Common Lisp 学习

作者：kenshinl

时间：Last Update: September 7, 2022

版本：0.1

封面图片作者：Elina Bernpaintner

Learn Common Lisp

目录

第 1 章 Common Lisp 语法

1.1 语法

```
;; 退出 sbcl 解释器
>:abort

;; 加载脚本
>: (load "test.lsp")
```

```
;; 定义全局变量
* (defparameter *glob* 99)

;; 定义全局常量
* (defconstant limit (+ *glob* 1))

;; 检查一个符号是否是全局变量或者全局常量
* (boundp '*glob*)
T

;; setf 给全局变量或者局部变量赋值
;; 如果 setf 的第一个实参是符号，且符号不是某个局部变量的名字，则 setf 把这个符号设为全局变量
* (setf *glob* 98)
98

;; 传入 setf 的第一个实参，还可以是表达式或变量名。在这种情况下，
;; 第二个实参的值被插入至第一个实参所引用的位置
* (setf x '(a b c))
(A B C)
* (setf (car x) 'n)
N
* x
(N B C)

;; 判断一个变量是不是列表
* (defun my-listp (x)
    (or (null x) (consp x)))

;; 判断一个变量是不是原子
* (defun my-atom (x)
    (or (null x) (not (consp x))))


;; nil 既是一个原子，又是一个列表
* (atom nil)
T
* (listp nil)
T

;; eql 比较两个对象的地址，equal 比较两个对象的值
* (eql (cons 'a nil) (cons 'a nil))
NIL
* (equal (cons 'a nil) (cons 'a nil))
T

;; 获取列表的第 n 个元素
```

```
* (nth 0 '(a b c))
A
;; 获取列表的第 n 个 cdr
* (nthcdr 2 '(a b c d))
(C D)

;; 返回列表的最后一个 cons 对象
* (last '(a b c d))
(D)
;; 获取列表的最后一个元素
* (car (last '(a b c d)))
D

;; mapcar 接受一个函数以及一个或多个列表，并返回把函数应用
;; 至每个列表的元素的结果，直到有的列表没有元素为止
* (mapcar #'(lambda (x) (+ x 10))
     '(1 2 3))
(11 12 13)

* (mapcar #'list
    '(a b c)
    '(1 2 3 4))
((A 1) (B 2) (C 3))


;; maplist 接受同样的参数，将列表的渐进的下一个 cdr 传入函数
* (maplist #'(lambda (x) x)
    '(a b c))
((A B C) (B C) (C))

;; member 返回从元素出现的位置开始的剩余列表元素，如果列表中不存在这个元素返回 nil
* (member 'b '(a b c))
(B C)

* (member 'd '(a b c))
NIL

;; 关键字参数
;; Common Lisp 函数接受一个或多个关键字参数。这些关键字参数不同的地方是，
他们不是把对应的参数放在特定的位置作匹配，而是在函数调用中用特殊标签，
称为关键字，来作匹配。一个关键字是一个前面有冒号的符号。关键字参数不关心顺序。

;; member 有一个 :test 关键字参数给使用者自定义比较函数
* (member '(a) '((a) (b)) :test #'equal)
((A) (B))

;; 函数 adjoin 像是条件式的 cons 。它接受一个对象及一个列表，
;; 如果对象还不是列表的成员，才构造对象至列表上。
* (adjoin 'b '(a b c))
(A B C)
* (adjoin 'd '(a b c))
(D A B C)

;; 并集、交集、补集
* (union '(a b c) '(b c d))
(A B C D)
* (intersection '(a b c) '(a c e))
(C A)
* (set-difference '(a b c d e) '(e d c))
(B A)

;; length 获取列表的长度
* (length '(a b c))
```

```
3

;; subseq 复制列表的一个子列表
* (subseq '(a b c d e) 1 3)
(B C)
* (subseq '(a b c d e) 1)
(B C D E)

;; reverse 返回一个反转过的列表
* (reverse '(a b c))
(C B A)

;; sort 接受一个序列及一个比较两个参数的函数，返回一个有同样元素的序列，
;; 根据比较函数来排序， sort 会修改传入的序列。
* (sort '(3 2 1 6 5 4) #'>)
(6 5 4 3 2 1)

;; 函数 every 和 some 接受一个比较函数及一个或多个序列。
;; 当我们仅输入一个序列时，它们测试序列元素是否满足判断式
* (every #'oddp '(1 2 3))
NIL
* (some #'oddp '(1 2 3))
T

;; 如果它们输入多于一个序列时，比较函数必须接受与序列一样多的元素作为参数，
;; every/some 从所有序列中提取一个元素作为比较函数的参数
* (every #'> '(4 5 6) '(1 2 3))
T

;; push/pop 在列表头部插入/删除元素
* (defparameter *y* '(1 2 3))
*Y*
* *y*
(1 2 3)
* (push 4 *y*)
(4 1 2 3)
* (pop *y*)
4
* *y*
(1 2 3)

;; pushnew 宏是 push 的变种，使用了 adjoin 而不是 cons
;; pushnew 只会往列表中插入不存在的元素
* (defparameter *y* '(1 2 3))
*Y*
* (pushnew 4 *y*)
(4 1 2 3)
* (pushnew 2 *y*)
(4 1 2 3)

;; 一个非正规列表的 Cons 对象称之为点状列表 (dotted list)
;; 在点状表示法，每个 Cons 对象的 car 与 cdr 由一个句点隔开来表示。
* (cons 'a 'b)
(A . B)

;; 用 Cons 对象来表示映射 (mapping)也是很自然的。一个由 Cons 对象组成
;; 的列表称之为关联列表(assoc-listor alist)。
* (defparameter *trans* '((+ . "add") (- . "sub")))
*TRANS*
* *trans*
((+ . "add") (- . "sub"))

;; assoc 用来取出在关联列表中，与给定的键值有关联的 Cons 对
```

```
;; 如果 assoc 没有找到对应的键时返回 nil
* (assoc '+ *trans*)
(+ . "add")


;; 可以调用 make-array 来构造一个数组，第一个实参为一个指定数组维度的列表。
;; 关键字参数 :initial-element 用来指定默认值
* (defparameter arr (make-array '(2 3) :initial-element nil))
ARR
* arr
#2A((NIL NIL NIL) (NIL NIL NIL))

;; 用 aref 取出数组内的元素
(aref arr 0 0)
NIL

;; 用 setf 和 aref 来设置数组的某个元素
(setf (aref arr 0 0) 'b)
B
arr
#2A((B NIL NIL) (NIL NIL NIL))

;; 我们只想要一维的数组，可以给 make-array 第一个实参传一个整数，而不是一个列表
* (defparameter vec (make-array '3 :initial-element 10))
VEC
* vec
#(10 10 10)

;; 一维数组又称为向量（vector）。可以通过 vector 函数来创建向量，向量的元素可以是任何类型
* (vector "a" 'b 10)
#("a" B 10)

;; 可以用 aref 来存取向量，但有一个更快的函数叫做 svref ，专门用来存取向量
* (svref vec 1)
10

;; 字符串是字符组成的向量。我们用一系列由双引号包住的字符，来表示一个字符串常量，而字符 c 用 #\c 表示。
;; 每个字符都有一个相关的整数   通常是 ASCII 码，但不一定是。在多数的 Lisp 实现里，函数 char-code
;; 返回与字符相关的数字，而 code-char 返回与数字相关的字符。
;; 字符比较函数 char<, char<=, char=, char>=, char>, 以及 char/=。
* (sort "hello" #'char<)
"ehllo"

;; 由于字符串是字符向量，序列与数组的函数都可以用在字符串。可以用 aref 来取出元素。
(aref "abc" 1)
#\b
;; 也可以使用针对字符串设计的 char 函数来取出元素。
(char "abc" 1)
#\b

;; 使用比较两个字符串
(equal "abc" "abc")
T
;; 使用 string-equal 忽略大小写比较两个字符串
(string-equal "abc" "ABC")
T

;; 使用 concatenate 连接字符串
(concatenate 'string "a " "b c")
"a b c"

;; 可以使用通用的 elt 函数来取序列的元素
(elt '(1 2 3) 1)
```

```
2
(elt "abc" 1)
#\b

;; 定义结构体
* (defstruct point
    x
    y)
POINT

;; 上面定义了一个 point 结构，具有两个字段 x 与 y 。
;; 同时隐式地定义了 make-point, point-p, copy-point, point-x, point-y 函数。
;; 每一个 make-point 的调用，会返回一个新的 point 。可以通过给予对应的关键字参数，来指定单一字段的值。
* (setf p (make-point :x 1 :y 2))
#S(POINT :X 1 :Y 2)

;; 使用生成的 point-x 函数访问 x 字段
* (point-x p)
1
;; 使用 setf 设置 x 字段
* (setf (point-x p) 3)
3
* p
#S(POINT :X 3 :Y 2)


;; 创建哈希表
* (setf ht (make-hash-table))
#<HASH-TABLE :TEST EQL :COUNT 0 {1001DEA573}>

;; 要取出与给定键值有关的数值，我们调用 gethash 并传入一个键值与哈希表。
;; 预设情况下，如果没有与这个键值相关的数值， gethash 会返回 nil
;; 函数 gethash 返回两个数值。第一个值是与键值有关的数值，第二个值说明了
;; 哈希表是否含有任何用此键值来储存的数值。由于第二个值是 nil ，我们知道
;; 第一个 nil是缺省的返回值，而不是因为 nil 是与 a 有关的数值。
* (gethash 'a ht)
NIL
NIL

;; 要把数值与键值作关联，使用 gethash 搭配 setf
* (setf (gethash 'a ht) 'hello)
HELLO
* (gethash 'a ht)
HELLO
T

;; 要从集合中移除一个对象，你可以调用 remhash ，它从一个哈希表中移除一个词条
* (remhash 'a ht)
T
* (gethash 'a ht)
NIL
NIL


;; 哈希表有一个迭代函数 maphash ，它接受两个实参，接受两个参数的函数以及哈希表。
;; 该函数会被每个键值对调用，没有特定的顺序。maphash 总是返回 nil。
* (setf (gethash 'a ht) 1
       (gethash 'b ht) 2)
2
* (maphash #'(lambda (k v)
    (format t "~A = ~A~%" k v))
    ht)
A = 1
```

```
B = 2
NIL

;; 和任何牵涉到查询的结构一样，哈希表一定有某种比较键值的概念。
;; 预设是使用 eql ，但你可以提供一个额外的关键字参数 :test 来告诉哈希表
;; 要使用 eq，equal，还是 equalp。
* (setf ht (make-hash-table :test #'equal))
#<HASH-TABLE :TEST EQUAL :COUNT 0 {1001E9F823}>
```

section 控制流

```
;; Common Lisp 有三个构造区块（block）的基本操作符： progn，block，tagbody。
;; 在 progn 主体中的表达式会依序求值，并返回最后一个表达式的值。
* (progn
    (format t "a")
    (format t "b")
    ( + 11 12))
ab
23

;; block 像是带有名字及紧急出口的 progn 。第一个实参应为符号，
;; 表示区块的名字。在主体中的任何地方，可以停止求值，并通过使
;; 用 return-from 指定区块的名字，来立即返回数值。
* (block head
    (format t "a")
    (return-from head 'val)
    (format t "b"))
a
VAL

;; 使用 defun 定义的函数主体，都隐含在一个与函数同名的区块，所以可以使用 return-form
* (defun foo ()
    (return-from foo 27))


;; 另一个我们用来区分表达式的操作符是 let。它接受一个代码主体，但允许我们在主体内设置新变量。
* (let (( x 7)
        (y 2))
    (format t "Number")
    ( + x y))
Number
9

;; 概念上说，一个 let 表达式等同于函数调用。
;; 函数可以用名字来引用，也可以通过使用一个 lambda 表达式从字面上来引用。由于 lambda 表达式
;; 是函数的名字，我们可以像使用函数名那样，把 lambda 表达式作为函数调用的第一个实参
* ((lambda (x) (+ x 1)) 3)
4

;; 所以，上面的 let 表达式实际上等价于
* ((lambda (x y)
    (format t "Number")
    (+ x y))
  7
  2)
Number
9

;; 这个模型清楚的告诉我们，由 let 创造的变量的值，不能依赖其它由同一个 let 所创造的变量。
* (let ((x 2)
        (y (+ x 1)))
    (+ x y))
```

```
;; 在 (+ x 1) 中的 x 不是前一行设置的值，因为这个let 表达式其实等价于
* ((lambda (x y) (+ x y)) 2
                        (+ x 1))

;; 从这里可以清晰的看出 (+ x 1) 中的 x 其实是外部的变量，而非 let 表达式的内部变量

;; 如果想要新变量的值依赖同一个表达式所设立的另一个变量，可以使用 let*
* (let* ((x 1)
         (y (+ x 1)))
    (+ x y))
3

;; 一个 let* 功能上等同于一系列嵌套的 let 。这个例子等同于
* (let ((x 1))
    (let ((y (+ x 1)))
      (+ x y)))
3

;; let 与 let* 将变量初始值都设为 nil 。nil 为初始值的变量，可以不赋值。
* (let ( x y)
    (list x y))
(NIL NIL)

;; destructuring-bind 宏是通用化的 let 。其接受单一变量，
;; 一个模式(pattern，一个或多个变量所构成的树)，并将它们与某个实际的树所对应的部份做绑定。
* (destructuring-bind (w (x y) . z) '(a (b c) d e)
    (list w x y z))
(A B C (D E))

;;;;;;;;;;; 条件表达式

;; if 表达式，测试表达式求值返回真时，则对主体求值
* (if (oddp 9)
    (progn
      (format t "odd")))
odd
NIL

;; when 表达式与 if 表达式等价，测试表达式求值返回真时，则对主体求值
* (when (oddp 9)
    (format t "odd"))
odd
NIL

;; unless 表达式与 when 表达式相反，测试表达式求值返回假时，则对主体求值
* (unless (oddp 10)
    (format t "not odd"))
not odd
NIL

;; cond 表达式，允许多个条件判断，与每个条件相关的代码隐含在 progn 里
* (defun check-num (num)
    (cond ((< num 0) (format t "num < 0"))
    ((= num 0) (format t "num = 0"))
    (t (format t "num > 0"))))

* (check-num -1)
num < 0
NIL
* (check-num 0)
num = 0
NIL
* (check-num 1)
```

```
num > 0
NIL

;; case 表达式
* (defun check-num (num)
    (case num
    ((-1) (format t "num = -1"))
    ((0) (format t "num = 0"))
    (otherwise (format t "num unknown"))))

* (check-num -1)
num = -1
NIL
* (check-num 0)
num = 0
NIL
* (check-num 1)
num unknown
NIL

;; do 表达式，do 的第一个参数必须是说明变量规格的列表
;; (variable initial update)
;; initial 与 update 形式是选择性的。若 update 形式忽略时，
;; 每次迭代时不会更新变量。若 initial 形式也忽略时，变量会使用nil 来初始化。
* (defun show-squares (start end)
    (do ((i start (+ i 1)))
        ((> i end) 'done)
        (format t "~A ~A~%" i (* i i))))

* (show-squares 1 3)
1 1
2 4
3 9
DONE

;; 当do 表达式同时更新超过一个变量时，如果一个 update 形式，引用到一个拥有自己的
;; update 形式的变量时，它获得迭代之前的值。
* (let ((x 'a))
    (do ((x 1 (+ x 1)) ;; update x
         (y x x))    ;; update y
      ((> x 5))
      (format t "(~A ~A) " x y)))
(1 A) (2 1) (3 2) (4 3) (5 4)
NIL

;; 但如果使用 do* ，它有着和 let 与 let* 一样的关系。任何 initial 或 update 形式
;; 可以参照到前一个子句的变量，并会获得当下的值
* (do* ((x 1 (+ x 1))
        (y x x))
    ((> x 5))
    (format t "(~A ~A) " x y))
(1 1) (2 2) (3 3) (4 4) (5 5)
NIL

;; 使用 dolist 进行迭代。当迭代结束时，初始列表内的第三个表达式(此处是 done)
;; 会被求值并作为 dolist 的返回值。缺省是 nil
* (dolist (x '(a b c d) 'done)
    (format t "~A " x))
A B C D
DONE

;; 类似的迭代操作符还有 dotimes，给定某个 n，将会从整数 0，迭代至 n-1
* (dotimes (x 5 'done)
```

```
   (format t "~A " x))
0 1 2 3 4
DONE

;; mapc 可以用来遍历多个列表
* (mapc #'(lambda (x y)
               (format t "~A ~A  " x y))
   '(a1 a2 a3)
   '(b1 b2 b3))
A1 B1 A2 B2 A3 B3
(A1 A2 A3)

;; 在 Common Lisp 里，一个表达式可以返回零个或多个数值，
;; 最多可以返回几个值取决于各编译器实现，但至少可以返回 19 个值。

;; values 函数返回多个数值，它一个不少地返回你作为数值所传入的实参
* (values 'a nil 10 (+ 1 2 3 4))
A
NIL
10
10

;; 要接收多个数值，我们使用 multiple-value-bind，
;; 如果接收的值数量大于返回的值数量，剩余的变量会是 nil
* (multiple-value-bind (x y z) (values 1 2 3)
   (list x y z))
(1 2 3)
* (multiple-value-bind (x y z) (values 1 2)
   (list x y z))
(1 2 NIL)

;; 可以借由 multiple-value-call 将多值作为实参传给第二个函数
* (multiple-value-call #'+ (values 1 2 3))
6

;; multiple-value-list 看起来像是使用 #'list 作为第一个参数的来调用 multiple-value-call
* (multiple-value-list (values 'a 'b 'c))
(A B C)

;; 可以使用 return 在任何时候离开一个 block。但有时候想在数个函数调用里将控制权转移回来。
;; 要实现这个功能，可以使用 catch 与 throw 。一个 catch 表达式接受一个标签(tag)，
;; 标签可以是任何类型的对象。
* (defun super ()
   (catch 'abort
     (sub)
     (format t "catch abort")))

* (defun sub ()
   (throw 'abort 100))

* (super)
100
;; 可以看到，这里只输出了 100，跳过了 (format t "catch abort") 这条语句的执行。
;; 一个带有给定标签的 throw ，为了要到达匹配标签的 catch ，会将控制权转移
;; (因此杀掉进程)给任何有标签的 catch 。如果没有一个 catch 符合欲匹配的标签时，
;; throw 会产生一个错误。

;; 有时候想要防止代码被 throw 与 error 打断。可以使用 unwind-protect ，确保像是前述的中断，
;; 不会让程序停在不一致的状态。一个 unwind-protect 接受任何数量的实参，并返回第一个实参的值。
;; 然而即便是第一个实参的求值被打断时，剩下的表达式仍会被求值：
* (setf x 1)
1
* (catch 'abort
```

```
    (unwind-protect
      (throw 'abort 100)
      (setf x 2)))
100
* x
2

;;;;;;;;;;;;;;;;;;;;;;;; 函数

;;;;; 全局函数 ;;;;;
;; fboundp 可以判断是否有个函数的名字与给定的符号绑定。
;; 如果一个符号是函数的名字，symbol-function 可以返回它
* (fboundp '+)
#<FUNCTION +>
* (fboundp 'useless)
NIL
* (symbol-function '+)
#<FUNCTION +>

;; 可通过 symbol-function 给函数配置某个名字
* (setf (symbol-function 'add2)
    #'(lambda (x) (+ x 2)))
#<FUNCTION (LAMBDA (X)) {5344D14B}>

;; 如果字符串成为 defun 定义的函数主体的第一个表达式，
;; 那么这个字符串会变成函数的文档字符串(documentation string)。
;; 要取得函数的文档字符串，可以通过调用 documentation 来取得
* (defun foo(x)
    "foo doc."
    x)
FOO
* (documentation 'foo 'function)
"foo doc."

;;;;; 局部函数 ;;;;;
;; 局部函数可以使用 labels 来定义，它是一种像是给函数使用的 let。
;; 它的第一个实参是一个新局部函数的定义列表，而不是一个变量规格说明的列表。
;; 列表中的元素形式为 (name parameters body)
* (labels ((add10 (x) (+ x 10))
           (consa (x) (cons 'a x)))
    (consa (add10 3)))
(A . 13)

;; 由 labels 表达式所定义的局部函数，可以被其他任何在此定义的函数引用，包括自己。
;; 所以这样定义一个递归的局部函数是可能的
* (labels ((len (lst)
            (if (null lst)
              0
              (+ (len (cdr lst)) 1))))
    (len '(a b c)))
3

;;;;;;;;;;;;;;;;;;;;;;;; 参数列表

;; 有了前序表达式， + 可以接受任何数量的参数。从那时开始，我们看过许多接受不定数量参数的函数。
;; 要写出这样的函数，我们需要使用一个叫做剩余（ rest ）参数的东西。如果我们在函数的形参列表里
;; 的最后一个变量前，插入 &rest 符号，那么当这个函数被调用时，这个变量会被设成一个带有剩余参数的列表。
;; 有的参数可以被忽略，并可以缺省设成特定的值。这样的参数称为选择性参数(optional parameters)，
;; 如果符号 &optional 出现在一个函数的形参列表时，那么在 &optional 之后的参数都是选择性的，缺省为 nil
* (defun myopt (arg1 &optional arg2)
    (list arg1 arg2))
MYOPT
* (myopt 1)
```

```
(1 NIL)

;; 可以通过将缺省值附在列表里明确指定缺省值。选择性参数的缺省值可以不是常量。可以是任何的 Lisp 表达式。
;; 若这个表达式不是常量，它会在每次需要用到缺省值时被重新求值。
;;
* (defun myopt (arg1 &optional (arg2 100))
    (list arg1 arg2))
MYOPT
* (myopt 1)
(1 100)

;; 关键字参数(keyword parameter)是一种更灵活的选择性参数。如果把符号 &key 放在一个形参列表，
;; 那在 &key 之后的形参都是选择性的。此外，当函数被调用时，这些参数会被识别出来，参数的位置在
;; 哪不重要，和普通的选择性参数一样，关键字参数缺省值为 nil，但可以在形参列表中明确地指定缺省值
* (defun keylist (a &key x y z)
    (list a x y z))
KEYLIST
* (keylist 1 :y 2 :z 3)
(1 NIL 2 3)


;;;;;;;;;;;; 闭包
;; 函数可以如表达式的值，或是其它对象那样被返回。以下是接受一个实参，并依其类型返回特定的结合函数
* (defun combiner (x)
    (typecase x
      (number #'+)
      (list #'append)
      (t #'list)))
COMBINER
;; 通用结合函数
* (defun combine (&rest args)
    (apply (combiner (car args))
      args))
COMBINE
* (combine 1 2 3)
6
* (combine '(a b) '(c d))
(A B C D)

;; 当函数引用到外部定义的变量时，这外部定义的变量称为自由变量(free variable)。
;; 函数引用到自由的词法变量时，称之为闭包(closure)。只要函数还存在，变量就必须一起存在。
;; 闭包结合了函数与环境（environment）；无论何时，当一个函数引用到周围词法环境的某个东西时，
;; 闭包就被隐式地创建出来了。
* (setf fn (let ((i 3))
     #'(lambda (x) (+ x i))))
#<FUNCTION (LAMBDA (X)) {5344E4DB}>
* (funcall fn 2)
5

* (defun make-adder (n)
    #'(lambda (x)
      (+ x n)))
MAKE-ADDER
* (setf add3 (make-adder 3))
#<FUNCTION (LAMBDA (X) :IN MAKE-ADDER) {100223001B}>
* (funcall add3 2)
5

;; 可以产生共享变量的数个闭包。下面定义共享一个计数器的两个函数
;; reset-counter 函数和 add-counter 函数都可以访问 counter
* (let ((counter 0))
    (defun reset-counter ()
      (setf counter 0))
```

```
   (defun add-counter ()
     (setf counter (+ counter 1))))
ADD-COUNTER
* (list (add-counter) (add-counter) (reset-counter) (add-counter))
(1 2 0 1)

;; 动态作用域 (Dynamic Scope)
;; 动态作用域，我们在环境中函数被调用的地方寻找变量。要使一个变量是动态作用域的，
;; 我们需要在任何它出现的上下文中声明它是 special 。如果我们这样定义 foo
* (let ((x 10))
   (defun foo ()
     (declare (special x))
     x))
FOO
;; 则函数内的 x 就不再引用到函数定义里的那个词法变量，
;; 但会引用到函数被调用时，当下所存在的任何特别变量 x
;; 新的变量被创建出来之后， 一个 declare 调用可以在代码的任何地方出现。
;; special 声明是独一无二的，因为它可以改变程序的行为。
* (let ((x 20))
   (declare (special x))
   (foo))
20

;;;;;;;;; 编译 (Compilation)
;; Common Lisp 函数可以独立被编译或挨个文件编译。如果你只是在顶层输入一个 defun 表达式，
;; 许多实现会创建一个直译的函数(interpreted function)。你可以将函数传给
;; compiled-function-p 来检查一个函数是否有被编译，也可以调用 compile 来编译函数
* (defun foo (x) (+ x 1))
FOO
* (compiled-function-p #'foo)
T
* (compile 'foo)
FOO
NIL
NIL

;; 通常要编译 Lisp 代码不是挨个函数编译，而是使用 compile-file 编译整个文件。
;; 这个函数接受一个文件名，并创建一个原始码的编译版本  通常会有同样的名称，但不同的扩展名。
;; 当编译过的文件被载入时， compiled-function-p 应给所有定义在文件内的函数返回真。
;; 当一个函数包含在另一个函数内时，包含它的函数会被编译，而且内部的函数也会被编译。

;;;;;;;;;;;;;;;;;; IO ;;;;;;;;;;;;;;;;;;;;;;;
;; 读入换行符 (newline)之前的所有字符，并用字符串返回它们。它接受一个选择性流参数
;; (optional stream argument)；若流忽略时，缺省为 *standard-input*
* (progn
   (format t "enter name:~%")
   (read-line))
enter name:
hello
"hello"
NIL

;; 如果我们想要把输入解析为 Lisp 对象，使用 read 。这个函数恰好读取一个表达式，在表达式结束时停止读取。
;; 所以可以读取多于或少于一行。而当然它所读取的内容必须是合法的 Lisp 语法。
* (read)
* (a
* b
* c
* )
(A B C)

;; 三个最简单的输出函数是 prin1 , princ 以及 terpri 。这三个函数的最后一个参数皆为选择性的流参数，
;; 缺省是 *standard-output*。prin1 与 princ 的差别大致在于 prin1 给程序产生输出，
```

```
;; 而 princ 给人类产生输出。所以举例来说， prin1 会印出字符串左右的双引号，而 princ 不会。
;; 两者皆返回它们的第一个参数。
* (prin1 "Hello")
"Hello"
"Hello"
* (princ "Hello")
Hello
"Hello"

;;;;;;;;;;;;; 宏字符 (Macro Characters)
;; 一个宏字符或宏字符组合也称作 read-macro (读取宏) 。许多 Common Lisp 预定义的读取宏是缩写。
;; 比如说引用 (Quote): 读入一个像是 'a 的表达式时，它被读取器展开成 (quote a) 。当你输入引用
;; 的表达式 (quoted expression)至顶层时，它们在读入之时就会被求值，所以一般来说你看不到这样的
;; 转换。你可以透过显式调用 read 使其现形。
* (car (read-from-string "'a"))
QUOTE
* (read-from-string "'a")
'A
2

;; 引用对于读取宏来说是不寻常的，因为它用单一字符表示。有了一个有限的字符集，你可以在 Common Lisp
;; 里有许多单一字符的读取宏，来表示一个或更多字符。这样的读取宏叫做派发 (dispatching)读取宏，
;; 而第一个字符叫做派发字符 (dispatching character)。所有预定义的派发读取宏使用井号 ( # )作为派发
;; 字符。我们已经见过好几个。举例来说， #' 是 (function ...) 的缩写，同样的 ' 是 (quote ...) 的缩写。
;; 其它我们见过的派发读取宏包括 #(...) ，产生一个向量； #nA(...) 产生数组； #\ 产生一个字符；
;; #S(n ...) 产生一个结构。当这些类型的每个对象被 prin1 显示时 (或是 format 搭配 ~S)，它们使用对应
;; 的读取宏

;;;;;;;;;;;;;;;;;;;;;; 符号 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; 符号是变量的名字，符号本身以对象所存在。符号可以用任何字符串当作名字，可以通过调用 symbol-name 来获得符
    号的名字
* (symbol-name 'abc)
"ABC"

;; 注意到这个符号的名字，打印出来都是大写字母。缺省情况下， Common Lisp 在读入时，
;; 会把符号名字所有的英文字母都转成大写。代表 Common Lisp 缺省是不分大小写的
* (eql 'abc 'Abc)
T

;; 一个名字包含空白，或其它可能被读取器认为是重要的字符的符号，要用特殊的语法来引用。
;; 任何存在垂直杠 (vertical bar)之间的字符序列将被视为符号。可以如下这般在符号的名字中，放入任何字符
;; 当这种符号被读入时，不会有大小写转换，而宏字符与其他的字符被视为一般字符。
;; 垂直杠是一种表示符号的特殊语法。它们不是符号的名字之一。
* (list '|Lisp lang| "||")
(|Lisp lang| "||")

;; 属性列表 (Property Lists)
;; 在 Common Lisp 里，每个符号都有一个属性列表(property-list)或称为 plist 。
;; 函数 get 接受符号及任何类型的键值，然后返回在符号的属性列表中，与键值相关的数值
;; 它使用 eql 来比较各个键。若某个特定的属性没有找到时， get 返回 nil
* (get 'a 'attr)
NIL

;; 要将值与键关联起来时，你可以使用 setf 及 get
* (setf (get 'a 'color) 'red)
RED
* (get 'a 'color)
RED

;; 可以使用 symbol-plist 查看符号的属性列表
* (symbol-plist 'a)
(COLOR RED)
```

```
;;;;;;;;;;;; 数值 ;;;;;;;;;;;;;;;;;;;;
;; 函数 float 将任何实数转换成浮点数
* (/ 2 3)
2/3
* (float 2/3)
0.6666667

;; 函数 truncate 返回任何实数的整数部分，二个返回值是传入的参数减去第一个返回值。
* (truncate 3.14)
3
0.1400001

;; 函数 floor 与 ceiling 以及 round 也从它们的参数中导出整数
;; 函数 round 返回最接近其参数的整数。当参数与两个整数的距离相等时，
;; Common Lisp 和很多程序语言一样，不会往上取(round up)整数。而是取最近的偶数


;; 比值与复数概念上是两部分的结构。(译注：像 Cons 这样的两部分结构) 函数
;; numerator 与 denominator 返回比值或整数的分子与分母(如果数字是整数，
;; 前者返回该数，而后者返回 1)。函数 realpart 与 imgpart 返回任何数字的实数
;; 与虚数部分。(如果数字不是复数，前者返回该数字，后者返回 0)。

;; 谓词 = 比较其参数，当数值上相等时，返回真。= 比起 eql 来得宽松，但参数的类型需一致。
;; 宏 incf 及 decf 分别递增与递减数字。

;; expt 是指数函数，log 是对数函数
* (expt 2 5)
32
* (log 32 2)
5.0

;;;;;;;;;;;;;;;;;;; 宏 ;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 宏是通过转换 (transformation)而实现的操作符。你通过说明你一个调用应该要翻译成什么，来定义一个宏。
;; 这个翻译称为宏展开(macro-expansion)，宏展开由编译器自动完成。所以宏所产生的代码，会变成程序的一
;; 个部分，就像你自己输入的程序一样。宏通常通过调用 defmacro 来定义。
* (defmacro nil! (x)
    (list 'setf x nil))
NIL!
* (setf x 100)
100
* x
100
* (nil! x)
NIL
* x
NIL

;; 上面定义了一个新的操作符，称为 nil! ，它接受一个参数。
;; 形式如 (nil! a) 的调用，会在求值或编译前，被翻译成 (setf a nil)

;; 要测试一个函数，我们调用它，但要测试一个宏，我们看它的展开式 (expansion)。
;; 函数 macroexpand-1 接受一个宏调用，并产生它的展开式：
* (macroexpand-1 '(nil! x))
(SETF X NIL)
T

;;;;;;;;;;;; 反引号 (Backquote)
;; 反引号读取宏 (read-macro)使得从模版 (templates)建构列表变得有可能。反引号广泛使用在宏定义中。
;; 一个反引号单独使用时，等于普通的引号
* `(a b c)
(A B C)
```

```
;; 和普通引号一样，单一个反引号保护其参数被求值。反引号的优点是，在一个反引号表达式里，
;; 你可以使用 ,(逗号)与 ,@(comma-at)来重启求值。如果你在反引号表达式里，在某个东西前
;; 面加逗号，则它会被求值。所以我们可以使用反引号与逗号来建构列表模版
* (setf a 1 b 2)
2
* `(a is ,a and b is ,b)
(A IS 1 AND B IS 2)


;; 通过使用反引号取代调用 list ，我们可以写出会产生出展开式的宏。举例来说 nil! 可以定义为
* (defmacro nil! (x)
    `(setf ,x nil))
NIL!

;; ,@ 与逗号相似，但将（本来应该是列表的）参数解开。将列表的元素插入模版来取代列表
* (setf lst '(a b c))
(A B C)
* `(lst = ,lst)
(LST = (A B C))
* `(lst = ,@lst)
(LST = A B C)

;;;;;;; 设计宏 (Macro Design)



;;;;;;; CLOS(Common Lisp 对象系统)
(defclass rectangle ()
  (height width))

(defclass circle ()
  (radius))

(defmethod area ((x rectangle))
  (* (slot-value x 'height) (slot-value x 'width)))

(defmethod area ((x circle))
  (* pi (expt (slot-value x 'radius) 2)))

;; 在面向对象模型里，我们的程序被拆成数个独一无二的方法，每个方法为某些特定类型的参数而生。
;; 我们调用 area 时，Lisp 检查参数的类型，并调用相对应的方法。
* (let ((r (make-instance 'rectangle)))
    (setf (slot-value r 'height) 2
          (slot-value r 'width) 3)
    (area r))
6

;; 类与实例(Class and Instances)
;; 使用 defclass 定义一个类(Class)
* (defclass circle ()
    (radius center))
#<STANDARD-CLASS COMMON-LISP-USER::CIRCLE>

;; 这个定义说明了 circle 类别的实例会有两个槽(slot)，分别名为 radius 与 center
;; 要创建这个类的实例，我们调用通用的 make-instance 函数，传入的第一个参数为类别名称
* (setf c (make-instance 'circle))
#<CIRCLE {1001958113}>

;; 要给这个实例的槽赋值，我们可以使用 setf 搭配 slot-value
(setf (slot-value c 'radius) 1)
1
```

```
;; 传给 defclass 的第三个参数必须是一个槽定义的列表。如上例所示，最简单的槽定义是一个表示
;; 其名称的符号。在一般情况下，一个槽定义可以是一个列表，第一个是槽的名称，伴随著一个或多个
;; 属性 (property)。属性像关键字参数那样指定。通过替一个槽定义一个访问器 (accessor)，我们
;; 隐式地定义了一个可以引用到槽的函数，使我们不需要再调用 slot-value 函数。如下更新 circle
;; 类定义，能够分别通过 circle-radius 及 circle-center 来引用槽
* (defclass circle ()
    ((radius :accessor circle-radius)
    (center :accessor circle-center)))

* (setf c (make-instance 'circle))
#<CIRCLE {1001964333}>
* (setf (circle-radius c) 1)
1
* (circle-radius c)
1

;; 通过指定一个 :writer 或是一个 :reader ，而不是 :accessor ，我们可以获得访问器的写入或读取行为。
;; 要指定一个槽的缺省值，我们可以给入一个 :initform 参数。若我们想要在 make-instance 调用期间就将
;; 槽初始化，我们可以用:initarg 定义一个参数名。加入刚刚所说的两件事，现在我们的类定义变成
* (setf c (make-instance 'circle :radius 3))
#<CIRCLE {100197BE43}>
* (circle-radius c)
3
* (circle-center c)
(0 . 0)

;; 我们可以指定某些槽是共享的    也就是每个产生出来的实例，共享槽的值都会是一样的。我们通过声明槽
;; :allocation:class 属性来实现(另一个办法是让一个槽有 :allocation :instance ，但由于
;; 这是缺省设置，不需要特别再声明一次)。当我们在一个实例中，改变了共享槽的值，则其它实例共享槽也
;; 会获得相同的值。所以我们会想要使用共享槽来保存所有实例都有的相同属性。
* (defclass kid ()
    ((age :accessor kid-age
          :allocation :class)))


* (setf kid-1 (make-instance 'kid)
       kid-2 (make-instance 'kid))
#<KID {100194A713}>
* (setf (kid-age kid-1) 12)
12
* (kid-age kid-2)
12

;; 上面可以看到，修改了对象 kid-1 的 age 属性，kid-2 的 age 属性也发生了变化

;;;;;;;;;;;; 基类 (Superclasses)

;; defclass 接受的第二个参数是一个列出其基类的列表。一个类别继承了所有基类槽。


;; 一个通用函数 (generic function) 是由一个或多个方法组成的一个函数。
;; 方法可用 defmethod 来定义，与 defun 的定义形式类似。
* (defmethod combine (x y)
    (list x y))
#<STANDARD-METHOD COMMON-LISP-USER::COMBINE (T T) {1001A57F43}>

;; 通用方法可以针对类型进行进行特化
;; 通过使用 eql，方法甚至可以对单一的对象做特化
;; 单一对象特化的优先级比类别特化高
* (defmethod combine ((x (eql 'a)) (y (eql 'b)))
    (format t "x=a, y=b"))
#<STANDARD-METHOD COMMON-LISP-USER::COMBINE ((EQL A) (EQL B)) {1001AA1F43}>
```

```
;; 方法可以像一般 Common Lisp 函数一样有复杂的参数列表，但所有组成通用函数方法的参数列表必须
;; 是一致的 (congruent)。参数的数量必须一致，同样数量的选择性参数（如果有的话），要么一起使用
;; &rest 或是 &key 参数，或者一起不要用。下面的参数列表对是全部一致的，
(x)           (a)
(x &optional y) (a &optional b)
(x y &rest z) (a b &key c)
(x y &key z)  (a b &key c d)

;; 下列的参数列表对不是一致的
(x)           (a b)
(x &optional y) (a &optional b c)
(x &optional y) (a &rest b)
(x &key x y)  (a)

;; 只有必要参数可以被特化。所以每个方法都可以通过名字及必要参数的特化独一无二地识别出来。


;; 方法可以通过如 :before ， :after 以及 :around 等辅助方法来增强。
;; :before 与 :after 方法允许我们将新的行为包在调用主方法的周围。 :around 方法提供了一个
;; 更戏剧的方式来办到这件事。如果:around 方法存在的话，会调用的是 :around 方法而不是主方法。
;; 则根据它自己的判断， :around 方法自己可能会调用主方法（通过函数 call-next-method ，这也
;; 是这个函数存在的目的）。

;; 辅助方法通过在 defmethod 调用中，在方法名后加上一个修饰关键字 (qualifying keyword)来定义。
;; 如果我们替 speaker 类别定义一个主要的 speak 方法如下:
* (defclass speaker () ())
#<STANDARD-CLASS COMMON-LISP-USER::SPEAKER>

(defmethod speak ((s speaker) string)
  (format t "~A" string))
#<STANDARD-METHOD COMMON-LISP-USER::SPEAK (SPEAKER T) {1001981FD3}>

;; 调用一下 speaker 类的 speak 方法 会正常输出参数
(speak (make-instance 'speaker)
  "Hello")
Hello
NIL

;; 定义一个 intel 子类和 speak 的 :before, :after 辅助方法
(defclass intel (speaker) ())
#<STANDARD-CLASS COMMON-LISP-USER::INTEL>

(defmethod speak :before ((i intel) string)
  (princ "Before "))
#<STANDARD-METHOD COMMON-LISP-USER::SPEAK :BEFORE (INTEL T) {10019D5773}>

(defmethod speak :after ((i intel) string)
  (princ " After"))
#<STANDARD-METHOD COMMON-LISP-USER::SPEAK :AFTER (INTEL T) {1001A0F2F3}>

;; 调用 intel 类的 speak 方法，可以看到先执行 :before 方法，再执行 speak 方法，
;; 最后执行 :after 方法
(speak (make-instance 'intel)
  "hello")
Before hello After
NIL

;;;; 共享结构 (Shared Structure)
;; 多个列表可以共享 cons 。在最简单的情况下，一个列表可以是另一个列表的一部分。
;; 下面的 part 是第二个 cons 的 cdr
* (setf part (list 'b 'c))
(B C)
* (setf whole (cons 'a part))
```

```
(A B C)

;; 使用 tailp 判断式来检测一下。将两个列表作为它的输入参数，
;; 如果第一个列表是第二个列表的一部分时，则返回 T
* (tailp part whole)
T

;; 如果想避免共享结构，可以使用 copy-list 来复制。
;; 它返回一个不与原始列表共享顶层列表结构的新列表。
```