\\' 表示。控制字符，退格、制表符，换行符，垂直制表符，换页符，空格，回车，删除和 escape 表示为 '?\a', '?\b', '?\t', '?\n', '?\v', '?\f', '?\s', '?\r', '?\d', 和'?\e'。对于没有特殊意义的字符，加上转义字符\ 是没有副作用的，比如 '?\+' 和 '?+' 是完全一样的。所以标点还是都用转义字符来表示吧。

```
?\a => 7                ; control-g, 'C-g'
?\b => 8                ; backspace, <BS>, 'C-h'
?\t => 9                ; tab, <TAB>, 'C-i'
?\n => 10               ; newline, 'C-j'
?\v => 11               ; vertical tab, 'C-k'
?\f => 12               ; formfeed character, 'C-l'
?\r => 13               ; carriage return, <RET>, 'C-m'
?\e => 27               ; escape character, <ESC>, 'C-['
?\s => 32               ; space character, <SPC>
?\\ => 92               ; backslash character, '\'
?\d => 127              ; delete character, <DEL>
```

控制字符可以有多种表示方式，比如 C-i，这些都是对的：

```
?\^I  ?\^i  ?\C-I  ?\C-i
```

它们都对应数字 9。 meta 字符是用¡META¿ 修饰键（通常就是 Alt 键）输入的字符。之所以称为修饰键，是因为这样输入的字符就是在其修饰字符的第 27 位由 0 变成 1 而成，也就是如下操作：

```
(logior (lsh 1 27) ?A)                  ; => 134217793
?\M-A                                   ; => 134217793
```

你可以用'\M-' 代表 meta 键，加上修饰的字符就是新生成的字符。比如：?\M-A, ?\M-\C-b. 后面这个也可以写成?\C-\M-b。

如果你还记得前面说过字符串里的字符不能超过 524287 的话，这就可以看出字符串是不能放下一个 meta 字符的。所以按键序列在这时只能用 vector 来储存。

其它的修饰键也是类似的。emacs 用 2**25 位来表示 shift 键，2**24 对应 hyper，2**23 对应 super，2**22 对应 alt。

4.1 测试函数

字符串测试使用 stringp，没有 charp，因为字符就是整数。 string-or-null-p 当对象是一个字符或 nil 时返回 t。char-or-string-p 测试是否是字符串或者字符类型。比较头疼的是 emacs 没有测试字符串是否为空的函数。这是我用的这个测试函数，使用前要测试字符串是否为 nil：

```
(defun string-emptyp (str)
  (not (string< "" str)))
```

4.2 构造函数

产生一个字符串可以用 make-string。这样生成的字符串包含的字符都是一样的。要生成不同的字符串可以用 string 函数。

```
(make-string 5 ?x)                    ; => "xxxxx"
(string ?a ?b ?c)                     ; => "abc"
```

在已有的字符串生成新的字符串的方法有 substring, concat。substring 的后两个参数是起点和终点的位置。如果终点越界或者终点比起点小都会产生一个错误。这个在使用 substring 时要特别小心。

```
(substring "0123456789" 3)            ; => "3456789"
(substring "0123456789" 3 5)          ; => "34"
(substring "0123456789" -3 -1)        ; => "78"
```

concat 函数相对简单，就是把几个字符串连接起来。

4.3 字符串比较

char-equal 可以比较两个字符是否相等。与整数比较不同，这个函数还考虑了大小写。如果 case-fold-search 变量是 t 时，这个函数的字符比较是忽略大小写的。编程时要小心，因为通常 case-fold-search 都是 t，这样如果要考虑字符的大小写时就不能用 char-equal 函数了。

字符串比较使用 string=，string-equal 是一个别名。string¡ 是按字典序比较两个字符串，string-less 是它的别名。空字符串小于所有字符串，除了空字符串。前面 string-emptyp 就是用这个特性。当然直接用 length 检测字符串长度应该也可以，还可以省去检测字符串是否为空。没有 string¿ 函数。

4.4 转换函数

字符转换成字符串可以用 char-to-string 函数，字符串转换成字符可以用 string-to-char。当然只是返回字符串的第一个字符。

数字和字符串之间的转换可以用 number-to-string 和 string-to-number。其中 string-to-number 可以设置字符串的进制，可以从 2 到 16。number-to-string 只能转换成 10 进制的数字。如果要输出八进制或者十六进制，可以用 format 函数：

```
(string-to-number "256")                ; => 256
(number-to-string 256)                  ; => "256"
(format "%#o" 256)                      ; => "0400"
(format "%#x" 256)                      ; => "0x100"
```

如果要输出成二进制，好像没有现成的函数了。calculator 库倒是可以，这是我写的函数：

```
(defun number-to-bin-string (number)
  (require 'calculator)
  (let ((calculator-output-radix 'bin)
        (calculator-radix-grouping-mode nil))
    (calculator-number-to-string number)))
(number-to-bin-string 256)              ; => "100000000"
```

其它数据类型现在还没有学到，不过可以先了解一下吧。concat 可以把一个字符构成的列表或者向量转换成字符串，vconcat 可以把一个字符串转换成一个向量，append 可以把一个字符串转换成一个列表。

```
(concat '(?a ?b ?c ?d ?e))              ; => "abcde"
(concat [?a ?b ?c ?d ?e])               ; => "abcde"
(vconcat "abdef")                       ; => [97 98 100 101 102]
(append "abcdef" nil)                   ; => (97 98 99 100 101 102)
```

大小写转换使用的是 downcase 和 upcase 两个函数。这两个函数的参数既可以字符串，也可以是字符。capitalize 可以使字符串中单词的第一个字符大写，其它字符小写。upcase-initials 只使第一个单词的第一个字符大写，其它字符小写。这两个函数的参数如果是一个字符，那么只让这个字符大写。比如：

```
(downcase "The cat in the hat")         ; => "the cat in the hat"
(downcase ?X)                           ; => 120
(upcase "The cat in the hat")           ; => "THE CAT IN THE HAT"
(upcase ?x)                             ; => 88
(capitalize "The CAT in tHe hat")       ; => "The Cat In The Hat"
(upcase-initials "The CAT in the hAt")  ; => "The CAT In The HAt"
```

4.5 格式化字符串

format 类似于 C 语言里的 printf 可以实现对象的字符串化。数字的格式化和 printf 的参数差不多，值得一提的是"输出形式转换成字符串，这在调试时是很有用的。

4.6 查找和替换

字符串查找的核心函数是 string-match。这个函数可以从指定的位置对字符串进行正则表达式匹配，如果匹配成功，则返回匹配的起点，如：

```
(string-match "34" "01234567890123456789")     ; => 3
(string-match "34" "01234567890123456789" 10) ; => 13
```

注意 string-match 的参数是一个 regexp。emacs 好象没有内建的查找子串的函数。如果你想把 string-match 作为一个查找子串的函数，可以先用 regexp-quote 函数先处理一下子串。比如：

```
(string-match "2*" "232*3=696")                 ; => 0
(string-match (regexp-quote "2*") "232*3=696") ; => 2
```

事实上，string-match 不只是查找字符串，它更重要的功能是捕捉匹配的字符串。如果你对正则表达式不了解，可能需要先找一本书，先了解一下什么是正则表达式。string-match 在查找的同时，还会记录下每个要捕捉的字符串的位置。这个位置可以在匹配后用 match-data、match-beginning 和 match-end 等函数来获得。先看一下例子：

```
(progn
  (string-match "3\\(4\\)" "01234567890123456789")
  (match-data))                          ; => (3 5 4 5)
```

最后返回这个数字是什么意思呢?正则表达式捕捉的字符串按括号的顺序对应一个序号，整个模式对应序号 0，第一个括号对应序号 1，第二个括号对应序号 2，以此类推。所以"3\\(4\\)"这个正则表达式中有序号 0 和 1，最后 match-data 返回的一系列数字对应的分别是要捕捉字符串的起点和终点位置，也就是说子串"34" 起点从位置 3 开始，到位置 5 结束，而捕捉的字符串"4" 的起点是从 4 开始，到 5 结束。这些位置可以用 match-beginning 和 match-end 函数用对应的序号得到。要注意的是，起点位置是捕捉字符串的第一个字符的位置，而终点位置不是捕捉的字符串最后一个字符的位置，而是下一个字符的位置。这个性质对于循环是很方便的。比如要查找上面这个字符串中所有 34 出现的位置：

```
(let ((start 0))
  (while (string-match "34" "01234567890123456789" start)
    (princ (format "find at %d\n" (match-beginning 0)))
    (setq start (match-end 0))))
```

查找会了，就要学习替换了。替换使用的函数是 replace-match。这个函数既可以用于字符串的替换，也可以用于缓冲区的文本替换。对于字符串的替换，replace-match 只是按给定的序号把字符串中的那一部分用提供的字符串替换了而已：

```
(let ((str "01234567890123456789"))
```

```
(string-match "34" str)
(princ (replace-match "x" nil nil str 0))
(princ "\n")
(princ str))
```

可以看出 replace-match 返回的字符串是替换后的新字符串，原字符串被没有改变。

如果你想挑战一下，想想怎样把上面这个字符串中所有的 34 都替换掉？如果想就使用同一个字符串来存储，可能对于固定的字符串，这个还容易一些，如果不是的话，就要花一些脑筋了，因为替换之后，新的字符串下一个搜索起点的位置就不能用 (match-end 0) 给出来的位置了，而是要扣除替换的字符串和被替换的字符串长度的差值。emacs 对字符串的替换有一个函数 replace-regexp-in-string。这个函数的实现方法是把每次匹配部分之前的子串收集起来，最后再把所有字符串连接起来。

单字符的替换有 subst-char-in-string 函数。但是 emacs 没有类似 perl 函数或者程序 tr 那样进行字符替换的函数。只能自己建表进行循环操作了。

4.7 函数列表

```
;; 测试函数
(stringp OBJECT)
(string-or-null-p OBJECT)
(char-or-string-p OBJECT)
;; 构建函数
(make-string LENGTH INIT)
(string &rest CHARACTERS)
(substring STRING FROM &optional TO)
(concat &rest SEQUENCES)
;; 比较函数
(char-equal C1 C2)
(string= S1 S2)
(string-equal S1 S2)
(string< S1 S2)
;; 转换函数
(char-to-string CHAR)
(string-to-char STRING)
(number-to-string NUMBER)
(string-to-number STRING &optional BASE)
(downcase OBJ)
(upcase OBJ)
(capitalize OBJ)
(upcase-initials OBJ)
(format STRING &rest OBJECTS)
;; 查找与替换
```

```
(string-match REGEXP STRING &optional START)
(replace-match NEWTEXT &optional FIXEDCASE LITERAL STRING SUBEXP)
(replace-regexp-in-string REGEXP REP STRING &optional FIXEDCASE LITERAL SUBEXP START)
(subst-char-in-string FROMCHAR TOCHAR STRING &optional INPLACE)
```

第五章　基本数据类型之三—— cons cell 和列表

如果从概念上来说，cons cell 其实非常简单的，就是两个有顺序的元素。第一个叫 CAR，第二个就 CDR。CAR 和 CDR 名字来自于 Lisp。它最初在 IBM 704 机器上的实现。在这种机器有一种取址模式，使人可以访问一个存储地址中的“地址（address）”部分和“减量（decrement）”部分。CAR 指令用于取出地址部分，表示 (Contents of Address part of Register)，CDR 指令用于取出地址的减量部分 (Contents of the Decrement part of Register)。cons cell 也就是 construction of cells。car 函数用于取得 cons cell 的 CAR 部分，cdr 取得 cons cell 的 CDR 部分。cons cell 如此简单，但是它却能衍生出许多高级的数据结构，比如链表，树，关联表等等。cons cell 的读入语法是用 . 分开两个部分，比如：

```
'(1 . 2)                                ; => (1 . 2)
'(?a . 1)                               ; => (97 . 1)
'(1 . "a")                              ; => (1 . "a")
'(1 . nil)                              ; => (1)
'(nil . nil)                            ; => (nil)
```

注意到前面的表达式中都有一个 ' 号，这是什么意思呢？其实理解了 eval-last-sexp 的作用就能明白了。eval-last-sexp 其实包含了两个步骤，一是读入前一个 S-表达式，二是对读入的 S-表达式求值。这样如果读入的 S-表达式是一个 cons cell 的话，求值时会把这个 cons cell 的第一个元素作为一个函数来调用。而事实上，前面这些例子的第一个元素都不是一个函数，这样就会产生一个错误 invalid-function。之所以前面没有遇到这个问题，那是因为前面数字和字符串是一类特殊的 S-表达式，它们求值后和求值前是不变，称为自求值表达式（self-evaluating form）。' 号其实是一个特殊的函数 quote，它的作用是将它的参数返回而不作求值。`'(1 . 2)` 等价于(quote (1 . 2))。为了证明 cons cell 的读入语法确实就是它的输出形式，可以看下面这个语句：

```
(read "(1 . 2)")                        ; => (1 . 2)
```

列表包括了 cons cell。但是列表中有一个特殊的元素——空表 nil。

```
nil                                     ; => nil
'()                                     ; => nil
```

空表不是一个 cons cell，因为它没有 CAR 和 CDR 两个部分，事实上空表里没有任何内容。但是为了编程的方便，可以认为 nil 的 CAR 和 CDR 都是 nil：

```
(car nil)                               ; => nil
(cdr nil)                               ; => nil
```

按列表最后一个 cons cell 的 CDR 部分的类型分，可以把列表分为三类。如果它是 nil 的话，这个列表也称为“真列表”(true list)。如果既不是 nil 也不是一个 cons cell，则这个列表称为“点列表”(dotted list)。还有一种可能，它指向列表中之前的一个 cons cell，则称为环形列表 (circular list)。这里分别给出一个例子：

```
'(1 2 3)                                ; => (1 2 3)
'(1 2 . 3)                              ; => (1 2 . 3)
```

```
'(1 . #1=(2 3 . #1#))                    ; => (1 2 3 . #1)
```

从这个例子可以看出前两种列表的读入语法和输出形式都是相同的，而环形列表的读入语法是很古怪的，输出形式不能作为环形列表的读入形式。

如果把真列表最后一个 cons cell 的 nil 省略不写，也就是 (1 . nil) 简写成 (1)，把 (obj1 . (obj2 . list)) 简写成 (obj1 obj2 . list)，那么列表最后可以写成一个用括号括起的元素列表：

```
'(1 . (2 . (3 . nil)))                   ; => (1 2 3)
```

尽管这样写是清爽多了，但是，我觉得看一个列表时还是在脑子里反映的前面的形式，这样在和复杂的 cons cell 打交道时就不会搞不清楚这个 cons cell 的 CDR 是一个列表呢，还是一个元素或者是嵌套的列表。

5.1 测试函数

测试一个对象是否是 cons cell 用 consp，是否是列表用 listp。

```
(consp '(1 . 2))                         ; => t
(consp '(1 . (2 . nil)))                 ; => t
(consp nil)                              ; => nil
(listp '(1 . 2))                         ; => t
(listp '(1 . (2 . nil)))                 ; => t
(listp nil)                              ; => t
```

没有内建的方法测试一个列表是不是一个真列表。通常如果一个函数需要一个真列表作为参数，都是在运行时发出错误，而不是进行参数检查，因为检查一个列表是真列表的代价比较高。

测试一个对象是否是 nil 用 null 函数。只有当对象是空表时，null 才返回空值。

5.2 构造函数

生成一个 cons cell 可以用 cons 函数。比如：

```
(cons 1 2)                               ; => (1 . 2)
(cons 1 '())                             ; => (1)
```

这也是在列表前面增加元素的方法。比如：

```
(setq foo '(a b))                        ; => (a b)
(cons 'x foo)                            ; => (x a b)
```

值得注意的是前面这个例子的 foo 值并没有改变。事实上有一个宏 push 可以加入元素的同时改变列表的值：

```
(push 'x foo)                            ; => (x a b)
foo                                      ; => (x a b)
```

生成一个列表的函数是 list。比如：

```
(list 1 2 3)                            ; => (1 2 3)
```

可能这时你有一个疑惑，前面产生一个列表，我常用 quote（也就是 ' 符号）这个函数，它和这个 cons 和 list 函数有什么区别呢？其实区别是很明显的，quote 是把参数直接返回不进行求值，而 list 和 cons 是对参数求值后再生成一个列表或者 cons cell。看下面这个例子：

```
'((+ 1 2) 3)                            ; => ((+ 1 2) 3)
(list (+ 1 2) 3)                        ; => (3 3)
```

前一个生成的列表的 CAR 部分是(+ 1 2) 这个列表，而后一个是先对(+ 1 2) 求值得到 3 后再生成列表。

> 思考题
> 如果你觉得你有点明白的话，我提一个问题考考你：怎样用 list 函数构造一个(a b c) 这样的列表呢？

前面提到在列表前端增加元素的方法是用 cons，在列表后端增加元素的函数是用 append。比如：

```
(append '(a b) '(c))                    ; => (a b c)
```

append 的功能可以认为它是把第一个参数最后一个列表的 nil 换成第二个参数，比如前面这个例子，第一个参数写成 cons cell 表示方式是 (a . (b . nil))，把这个 nil 替换成 (c) 就成了 (a . (b . (c)))。对于多个参数的情况也是一样的，依次把下一个参数替换新列表最后一个 nil 就是最后的结果了。

```
(append '(a b) '(c) '(d))               ; => (a b c d)
```

一般来说 append 的参数都要是列表，但是最后一个参数可以不是一个列表，这也不违背前面说的，因为 cons cell 的 CDR 部分本来就可以是任何对象：

```
(append '(a b) 'c)                      ; => (a b . c)
```

这样得到的结果就不再是一个真列表了，如果再进行 append 操作就会产生一个错误。

如果你写过 c 的链表类型，可能就知道如果链表只保留一个指针，那么链表只能在一端增加元素。elisp 的列表类型也是类似的，用 cons 在列表前增加元素比用 append 要快得多。append 的参数不限于列表，还可以是字符串或者向量。前面字符串里已经提到可以把一个字符串转换成一个字符列表，同样可能把向量转换成一个列表：

```
(append [a b] "cd" nil)                 ; => (a b 99 100)
```

注意前面最后一个参数 nil 是必要的，不然你可以想象得到的结果是什么。

5.3 把列表当数组用

要得到列表或者 cons cell 里元素，唯一的方法是用 car 和 cdr 函数。很容易明白，car 就是取得 cons cell 的 CAR 部分，cdr 函数就是取得 cons cell 的 CDR 部分。通过这两个函数，我们就能访问 cons cell 和列表中的任何元素。

> 思考题
> 你如果知道 elisp 的函数如果定义，并知道 if 的使用方法，不妨自己写一个函数来取得一个列表的第 n 个 CDR。

通过使用 elisp 提供的函数，我们事实上是可以把列表当数组来用。依惯例，我们用 car 来访问列表的第一个元素，cadr 来访问第二个元素，再往后就没有这样的函数了，可以用 nth 函数来访问：

```
(nth 3 '(0 1 2 3 4 5))                  ; => 3
```

获得列表一个区间的函数有 nthcdr、last 和 butlast。nthcdr 和 last 比较类似，它们都是返回列表后端的列表。nthcdr 函数返回第 n 个元素后的列表：

```
(nthcdr 2 '(0 1 2 3 4 5))               ; => (2 3 4 5)
```

last 函数返回倒数 n 个长度的列表：

```
(last '(0 1 2 3 4 5) 2)                 ; => (4 5)
```

butlast 和前两个函数不同，返回的除了倒数 n 个元素的列表。

```
(butlast '(0 1 2 3 4 5) 2)              ; => (0 1 2 3)
```

> 思考题
> 如何得到某个区间(比如从 3 到 5 之间)的列表(提示列表长度可以用 length 函数得到):

```
(my-subseq '(0 1 2 3 4 5) 2 5)          ; => (2 3 4)
```

使用前面这几个函数访问列表是没有问题了。但是你也可以想象，链表这种数据结构是不适合随机访问的，代价比较高，如果你的代码中频繁使用这样的函数或者对一个很长的列表使用这样的函数，就应该考虑是不是应该用数组来实现。

直到现在为止，我们用到的函数都不会修改一个已有的变量。这是函数式编程的一个特点。只用这些函数编写的代码是很容易调试的，因为你不用去考虑一个变量在执行一个代码后就改变了，不用考虑变量的引用情况等等。下面就要结束这样轻松的学习了。

首先学习怎样修改一个 cons cell 的内容。首先 setcar 和 setcdr 可以修改一个 cons cell 的 CAR 部分和 CDR 部分。比如：

```
(setq foo '(a b c))                     ; => (a b c)
(setcar foo 'x)                         ; => x
foo                                     ; => (x b c)
(setcdr foo '(y z))                     ; => (y z)
foo                                     ; => (x y z)
```

思考题
好像很简单是吧。我出一个比较 bt 的一个问题，下面代码运行后 foo 是什么东西呢？

```
(setq foo '(a b c))
(setcdr foo foo)
```

现在来考虑一下，怎样像数组那样直接修改列表。使用 setcar 和 nthcdr 的组合就可以实现了：

```
(setq foo '(1 2 3))                    ; => (1 2 3)
(setcar foo 'a)                        ; => a
(setcar (cdr foo) 'b)                  ; => b
(setcar (nthcdr 2 foo) 'c)             ; => c
foo                                    ; => (a b c)
```

5.4 把列表当堆栈用

前面已经提到过可以用 push 向列表头端增加元素，在结合 pop 函数，列表就可以做为一个堆栈了。

```
(setq foo nil)                         ; => nil
(push 'a foo)                          ; => (a)
(push 'b foo)                          ; => (b a)
(pop foo)                              ; => b
foo                                    ; => (a)
```

5.5 重排列表

如果一直用 push 往列表里添加元素有一个问题是这样得到的列表和加入的顺序是相反的。通常我们需要得到一个反向的列表。reverse 函数可以做到这一点：

```
(setq foo '(a b c))                    ; => (a b c)
(reverse foo)                          ; => (c b a)
```

需要注意的是使用 reverse 后 foo 值并没有改变。不要怪我太啰唆，如果你看到一个函数 nreverse，而且确实它能返回逆序的列表，不明所以就到处乱用，迟早会写出一个错误的函数。这个 nreverse 和前面的 reverse 差别就在于它是一个有破坏性的函数，也就是说它会修改它的参数。

```
(nreverse foo)                         ; => (c b a)
foo                                    ; => (a)
```

为什么现在 foo 指向的是列表的末端呢？如果你实现过链表就知道，逆序操作是可以在原链表上进行的，这样原来头部指针会变成链表的尾端。列表也是（应该是，我也没有看过实现）这个原理。使用 nreverse 的唯一的好处是速度快，省资源。所以如果你只是想得到逆序后的列表就放心用 nreverse，否则还是用 reverse 的好。

elisp 还有一些是具有破坏性的函数。最常用的就是 sort 函数：

```
(setq foo '(3 2 4 1 5))                 ; => (3 2 4 1 5)
(sort foo '<)                           ; => (1 2 3 4 5)
foo                                     ; => (3 4 5)
```

这一点请一定要记住，我就曾经在 sort 函数上犯了好几次错误。那如果我既要保留原列表，又要进行 sort 操作怎么办呢？可以用 copy-sequence 函数。这个函数只对列表进行复制，返回的列表的元素还是原列表里的元素，不会拷贝列表的元素。

nconc 和 append 功能相似，但是它会修改除最后一个参数以外的所有的参数，nbutlast 和 butlast 功能相似，也会修改参数。这些函数都是在效率优先时才使用。总而言之，以 n 开头的函数都要慎用。

5.6 把列表当集合用

列表可以作为无序的集合。合并集合用 append 函数。去除重复的 equal 元素用 delete-dups。查找一个元素是否在列表中，如果测试函数是用 eq，就用 memq，如果测试用 equal，可以用 member。删除列表中的指定的元素，测试函数为 eq 对应 delq 函数，equal 对应 delete。还有两个函数 remq 和 remove 也是删除指定元素。它们的差别是 delq 和 delete 可能会修改参数，而 remq 和 remove 总是返回删除后列表的拷贝。注意前面这是说的是可能会修改参数的值，也就是说可能不会，所以保险起见，用 delq 和 delete 函数要么只用返回值，要么用 setq 设置参数的值为返回值。

```
(setq foo '(a b c))                     ; => (a b c)
(remq 'b foo)                           ; => (a c)
foo                                     ; => (a b c)
(delq 'b foo)                           ; => (a c)
foo                                     ; => (a c)
(delq 'a foo)                           ; => (c)
foo                                     ; => (a c)
```

5.7 把列表当关联表

用在 elisp 编程中，列表最常用的形式应该是作为一个关联表了。所谓关联表，就是可以用一个字符串（通常叫关键字，key）来查找对应值的数据结构。由列表实现的关联表有一个专门的名字叫 association list。尽管 elisp 里也有 hash table，但是 hash table 相比于 association list 至少这样几个缺点：

- hash table 里的关键字（ key）是无序的，而 association list 的关键字可以按想要的顺序排列

- hash table 没有列表那样丰富的函数，只有一个 maphash 函数可以遍历列表。而 association list 就是一个列表，所有列表函数都能适用
- hash table 没有读入语法和输入形式，这对于调试和使用都带来很多不便

所以 elisp 的 hash table 不是一个首要的数据结构，只要不对效率要求很高，通常直接用 association list。数组可以作为关联表，但是数组不适合作为与人交互使用数据结构（毕竟一个有意义的名字比纯数字的下标更适合人脑）。所以关联表的地位在 elisp 中就非比寻常了，emacs 为关联表专门用 c 程序实现了查找的相关函数以提高程序的效率。在 association list 中关键字是放在元素的 CAR 部分，与它对应的数据放在这个元素的 CDR 部分。根据比较方法的不同，有 assq 和 assoc 两个函数，它们分别对应查找使用 eq 和 equal 两种方法。例如：

```
(assoc "a" '(("a" 97) ("b" 98)))        ; => ("a" 97)
(assq 'a '((a . 97) (b . 98)))          ; => (a . 97)
```

通常我们只需要查找对应的数据，所以一般来说都要用 cdr 来得到对应的数据：

```
(cdr (assoc "a" '(("a" 97) ("b" 98))))  ; => (97)
(cdr (assq 'a '((a . 97) (b . 98))))    ; => 97
```

assoc-default 可以一步完成这样的操作：

```
(assoc-default "a" '(("a" 97) ("b" 98)))          ; => (97)
```

如果查找用的键值（ key）对应的数据也可以作为一个键值的话，还可以用 rassoc 和 rassq 来根据数据查找键值：

```
(rassoc '(97) '(("a" 97) ("b" 98)))     ; => ("a" 97)
(rassq '97 '((a . 97) (b . 98)))        ; => (a . 97)
```

如果要修改关键字对应的值，最省事的作法就是用 cons 把新的键值对加到列表的头端。但是这会让列表越来越长，浪费空间。如果要替换已经存在的值，一个想法就是用 setcdr 来更改键值对应的数据。但是在更改之前要先确定这个键值在对应的列表里，否则会产生一个错误。另一个想法是用 assoc 查找到对应的元素，再用 delq 删除这个数据，然后用 cons 加到列表里：

```
(setq foo '(("a" . 97) ("b" . 98)))     ; => (("a" . 97) ("b" . 98))

;; update value by setcdr
(if (setq bar (assoc "a" foo))
    (setcdr bar "this is a")
  (setq foo (cons '("a" . "this is a") foo))) ; => "this is a"
foo                           ; => (("a" . "this is a") ("b" . 98))
;; update value by delq and cons
(setq foo (cons '("a" . 97)
                (delq (assoc "a" foo) foo))) ; => (("a" . 97) ("b" . 98))
```

如果不对顺序有要求的话，推荐用后一种方法吧。这样代码简洁，而且让最近更新的元素放到列表前端，查找更快。

5.8 把列表当树用

列表的第一个元素如果作为结点的数据，其它元素看作是子节点，就是一个树了。由于树的操作都涉及递归，现在还没有说到函数，我就不介绍了。（其实是我不太熟，就不班门弄斧了）。

5.9 遍历列表

遍历列表最常用的函数就是 mapc 和 mapcar 了。它们的第一个参数都是一个函数，这个函数只接受一个参数，每次处理一个列表里的元素。这两个函数唯一的差别是前者返回的还是输入的列表，而 mapcar 返回的函数返回值构成的列表：

```
(mapc '1+ '(1 2 3))                    ; => (1 2 3)
(mapcar '1+ '(1 2 3))                  ; => (2 3 4)
```

另一个比较常用的遍历列表的方法是用 dolist。它的形式是：

```
(dolist (var list [result]) body...)
```

其中 var 是一个临时变量，在 body 里可以用来得到列表中元素的值。使用 dolist 的好处是不用写 lambda 函数。一般情况下它的返回值是 nil，但是你也可以指定一个值作为返回值（我觉得这个特性没有什么用，只省了一步而已）：

```
(dolist (foo '(1 2 3))
  (incf foo))                          ; => nil
(setq bar nil)
(dolist (foo '(1 2 3) bar)
  (push (incf foo) bar))               ; => (4 3 2)
```

5.10 其它常用函数

如果看过一些函数式语言教程的话，一定对 fold（或叫 accumulate、reduce）和 filter 这些函数记忆深刻。不过 elisp 里好像没有提供这样的函数。remove-if 和 remove-if-not 可以作 filter 函数，但是它们是 cl 里的，自己用用没有关系，不能强迫别人也跟着用，所以不能写到 elisp 里。如果不用这两个函数，也不用别人的函数的话，自己实现不妨用这样的方法：

```
(defun my-remove-if (predicate list)
  (delq nil (mapcar (lambda (n)
                      (and (not (funcall predicate n)) n))
                    list)))
(defun evenp (n)
  (= (% n 2) 0))
(my-remove-if 'evenp '(0 1 2 3 4 5))    ; => (1 3 5)
```

fold 的操作只能用变量加循环或 mapc 操作来代替了：

```
(defun my-fold-left (op initial list)
  (dolist (var list initial)
    (setq initial (funcall op initial var))))
(my-fold-left '+ 0 '(1 2 3 4))          ; => 10
```

这里只是举个例子，事实上你不必写这样的函数，直接用函数里的遍历操作更好一些。

产生数列常用的方法是用 number-sequence（这里不禁用说一次，不要再用 loop 产生 tab-stop-list 了，你们 too old 了）。不过这个函数好像在 emacs21 时好像还没有。

解析文本时一个很常用的操作是把字符串按分隔符分解，可以用 split-string 函数：

```
(split-string "key = val" "\\s-*=\\s-*")  ; => ("key" "val")
```

与 split-string 对应是把几个字符串用一个分隔符连接起来，这可以用 mapconcat 完成。比如：

```
(mapconcat 'identity '("a" "b" "c") "\t") ; => "a       b       c"
```

identity 是一个特殊的函数，它会直接返回参数。mapconcat 第一个参数是一个函数，可以很灵活的使用。

5.11 函数列表

```
;; 列表测试
(consp OBJECT)
(listp OBJECT)
(null OBJECT)
;; 列表构造
(cons CAR CDR)
(list &rest OBJECTS)
(append &rest SEQUENCES)
;; 访问列表元素
(car LIST)
(cdr LIST)
(cadr X)
(caar X)
(cddr X)
(cdar X)
(nth N LIST)
(nthcdr N LIST)
(last LIST &optional N)
(butlast LIST &optional N)
;; 修改 cons cell
(setcar CELL NEWCAR)
```

```
(setcdr CELL NEWCDR)
;; 列表操作
(push NEWELT LISTNAME)
(pop LISTNAME)
(reverse LIST)
(nreverse LIST)
(sort LIST PREDICATE)
(copy-sequence ARG)
(nconc &rest LISTS)
(nbutlast LIST &optional N)
;; 集合函数
(delete-dups LIST)
(memq ELT LIST)
(member ELT LIST)
(delq ELT LIST)
(delete ELT SEQ)
(remq ELT LIST)
(remove ELT SEQ)
;; 关联列表
(assoc KEY LIST)
(assq KEY LIST)
(assoc-default KEY ALIST &optional TEST DEFAULT)
(rassoc KEY LIST)
(rassq KEY LIST)
;; 遍历函数
(mapc FUNCTION SEQUENCE)
(mapcar FUNCTION SEQUENCE)
(dolist (VAR LIST [RESULT]) BODY...)
;; 其它
(number-sequence FROM &optional TO INC)
(split-string STRING &optional SEPARATORS OMIT-NULLS)
(mapconcat FUNCTION SEQUENCE SEPARATOR)
(identity ARG)
```

5.12 问题解答

5.12.1 用list 生成(a b c)

答案是`(list 'a 'b 'c)`。很简单的一个问题。从这个例子可以看出为什么要想出用，来输入列表。这就是程序员“懒”的美德呀!

5.12.2 nthcdr 的一个实现

```
(defun my-nthcdr (n list)
  (if (or (null list) (= n 0))
      (car list)
    (my-nthcdr (1- n) (cdr list))))
```

这样的实现看上去很简洁，但是一个最大的问题的 elisp 的递归是有限的，所以如果想这个函数没有问题，还是用循环还实现比较好。

5.12.3 my-subseq 函数的定义

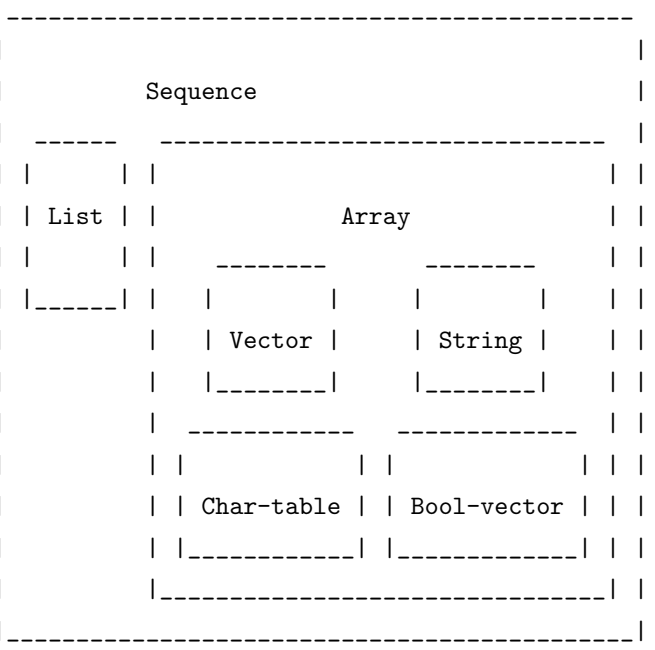
```
(defun my-subseq (list from &optional to)
  (if (null to) (nthcdr from list)
    (butlast (nthcdr from list) (- (length list) to))))
```

5.12.4 (setcdr foo foo) 是什么怪东西?

可能你已经想到了，这就是传说中的环呀。这在 info elisp - Circular Objects 里有介绍。elisp 里用到这样的环状列表并不多见，但是也不是没有， org 和 session 那个 bug 就是由于一个环状列表造成的。

第六章　基本数据类型之四——序列和数组

序列是列表和数组的统称，也就是说列表和数组都是序列。它们的共性是内部的元素都是有序的。elisp 里的数组包括字符串、向量、char-table 和布尔向量。它们的关系可以用下面图表示：



数组有这样一些特性：

- 数组内的元素都对应一个下标，第一个元素下标为 0，接下来是 1。数组内的元素可以在常数时间内访问。
- 数组在创建之后就无法改变它的长度。
- 数组是自求值的。
- 数组里的元素都可以用 aref 来访问，用 aset 来设置。

向量可以看成是一种通用的数组，它的元素可以是任意的对象。而字符串是一种特殊的数组，它的元素只能是字符。如果元素是字符时，使用字符串相比向量更好，因为字符串需要的空间更少（只需要向量的 1/4），输出更直观，能用文本属性（ text property），能使用 emacs 的 IO 操作。但是有时必须使用向量，比如存储按键序列。

由于 char-table 和 bool-vector 使用较少，而且较难理解，这里就不介绍了。

6.1　测试函数

sequencep 用来测试一个对象是否是一个序列。arrayp 测试对象是否是数组。vectorp、char-table-p 和 bool-vector-p 分别测试对象是否是向量、 char-table、bool-vector。

6.2 序列的通用函数

一直没有提到一个重要的函数 length，它可以得到序列的长度。但是这个函数只对真列表有效。对于一个点列表和环形列表这个函数就不适用了。点列表会出参数类型不对的错误，而环形列表就更危险，会陷入死循环。如果不确定参数类型，不妨用 safe-length。比如：

```
(safe-length '(a . b))                  ; => 1
(safe-length '#1=(1 2 . #1#))           ; => 3
```

> 思考题
> 写一个函数来检测列表是否是一个环形列表。由于现在还没有介绍 let 绑定和循环，不过如果会函数定义，还是可以用递归来实现的。

取得序列里第 n 个元素可以用 elt 函数。但是我建议，对于已知类型的序列，还是用对应的函数比较好。也就是说，如果是列表就用 nth，如果是数组就用 aref。这样一方面是省去 elt 内部的判断，另一方面读代码时能很清楚知道序列的类型。

copy-sequence 在前面已经提到了。不过同样 copy-sequence 不能用于点列表和环形列表。对于点列表可以用 copy-tree 函数。环形列表就没有办法复制了。好在这样的数据结构很少用到。

6.3 数组操作

创建字符串已经说过了。创建向量可以用 vector 函数：

```
(vector 'foo 23 [bar baz] "rats")
```

当然也可以直接用向量的读入语法创建向量，但是由于数组是自求值的，所以这样得到的向量和原来是一样的，也就是说参数不进行求值，看下面的例子就明白了：

```
foo                                     ; => (a b)
[foo]                                   ; => [foo]
(vector foo)                            ; => [(a b)]
```

用 make-vector 可以生成元素相同的向量。

```
(make-vector 9 'Z)                      ; => [Z Z Z Z Z Z Z Z Z]
```

fillarray 可以把整个数组用某个元素填充。

```
(fillarray (make-vector 3 'Z) 5)        ; => [5 5 5]
```

aref 和 aset 可以用于访问和修改数组的元素。如果使用下标超出数组长度的话，会产生一个错误。所以要先确定数组的长度才能用这两个函数。

vconcat 可以把多个序列用 vconcat 连接成一个向量。但是这个序列必须是真列表。这也是把列表转换成向量的方法。

```
(vconcat [A B C] "aa" '(foo (6 7)))     ; => [A B C 97 97 foo (6 7)]
```

把向量转换成列表可以用 append 函数，这在前一节中已经提到。

思考题
如果知道 elisp 的 let 绑定和循环的使用方法，不妨试试实现一个 elisp 的 tr 函数，它接受三个参数，一是要操作的字符串，另外两个分别是要替换的字符集，和对应的替换后的字符集（当它是空集时，删除字符串中所有对应的字符）。

6.4 函数列表

```
;; 测试函数
(sequencep OBJECT)
(arrayp OBJECT)
(vectorp OBJECT)
(char-table-p OBJECT)
(bool-vector-p OBJECT)
;; 序列函数
(length SEQUENCE)
(safe-length LIST)
(elt SEQUENCE N)
(copy-sequence ARG)
(copy-tree TREE &optional VECP)
;; 数组函数
(vector &rest OBJECTS)
(make-vector LENGTH INIT)
(aref ARRAY IDX)
(aset ARRAY IDX NEWELT)
(vconcat &rest SEQUENCES)
(append &rest SEQUENCES)
```

6.5 问题解答

6.5.1 测试列表是否是环形列表

这个算法是从 safe-length 定义中得到的。你可以直接看它的源码。下面是我写的函数。

```
(defun circular-list-p (list)
  (and (consp list)
       (circular-list-p-1 (cdr list) list 0)))

(defun circular-list-p-1 (tail halftail len)
  (if (eq tail halftail)
      t
    (if (consp tail)
        (circular-list-p-1 (cdr tail)
```

```
                          (if (= (% len 2) 0)
                              (cdr halftail)
                            halftail)
                          (1+ len))
      nil)))
```

6.5.2 转换字符的 tr 函数

```
(defun my-tr (str from to)
  (if (= (length to) 0)                ; 空字符串
      (progn
        (setq from (append from nil))
        (concat
         (delq nil
               (mapcar (lambda (c)
                         (if (member c from)
                             nil c))
                       (append str nil)))))
    (let (table newstr pair)
      ;; 构建转换表
      (dotimes (i (length from))
        (push (cons (aref from i) (aref to i)) table))
      (dotimes (i (length str))
        (push
         (if (setq pair (assoc (aref str i) table))
             (cdr pair)
           (aref str i))
         newstr))
      (concat (nreverse newstr) nil))))
```

这里用到的 dotimes 函数相当于一个 C 里的 for 循环。如果改写成 while 循环，相当于：

```
(let (var)
  (while (< var count)
    body
    (setq var (1+ var)))
  result)
```

从这个例子也可以看出，由于列表具有丰富的函数和可变长度，使列表比数组使用更方便，而且效率往往更高。

第七章　基本数据类型之五——符号

符号是有名字的对象。可能这么说有点抽象。作个不恰当的比方，符号可以看作是 C 语言里的指针。通过符号你可以得到和这个符号相关联的信息，比如值，函数，属性列表等等。

首先必须知道的是符号的命名规则。符号名字可以含有任何字符。大多数的符号名字只含有字母、数字和标点“-+=*/”。这样的名字不需要其它标点。名字前缀要足够把符号名和数字区分开来，如果需要的话，可以在前面用 \ 表示为符号，比如：

```
(symbolp '+1)                           ; => nil
(symbolp '\+1)                          ; => t
(symbol-name '\+1)                      ; => "+1"
```

其它字符 _~!@$%^&:<>{}? 用的比较少。但是也可以直接作为符号的名字。任何其它字符都可以用\ 转义后用在符号名字里。但是和字符串里字符表示不同，\ 转义后只是表示其后的字符，比如\t 代表的字符 t，而不是制表符。如果要在符号名里使用制表符，必须在\ 后加上制表符本身。

符号名是区分大小写的。这里有一些符号名的例子：

```
foo                 ; 名为 'foo' 的符号
FOO                 ; 名为 'FOO' 的符号, 和 'foo' 不同
char-to-string      ; 名为 'char-to-string' 的符号
1+                  ; 名为 '1+' 的符号 (不是整数 '+1')
\+1                 ; 名为 '+1' 的符号 (可读性很差的名字)
\(*\ 1\ 2\)         ; 名为 '(* 1 2)' 的符号 (更差劲的名字).
+-*/_~!@$%^&=:<>{}  ; 名为 '+-*/_~!@$%^&=:<>{}' 的符号.
                    ;   这些字符无须转义
```

7.1　创建符号

一个名字如何与数据对应上呢？这就要了解一下符号是如何创建的了。符号名要有唯一性，所以一定会有一个表与名字关联，这个表在 elisp 里称为 obarray。从这个名字可以看出这个表是用数组类型，事实上是一个向量。当 emacs 创建一个符号时，首先会对这个名字求 hash 值以得到一个在 obarray 这个向量中查找值所用的下标。hash 是查找字符串的很有效的方法。这里强调的是 obarray 不是一个特殊的数据结构，就是一个一般的向量。全局变量 obarray 里 emacs 所有变量、函数和其它符号所使用的 obarray（注意不同语境中 obarray 的含义不同。前一个 obarray 是变量名，后一个 obarray 是数据类型名）。也可以自己建立向量，把这个向量作为 obarray 来使用。这是一种代替散列的一种方法。它比直接使用散列有这样一些好处：

- 符号不仅可以有一个值，还可以用属性列表，后者又可以相当于一个关联列表。这样有很高的扩展性，而且可以表达更高级的数据结构。
- emacs 里有一些函数可以接受 obarray 作为参数，比如补全相关的函数。

当 lisp 读入一个符号时，通常会先查找这个符号是否在 obarray 里出现过，如果没有则会把这个符号加入到 obarray 里。这样查找并加入一个符号的过程称为是 intern。intern 函数可以查找或加入一个名字到 obarray 里，返回对应的符号。默认是全局的 obarray，也可以指定一个 obarray。intern-soft 与 intern 不同的是，当名字不在 obarray 里时，intern-soft 会返回 nil，而 intern 会加入到 obarray 里。为了不污染 obarray，我下面的例子中尽量在 foo 这个 obarray 里进行。一般来说，去了 foo 参数，则会在 obarray 里进行。其结果应该是相同的：

```
(setq foo (make-vector 10 0))           ; => [0 0 0 0 0 0 0 0 0 0]
(intern-soft "abc" foo)                 ; => nil
(intern "abc" foo)                      ; => abc
(intern-soft "abc" foo)                 ; => abc
```

lisp 每读入一个符号都会 intern 到 obarray 里，如果想避免，可以用在符号名前加上#:：

```
(intern-soft "abc")                     ; => nil
'abc                                    ; => abc
(intern-soft "abc")                     ; => abc
(intern-soft "abcd")                    ; => nil
'#:abcd                                 ; => abcd
(intern-soft "abcd")                    ; => nil
```

如果想除去 obarray 里的符号，可以用 unintern 函数。unintern 可以用符号名或符号作参数在指定的 obarray 里去除符号，成功去除则返回 t，如果没有查找到对应的符号则返回 nil：

```
(intern-soft "abc" foo)                 ; => abc
(unintern "abc" foo)                    ; => t
(intern-soft "abc" foo)                 ; => nil
```

和 hash-table 一样，obarray 也提供一个 mapatoms 函数来遍历整个 obarray。比如要计算 obarray 里所有的符号数量：

```
(setq count 0)                          ; => 0
(defun count-syms (s)
  (setq count (1+ count)))              ; => count-syms
(mapatoms 'count-syms)                  ; => nil
count                                   ; => 28371
(length obarray)                        ; => 1511
```

> 思考题
> 由前面的例子可以看出 elisp 中的向量长度都是有限的，而 obarray 里的符号有成千上万个。那这些符号是怎样放到 obarray 里的呢？

7.2 符号的组成

每个符号可以对应四个组成部分，一是符号的名字，可以用 symbol-name 访问。二是符号的值。符号的值可以通过 set 函数来设置，用 symbol-value 来访问。

```
(set (intern "abc" foo) "I'm abc")      ; => "I'm abc"
(symbol-value (intern "abc" foo))       ; => "I'm abc"
```

可能大家最常见到 setq 函数，而 set 函数确很少见到。setq 可以看成是一个宏，它可以让你用(setq sym val) 代替(set (quote sym) val)。事实上这也是它名字的来源(q 代表 quoted)。但是 setq 只能设置 obarray 里的变量，前面这个例子中就只能用 set 函数。

> 思考题
> 参考 assoc-default 的代码，写一个函数从一个关联列表中除去一个关键字对应的元素。这个函数可以直接修改关联列表符号的值。要求可以传递一个参数作为测试关键字是否相同的函数。比如:
>
> ```
> (setq foo '((?a . a) (?A . c) (?B . d)))
> (remove-from-list 'foo ?b 'char-equal)  ; => ((97 . a) (65 . c))
> foo                                     ; => ((97 . a) (65 . c))
> ```

如果一个符号的值已经有设置过的话，则 boundp 测试返回 t，否则为 nil。对于 boundp 测试返回 nil 的符号，使用符号的值会引起一个变量值为 void 的错误。

符号的第三个组成部分是函数。它可以用 symbol-function 来访问，用 fset 来设置：

```
(fset (intern "abc" foo) (symbol-function 'car)) ; => #<subr car>
(funcall (intern "abc" foo) '(a . b))            ; => a
```

类似的，可以用 fboundp 测试一个符号的函数部分是否有设置。

符号的第四个组成部分是属性列表(property list)。通常属性列表用于存储和符号相关的信息，比如变量和函数的文档，定义的文件名和位置，语法类型。属性名和值可以是任意的 lisp 对象，但是通常名字是符号，可以用 get 和 put 来访问和修改属性值，用 symbol-plist 得到所有的属性列表：

```
(put (intern "abc" foo) 'doc "this is abc")      ; => "this is abc"
(get (intern "abc" foo) 'doc)                    ; => "this is abc"
(symbol-plist (intern "abc" foo))                ; => (doc "this is abc")
```

关联列表和属性列表很相似。符号的属性列表在内部表示上是用 (prop1 value1 prop2 value2 ...) 的形式，和关联列表也是很相似的。属性列表在查找和这个符号相关的信息时，要比直接用关联列表要简单快捷的多。所以变量的文档等信息都是放在符号的属性列表里。但是关联表在头端加入元素是很快的，而且它可以删除表里的元素。而属性列表则不能删除一个属性。

如果已经把属性列表取出，那么还可以用 plist-get 和 plist-put 的方法来访问和设置属性列表：

```
(plist-get '(foo 4) 'foo)               ; => 4
(plist-get '(foo 4 bad) 'bar)           ; => nil
(setq my-plist '(bar t foo 4))          ; => (bar t foo 4)
```

```
(setq my-plist (plist-put my-plist 'foo 69)) ; => (bar t foo 69)
(setq my-plist (plist-put my-plist 'quux '(a))) ; => (bar t foo 69 quux (a))
```

> 思考题
> 你能不能用已经学过的函数来实现 plist-get 和 plist-put?

7.3 函数列表

```
(symbolp OBJECT)
(intern-soft NAME &optional OBARRAY)
(intern STRING &optional OBARRAY)
(unintern NAME &optional OBARRAY)
(mapatoms FUNCTION &optional OBARRAY)
(symbol-name SYMBOL)
(symbol-value SYMBOL)
(boundp SYMBOL)
(set SYMBOL NEWVAL)
(setq SYM VAL SYM VAL ...)
(symbol-function SYMBOL)
(fset SYMBOL DEFINITION)
(fboundp SYMBOL)
(symbol-plist SYMBOL)
(get SYMBOL PROPNAME)
(put SYMBOL PROPNAME VALUE)
```

7.4 问题解答

7.4.1 obarray 里符号数为什么大于向量长度

其实这和散列的的实现是一样的。obarray 里的每一个元素通常称为 bucket。一个 bucket 是可以容纳多个相同 hash 值的字符串和它们的数据。我们可以用这样的方法来模拟一下：

```
(defun hash-string (str)
  (let ((hash 0) c)
    (dotimes (i (length str))
      (setq c (aref str i))
      (if (> c #o140)
          (setq c (- c 40)))
      (setq hash (+ (setq hash (lsh hash 3))
                    (lsh hash -28)
                    c)))
    hash))
```

```
(let ((len 10) str hash)
  (setq foo (make-vector len 0))
  (dotimes (i (1+ len))
    (setq str (char-to-string (+ ?a i))
          hash (% (hash-string str) len))
    (message "I put %s in slot %d"
             str hash)
    (if (eq (aref foo hash) 0)
        (intern str foo)
      (message "I found %S is already taking the slot: %S"
               (aref foo hash) foo)
      (intern str foo)
      (message "Now I'am in the slot too: %S" foo))))
```

在我这里的输出是：

```
I put a in slot 7
I put b in slot 8
I put c in slot 9
I put d in slot 0
I put e in slot 1
I put f in slot 2
I put g in slot 3
I put h in slot 4
I put i in slot 5
I put j in slot 6
I put k in slot 7
I found a is already taking the slot: [d e f g h i j a b c]
Now I'am in the slot too: [d e f g h i j k b c]
```

当然，这个 hash-string 和实际 obarray 里用的 hash-string 只是算法上是相同的，但是由于数据类型和 c 不是完全相同，所以对于长一点的字符串结果可能不一样，我只好用单个字符来演示一下。

7.4.2 根据关键字删除关联列表中的元素

```
(defun remove-from-list (list-var key &optional test)
  (let ((prev (symbol-value list-var))
        tail found value elt)
    (or test (setq test 'equal))
    (if (funcall test (caar prev) key)
        (set list-var (cdr prev))
      (setq tail (cdr prev))
      (while (and tail (not found))
```

```
        (setq elt (car tail))
        (if (funcall test (car elt) key)
            (progn
              (setq found t)
              (setcdr prev (cdr tail)))
          (setq tail (cdr tail)
                prev (cdr prev)))))
    (symbol-value list-var)))
```

注意这个函数的参数 list-var 是一个符号，所以这个函数不能直接传递一个列表。这和 add-to-list 的参数是一样的。

7.4.3 plist-get 和 plist-put 的实现

```
(defun my-plist-get (plist prop)
  (cadr (memq plist prop)))
(defun my-plist-put (plist prop val)
  (let ((tail (memq prop plist)))
    (if tail
        (setcar (cdr tail) val)
      (setcdr (last plist) (list prop val))))
  plist)
```

my-plist-put 函数没有 plist-put 那样 robust，如果属性列表是`'(bar t foo)` 这样的话，这个函数就会出错。而且加入一个属性的时间复杂度比 plist 更高（ memq 和 last 都是 O(n)），不过可以用循环来达到相同的时间复杂度。

第八章　求值规则

至此，elisp 中最常见的数据类型已经介绍完了。我们可以真正开始学习怎样写一个 elisp 程序。如果想深入了解一下 lisp 是如何工作的，不妨先花些时间看看 lisp 的求值过程。当然忽略这一部分也是可以的，因为我觉得这个求值规则是那么自然，以至于你会认为它就是应该这样的。

求值是 lisp 解释器的核心，理解了求值过程也就学会了 lisp 编程的一半。正因为这样，我有点担心自己说得不清楚或者理解错误，会误导了你。所以如果真想深入了解的话，还是自己看 info elisp - Evaluation 这一章吧。

一个要求值的 lisp 对象被称为表达式（ form）。所有的表达式可以分为三种：符号、列表和其它类型（废话）。下面一一说明各种表达式的求值规则。

第一种表达式是最简单的，自求值表达式。前面说过数字、字符串、向量都是自求值表达式。还有两个特殊的符号 t 和 nil 也可以看成是自求值表达式。

第二种表达式是符号。符号的求值结果就是符号的值。如果它没有值，就会出现 void-variable 的错误。

第三种表达式是列表表达式。而列表表达式又可以根据第一个元素分为函数调用、宏调用和特殊表达式（ special form）三种。列表的第一个表达式如果是一个符号，解释器会查找这个表达式的函数值。如果函数值是另一个符号，则会继续查找这个符号的函数值。这称为“ symbol function indirection”。最后直到某个符号的函数值是一个 lisp 函数（ lambda 表达式）、byte-code 函数、原子函数（ primitive function）、宏、特殊表达式或 autoload 对象。如果不是这些类型，比如某个符号的函数值是前面出现的某个符号导致无限循环，或者某个符号函数值为空，都会导致一个错误 invalid-function。

这个函数显示 indirection function：

```
(symbol-function 'car)                  ; => #<subr car>
(fset 'first 'car)                      ; => car
(fset 'erste 'first)                    ; => first
(erste '(1 2 3))                        ; => 1
```

对于第一个元素是 lisp 函数对象、byte-code 对象和原子函数时，这个列表也称为函数调用（ funtion call）。对这样的列表求值时，先对列表中其它元素先求值，求值的结果作为函数调用的真正参数。然后使用 apply 函数用这些参数调用函数。如果函数是用 lisp 写的，可以理解为把参数和变量绑定到函数后，对函数体顺序求值，返回最后一个 form 的值。

如果第一个元素是一个宏对象，列表里的其它元素不会立即求值，而是根据宏定义进行扩展。如果扩展后还是一个宏调用，则会继续扩展下去，直到扩展的结果不再是一个宏调用为止。例如：

```
(defmacro cadr (x)
  (list 'car (list 'cdr x)))
```

这样`(cadr (assq 'handler list))`扩展后成为`(car (cdr (assq 'handler list)))`。

第一个元素如果是一个特殊表达式时，它的参数可能并不会全求值。这些特殊表达式通常是用于控制结构或者变量绑定。每个特殊表达式都有对应的求值规则。这在下面会提到。

最后用这个伪代码来说明一下 elisp 中的求值规则：

```
(defun (eval exp)
  (cond
   ((numberp exp) exp)
   ((stringp exp) exp)
   ((arrayp exp) exp)
   ((symbolp exp) (symbol-value exp))
   ((special-form-p (car exp))
    (eval-special-form exp))
   ((fboundp (car exp))
    (apply (car exp) (cdr exp)))
   (t
    (error "Unknown expression type -- EVAL %S" exp))))
```

第九章　变量

在此之前，我们已经见过 elisp 中的两种变量，全局变量和 let 绑定的局部变量。它们相当于其它语言中的全局变量和局部变量。

关于 let 绑定的变量，有两点需要补充的。当同一个变量名既是全局变量也是局部变量，或者用 let 多层绑定，只有最里层的那个变量是有效的，用 setq 改变的也只是最里层的变量，而不影响外层的变量。比如：

```
(progn
  (setq foo "I'm global variable!")
  (let ((foo 5))
    (message "foo value is: %S" foo)
    (let (foo)
      (setq foo "I'm local variable!")
      (message foo))
    (message "foo value is still: %S" foo))
  (message foo))
```

另外需要注意一点的是局部变量的绑定不能超过一定的层数，也就是说，你不能把 foo 用 let 绑定 10000 层。当然普通的函数是不可能写成这样的，但是递归函数就不一定了。限制层数的变量在 max-specpdl-size 中定义。如果你写的递归函数有这个需要的话，可以先设置这个变量的值。

emacs 有一种特殊的局部变量——buffer-local 变量。

9.1　buffer-local 变量

emacs 能有如此丰富的模式，各个缓冲区之间能不相互冲突，很大程度上要归功于 buffer-local 变量。

声明一个 buffer-local 的变量可以用 make-variable-buffer-local 或用 make-local-variable。这两个函数的区别在于前者是相当于在所有变量中都产生一个 buffer-local 的变量。而后者只在声明时所在的缓冲区内产生一个局部变量，而其它缓冲区仍然使用的是全局变量。一般来说推荐使用 make-local-variable。

为了方便演示，下面的代码我假定你是在*scratch* 缓冲区里运行。我使用另一个一般都会有的缓冲区*Messages* 作为测试。先介绍两个用到的函数（ with-current-buffer 其实是一个宏）。

with-current-buffer 的使用形式是：

```
(with-current-buffer buffer
  body)
```

其中 buffer 可以是一个缓冲区对象，也可以是缓冲区的名字。它的作用是使其中的 body 表达式在指定的缓冲区里执行。

get-buffer 可以用缓冲区的名字得到对应的缓冲区对象。如果没有这样名字的缓冲区会返回 nil。

下面是使用 buffer-local 变量的例子：

```
(setq foo "I'm global variable!")       ; => "I'm global variable!"
(make-local-variable 'foo)              ; => foo
foo                                     ; => "I'm global variable!"
(setq foo "I'm buffer-local variable!") ; => "I'm buffer-local variable!"
foo                                  ; => "I'm buffer-local variable!"
(with-current-buffer "*Messages*" foo)  ; => "I'm global variable!"
```

从这个例子中可以看出，当一个符号作为全局变量时有一个值的话，用 make-local-variable 声明为 buffer-local 变量时，这个变量的值还是全局变量的值。这时候全局的值也称为缺省值。你可以用 default -value 来访问这个符号的全局变量的值：

```
(default-value 'foo)                    ; => "I'm global variable!"
```

如果一个变量是 buffer-local，那么在这个缓冲区内使用用 setq 就只能用改变当前缓冲区里这个变量的值。setq-default 可以修改符号作为全局变量的值。通常在.emacs 里经常使用 setq-default，这样可以防止修改的是导入.emacs 文件对应的缓冲区里的 buffer-local 变量，而不是设置全局的值。

测试一个变量是不是 buffer-local 可以用 local-variable-p：

```
(local-variable-p 'foo)                           ; => t
(local-variable-p 'foo (get-buffer "*Messages*")) ; => nil
```

如果要在当前缓冲区里得到其它缓冲区的 buffer-local 变量可以用 buffer-local-value：

```
(with-current-buffer "*Messages*"
  (buffer-local-value 'foo (get-buffer "*scratch*")))
                   ; => "I'm buffer local variable!"
```

9.2 变量的作用域

我们现在已经学习这样几种变量：

- 全局变量
- buffer-local 变量
- let 绑定局部变量

如果还要考虑函数的参数列表声明的变量，也就是 4 种类型的变量。那这种变量的作用范围(scope)和生存期（ extent）分别是怎样的呢？

作用域（ scope）是指变量在代码中能够访问的位置。emacs lisp 这种绑定称为 indefinite scope。indefinite scope 也就是说可以在任何位置都可能访问一个变量名。而 lexical scope（词法作用域）指局部变量只能作用在函数中和一个块里（ block）。

比如 let 绑定和函数参数列表的变量在整个表达式内都是可见的，这有别于其它语言词法作用域的变量。先看下面这个例子：

```
(defun binder (x)                    ; ‘x’ is bound in ‘binder’.
  (foo 5))                           ; ‘foo’ is some other function.
(defun user ()                       ; ‘x’ is used "free" in ‘user’.
  (list x))
(defun foo (ignore)
  (user))
(binder 10)                          ; => (10)
```

对于词法作用域的语言，在 user 函数里无论如何是不能访问 binder 函数中绑定的 x。但是在 elisp 中可以。

生存期是指程序运行过程中，变量什么时候是有效的。全局变量和 buffer-local 变量都是始终存在的，前者只能当关闭 emacs 或者用 unintern 从 obarray 里除去时才能消除。而 buffer-local 的变量也只能关闭缓冲区或者用 kill-local-variable 才会消失。而对于局部变量，emacs lisp 使用的方式称为动态生存期：只有当绑定了这个变量的表达式运行时才是有效的。这和 C 和 Pascal 里的 Local 和 automatic 变量是一样的。与此相对的是 indefinite extent，变量即使离开绑定它的表达式还能有效。比如：

```
(defun make-add (n)
  (function (lambda (m) (+ n m))))      ; Return a function.
(fset 'add2 (make-add 2))               ; Define function ‘add2’
                                        ;   with ‘(make-add 2)’.
(add2 4)                                ; Try to add 2 to 4.
```

其它 Lisp 方言中有闭包，但是 emacs lisp 中没有。

说完这些概念，可能你还是一点雾水。我给一个判断变量是否有效的方法吧：

1. 看看包含这个变量的 form 中是否有 let 绑定这个局部变量。如果这个 form 不是在定义一个函数，则跳到第 3 步。
2. 如果是在定义函数，则不仅要看这个函数的参数中是否有这个变量，而且还要看所有直接或间接调用这个函数的函数中是否有用 let 绑定或者参数列表里有这个变量名。这就没有办法确定了，所以你永远无法判断一个函数中出现的没有用 let 绑定，也不在参数列表中的变量是否是没有定义过的。但是一般来说这不是一个好习惯。
3. 看这个变量是否是一个全局变量或者是 buffer-local 变量。

对于在一个函数中绑定一个变量，而在另一个函数中还在使用，manual 里认为这两个种情况下是比较好的：

- 这个变量只有相关的几个函数中使用，在一个文件中放在一起。这个变量起程序里通信的作用。而且需要写好注释告诉其它程序员怎样使用它。
- 如果这个变量是定义明确、有很好文档作用的，可能让所有函数使用它，但是不要设置它。比如 case-fold-search。（我怎么觉得这里是用全局变量呢。）

思考题
先在*scratch* 缓冲区里运行了 (kill-local-variable 'foo) 后，运行几次下面的表达式，你能预测它们结果吗？

```
(progn
 (setq foo "I'm local variable!")
 (let ((foo "I'm local variable!"))
   (set (make-local-variable 'foo) "I'm buffer-local variable!")
   (setq foo "This is a variable!")
   (message foo))
 (message foo))
```

9.3 其它函数

一个符号如果值为空，直接使用可能会产生一个错误。可以用 boundp 来测试一个变量是否有定义。这通常用于 elisp 扩展的移植（用于不同版本或 XEmacs）。对于一个 buffer-local 变量，它的缺省值可能是没有定义的，这时用 default-value 函数可能会出错。这时就先用 default-boundp 先进行测试。

使一个变量的值重新为空，可以用 makunbound。要消除一个 buffer-local 变量用函数 kill-local-variable。可以用 kill-all-local-variables 消除所有的 buffer-local 变量。但是有属性 permanent-local 的不会消除，带有这些标记的变量一般都是和缓冲区模式无关的，比如输入法。

```
foo                                ; => "I'm local variable!"
(boundp 'foo)                      ; => t
(default-boundp 'foo)              ; => t
(makunbound 'foo)                  ; => foo
foo                                ; This will signal an error
(default-boundp 'foo)              ; => t
(kill-local-variable 'foo)         ; => foo
```

9.4 变量名习惯

对于变量的命名，有一些习惯，这样可以从变量名就能看出变量的用途：

- -hook 一个在特定情况下调用的函数列表，比如关闭缓冲区时，进入某个模式时。
- -function 值为一个函数
- -functions 值为一个函数列表
- -flag 值为 nil 或 non-nil

- -predicate 值是一个作判断的函数，返回 nil 或 non-nil
- -program 或-command 一个程序或 shell 命令名
- -form 一个表达式
- -forms 一个表达式列表。
- -map 一个按键映射（ keymap）

9.5 函数列表

```
(make-local-variable VARIABLE)
(make-variable-buffer-local VARIABLE)
(with-current-buffer BUFFER &rest BODY)
(get-buffer NAME)
(default-value SYMBOL)
(local-variable-p VARIABLE &optional BUFFER)
(buffer-local-value VARIABLE BUFFER)
(boundp SYMBOL)
(default-boundp SYMBOL)
(makunbound SYMBOL)
(kill-local-variable VARIABLE)
(kill-all-local-variables)
```

9.6 变量列表

```
max-specpdl-size
```

9.7 问题解答

9.7.1 同一个表达式运行再次结果不同?

运行第一次时，foo 缺省值为”I’m local variable!”，而 buffer-local 值为”This is a variable!”。第一个和第二个 message 都会显示”This is a variable!”。运行第二次时，foo 缺省值和 buffer-local 值都成了”I’m local variable!”，而第一次 message 显示”This is a variable!”，第二次显示”I’m local variable!”。这是由于 make-local-variable 在这个符号是否已经是 buffer-local 变量时有不同表现造成的。如果已经是一个 buffer-local 变量，则它什么也不做，而如果不是，则会生成一个 buffer-local 变量，这时在这个表达式内的所有 foo 也被重新绑定了。希望你写的函数能想到一点。

第十章　函数和命令

在 elisp 里类似函数的对象很多，比如：

- 函数。这里的函数特指用 lisp 写的函数。
- 原子函数（ primitive）。用 C 写的函数，比如 car、append。
- lambda 表达式
- 特殊表达式
- 宏(macro)。宏是用 lisp 写的一种结构，它可以把一种 lisp 表达式转换成等价的另一个表达式。
- 命令。命令能用 command-execute 调用。函数也可以是命令。

以上这些用 functionp 来测试都会返回 t。

我们已经学过如何定义一个函数。但是这些函数的参数个数都是确定。但是你可以看到 emacs 里有很多函数是接受可选参数，比如 random 函数。还有一些函数可以接受不确定的参数，比如加减乘除。这样的函数在 elisp 中是如何定义的呢？

10.1　参数列表的语法

这是参数列表的方法形式：

```
(REQUIRED-VARS...
 [&optional OPTIONAL-VARS...]
 [&rest REST-VAR])
```

它的意思是说，你必须把必须提供的参数写在前面，可选的参数写在后面，最后用一个符号表示剩余的所有参数。比如：

```
(defun foo (var1 var2 &optional opt1 opt2 &rest rest)
  (list var1 var2 opt1 opt2 rest))

(foo 1 2)                               ; => (1 2 nil nil nil)
(foo 1 2 3)                             ; => (1 2 3 nil nil)
(foo 1 2 3 4 5 6)                       ; => (1 2 3 4 (5 6))
```

从这个例子可以看出，当可选参数没有提供时，在函数体里，对应的参数值都是 nil。同样调用函数时没有提供剩余参数时，其值也为 nil，但是一旦提供了剩余参数，则所有参数是以列表的形式放在对应变量里。

> 思考题
> 写一个函数测试两个浮点数是否相等，设置一个可选参数，如果提供这个参数，则用这个参数作为测试误差，否则用 1.0e-6 作为误差。

10.2 关于文档字符串

最好为你的函数都提供一个文档字符串。关于文档字符串有一些规范，最好遵守这些约定。

字符串的第一行最好是独立的。因为 apropos 命令只能显示第一行的文档。所以最好用一行（一两个完整的句子）总结这个函数的目的。

文档的缩进最好要根据最后的显示的效果来调用。因为引号之类字符会多占用一个字符，所以在源文件里缩进最好看，不一定显示的最好。

如果你想要让你的函数参数显示的与函数定义的不同（比如提示用户如何调用这个函数），可以在文档最后一行，加上一行：

```
\(fn ARGLIST)
```

注意这一行前面要有一个空行，这一行后不能再有空行。比如：

```
(defun foo (var1 var2 &optional opt1 opt2 &rest rest)
  "You should call the function like:

\(fn v1 v2)"
  (list var1 var2 opt1 opt2 rest))
```

还有一些有特殊标记功能的符号，比如‘’ 引起的符号名可以生成一个链接，这样可以在*Help* 中更方便的查看相关变量或函数的文档。\\{major-mode-map} 可以显示扩展成这个模式按键的说明，例如：

```
(defun foo ()
  "A simple document string to show how to use ‘’ and \\=\\{}.
You can press this button ‘help’ to see the document of
function \"help\".

This is keybind of text-mode(substitute from \\=\\{text-mode-map}):
\\{text-mode-map}

See also ‘substitute-command-keys’ and ‘documentation’"
  )
```

10.3 调用函数

通常函数的调用都是用 eval 进行的，但是有时需要在运行时才决定使用什么函数，这时就需要用 funcall 和 apply 两个函数了。这两个函数都是把其余的参数作为函数的参数进行调用。那这两个函数有什么参数呢？唯一的区别就在于 funcall 是直接把参数传递给函数，而 apply 的最后一个参数是一个列表，传入函数的参数把列表进行一次平铺后再传给函数，看下面这个例子就明白了：

```
(funcall 'list 'x '(y) '(z))                ; => (x (y) (z))
```

```
(apply 'list 'x '(y ) '(z))                ; => (x (y) z)
```

> 思考题
> 如果一个 list 作为一个树的结构，任何是 cons cell 的元素都是一个内部节点（不允许有 dotted list 出现），任何不是 cons cell 的元素都是树的叶子。请写一个函数，调用的一个类似 mapcar 的函数，调用一个函数遍历树的叶子，并收集所有的结果，返回一个结构相同的树，比如：
>
> ```
> (tree-mapcar '1+ '(1 (2 (3 4)) (5)))    ; => (2 (3 (4 5)) (6))
> ```

10.4 宏

前面在已经简单介绍过宏。宏的调用和函数是很类似的，它的求值和函数差不多，但是有一个重要的区别是，宏的参数是出现在最后扩展后的表达式中，而函数参数是求值后才传递给这个函数：

```
(defmacro foo (arg)
  (list 'message "%d %d" arg arg))

(defun bar (arg)
  (message "%d %d" arg arg))

(let ((i 1))
  (bar (incf i)))                          ; => "2 2"

(let ((i 1))
  (foo (incf i)))                          ; => "2 3"
```

也许你对前面这个例子 foo 里为什么要用 list 函数很不解。其实宏可以这样看，如果把宏定义作一个表达式来运行，最后把参数用调用时的参数替换，这样就得到了宏调用最后用于求值的表达式。这个过程称为扩展。可以用 macroexpand 函数进行模拟：

```
(macroexpand '(foo (incf i))) ; => (message "%d %d" (incf i) (incf i))
```

上面用 macroexpand 得到的结果就是用于求值的表达式。

使用 macroexpand 可以使宏的编写变得容易一些。但是如果不能进行 debug 是很不方便的。在宏定义里可以引入 declare 表达式，它可以增加一些信息。目前只支持两类声明：debug 和 indent。debug 可选择的类型很多，具体参考 info elisp - Edebug 一章，一般情况下用 t 就足够了。indent 的类型比较简单，它可以使用这样几种类型：

- nil 也就是一般的方式缩进
- defun 类似 def 的结构，把第二行作为主体，对主体里的表达式使用同样的缩进

- 整数表示从第 n 个表达式后作为主体。比如 if 设置为 2，而 when 设置为 1
- 符号这个是最坏情况，你要写一个函数自己处理缩进。

看 when 的定义就能知道 declare 如何使用了：

```
(defmacro when (cond &rest body)
  (declare (indent 1) (debug t))
  (list 'if cond (cons 'progn body)))
```

实际上，declare 声明只是设置这个符号的属性列表：

```
(symbol-plist 'when)    ; => (lisp-indent-function 1 edebug-form-spec t)
```

> 思考题
> 一个比较常用的结构是当 buffer 是可读情况下，绑定 inhibit-read-only 值为 t 来强制插入字符串。请写一个这样的宏，处理好缩进和调用。

从前面宏 when 的定义可以看出直接使用 list，cons，append 构造宏是很麻烦的。为了使记号简洁，lisp 中有一个特殊的宏‘，称为 backquote。在这个宏里，所有的表达式都是引起（quote）的，如果要让一个表达式不引起（也就是列表中使用的是表达式的值），需要在前面加“,”，如果要让一个列表作为整个列表的一部分（ slice），可以用“,@”。

```
`(a list of ,(+ 2 3) elements)          ; => (a list of 5 elements)
(setq some-list '(2 3))                 ; => (2 3)
`(1 ,some-list 4 ,@some-list)           ; => (1 (2 3) 4 2 3)
```

有了这些标记，前面 when 这个宏可以写成：

```
(defmacro when (cond &rest body)
  `(if ,cond
       (progn ,@body)))
```

值得注意的是这个 backquote 本身就是一个宏，从这里可以看出宏除了减少重复代码这个作用之外的另一个用途：定义新的控制结构，甚至增加新的语法特性。

10.5 命令

emacs 运行时就是处于一个命令循环中，不断从用户那得到按键序列，然后调用对应命令来执行。lisp 编写的命令都含有一个 interactive 表达式。这个表达式指明了这个命令的参数。比如下面这个命令：

```
(defun hello-world (name)
  (interactive "sWhat you name? ")
  (message "Hello, %s" name))
```

现在你可以用 M-x 来调用这个命令。让我们来看看 interactive 的参数是什么意思。这个字符串的第一个字符（也称为代码字符）代表参数的类型，比如这里 s 代表参数的类型是一个字

符串，而其后的字符串是用来提示的字符串。如果这个命令有多个参数，可以在这个提示字符串后使用换行符分开，比如：

```
(defun hello-world (name time)
  (interactive "sWhat you name? \nnWhat the time? ")
  (message "Good %s, %s"
           (cond ((< time 13) "morning")
                 ((< time 19) "afternoon")
                 (t "evening"))
           name))
```

interactive 可以使用的代码字符很多，虽然有一定的规则，比如字符串用 s，数字用 n，文件用 f，区域用 r，但是还是很容易忘记，用的时候看 interactive 函数的文档还是很有必要的。但是不是所有时候都参数类型都能使用代码字符，而且一个好的命令，应该尽可能的让提供默认参数以让用户少花时间在输入参数上，这时，就有可能要自己定制参数。

首先学习和代码字符等价的几个函数。s 对应的函数是 read-string。比如：

```
(read-string "What your name? " user-full-name)
```

n 对应的函数是 read-number，文件对应 read-file-name。很容易记对吧。其实大部分代码字符都是有这样对应的函数或替换的方法（见表10.1）。

知道这些表达式如何用于 interactive 表达式里呢？简而言之，如果 interactive 的参数是一个表达式，则这个表达式求值后的列表元素对应于这个命令的参数。请看这个例子：

```
(defun read-hiden-file (file arg)
  (interactive
   (list (read-file-name "Choose a hiden file: " "~/" nil nil nil
                         (lambda (name)
                           (string-match "^\\." (file-name-nondirectory name))))
         current-prefix-arg))
  (message "%s, %S" file arg))
```

第一个参数是读入一个以“.” 开头的文件名，第二个参数为当前的前缀参数（ prefix argument），它可以用 C-u 或 C-u 加数字提供。list 把这两个参数构成一个列表。这就是命令一般的自定义设定参数的方法。

需要注意的是 current-prefix-arg 这个变量。这个变量当一个命令被调用，它就被赋与一个值，你可以用 C-u 就能改变它的值。在命令运行过程中，它的值始终都存在。即使你的命令不用参数，你也可以访问它：

```
(defun foo ()
  (interactive)
  (message "%S" current-prefix-arg))
```

用 C-u foo 调用它，你可以发现它的值是 (4)。那为什么大多数命令还单独为它设置一个参数呢？这是因为命令不仅是用户可以调用，很可能其它函数也可以调用，单独设置一个参数可以方便的用参数传递的方法调用这个命令。事实上所有的命令都可以不带参数，而使用前面介

表 10.1: interactive 字符代码等价的 lisp 函数

代码字符	代替的表达式
a	(completing-read prompt obarray 'fboundp t)
b	(read-buffer prompt nil t)
B	(read-buffer prompt)
c	(read-char prompt)
C	(read-command prompt)
d	(point)
D	(read-directory-name prompt)
e	(read-event)
f	(read-file-name prompt nil nil t)
F	(read-file-name prompt)
G	暂时不知道和 f 的差别
k	(read-key-sequence prompt)
K	(read-key-sequence prompt nil t)
m	(mark)
n	(read-number prompt)
N	(if current-prefix-arg (prefix-numeric-value current-prefix-arg) (read-number prompt))
p	(prefix-numeric-value current-prefix-arg)
P	current-prefix-arg
r	(region-beginning) (region-end)
s	(read-string prompt)
S	(completing-read prompt obarray nil t)
v	(read-variable prompt)
x	(read-from-minibuffer prompt nil nil t)
X	(eval (read-from-minibuffer prompt nil nil t))
z	(read-coding-system prompt)
Z	(and current-prefix-arg (read-coding-system prompt))

绍的方法在命令定义的部分读入需要的参数，但是为了提高命令的可重用性和代码的可读性，还是把参数分离到 interactive 表达式里好。

从现在开始可能会遇到很多函数，它们的用法有的简单，有的却复杂的要用大段篇幅来解释。我可能就会根据需要来解释一两个函数，就不一一介绍了。自己看 info elisp，用 i 来查找对应的函数。

> 思考题
> 写一个命令用来切换 major-mode。要求用户输入一个 major-mode 的名字，就切换到这个 major-mode，而且要提供一种补全的办法，去除所有不是 major-mode 的符号，这样用户需要输入少量词就能找到对应的 major-mode。

10.6 函数列表

```
(functionp OBJECT)
(apply FUNCTION &rest ARGUMENTS)
(funcall FUNCTION &rest ARGUMENTS)
(defmacro NAME ARGLIST [DOCSTRING] [DECL] BODY...)
(macroexpand FORM &optional ENVIRONMENT)
(declare &rest SPECS)
(‘ ARG)
(interactive ARGS)
(read-string PROMPT &optional INITIAL-INPUT HISTORY DEFAULT-VALUE
             INHERIT-INPUT-METHOD)
(read-file-name PROMPT &optional DIR DEFAULT-FILENAME MUSTMATCH
                INITIAL PREDICATE)
(completing-read PROMPT COLLECTION &optional PREDICATE
                 REQUIRE-MATCH INITIAL-INPUT HIST DEF
                 INHERIT-INPUT-METHOD)
(read-buffer PROMPT &optional DEF REQUIRE-MATCH)
(read-char &optional PROMPT INHERIT-INPUT-METHOD SECONDS)
(read-command PROMPT &optional DEFAULT-VALUE)
(read-directory-name PROMPT &optional DIR DEFAULT-DIRNAME
                     MUSTMATCH INITIAL)
(read-event &optional PROMPT INHERIT-INPUT-METHOD SECONDS)
(read-key-sequence PROMPT &optional CONTINUE-ECHO
                   DONT-DOWNCASE-LAST CAN-RETURN-SWITCH-FRAME
                   COMMAND-LOOP)
(read-number PROMPT &optional DEFAULT)
(prefix-numeric-value RAW)
(read-from-minibuffer PROMPT &optional INITIAL-CONTENTS KEYMAP
                      READ HIST DEFAULT-VALUE INHERIT-INPUT-METHOD)
(read-coding-system PROMPT &optional DEFAULT-CODING-SYSTEM)
```

10.7 变量列表

```
current-prefix-arg
```

10.8 问题解答

10.8.1 可选误差的浮点数比较

```
(defun approx-equal (x y &optional err)
  (if err
      (setq err (abs err))
    (setq err 1.0e-6))
  (or (and (= x 0) (= y 0))
      (< (/ (abs (- x y))
            (max (abs x) (abs y)))
         err)))
```

这个应该是很简单的一个问题。

10.8.2 遍历树的函数

```
(defun tree-mapcar (func tree)
  (if (consp tree)
      (mapcar (lambda (child)
                (tree-mapcar func child))
              tree)
    (funcall func tree)))
```

这个函数可能对于树算法比较熟悉的人一点都不难，就当练手吧。

10.8.3 宏 with-inhibit-read-only-t

```
(defmacro with-inhibit-read-only-t (&rest body)
  (declare (indent 0) (debug t))
  (cons 'let (cons '((inhibit-read-only t))
                   body)))
```

如果用 backquote 来改写一个就会发现这个宏会很容易写，而且更容易读了。

10.8.4 切换 major-mode 的命令

```
(defvar switch-major-mode-history nil)
(defun switch-major-mode (mode)
  (interactive
```

```
 (list
  (intern
   (completing-read "Switch to mode: "
                    obarray (lambda (s)
                              (and (fboundp s)
                                   (string-match "-mode$" (symbol-name s))))
                    t nil 'switch-major-mode-history))))
(setq switch-major-mode-history
      (cons (symbol-name major-mode) switch-major-mode-history))
(funcall mode))
```

这是我常用的一个命令之一。这个实现也是一个使用 minibuffer 历史的例子。

第十一章　正则表达式

如果你不懂正则表达式，而你还想进一步学习编程的话，那你应该停下手边的事情，先学学正则表达式。即使你不想学习编程，也不喜欢编程，学习一点正则表达式，也可能让你的文本编辑效率提高很多。

在这里，我不想详细介绍正则表达式，因为我觉得这类的文档已经很多了，比我写得好的文章多的是。如果你找不到一个好的入门教程，我建议你不妨看看 perlretut。我想说的是和 emacs 有关的正则表达式的内容，比如，和 Perl 正则表达式的差异、语法表格（ syntax table）和字符分类（ category）等。

11.1　与 Perl 正则表达式比较

Perl 是文本处理的首选语言。它内置强大而简洁的正则表达式，许多程序也都兼容 Perl 的正则表达式。说实话，就简洁而言，我对 emacs 的正则表达式是非常反感的，那么多的反斜线经常让我抓狂。首先，emacs 里的反斜线构成的特殊结构（ backslash construct）出现是相当频繁的。在 Perl 正则表达式里， “()[]—” 都是特殊字符，而 emacs 它们不是这样。所以它们匹配字符时是不用反斜线，而作为特殊结构时就要用反斜线。而事实上 “()—”作为字符来匹配的情形远远小于作为捕捉字符串和作或运算的情形概率小。而 emacs 的正则表达式又没有 Perl 那种简洁的记号，完全用字符串来表示，这样使得一个正则表达式常常一眼看去全是 “\\”。

到底要用多少个 \? 经常会记不住在 emacs 的正则表达式中应该用几个 \。有一个比较好的方法，首先想想没有引号引起时的正则表达式是怎样。比如对于特殊字符 \$ 要用\$，对于反斜线结构是 \(，\{，\| 等等。知道这个怎样写之后，再把所有 \ 替换成 \\，这就是最后写到双引号里形式。所以要匹配一个\，应该用\\，而写在引号里就应该用 \\\\ 来匹配。

emacs 里匹配的对象不仅包括字符串，还有 buffer，所以有一些对字符串和 buffer 有区分的结构。比如 \$ 对于字符串是匹配字符串的末尾，而在 buffer 里是行尾。而 \’ 匹配的是字符串和 buffer 的末尾。对应 ^ 和 \‘ 也是这样。

emacs 对字符有很多种分类方法，在正则表达式里也可以使用。比如按语法类型分类，可以用 \s 结构匹配一类语法分类的字符，最常用的比如匹配空格的 \s- 和匹配词的 \sw（等价于 \w）。这在 Perl 里是没有的。另外 emacs 里字符还对应一个或多个分类（ category），比如所有汉字在分类”c” 里。这样可以用 \cc 来匹配一个汉字。这在 Perl 里也有类似的分类。除此之外，还有一些预定义的字符分类，可以用 [:class:] 的形式，比如 [:digit:] 匹配 0-9 之间的数，[:ascii:] 匹配所有 ASCII 字符等等。在 Perl 里只定义几类最常用的字符集，比如 \d，\s，\w，但是我觉得非常实用。比 emacs 这样长的标记好用的多。

另外在用[] 表示一个字符集时，emacs 里不能用 \ 进行转义，事实上 \ 在这里不是一个特殊字符。所以 emacs 里的处理方法是，把特殊字符提前或放在后面，比如如果要在字符集里包括] 的话，要把] 放在第一位。而如果要包括 -，只能放在最后一位，如果要包括 ^ 不能放在第一位。如果只想匹配一个 ^，就只能用\^ 的形式。比较拗口，希望下面这个例子能帮你理解：

```
(let ((str "abc]-^]123"))
  (string-match "[]^0-9-]+" str)
  (match-string 0 str))                ; => "]-^]123"
```

最后提示一下，emacs 提供一个很好的正则表达式调试工具：M-x re-builder。它能显示 buffer 匹配正则表达式的字符串，并用不同颜色显示捕捉的字符串。

11.2 语法表格和分类表格简介

语法表格指的是 emacs 为每个字符都指定了语法功能，这为解析函数，复杂的移动命令等等提供了各种语法结构的起点和终点。语法表使用的数据结构是一种称为字符表（ char-table）的数组，它能以字符作为下标（还记得 emacs 里的字符就是整数吗）来得到对应的值。语法表里一个字符对应一个语法分类的列表。每一个分类都有一个助记字符（ mnemonic character）。一共有哪几类分类呢？

名称	助记符	说明
空白（ whitespace）	- 或␣	
词（ word）	w	
符号（ symbol）	_	这是除 word 之外其它用于变量和命令名的字符。
标点（ punctuation）	.	
open 和 close	(和)	一般是括号()[]
字符串引号（ string quote）	"	
转义符（ escape-syntax）	\	用于转义序列，比如 C 和 Lisp 字符串中的 \。
字符引号（ character quote）	/	
paired delimiter	$	只有 TeX 模式中使用
expression prefix	'	
注释开始和注释结束	< 和 >	
inherit standard syntax	@	
generic comment delimiter	!	

语法表格可以继承，所以基本上所有语法表格都是从 standard-syntax-table 继承而来，作少量修改，加上每个模式特有的语法构成就行了。一般来说记住几类重要的分类就行了，比如，空白包括空格，制表符，换行符，换页符。词包括所有的大小写字母，数字。符号一般按使用的模式而定，比如 C 中包含“_”，而 Lisp 中是 $&*+-_<>。可以用 M-x describe-syntax 来查看所有字符的语法分类。

字符分类（ category）是另一种分类方法，每个分类都有一个名字，对应一个从 到 ~ 的 ASCII 字符。可以用 M-x describe-categories 查看所有字符的分类。每一种分类都有说明，我就不详细解释了。

11.3 几个常用的函数

如果你要匹配的字符串中含有很多特殊字符，而你又想用正则表达式进行匹配，可以使用 regexp-quote 函数，它可以让字符串中的特殊字符自动转义。

一般多个可选词的匹配可以用或运算连接起来，但是这有两个不好的地方，一是要写很长的正则表达式，还含有很多反斜线，不好看，容易出错，也不好修改，二是效率很低。这时可以使用 regexp-opt 还产生一个更好的正则表达式：

```
(regexp-opt '("foobar" "foobaz" "foo")) ; => "foo\\(?:ba[rz]\\)?"
```

11.4 函数列表

```
(regexp-quote STRING)
(regexp-opt STRINGS &optional PAREN)
```

11.5 命令列表

```
describe-syntax
describe-categories
```

第十二章　操作对象之一——缓冲区

缓冲区（ buffer）是用来保存要编辑文本的对象。通常缓冲区都是和文件相关联的，但是也有很多缓冲区没有对应的文件。emacs 可以同时打开多个文件，也就是说能同时有很多个缓冲区存在，但是在任何时候都只有一个缓冲区称为当前缓冲区（ current buffer）。即使在 lisp 编程中也是如此。许多编辑和移动的命令只能针对当前缓冲区。

12.1　缓冲区的名字

emacs 里的所有缓冲区都有一个不重复的名字。所以和缓冲区相关的函数通常都是可以接受一个缓冲区对象或一个字符串作为缓冲区名查找对应的缓冲区。下面的函数列表中如果参数是 BUFFER-OR-NAME 则是能同时接受缓冲区对象和缓冲区名的函数，否则只能接受一种参数。有一个习惯是名字以空格开头的缓冲区是临时的，用户不需要关心的缓冲区。所以现在一般显示缓冲区列表的命令都不会显示这样的变量，除非这个缓冲区关联一个文件。

要得到缓冲区的名字，可以用 buffer-name 函数，它的参数是可选的，如果不指定参数，则返回当前缓冲区的名字，否则返回指定缓冲区的名字。更改一个缓冲区的名字用 rename-buffer，这是一个命令，所以你可以用 M-x 调用来修改当前缓冲区的名字。如果你指定的名字与现有的缓冲区冲突，则会产生一个错误，除非你使用第二个可选参数以产生一个不相同的名字，通常是在名字后加上¡序号¿ 的方式使名字变得不同。你也可以用 generate-new-buffer-name 来产生一个唯一的缓冲区名。

12.2　当前缓冲区

当前缓冲区可以用 current-buffer 函数得到。当前缓冲区不一定是显示在屏幕上的那个缓冲区，你可以用 set-buffer 来指定当前缓冲区。但是需要注意的是，当命令返回到命令循环时，光标所在的缓冲区会自动成为当前缓冲区。这也是单独在*scratch* 中执行 set-buffer 后并不能改变当前缓冲区，而必须使用 progn 语句同时执行多个语句才能改变当前缓冲区的原因：

```
(set-buffer "*Messages*")   ; => #<buffer *Messages*>
(message (buffer-name))                  ; => "*scratch*"
(progn
  (set-buffer "*Messages*")
  (message (buffer-name)))               ; "*Messages*"
```

但是你不能依赖命令循环来把当前缓冲区设置成使用 set-buffer 之前的。因为这个命令很可以会被另一个程序员来调用。你也不能直接用 set-buffer 设置成原来的缓冲区，比如：

```
(let (buffer-read-only
      (obuf (current-buffer)))
  (set-buffer ...)
  ...
  (set-buffer obuf))
```

因为 set-buffer 不能处理错误或退出情况。正确的作法是使用 save-current-buffer、with-current-buffer 和 save-excursion 等方法。save-current-buffer 能保存当前缓冲区，执行其中的表达式，最后恢复为原来的缓冲区。如果原来的缓冲区被关闭了，则使用最后使用的那个当前缓冲区作为语句返回后的当前缓冲区。lisp 中很多以 with 开头的宏，这些宏通常是在不改变当前状态下，临时用另一个变量代替现有变量执行语句。比如 with-current-buffer 使用另一个缓冲区作为当前缓冲区，语句执行结束后恢复成执行之前的那个缓冲区。

```
(with-current-buffer BUFFER-OR-NAME
   body)
```

相当于：

```
(save-current-buffer
  (set-buffer BUFFER-OR-NAME)
  body)
```

save-excursion 与 save-current-buffer 不同之处在于，它不仅保存当前缓冲区，还保存了当前的位置和 mark。在*scratch* 缓冲区中运行下面两个语句就能看出它们的差别了：

```
(save-current-buffer
  (set-buffer "*scratch*")
  (goto-char (point-min))
  (set-buffer "*Messages*"))

(save-excursion
  (set-buffer "*scratch*")
  (goto-char (point-min))
  (set-buffer "*Messages*"))
```

12.3 创建和关闭缓冲区

产生一个缓冲区必须用给这个缓冲区一个名字，所以两个能产生新缓冲区的函数都是以一个字符串为参数：get-buffer-create 和 generate-new-buffer。这两个函数的差别在于前者如果给定名字的缓冲区已经存在，则返回这个缓冲区对象，否则新建一个缓冲区，名字为参数字符串，而后者在给定名字的缓冲区存在时，会使用加上后缀 ¡N¿（ N 是一个整数，从 2 开始）的名字创建新的缓冲区。

关闭一个缓冲区可以用 kill-buffer。当关闭缓冲区时，如果要用户确认是否要关闭缓冲区，可以加到 kill-buffer-query-functions 里。如果要做一些善后处理，可以用 kill-buffer-hook。

通常一个接受缓冲区作为参数的函数都需要参数所指定的缓冲区是存在的。如果要确认一个缓冲区是否依然还存在可以使用 buffer-live-p。

要对所有缓冲区进行某个操作，可以用 buffer-list 获得所有缓冲区的列表。

如果你只是想使用一个临时的缓冲区，而不想先建一个缓冲区，使用结束后又需要关闭这个缓冲区，可以用 with-temp-buffer 这个宏。从这个宏的名字可以看出，它所做的事情是先新建一个临时缓冲区，并把这个缓冲区作为当前缓冲区，使用结束后，关闭这个缓冲区，并恢复之前的缓冲区为当前缓冲区。

12.4 在缓冲区内移动

在学会移动函数之前，先要理解两个概念：位置（ position）和标记（ mark）。位置是指某个字符在缓冲区内的下标，它从 1 开始。更准确的说位置是在两个字符之间，所以有在位置之前的字符和在位置之后的字符之说。但是通常我们说在某个位置的字符都是指在这个位置之后的字符。

标记和位置的区别在于位置会随文本插入和删除而改变位置。一个标记包含了缓冲区和位置两个信息。在插入和删除缓冲区里的文本时，所有的标记都会检查一遍，并重新设置位置。这对于含有大量标记的缓冲区处理是很花时间的，所以当你确认某个标记不用的话应该释放这个标记。

创建一个标记使用函数 make-marker。这样产生的标记不会指向任何地方。你需要用 set-marker 命令来设置标记的位置和缓冲区。

```
(setq foo (make-marker))                ; => #<marker in no buffer>
(set-marker foo (point))                ; => #<marker at 3594 in *scratch*>
```

也可以用 point-marker 直接得到 point 处的标记。或者用 copy-marker 复制一个标记或者直接用位置生成一个标记。

```
(point-marker)                          ; => #<marker at 3516 in *scratch*>
(copy-marker 20)                        ; => #<marker at 20 in *scratch*>
(copy-marker foo)                       ; => #<marker at 3502 in *scratch*>
```

如果要得一个标记的内容，可以用 marker-position 和 marker-buffer。

```
(marker-position foo)                   ; => 3502
(marker-buffer foo)                     ; => #<buffer *scratch*>
```

位置就是一个整数，而标记在一般情况下都是以整数的形式使用，所以很多接受整数运算的函数也可以接受标记为参数。比如加减乘。

和缓冲区相关的变量，有的可以用变量得到，比如缓冲区关联的文件名，有的只能用函数来得到，比如 point。point 是一个特殊的缓冲区位置，许多命令在这个位置进行文本插入。每个缓冲区都有一个 point 值，它总是比函数 point-min 大，比另一个函数 point-max 返回值小。注意，point-min 的返回值不一定是 1，point-max 的返回值也不定是比缓冲区大小函数 buffer-size 的返回值大 1 的数，因为 emacs 可以把一个缓冲区缩小（ narrow）到一个区域，这时 point-min 和 point-max 返回值就是这个区域的起点和终点位置。所以要得到 point 的范围，只能用这两个函数，而不能用 1 和 buffer-size 函数。

和 point 类似，有一个特殊的标记称为 “the mark”。它指定了一个区域的文本用于某些命令，比如 kill-region，indent-region。可以用 mark 函数返回当前 mark 的值。如果使用 transient-mark-mode，而且 mark-even-if-inactive 值是 nil 的话，在 mark 没有激活时（也就是 mark-active 的值为 nil），调用 mark 函数会产生一个错误。如果传递一个参数 force 才能返回当前缓冲区 mark 的位置。mark-marker 能返回当前缓冲区的 mark，这不是 mark 的拷贝，所以设置它的值会改变当前 mark 的值。set-mark 可以设置 mark 的值，并激活 mark。每个缓冲区还维护一个 mark-ring，这个列表里保存了 mark 的前一个值。当一个命令修改了 mark 的值时，通常要把旧的值放到 mark-ring 里。可以用 push-mark 和 pop-mark 加入或删除 mark-ring 里的元素。当

缓冲区里 mark 存在且指向某个位置时，可以用 region-beginning 和 region-end 得到 point 和 mark 中较小的和较大的值。当然如果使用 transient-mark-mode 时，需要激活 mark，否则会产生一个错误。

> 思考题
> 写一个命令，对于使用 transient-mark-mode 时，当选中一个区域时显示区域的起点和终点，否则显示 point-min 和 point-max 的位置。如果不使用 transient-mark-mode，则显示 point 和 mark 的位置。

按单个字符位置来移动的函数主要使用 goto-char 和 forward-char、backward-char。前者是按缓冲区的绝对位置移动，而后者是按 point 的偏移位置移动比如：

```
(goto-char (point-min))                ; 跳到缓冲区开始位置
(forward-char 10)                      ; 向前移动~10 个字符
(forward-char -10)                     ; 向后移动~10 个字符
```

可能有一些写 elisp 的人没有读文档或者贪图省事，就在写的 elisp 里直接用 beginning-of-buffer 和 end-of-buffer 来跳到缓冲区的开头和末尾，这其实是不对的。因为这两个命令还做了其它事情，比如设置标记等等。同样，还有一些函数都是不推荐在 elisp 中使用的，如果你准备写一个要发布 elisp，还是要注意一下。

按词移动使用 forward-word 和 backward-word。至于什么是词，这就要看语法表格的定义了。

按行移动使用 forward-line。没有 backward-line。forward-line 每次移动都是移动到行首的。所以，如果要移动到当前行的行首，使用(forward-line 0)。如果不想移动就得到行首和行尾的位置，可以用 line-beginning-position 和 line-end-position。得到当前行的行号可以用 line-number-at-pos。需要注意的是这个行号是从当前状态下的行号，如果使用 narrow-to-region 或者用 widen 之后都有可能改变行号。

由于 point 只能在 point-min 和 point-max 之间，所以 point 位置测试有时是很重要的，特别是在循环条件测试里。常用的测试函数是 bobp（ beginning of buffer predicate）和 eobp（ end of buffer predicate）。对于行位置测试使用 bolp（ beginning of line predicate）和 eolp（ end of line predicate）。

12.5 缓冲区的内容

要得到整个缓冲区的文本，可以用 buffer-string 函数。如果只要一个区间的文本，使用 buffer-substring 函数。point 附近的字符可以用 char-after 和 char-before 得到。point 处的词可以用 current-word 得到，其它类型的文本，比如符号，数字，S 表达式等等，可以用 thing-at-point 函数得到。

> 思考题
> 参考 thing-at-point 写一个命令标记光标处的 S 表达式。这个命令和 mark-sexp 不同的是，它能从整个 S 表达式的开始标记。

12.6 修改缓冲区的内容

要修改缓冲区的内容，最常见的就是插入、删除、查找、替换了。下面就分别介绍这几种操作。

插入文本最常用的命令是 insert。它可以插入一个或者多个字符串到当前缓冲区的 point 后。也可以用 insert-char 插入单个字符。插入另一个缓冲区的一个区域使用 insert-buffer-substring。

删除一个或多个字符使用 delete-char 或 delete-backward-char。删除一个区间使用 delete-region。如果既要删除一个区间又要得到这部分的内容使用 delete-and-extract-region，它返回包含被删除部分的字符串。

最常用的查找函数是 re-search-forward 和 re-search-backward。这两个函数参数如下：

```
(re-search-forward REGEXP &optional BOUND NOERROR COUNT)
(re-search-backward REGEXP &optional BOUND NOERROR COUNT)
```

其中 BOUND 指定查找的范围，默认是 point-max（对于 re-search-forward）或 point-min（对于 re-search-backward），NOERROR 是当查找失败后是否要产生一个错误，一般来说在 elisp 里都是自己进行错误处理，所以这个一般设置为 t，这样在查找成功后返回区配的位置，失败后会返回 nil。COUNT 是指定查找匹配的次数。

替换一般都是在查找之后进行，也是使用 replace-match 函数。和字符串的替换不同的是不需要指定替换的对象了。

> 思考题
> 从 OpenOffice 字处理程序里拷贝到 emacs 里的表格通常都是每一个单元格就是一行的。写一个命令，让用户输入表格的列数，把选中区域转换成用制表符分隔的表格。

12.7 函数列表

```
(buffer-name &optional BUFFER)
(rename-buffer NEWNAME &optional UNIQUE)
(generate-new-buffer-name NAME &optional IGNORE)
(current-buffer)
(set-buffer BUFFER-OR-NAME))
(save-current-buffer &rest BODY)
(with-current-buffer BUFFER-OR-NAME &rest BODY)
(save-excursion &rest BODY)
(get-buffer-create NAME)
(generate-new-buffer NAME)
(kill-buffer BUFFER-OR-NAME)
(buffer-live-p OBJECT)
(buffer-list &optional FRAME)
(with-temp-buffer &rest BODY)
(make-marker)
```

```
(set-marker MARKER POSITION &optional BUFFER)
(point-marker)
(copy-marker MARKER &optional TYPE)
(marker-position MARKER)
(marker-buffer MARKER)
(point)
(point-min)
(point-max)
(buffer-size &optional BUFFER)
(mark &optional FORCE)
(mark-marker)
(set-mark POS)
(push-mark &optional LOCATION NOMSG ACTIVATE)
(pop-mark)
(region-beginning)
(region-end)
(goto-char POSITION)
(forward-char &optional N)
(backward-char &optional N)
(beginning-of-buffer &optional ARG)
(end-of-buffer &optional ARG)
(forward-word &optional ARG)
(backward-word &optional ARG)
(forward-line &optional N)
(line-beginning-position &optional N)
(line-end-position &optional N)
(line-number-at-pos &optional POS)
(narrow-to-region START END)
(widen)
(bobp)
(eobp)
(bolp)
(eolp)
(buffer-string)
(buffer-substring START END)
(char-after &optional POS)
(char-before &optional POS)
(current-word &optional STRICT REALLY-WORD)
(thing-at-point THING)
(insert &rest ARGS)
(insert-char CHARACTER COUNT &optional INHERIT)
(insert-buffer-substring BUFFER &optional START END)
(delete-char N &optional KILLFLAG)
```

```
(delete-backward-char N &optional KILLFLAG)
(delete-region START END)
(delete-and-extract-region START END)
(re-search-forward REGEXP &optional BOUND NOERROR COUNT)
(re-search-backward REGEXP &optional BOUND NOERROR COUNT)
```

12.8 问题解答

12.8.1 可选择区域也可不选择区域的命令

```
(defun show-region (beg end)
  (interactive
   (if (or (null transient-mark-mode)
           mark-active)
       (list (region-beginning) (region-end))
     (list (point-min) (point-max))))
  (message "Region start from %d to %d" beg end))
```

这是通常那种如果选择区域则对这个区域应用命令，否则对整个缓冲区应用命令的方法。我喜欢用 transient-mark-mode，因为它让这种作用于区域的命令更灵活。当然也有人反对，无所谓了，emacs 本身就是很个性化的东西。

12.8.2 标记整个 S 表达式

```
(defun mark-whole-sexp ()
  (interactive)
  (let ((bound (bounds-of-thing-at-point 'sexp)))
    (if bound
        (progn
          (goto-char (car bound))
          (set-mark (point))
          (goto-char (cdr bound)))
      (message "No sexp found at point!"))))
```

学习过程中应该可以看看其它一些函数是怎样实现的，从这些源代码中常常能学到很多有用的技巧和方法。比如要标记整个 S 表达式，联想到 thing-at-point 能得到整个 S 表达式，那自然能得到整个 S 表达式的起点和终点了。所以看看 thing-at-point 的实现，一个很简单的函数，一眼就能发现其中最关键的函数是 bounds-of-thing-at-point，它能返回某个语法实体（ syntactic entity）的起点和终点。这样这个命令就很容易就能写出来了。从这个命令中还应该注意到的是对于错误应该很好的处理，让用户明白发生什么错了。比如这里，如果当前光标下没有 S 表达式时，bound 变量为 nil，如果不进行判断，会出现错误：

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

加上这个判断，用户就明白发生什么事了。

12.8.3 oowriter 表格转换

实现这个目的有多种方法：

1. 一行一行移动，删除回车，替换成制表符：

```
(defun oowrite-table-convert (col beg end)
  (interactive "nColumns of table: \nr")
  (setq col (1- col))
  (save-excursion
    (save-restriction
      (narrow-to-region beg end)
      (goto-char (point-min))
      (while (not (eobp))
        (dotimes (i col)
          (forward-line 1)
          (backward-delete-char 1)
          (insert-char ?\t 1))
        (forward-line 1)))))
```

2. 用 subst-char-in-region 函数直接替换：

```
(defun oowrite-table-convert (col beg end)
  (interactive "nColumns of table: \nr")
  (save-excursion
    (save-restriction
      (narrow-to-region beg end)
      (goto-char (point-min))
      (while (not (eobp))
        (subst-char-in-region
         (point) (progn (forward-line col) (1- (point)))
         ?\n ?\t)))))
```

3. 用 re-search-forward 和 replace-match 查找替换

```
(defun oowrite-table-convert (col beg end)
  (interactive "nColumns of table: \nr")
  (let (start bound)
    (save-excursion
      (save-restriction
        (narrow-to-region beg end)
        (goto-char (point-min))
        (while (not (eobp))
          (setq start (point))
          (forward-line col)
          (setq bound (copy-marker (1- (point))))
```

```
(goto-char start)
(while (re-search-forward "\n" bound t)
  (replace-match "\t"))
(goto-char (1+ bound)))))))
```

之所以要给出这三种方法，是想借此说明 elisp 编程其实要实现一个目的通常有很多种方法，选择一种适合的方法。比如这个问题较好的方法是使用第二种方法，前提是你要知道有 subst-char-in-region 这个函数，这就要求你对 emacs 提供的内置的函数比较熟悉了，没有别的办法，只有自己多读 elisp manual，如果你真想学习 elisp 的话，读 manual 还是值得的，我每读一遍都会有一些新的发现。如果你不知道这个函数，只知道常用的函数，那么相比较而言，第一种方法是比较容易想到，也比较容易实现的。但是事实上第三种方法才是最重要的方法，因为这个方法是适用范围最广的。试想一下你如果不是替换两个字符，而是字符串的话，前面两种方法都没有办法使用了，而这个方法只要稍微修改就能用了。

另外，需要特别说明的是这个命令中 bound 使用的是一个标记而不是一个位置，如果替换的字符串和删除的字符串是相同长度的，当前用什么都可以，否则就要注意了，因为在替换之后，边界就有可能改变。这也是写查找替换的函数中很容易出现的一个错误。解决的办法，一是像我这样用一个标记来记录边界位置。另一种就是用 narrow-to-region 的方法，先把缓冲区缩小到查找替换的区域，结束后用 widen 展开。当然为了省事，可以直接用 save-restriction。

第十三章　操作对象之二——窗口

首先还是要定义一下什么是窗口（ window）。窗口是屏幕上用于显示一个缓冲区的部分。和它要区分开来的一个概念是 frame。frame 是 Emacs 能够使用屏幕的部分。可以用窗口的观点来看 frame 和窗口，一个 frame 里可以容纳多个（至少一个）窗口，而 Emacs 可以有多个 frame。（可能需要和通常所说的窗口的概念要区分开来，一般来说，我们所说的其它程序的窗口更类似于 Emacs 的一个 frame，所以也有人认为这里 window 译为窗格更好一些。但是窗格这个词是一个生造出来的词，我还是用窗口比较顺一些，大家自己注意就行了。）在任何时候，都有一个被选中的 frame，而在这个 frame 里有一个被选中的窗口，称为选择的窗口（ selected window）。

13.1　分割窗口

刚启动时，emacs 都是只有一个 frame 一个窗口。多个窗口都是用分割窗口的函数生成的。分割窗口的内建函数是 split-window。这个函数的参数如下：

```
(split-window &optional window size horizontal)
```

这个函数的功能是把当前或者指定窗口进行分割，默认分割方式是水平分割，可以将参数中的 horizontal 设置为 non-nil 的值，变成垂直分割。如果不指定大小，则分割后两个窗口的大小是一样的。分割后的两个窗口里的缓冲区是同一个缓冲区。使用这个函数后，光标仍然在原窗口，而返回的新窗口对象：

```
(selected-window)                       ; => #<window 136 on *scratch*>
(split-window)                          ; => #<window 138 on *scratch*>
```

需要注意的是，窗口的分割也需要用树的结构来看分割后的窗口，比如图13.1这样一个过程可以看成是这样一种结构：

```
(win1) ->  (win1 win2) -> ((win1 win3) win2) -> ((win1 (win3 win4)) win2)
```

事实上可以用 window-tree 函数得到当前窗口的结构，如果忽略 minibuffer 对应的窗口，得到的应该类似这样的一个结果：

```
(nil (0 0 170 42)
     (t (0 0 85 42)
        #<win 3>
        (nil (0 21 85 42) #<win 8> #<win 10>))
     #<win 6>)
```

window-tree 返回值的第一个元素代表子窗口的分割方式，nil 表示水平分割， t 表示垂直分割。第二个元素代表整个结构的大小，这四个数字可以看作是左上和右下两个顶点的坐标。其余元素是子窗口。每个子窗口也是同样的结构。所以把前面这个列表还原成窗口排列应该是如图13.2所示。

图 13.1: 一个窗口分割操作

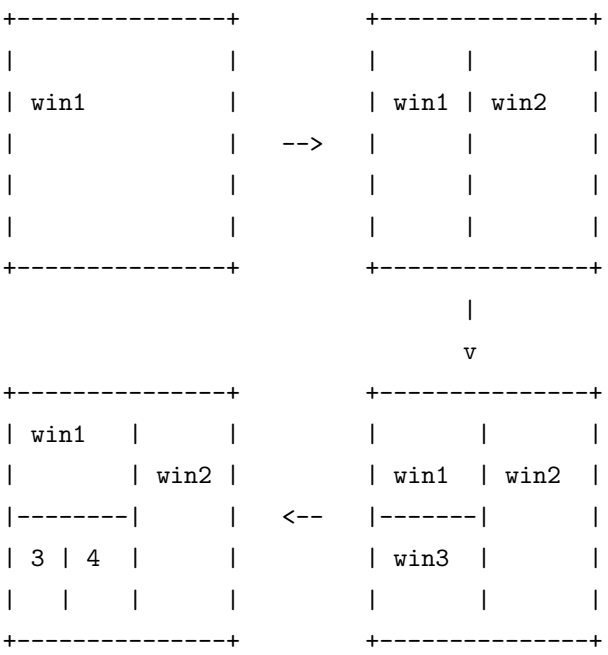
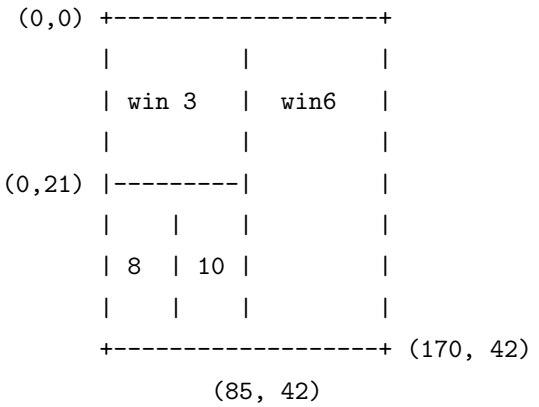


图 13.2: 窗口位置信息表示



由图可以注意到由 window-tree 返回的结果一些窗口的大小不能确定，比较上面的 win 8 和 win 10 只能知道它们合并起来的大小，不能确定它们分别的宽度是多少。

13.2 删除窗口

如果要让一个窗口不显示在屏幕上，要使用 delete-window 函数。如果没有指定参数，删除的窗口是当前选中的窗口，如果指定了参数，删除的是这个参数对应的窗口。删除的窗口多出来的空间会自动加到它的邻接的窗口中。如果要删除除了当前窗口之外的窗口，可以用 delete-other-windows 函数。

当一个窗口不可见之后，这个窗口对象也就消失了。

```
(setq foo (selected-window))            ; => #<window 90 on *scratch*>
(delete-window)
(windowp foo)                           ; => t
(window-live-p foo)                     ; => nil
```

13.3 窗口配置

窗口配置(window configuration) 包含了 frame 中所有窗口的位置信息：窗口大小，显示的缓冲区，缓冲区中光标的位置和 mark，还有 fringe，滚动条等等。用 current-window-configuration 得到当前窗口配置，用 set-window-configuration 来还原。

```
(setq foo (current-window-configuration))
;; do sth to make some changes on windows
(set-window-configuration foo)
```

13.4 选择窗口

可以用 selected-window 得到当前光标所在的窗口。

```
(selected-window)                       ; => #<window 104 on *scratch*>
```

可以用 select-window 函数使某个窗口变成选中的窗口。

```
(progn
  (setq foo (selected-window))
  (message "Original window: %S" foo)
  (other-window 1)
  (message "Current window: %S" (selected-window))
  (select-window foo)
  (message "Back to original window: %S" foo))
```

两个特殊的宏可以保存窗口位置执行语句：save-selected-window 和 with-selected-window。它们的作用是在执行语句结束后选择的窗口仍留在执行语句之前的窗口。with-selected-window 和 save-selected-window 几乎相同，只不过 save-selected-window 选择了其它窗口。这两个宏不

会保存窗口的位置信息，如果执行语句结束后，保存的窗口已经消失，则会选择最后一个选择的窗口。

```
;; 让另一个窗口滚动到缓冲区开始
(save-selected-window
  (select-window (next-window))
  (goto-char (point-min)))
```

当前 frame 里所有的窗口可以用 window-list 函数得到。可以用 next-window 来得到在 window-list 里排在某个 window 之后的窗口。对应的用 previous-window 得到排在某个 window 之前的窗口。

```
(selected-window)                       ; => #<window 245 on *scratch*>
(window-list)
;; => (#<window 245 on *scratch*> #<window 253 on *scratch*>
;;     #<window 251 on *info*>)
(next-window)                           ; => #<window 253 on *scratch*>
(next-window (next-window))             ; => #<window 251 on *info*>
(next-window (next-window (next-window))) ; => #<window 245 on *scratch*>
```

walk-windows 可以遍历窗口，相当于(mapc proc (window-list))。get-window-with-predicate 用于查找符合某个条件的窗口。

13.5 窗口大小信息

窗口是一个长方形区域，所以窗口的大小信息包括它的高度和宽度。用来度量窗口大小的单位都是以字符数来表示，所以窗口高度为 45 指的是这个窗口可以容纳 45 行字符，宽度为 140 是指窗口一行可以显示 140 个字符。

mode line 和 header line 都包含在窗口的高度里，所以有 window-height 和 window-body-height 两个函数，后者返回把 mode-line 和 header line 排除后的高度。

```
(window-height)                         ; => 45
(window-body-height)                    ; => 44
```

滚动条和 fringe 不包括在窗口的亮度里，window-width 返回窗口的宽度：

```
(window-width)                          ; => 72
```

也可以用 window-edges 返回各个顶点的坐标信息：

```
(window-edges)                          ; => (0 0 73 45)
```

window-edges 返回的位置信息包含了滚动条、fringe、mode line、header line 在内，window-inside-edges 返回的就是窗口的文本区域的位置：

```
(window-inside-edges)                   ; => (1 0 73 44)
```

如果需要的话也可以得到用像素表示的窗口位置信息：

```
(window-pixel-edges)                    ; => (0 0 511 675)
(window-inside-pixel-edges)             ; => (7 0 511 660)
```

思考题

current-window-configuration 可以将当前窗口的位置信息保存到一个变量中以便将来恢复窗口。但是这个对象没有读入形式，所以不能保存到文件中。请写一个函数可以把当前窗口的位置信息生成一个列表，然后用一个函数就能从这个列表恢复窗口。提示：这个列表结构用窗口的分割顺序表示。比如用这样一个列表表示对应的窗口：

```
;; +---------------+
;; |   |   |       |
;; |   |   |       |
;; |-------|       |
;; |       |       |
;; |       |       |
;; +---------------+
;; =>
(horizontal 73
            (vertical 22
                      (horizontal 36 win win)
                      win)
            win)
```

13.6 窗口对应的缓冲区

窗口对应的缓冲区可以用 window-buffer 函数得到：

```
(window-buffer)                         ; => #<buffer *scratch*>
(window-buffer (next-window))           ; => #<buffer *info*>
```

缓冲区对应的窗口也可以用 get-buffer-window 得到。如果有多个窗口显示同一个缓冲区，那这个函数只能返回其中的一个，由 window-list 决定。如果要得到所有的窗口，可以用 get-buffer-window-list。

```
(get-buffer-window (get-buffer "*scratch*"))
;; => #<window 268 on *scratch*>
(get-buffer-window-list (get-buffer "*scratch*"))
;; => (#<window 268 on *scratch*> #<window 270 on *scratch*>)
```

让某个窗口显示某个缓冲区可以用 set-window-buffer 函数。让选中窗口显示某个缓冲区也可以用 switch-to-buffer，但是一般不要在 elisp 编程中用这个命令，如果需要让某个缓冲区成为

当前缓冲区使用 set-buffer 函数，如果要让当前窗口显示某个缓冲区，使用 set-window-buffer 函数。

让一个缓冲区可见可以用 display-buffer。默认的行为是当缓冲区已经显示在某个窗口中时，如果不是当前选中窗口，则返回那个窗口，如果是当前选中窗口，且如果传递的 not-this-window 参数为 non-nil 时，会新建一个窗口，显示缓冲区。如果没有任何窗口显示这个缓冲区，则新建一个窗口显示缓冲区，并返回这个窗口。display-buffer 是一个比较高级的命令，用户可以通过一些变量来改变这个命令的行为。比如控制显示的 pop-up-windows， display-buffer-reuse-frames，pop-up-frames，控制新建窗口高度的 split-height-threshold，even-window-heights，控制显示的 frame 的 special-display-buffer-names，special-display-regexps， special-display-function，控制是否应该显示在当前选中窗口 same-window-buffer-names，same-window-regexps 等等。如果这些还不能满足你的要求（事实上我觉得已经太复杂了），你还可以自己写一个函数，将 display-buffer-function 设置成这个函数。

> 思考题
> 前一个思考题只能还原窗口，不能还原缓冲区。请修改一下使它能保存缓冲区信息，还原时让对应的窗口显示对应的缓冲区。

13.7 改变窗口显示区域

每个窗口会保存一个显示缓冲区的起点位置，这个位置对应于窗口左上角光标在缓冲区里的位置。可以用 window-start 函数得到某个窗口的起点位置。可以通过 set-window-start 来改变显示起点位置。可以通过 pos-visible-in-window-p 来检测缓冲区中某个位置是否是可见的。但是直接通过 set-window-start 来控制显示比较容易出现错误，因为 set-window-start 并不会改变 point 所在的位置，在窗口调用 redisplay 函数之后 point 会跳到相应的位置。如果你确实有这个需要，我建议还是用：(with-selected-window window (goto-char pos)) 来代替。

13.8 函数列表

```
(windowp OBJECT)
(split-window &optional WINDOW SIZE HORFLAG)
(selected-window)
(window-tree &optional FRAME)
(delete-window &optional WINDOW)
(delete-other-windows &optional WINDOW)
(current-window-configuration &optional FRAME)
(set-window-configuration CONFIGURATION)
(other-window ARG &optional ALL-FRAMES)
(save-selected-window &rest BODY)
(with-selected-window WINDOW &rest BODY)
(window-list &optional FRAME MINIBUF WINDOW)
(next-window &optional WINDOW MINIBUF ALL-FRAMES)
(previous-window &optional WINDOW MINIBUF ALL-FRAMES)
```

```
(walk-windows PROC &optional MINIBUF ALL-FRAMES)
(get-window-with-predicate PREDICATE &optional MINIBUF ALL-FRAMES DEFAULT)
(window-height &optional WINDOW)
(window-body-height &optional WINDOW)
(window-width &optional WINDOW)
(window-edges &optional WINDOW)
(window-inside-edges &optional WINDOW)
(window-pixel-edges &optional WINDOW)
(window-inside-pixel-edges &optional WINDOW)
(window-buffer &optional WINDOW)
(get-buffer-window BUFFER-OR-NAME &optional FRAME)
(get-buffer-window-list BUFFER-OR-NAME &optional MINIBUF FRAME)
(set-window-buffer WINDOW BUFFER-OR-NAME &optional KEEP-MARGINS)
(switch-to-buffer BUFFER-OR-NAME &optional NORECORD)
(display-buffer BUFFER-OR-NAME &optional NOT-THIS-WINDOW FRAME)
(window-start &optional WINDOW)
(set-window-start WINDOW POS &optional NOFORCE)
```

13.9 问题解答

13.9.1 保存窗口位置信息

这是我的答案。欢迎提出改进意见。

```
(defun my-window-tree-to-list (tree)
  (if (windowp tree)
      'win
    (let ((dir (car tree))
          (children (cddr tree)))
      (list (if dir 'vertical 'horizontal)
            (if dir
                (my-window-height (car children))
              (my-window-width (car children)))
            (my-window-tree-to-list (car children))
            (if (> (length children) 2)
                (my-window-tree-to-list (cons dir (cons nil (cdr children))))
              (my-window-tree-to-list (cadr children)))))))

(defun my-window-width (win)
  (if (windowp win)
      (window-width win)
    (let ((edge (cadr win)))
      (- (nth 2 edge) (car edge)))))
```

```
(defun my-window-height (win)
  (if (windowp win)
      (window-height win)
    (let ((edge (cadr win)))
      (- (nth 3 edge) (cadr edge)))))

(defun my-list-to-window-tree (conf)
  (when (listp conf)
    (let (newwin)
      (setq newwin (split-window nil (cadr conf)
                                 (eq (car conf) 'horizontal)))
      (my-list-to-window-tree (nth 2 conf))
      (select-window newwin)
      (my-list-to-window-tree (nth 3 conf)))))

(defun my-restore-window-configuration (winconf)
  (delete-other-windows)
  (my-list-to-window-tree winconf))

(defun my-current-window-configuration ()
  (my-window-tree-to-list (car (window-tree))))

;; test code here
(setq foo (my-current-window-configuration))
;; do sth to change windows
(my-restore-window-configuration foo)
```

13.9.2 改进的保存窗口信息的函数

由于缓冲区对象也是没有读入形式的，所以返回的列表里只能用缓冲区名来代表缓冲区，只要没有修改过缓冲区的名字，就能正确的还原缓冲区。如果对于访问文件的缓冲区，使用文件名可能是更好的想法。保存信息只要对 my-window-tree-to-list 函数做很小的修改就能用了。而恢复窗口则要做较大改动。my-list-to-window-tree 加了一个函数参数，这样这个函数的可定制性更高一些。

```
(defun my-window-tree-to-list (tree)
  (if (windowp tree)
      (buffer-name (window-buffer tree))
    (let ((dir (car tree))
          (children (cddr tree)))
      (list (if dir 'vertical 'horizontal)
            (if dir
                (my-window-height (car children))
              (my-window-width (car children)))
```

```
            (my-window-tree-to-list (car children))
            (if (> (length children) 2)
                (my-window-tree-to-list (cons dir (cons nil (cdr children))))
              (my-window-tree-to-list (cadr children)))))))

(defun my-list-to-window-tree (conf set-winbuf)
  (let ((newwin (split-window nil (cadr conf)
                              (eq (car conf) 'horizontal))))
    (if (eq (car conf) 'horizontal)
        (progn
          (funcall set-winbuf (selected-window) (nth 2 conf))
          (select-window newwin)
          (if (listp (nth 3 conf))
              (my-list-to-window-tree (nth 3 conf))
            (funcall set-winbuf newwin (nth 3 conf))))
      (if (listp (nth 2 conf))
          (my-list-to-window-tree (nth 2 conf))
        (funcall set-winbuf (selected-window) (nth 2 conf)))
      (select-window newwin)
      (funcall set-winbuf newwin (nth 3 conf)))))

(defun my-restore-window-configuration (winconf)
  (let ((buf (current-buffer)))
    (delete-other-windows)
    (my-list-to-window-tree winconf
                            (lambda (win name)
                              (set-window-buffer win (or (get-buffer name)
                                                         buf))))))
```

第十四章　操作对象之三——文件

作为一个编辑器，自然文件是最重要的操作对象之一。这一节要介绍有关文件的一系列命令，比如查找文件，读写文件，文件信息、读取目录、文件名操作等。

14.1　打开文件的过程

当你打开一个文件时，实际上 emacs 做了很多事情：

- 把文件名展开成为完整的文件名
- 判断文件是否存在
- 判断文件是否可读或者文件大小是否太大
- 查看文件是否已经打开，是否被锁定
- 向缓冲区插入文件内容
- 设置缓冲区的模式

这还只是简单的一个步骤，实际情况比这要复杂的多，许多异常需要考虑。而且为了所有函数的可扩展性，许多变量、handler 和 hook 加入到文件操作的函数中，使得每一个环节都可以让用户或者 elisp 开发者可以定制，甚至完全接管所有的文件操作。

这里需要区分两个概念：文件和缓冲区。它们是两个不同的对象，文件是在计算机上可持久保存的信息，而缓冲区是 Emacs 中包含文件内容信息的对象，在 emacs 退出后就会消失，只有当保存缓冲区之后缓冲区里的内容才写到文件中去。

14.2　文件读写

打开一个文件的命令是 find-file。这命令使一个缓冲区访问某个文件，并让这个缓冲区成为当前缓冲区。在打开文件过程中会调用 find-file-hook。 find-file-noselect 是所有访问文件的核心函数。与 find-file 不同，它只返回访问文件的缓冲区。这两个函数都有一个特点，如果 emacs 里已经有一个缓冲区访问这个文件的话，emacs 不会创建另一个缓冲区来访问文件，而只是简单返回或者转到这个缓冲区。怎样检查有没有缓冲区是否访问某个文件呢？所有和文件关联的缓冲区里都有一个 buffer-local 变量 buffer-file-name。但是不要直接设置这个变量来改变缓冲区关联的文件。而是使用 set-visited-file-name 来修改。同样不要直接从 buffer-list 里搜索 buffer-file-name 来查找和某个文件关联的缓冲区。应该使用 get-file-buffer 或者 find-buffer-visiting。

```
(find-file "~/temp/test.txt")
(with-current-buffer
    (find-file-noselect "~/temp/test.txt")
  buffer-file-name)                    ; => "/home/ywb/temp/test.txt"
(find-buffer-visiting "~/temp/test.txt") ; => #<buffer test.txt>
```

```
(get-file-buffer "~/temp/test.txt")      ; => #<buffer test.txt>
```

保存一个文件的过程相对简单一些。首先创建备份文件，处理文件的位模式，将缓冲区写入文件。保存文件的命令是 save-buffer。相当于其它编辑器里另存为的命令是 write-file。在这个过程中会调用一些函数或者 hook。 write-file-functions 和 write-contents-functions 几乎功能完全相同。它们都是在写入文件之前运行的函数，如果这些函数中有一个返回了 non-nil 的值，则会认为文件已经写入了，后面的函数都不会运行，而且也不会使用再调用其它写入文件的函数。这两个变量有一个重要的区别是 write-contents-functions 在改变主模式之后会被修改，因为它没有 permanent-local 属性，而 write-file-functions 则会仍然保留。before-save-hook 和 write-file-functions 功能也比较类似，但是这个变量里的函数会逐个执行，不论返回什么值也不会影响后面文件的写入。after-save-hook 是在文件已经写入之后才调用的 hook，它是 save-buffer 最后一个动作。

但是实际上在 elisp 编程过程中经常遇到的一个问题是读取一个文件中的内容，读取完之后并不希望这个缓冲区还留下来，如果直接用 kill-buffer 可能会把用户打开的文件关闭。而且 find-file-noselect 做的事情实在超出我们的需要的。这时你可能需要的是更底层的文件读写函数，它们是 insert-file-contents 和 write-region，调用形式分别是：

```
(insert-file-contents filename &optional visit beg end replace)
(write-region start end filename &optional append visit lockname mustbenew)
```

insert-file-contents 可以插入文件中指定部分到当前缓冲区中。如果指定 visit 则会标记缓冲区的修改状态并关联缓冲区到文件，一般是不用的。replace 是指是否要删除缓冲区里其它内容，这比先删除缓冲区其它内容后插入文件内容要快一些，但是一般也用不上。insert-file-contents 会处理文件的编码，如果不需要解码文件的话，可以用 insert-file-contents-literally。

write-region 可以把缓冲区中的一部分写入到指定文件中。如果指定 append 则是添加到文件末尾。和 insert-file-contents 相似，visit 参数也会把缓冲区和文件关联，lockname 则是文件锁定的名字，mustbenew 确保文件存在时会要求用户确认操作。

> 思考题
> 写一个函数提取出某个 c 头文件中的函数声明中的函数名和声明位置。

14.3 文件信息

文件是否存在可以使用 file-exists-p 来判断。对于目录和一般文件都可以用这个函数进行判断，但是符号链接只有当目标文件存在时才返回 t。

如何判断文件是否可读或者可写呢？file-readable-p、file-writable-p， file-executable-p 分用来测试用户对文件的权限。文件的位模式还可以用 file-modes 函数得到。

```
(file-exists-p "~/temp/test.txt")              ; => t
(file-readable-p "~/temp/test.txt")            ; => t
(file-writable-p "~/temp/test.txt")            ; => t
(file-executable-p "~/temp/test.txt")          ; => nil
(format "%o" (file-modes "~/temp/test.txt"))   ; => "644"
```

文件类型判断可以使用 file-regular-p、file-directory-p、file-symlink-p，分别判断一个文件名是否是一个普通文件（不是目录，命名管道、终端或者其它 IO 设备）、文件名是否一个存在的目录、文件名是否是一个符号链接。其中 file-symlink-p 当文件名是一个符号链接时会返回目标文件名。文件的真实名字也就是除去相对链接和符号链接后得到的文件名可以用 file-truename 得到。事实上每个和文件关联的 buffer 里也有一个缓冲区局部变量 buffer-file-truename 来记录这个文件名。

```
$ ls -l t.txt
lrwxrwxrwx 1 ywb ywb 8 2007-07-15 15:51 t.txt -> test.txt
```

```
(file-regular-p "~/temp/t.txt")          ; => t
(file-directory-p "~/temp/t.txt")        ; => nil
(file-symlink-p "~/temp/t.txt")          ; => "test.txt"
(file-truename "~/temp/t.txt")           ; => "/home/ywb/temp/test.txt"
```

文件更详细的信息可以用 file-attributes 函数得到。这个函数类似系统的 stat 命令，返回文件几乎所有的信息，包括文件类型，用户和组用户，访问日期、修改日期、status change 日期、文件大小、文件位模式、inode number、 system number。这是我写的方便使用的帮助函数：

```
(defun file-stat-type (file &optional id-format)
  (car (file-attributes file id-format)))
(defun file-stat-name-number (file &optional id-format)
  (cadr (file-attributes file id-format)))
(defun file-stat-uid (file &optional id-format)
  (nth 2 (file-attributes file id-format)))
(defun file-stat-gid (file &optional id-format)
  (nth 3 (file-attributes file id-format)))
(defun file-stat-atime (file &optional id-format)
  (nth 4 (file-attributes file id-format)))
(defun file-stat-mtime (file &optional id-format)
  (nth 5 (file-attributes file id-format)))
(defun file-stat-ctime (file &optional id-format)
  (nth 6 (file-attributes file id-format)))
(defun file-stat-size (file &optional id-format)
  (nth 7 (file-attributes file id-format)))
(defun file-stat-modes (file &optional id-format)
  (nth 8 (file-attributes file id-format)))
(defun file-stat-guid-changep (file &optional id-format)
  (nth 9 (file-attributes file id-format)))
(defun file-stat-inode-number (file &optional id-format)
  (nth 10 (file-attributes file id-format)))
(defun file-stat-system-number (file &optional id-format)
  (nth 11 (file-attributes file id-format)))
(defun file-attr-type (attr)
```

```
  (car attr))
(defun file-attr-name-number (attr)
  (cadr attr))
(defun file-attr-uid (attr)
  (nth 2 attr))
(defun file-attr-gid (attr)
  (nth 3 attr))
(defun file-attr-atime (attr)
  (nth 4 attr))
(defun file-attr-mtime (attr)
  (nth 5 attr))
(defun file-attr-ctime (attr)
  (nth 6 attr))
(defun file-attr-size (attr)
  (nth 7 attr))
(defun file-attr-modes (attr)
  (nth 8 attr))
(defun file-attr-guid-changep (attr)
  (nth 9 attr))
(defun file-attr-inode-number (attr)
  (nth 10 attr))
(defun file-attr-system-number (attr)
  (nth 11 attr))
```

前一组函数是直接由文件名访问文件信息，而后一组函数是由 file-attributes 的返回值来得到文件信息。

14.4 修改文件信息

重命名和复制文件可以用 rename-file 和 copy-file。删除文件使用 delete-file。创建目录使用 make-directory 函数。不能用 delete-file 删除目录，只能用 delete-directory 删除目录。当目录不为空时会产生一个错误。

设置文件修改时间使用 set-file-times。设置文件位模式可以用 set-file-modes 函数。set-file-modes 函数的参数必须是一个整数。你可以用位函数 logand、logior 和 logxor 函数来进行位操作。

> 思考题
> 写一个函数模拟 chmod 命令的行为。

14.5 文件名操作

虽然 MSWin 的文件名使用的路径分隔符不同，但是这里介绍的函数都能用于 MSWin 形式的文件名，只是返回的文件名都是 Unix 形式了。路径一般由目录和文件名，而文件名一般由

主文件名(basename)、文件名后缀和版本号构成。Emacs 有一系列函数来得到路径中的不同部分：

```
(file-name-directory "~/temp/test.txt")      ; => "~/temp/"
(file-name-nondirectory "~/temp/test.txt")   ; => "test.txt"
(file-name-sans-extension "~/temp/test.txt") ; => "~/temp/test"
(file-name-extension "~/temp/test.txt")      ; => "txt"
(file-name-sans-versions "~/temp/test.txt~") ; => "~/temp/test.txt"
(file-name-sans-versions "~/temp/test.txt.~1~") ; => "~/temp/test.txt"
```

路径如果是从根目录开始的称为是绝对路径。测试一个路径是否是绝对路径使用 file-name-absolute-p。如果在 Unix 或 GNU/Linux 系统，以~ 开头的路径也是绝对路径。在 MSWin 上，以 “/”、\\、“X:” 开头的路径都是绝对路径。如果不是绝对路径，可以使用 expand-file-name 来得到绝对路径。把一个绝对路径转换成相对某个路径的相对路径的可以用 file-relative-name 函数。

```
(file-name-absolute-p "~rms/foo")       ; => t
(file-name-absolute-p "/user/rms/foo")  ; => t
(expand-file-name "foo")                ; => "/home/ywb/foo"
(expand-file-name "foo" "/usr/spool/")  ; => "/usr/spool/foo"
(file-relative-name "/foo/bar" "/foo/") ; => "bar"
(file-relative-name "/foo/bar" "/hack/") ; => "../foo/bar"
```

对于目录，如果要将其作为目录，也就是确保它是以路径分隔符结束，可以用 file-name-as-directory。不要用(concat dir ”/”) 来转换，这会有移植问题。和它相对应的函数是 directory-file-name。

```
(file-name-as-directory "~rms/lewis")   ; => "~rms/lewis/"
(directory-file-name "~lewis/")         ; => "~lewis"
```

如果要得到所在系统使用的文件名，可以用 convert-standard-filename。比如在 MSWin 系统上，可以用这个函数返回用\\ 分隔的文件名。

```
(convert-standard-filename "c:/windows")  ;=> "c:\\windows"
```

14.6 临时文件

如果需要产生一个临时文件，可以使用 make-temp-file。这个函数按给定前缀产生一个不和现有文件冲突的文件，并返回它的文件名。如果给定的名字是一个相对文件名，则产生的文件名会用 temporary-file-directory 进行扩展。也可以用这个函数产生一个临时文件夹。如果只想产生一个不存在的文件名，可以用 make-temp-name 函数。

```
(make-temp-file "foo")                  ; => "/tmp/foo5611dxf"
(make-temp-name "foo")                  ; => "foo5611q7l"
```

14.7 读取目录内容

可以用 directory-files 来得到某个目录中的全部或者符合某个正则表达式的文件名。

```
(directory-files "~/temp/dir/")
;; =>
;; ("#foo.el#" "." ".#foo.el" ".." "foo.el" "t.pl" "t2.pl")
(directory-files "~/temp/dir/" t)
;; =>
;; ("/home/ywb/temp/dir/#foo.el#"
;;  "/home/ywb/temp/dir/."
;;  "/home/ywb/temp/dir/.#foo.el"
;;  "/home/ywb/temp/dir/.."
;;  "/home/ywb/temp/dir/foo.el"
;;  "/home/ywb/temp/dir/t.pl"
;;  "/home/ywb/temp/dir/t2.pl")
(directory-files "~/temp/dir/" nil "\\.pl$") ; => ("t.pl" "t2.pl")
```

directory-files-and-attributes 和 directory-files 相似，但是返回的列表中包含了 file-attributes 得到的信息。file-name-all-versions 用于得到某个文件在目录中的所有版本，file-expand-wildcards 可以用通配符来得到目录中的文件列表。

> 思考题
> 写一个函数返回当前目录包括子目录中所有文件名。

14.8 神奇的 Handle

如果不把文件局限在存储在本地机器上的信息，如果有一套基本的文件操作，比如判断文件是否存在、打开文件、保存文件、得到目录内容之类，那远程的文件和本地文件的差别也仅在于文件名表示方法不同而已。在 Emacs 里，底层的文件操作函数都可以托管给 elisp 中的函数，这样只要用 elisp 实现了某种类型文件的基本操作，就能像编辑本地文件一样编辑其它类型文件了。

决定何种类型的文件名使用什么方式来操作是在 file-name-handler-alist 变量定义的。它是由形如=(REGEXP . HANDLER)= 的列表。如果文件名匹配这个 REGEXP 则使用 HANDLER 来进行相应的文件操作。这里所说的文件操作，具体的来说有这些函数：

```
‘access-file’, ‘add-name-to-file’, ‘byte-compiler-base-file-name’,
‘copy-file’, ‘delete-directory’, ‘delete-file’,
‘diff-latest-backup-file’, ‘directory-file-name’, ‘directory-files’,
‘directory-files-and-attributes’, ‘dired-call-process’,
‘dired-compress-file’, ‘dired-uncache’,
‘expand-file-name’, ‘file-accessible-directory-p’, ‘file-attributes’,
‘file-directory-p’, ‘file-executable-p’, ‘file-exists-p’,
‘file-local-copy’, ‘file-remote-p’, ‘file-modes’,
```

‘file-name-all-completions’, ‘file-name-as-directory’,
‘file-name-completion’, ‘file-name-directory’, ‘file-name-nondirectory’,
‘file-name-sans-versions’, ‘file-newer-than-file-p’,
‘file-ownership-preserved-p’, ‘file-readable-p’, ‘file-regular-p’,
‘file-symlink-p’, ‘file-truename’, ‘file-writable-p’,
‘find-backup-file-name’, ‘find-file-noselect’,
‘get-file-buffer’, ‘insert-directory’, ‘insert-file-contents’,
‘load’, ‘make-auto-save-file-name’, ‘make-directory’,
‘make-directory-internal’, ‘make-symbolic-link’,
‘rename-file’, ‘set-file-modes’, ‘set-file-times’,
‘set-visited-file-modtime’, ‘shell-command’, ‘substitute-in-file-name’,
‘unhandled-file-name-directory’, ‘vc-registered’,
‘verify-visited-file-modtime’,
‘write-region’.

在 HANDLE 里，可以只接管部分的文件操作，其它仍交给 emacs 原来的函数来完成。举一个简单的例子。比如最新版本的 emacs 把*scratch* 的 auto-save-mode 打开了。如果你不想这个缓冲区的自动保存的文件名散布得到处都是，可以想办法让这个缓冲区的自动保存文件放到指定的目录中。刚好 make-auto-save-file-name 是在上面这个列表里的，但是不幸的是在函数定义里 make-auto-save-file-name 里不对不关联文件的缓冲区使用 handler（我觉得是一个 bug 呀），继续往下看，发现生成保存文件名是使用了 expand-file-name 函数，所以解决办法就产生了：

```
(defun my-scratch-auto-save-file-name (operation &rest args)
  (if (and (eq operation 'expand-file-name)
           (string= (car args) "#*scratch*#"))
      (expand-file-name (concat "~/.emacs.d/backup/" (car args)))
    (let ((inhibit-file-name-handlers
           (cons 'my-scratch-auto-save-file-name
                 (and (eq inhibit-file-name-operation operation)
                      inhibit-file-name-handlers)))
          (inhibit-file-name-operation operation))
      (apply operation args))))
```

14.9 函数列表

```
(find-file FILENAME &optional WILDCARDS)
(find-file-noselect FILENAME &optional NOWARN RAWFILE WILDCARDS)
(set-visited-file-name FILENAME &optional NO-QUERY ALONG-WITH-FILE)
(get-file-buffer FILENAME)
(find-buffer-visiting FILENAME &optional PREDICATE)
(save-buffer &optional ARGS)
(insert-file-contents FILENAME &optional VISIT BEG END REPLACE)
```

```
(insert-file-contents-literally FILENAME &optional VISIT BEG END REPLACE)
(write-region START END FILENAME &optional APPEND VISIT LOCKNAME MUSTBENEW)
(file-exists-p FILENAME)
(file-readable-p FILENAME)
(file-writable-p FILENAME)
(file-executable-p FILENAME)
(file-modes FILENAME)
(file-regular-p FILENAME)
(file-directory-p FILENAME)
(file-symlink-p FILENAME)
(file-truename FILENAME)
(file-attributes FILENAME &optional ID-FORMAT)
(rename-file FILE NEWNAME &optional OK-IF-ALREADY-EXISTS)
(copy-file FILE NEWNAME &optional OK-IF-ALREADY-EXISTS KEEP-TIME PRESERVE-UID-GID)
(delete-file FILENAME)
(make-directory DIR &optional PARENTS)
(delete-directory DIRECTORY)
(set-file-modes FILENAME MODE)
(file-name-directory FILENAME)
(file-name-nondirectory FILENAME)
(file-name-sans-extension FILENAME)
(file-name-sans-versions NAME &optional KEEP-BACKUP-VERSION)
(file-name-absolute-p FILENAME)
(expand-file-name NAME &optional DEFAULT-DIRECTORY)
(file-relative-name FILENAME &optional DIRECTORY)
(file-name-as-directory FILE)
(directory-file-name DIRECTORY)
(convert-standard-filename FILENAME)
(make-temp-file PREFIX &optional DIR-FLAG SUFFIX)
(make-temp-name PREFIX)
(directory-files DIRECTORY &optional FULL MATCH NOSORT)
(dired-files-attributes DIR)
```

14.10 问题解答

14.10.1 提取头文件中函数名

这是我写的一个版本，主要是函数声明的正则表达式不好写，函数是很简单的。从这个例子也可以看出它错误的把那个 typedef void 当成函数声明了。如果你知道更好的正则表达式，请告诉我一下。

```
(defvar header-regexp-list
  '(("^\\(?:\\(?:G_CONST_RETURN\\|extern\\|const\\)\\s-+\\)?[a-zA-Z][_a-zA-Z0-9]*\
```

```
\\(?:\\s-*[*]*[ \t\n]+\\|\\s-+[*]*\\)\\([a-zA-Z][_a-zA-Z0-9]*\\)\\s-*(" . 1)
    ("^\\s-*#\\s-*define\\s-+\\([a-zA-Z][_a-zA-Z0-9]*\\)" . 1)))
(defun parse-c-header (file)
  "Extract function name and declaration position using
‘header-regexp-list’."
  (interactive "fHeader file: \nP")
  (let (info)
    (with-temp-buffer
      (insert-file-contents file)
      (dolist (re header-regexp-list)
        (goto-char (point-min))
        (while (re-search-forward (car re) nil t)
          (push (cons (match-string (cdr re)) (line-beginning-position)) info))))
    info))
(parse-c-header "/usr/include/glib-2.0/gmodule.h")
;; =>
;; (("g_module_name" . 1788)
;;  ("g_module_open" . 1747)
;;  ("G_MODULE_EXPORT" . 1396)
;;  ("G_MODULE_EXPORT" . 1317)
;;  ("G_MODULE_IMPORT" . 1261)
;;  ("g_module_build_path" . 3462)
;;  ("g_module_name" . 2764)
;;  ("g_module_symbol" . 2570)
;;  ("g_module_error" . 2445)
;;  ("g_module_make_resident" . 2329)
;;  ("g_module_close" . 2190)
;;  ("g_module_open" . 2021)
;;  ("g_module_supported" . 1894)
;;  ("void" . 1673))
```

14.10.2 模拟 chmod 的函数

这是一个改变单个文件模式的 chmod 版本。递归版本的就自己作一个练习吧。最好不要直接调用这个函数，因为每次调用都要解析一次 mode 参数，想一个只解析一次的方法吧。

```
(defun chmod (mode file)
  "A elisp function to simulate command chmod.
Note that the command chmod can accept MODE match
‘[ugoa]*([-+=]([rwxXst]*|[ugo]))+’, but this version only can process
MODE match ‘[ugoa]*[-+=]([rwx]*|[ugo])’.
"
  (cond ((integerp mode)
         (if (> mode #o777)
```

```
          (error "Unknown mode option: %d" mode)))
       ((string-match "^[0-7]\\{3\\}$" mode)
        (setq mode (string-to-number mode 8)))
       ((string-match "^\\([ugoa]*\\)\\([-+=]\\)\\([rwx]*\\|[ugo]\\)$" mode)
        (let ((users (append (match-string 1 mode) nil))
              (mask-func (string-to-char (match-string 2 mode)))
              (bits (append (match-string 3 mode) nil))
              (oldmode (file-modes file))
              (user-list '((?a . #o777)
                           (?u . #o700)
                           (?g . #o070)
                           (?o . #o007)))
              mask)
          (when bits
            (setq bits (* (cond ((= (car bits) ?u)
                                 (lsh (logand oldmode #o700) -6))
                                ((= (car bits) ?g)
                                 (lsh (logand oldmode #o070) -3))
                                ((= (car bits) ?o)
                                 (logand oldmode #o007))
                                (t
                                 (+ (if (member ?r bits) 4 0)
                                    (if (member ?w bits) 2 0)
                                    (if (member ?x bits) 1 0))))
                          #o111))
            (if users
                (setq mask (apply 'logior
                                  (delq nil (mapcar
                                             (lambda (u)
                                               (assoc-default u user-list))
                                             users))))
              (setq mask #o777))
            (setq mode
                  (cond ((= mask-func ?\=)
                         (logior (logand bits mask)
                                 (logand oldmode (logxor mask #o777))))
                        ((= mask-func ?\+)
                         (logior oldmode (logand bits mask)))
                        (t
                         (logand oldmode
                                 (logxor (logand bits mask) #o777)))))))))
       (t (error "Unknow mode option: %S" mode)))
  (set-file-modes file mode))
```

14.10.3 列出目录中所有文件

为了让这个函数更类似 directory-files 函数，我把参数设置为和它一样的：

```
(defun my-directory-all-files (dir &optional full match nosort)
  (apply 'append
   (delq nil
    (mapcar
     (lambda (file)
       (if (and (not (string-match "^[.]+$" (file-name-nondirectory file)))
                (file-directory-p (expand-file-name file dir)))
           (if full
               (my-directory-all-files file full match nosort)
             (mapcar (lambda (f)
                       (concat (file-name-as-directory file) f))
                     (my-directory-all-files (expand-file-name file dir)
                                             full match nosort)))
         (if (string-match match file)
             (list file))))
     (directory-files dir full nil nosort)))))
```

后记

到现在为止，我计划写的 elisp 入门内容已经写完了。如果你都看完看懂这些内容，我想写一些简单的 elisp 应用应该是没有什么问题了。还有一些比较重要的内容没有涉及到，我在这列一下，如果你对此有兴趣，可以自己看 elisp manual 里相关章节：

- 按键映射(keymap) 和菜单
- Minibuffer 和补全
- 进程
- 调试
- 主模式(major mode) 和从属模式(minor mode)
- 定制声明(Customization)
- 修正函数(advising function)
- 非 ASCII 字符

其实看一遍 elisp manual 也是很好的选择。我在写这些文字时就是一边参考 elisp manual 一边写的。写的时候我一直有种不安的感觉，这近 3M 的文字被我压缩到这么一点点是不是太过份了。在 elisp manual 里一些很重要的说明经常被我一两句话就带过了，有时根本就没有提到，这会不会让刚学 elisp 的人误入歧途呢？每每想到这一点，我就想就此停住。但是半途而废总是不好的。所以我还是决定写完应该写的就好了。其它的再说吧。

如果你是一个新手，我很想知道你看完这个入门教程的感受。当然如果实在没有兴趣看，也可以告诉我究竟哪里写的不好。我希望在这份文档上花的时间和精力没有白费。