函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

http://blog.sina.com.cn/u/3266490431

第三讲

数据抽象

●程序中可能需要处理用多个数据描述的复杂对象，比如学生。关于学生的数据，有姓名，年龄，GPA等。

●应该用一种抽象的数据来描述和操作“学生”这个整体，而不应该分离地看待和操作学生的姓名，年龄，GPA等

●抽象数据和外部应该有一个接口，具体实现则对外部屏蔽（便于修改）。外部通过接口使用抽象数据。

●C++用“类”实现数据抽象

●Lisp怎么办？

引入“构造函数”和“选择函数”的思想来实现数据抽象

数据抽象实例：有理数的算术运算

- 有理数由分子和分母组成
- 构造函数：

  (make-rat <*n*> <*d*>) ***;构造以整数n为分子整数d 为分母的有理数***

- 选择函数：

  (numer <*x*>) ***;取得有理数x 的分子***

  (denom <*x*>) ***;取得有理数x 的分母***

- 构造函数和选择函数构成了“有理数”这种抽象数据和外部的接口

数据抽象实例：有理数的算术运算

- 有理数的计算规则：

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \qquad \frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2} \qquad \frac{n_1}{d_1} / \frac{n_2}{d_2} = \frac{n_1 d_2}{n_2 d_1}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \quad \text{iff} \quad n_1 d_2 = n_2 d_1$$

数据抽象实例：有理数的算术运算

有理数算数运算的实现：

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

数据抽象实例：有理数的算术运算

有理数算数运算的实现：

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

输出有理数：

```
(define (print-rat x)
  (newline) ;换行
  (display (numer x))
  (display "/")
  (display (denom x)))
```

数据抽象实例：有理数的算术运算

有理数算数运算的实现：

```
(define (make-rat n d)
          (cons n d))
;n,d是整数，代表分子分母

(define (numer x) (car x))
(define (denom x) (cdr x))

;x是有理数
```

输出有理数：

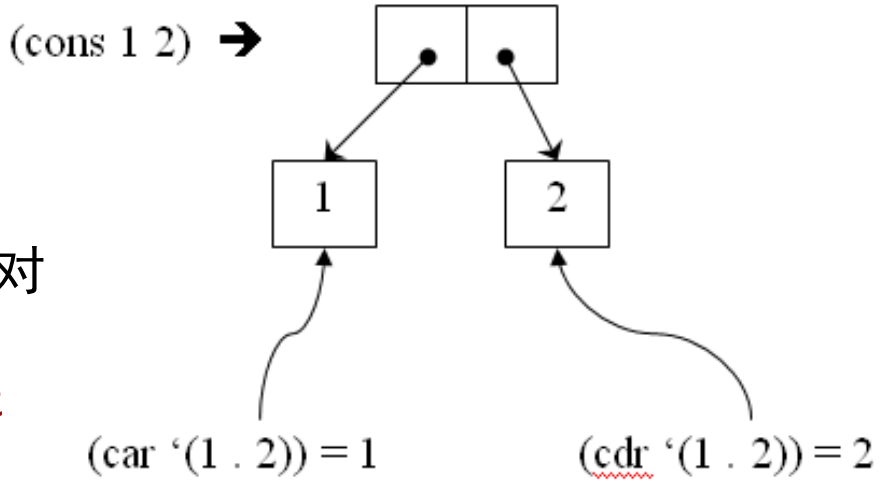
```
(define (print-rat x)
  (newline) ;换行
  (display (numer x))
  (display "/")
  (display (denom x)))
```

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

序对(pair)

- (cons a b) 即形成一个“序对”，一个序对由两部分(两个指针）构成

- car 取前部，cdr取后部



- 基本函数pair?判断是否是序对

```
(pair? (cons 3 4)) ;  => #t
(pair? 2) ;=> #f
```

序对(pair)

```
(define p (cons "good" 100))
p
(car p)
(cdr p)
```

输出：
'("good" . 100)
"good"
100

序对也是数据对象，可以用于构造更复杂的数据对象，如：

```
(define x (cons 1 2))
(define y (cons 3 4))
(define z (cons x y))
(car (car z))
=> 1
(car (cdr z))
=> 3
```

数据抽象实例：有理数的算术运算

```
(define one-half (make-rat 1 2))

(print-rat one-half)
 => 1/2

(define one-third (make-rat 1 3))
(print-rat (add-rat one-half one-third))
 => 5/6

(print-rat (mul-rat one-half one-third))
 => 1/6

(print-rat (add-rat one-third one-third))
 => 6/9
```

数据抽象实例：有理数的算术运算

有理数算数运算的实现(最简形式)：

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
;n,d是整数，代表分子分母
(define (numer x) (car x))
(define (denom x) (cdr x))
;x是有理数
```

```
(define (gcd a b)
   (if (= b 0)
      a
      (gcd b (remainder a b))))
```

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

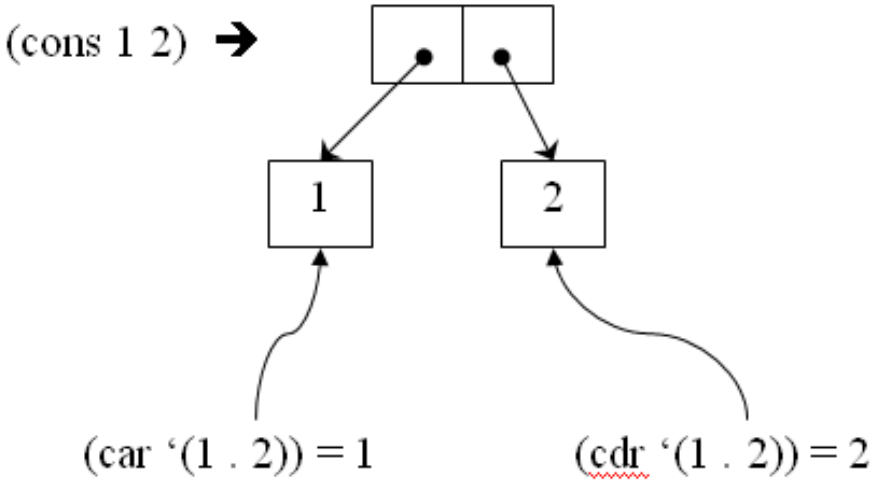
数据抽象实例：有理数的算术运算

有理数算数运算的实现（在求分子分母时取最简形式）：

```
(define (make-rat n d)
  (cons n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

基于过程实现的序对

序对，看起来似乎要用一个数据结构，
比如，两个指针来实现。



基于过程实现的序对

序对实际上可以完全用过程实现(一个序对就是一个过程，而不是一个数据结构)：

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- CONS" m)))))
```

; error 结束执行并输出各参数

```
(define (car z) (z 0))   ;z是一个序对，即一个过程
(define (cdr z) (z 1))
```

显然：`(car (cons 1 2))  => 1`

`(cdr (cons 1 2))  => 2`

闭包

闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。也可以说闭包是由函数和与其相关的引用环境组合而成的实体。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- CONS" m)))))
```

cons就是一个闭包

闭包

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- CONS" m)))))

(define a (cons 1 2))
(a 0)   => 1
(a 1)   => 2

(define b (cons 3 4))
(b 0)   => 3
(b 1)   => 4
```

闭包

```
(define closure-demo
  (let ((y 5))
    (lambda (x)
      (set! y (+ y x)) ; (set! a b) 等价于赋值语句 a = b;
      y)))


(closure-demo 6)
```

闭包

```
(define closure-demo
  (let ((y 5))
    (lambda (x)
      (set! y (+ y x)) ; (set! a b) 等价于赋值语句 a = b;
      y)))


(closure-demo 6)
=> 11
(closure-demo 10)
```

闭包

```
(define closure-demo
  (let ((y 5))
    (lambda (x)
      (set! y (+ y x)) ; (set! a b) 等价于赋值语句 a = b;
      y)))


(closure-demo 6)
=> 11
(closure-demo 10)
=> 21
```

闭包

```
(define closure-demo
  (let ((y 5))
    (lambda (x)
      (set! y (+ y x)) ; (set! a b) 等价于赋值语句 a = b;
      y)))
```

等价于：

```
(define closure-demo
  ((lambda (y)
    (lambda (x)
      (set! y (+ y x))
      y))
   5)) ;define则立即对closure-demo求值，即调用 lambda (y)
```

let的等价形式：

```
((lambda (<var1> ...<varn>)
    <body>)
 <exp1>
 ...
 <expn>)
```

闭包

```
(define closure-demo
  (let ((y 5)
        (k (display "kk")))
    (lambda (x)
      (set! y (+ y x))
      y))) ;=> kk

(closure-demo 6)   ;=>11
(closure-demo 10)  ;=>21
```

```
(define (closure-demo)
  (let ((y 5)
        (k (display "kk")))
    (lambda (x)
      (set! y (+ y x))
      y)))


((closure-demo) 6)
((closure-demo) 10)
(define x (closure-demo))
(x 6)
(x 10)
```

闭包

```
(define closure-demo
  (let ((y 5)
        (k (display "kk")))
    (lambda (x)
      (set! y (+ y x))
      y))) ;=> kk

(closure-demo 6)   ;=>11
(closure-demo 10)  ;=>21
```

```
(define (closure-demo)
  (let ((y 5)
        (k (display "kk")))
    (lambda (x)
      (set! y (+ y x))
      y)))


((closure-demo) 6)   ;=>kk11
((closure-demo) 10)  ;=>kk15
(define x (closure-demo))  ;=>kk
(x 6)  ;=>11
(x 10);=>21
```

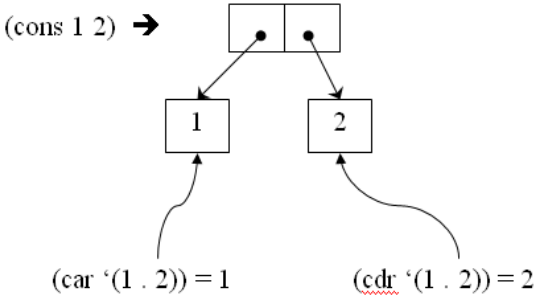
闭包

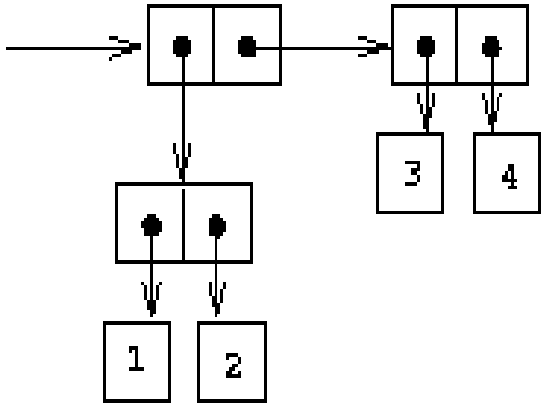
```
#lang racket
(define (func)
        (define x 100)
        (define (plus) (set! x (+ x 10)))
        (define (show) (displayln x))
        (cons plus show)
)
(define m (func))
((car m))
((cdr m)) =>?
```

闭包

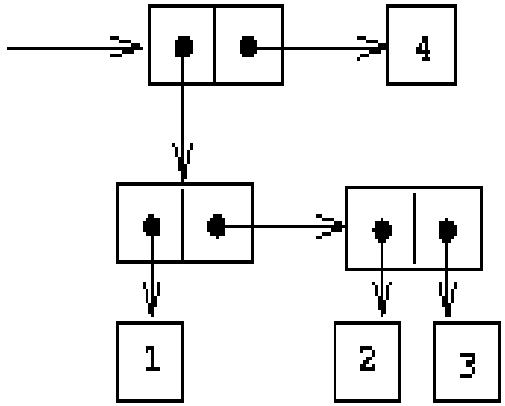
```
#lang racket
(define (func)
        (define x 100)
        (define (plus) (set! x (+ x 10)))
        (define (show) (displayln x))
        (cons plus show)
)
(define m (func))
((car m))
((cdr m)) =>110
```

层次数据：用序对组合成序对






```
(cons (cons 1 2)
      (cons 3 4))
```




```
(cons (cons 1
            (cons 2 3))
      4)
```

序列（列表）

列表是由多个类型相同或不同的数据连续组成的数据类型。

```
(define lst (list 1 2 3 4 ))
lst
(length lst)  ; 取得列表的长度
(list-ref lst 3)  ; 取得列表第3项的值（从0开始）
(define y (make-list 5 "a"))  ;创建列表
y
```

输出：
'(1 2 3 4)
4
4
'("a" "a" "a" "a" "a")

列表操作

列表是由多个类型相同或不同的数据连续组成的数据类型。
car用于取列表第0项，cdr代表除第0项外的其余部分组成的新表

```
(define lst (list 1 2 3 4 5))
(car lst)
(cdr lst)
(cadr lst)
(caddr lst)
(cadddr lst)
(cddr lst)
(cdddr lst)
(cddddr lst)
```

输出：
1
'(2 3 4 5)
2
3
4
'(3 4 5)
'(4 5)
'(5)

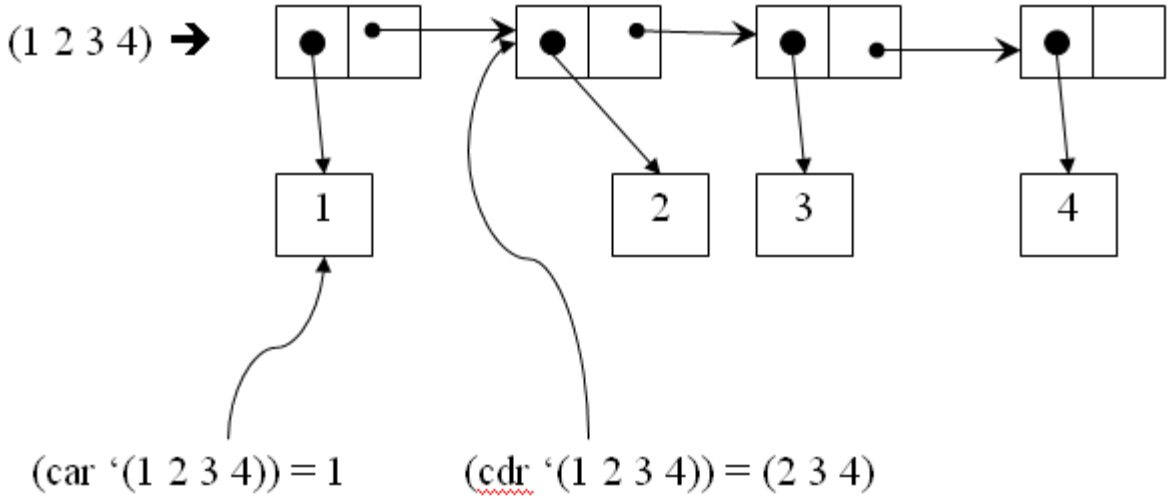
列表操作

cdr是构造一张新表，还是仅返回一个指向原表的指针？

列表操作

cdr是构造一张新表，还是仅返回一个指向原表的指针？

答案：在Racket中是返回原表指针



多重列表

多重列表：列表的元素可以是列表

```
(define x (list 0 (list 1 2 3) 4 (list "a" 200)))
x
=>'(0 (1 2 3) 4 ("a" 200))
(caadr x) ;若x的第二项是列表或序对，则取第二项的第一项
=>1

(define y (list 0 (cons 1 2) 4 (list "a" 200)))
(caadr y)
=>1
```

空表

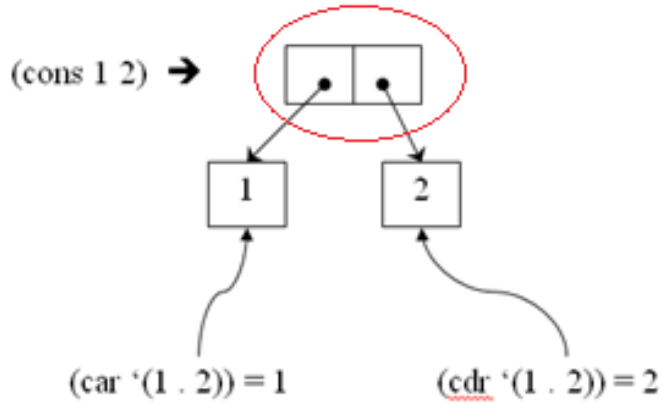
- 空表： nil （Racket里nil无定义，用 ‘()表示空表）

对空表进行 car 和 cdr操作，均为非法

```
(null? lst)   ;判断lst是否是空表，返回值为#t或#f
(pair? '())   ;=>#f ， 空表不是序对
```

cons和列表

- cons会形成一个新的“盒子”



cons和列表

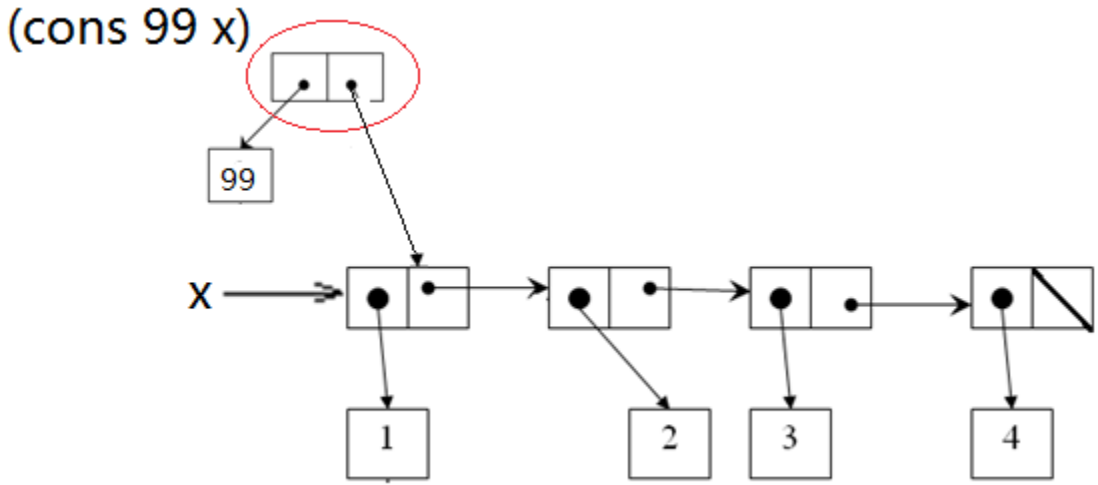
●如果 b是列表，则 (cons a b)形成一张新列表，新列表是在b的头部添加了a

```
(cons 1 '())
=>'(1)
(cons 1 (list 2 3))
=>'(1 2 3)

(list <a1> <a2> ... <an>)
```
等价于
```
(cons <a1> (cons <a2> (cons ... (cons <an> nil) ...)))
```

cons和列表

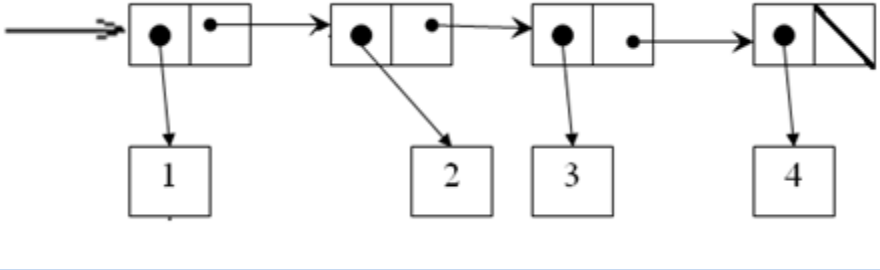
- cons会形成一个新的“盒子”

```
(define x (list 1 2 3 4))
(cons 99 x)
```



cons和列表

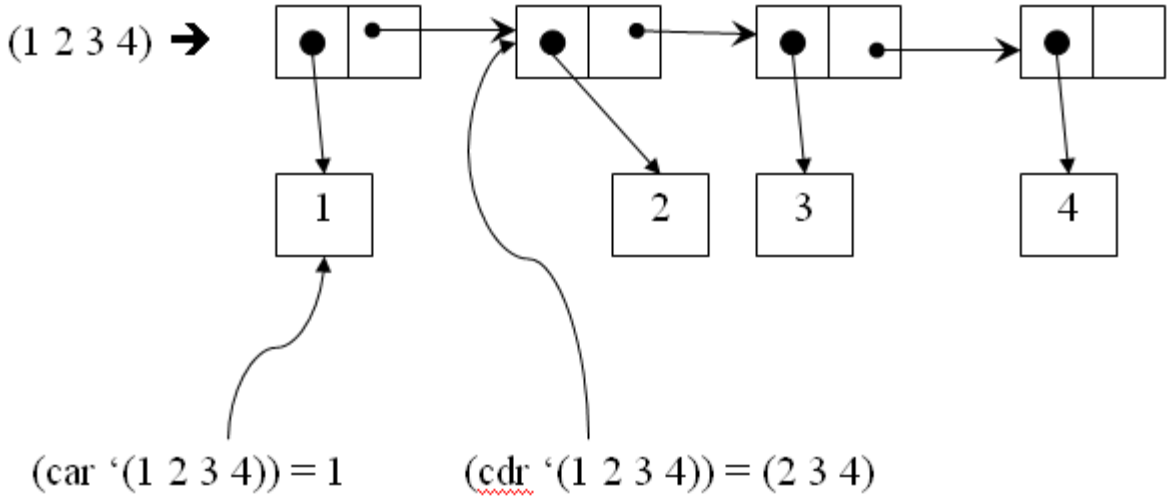
```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
=>'(1 2 3 4)
(list 1 2 3 4) ;和前者等价
=>'(1 2 3 4)
```




```
(cons 1 (cons 2 (cons 3 4))) ;和前两者不等价!
=>'(1 2 3 . 4)
```

列表和序对的关系

非空列表就是一个序对。序对的前部指向列表的car，后部是一个指针，指向列表的cdr。

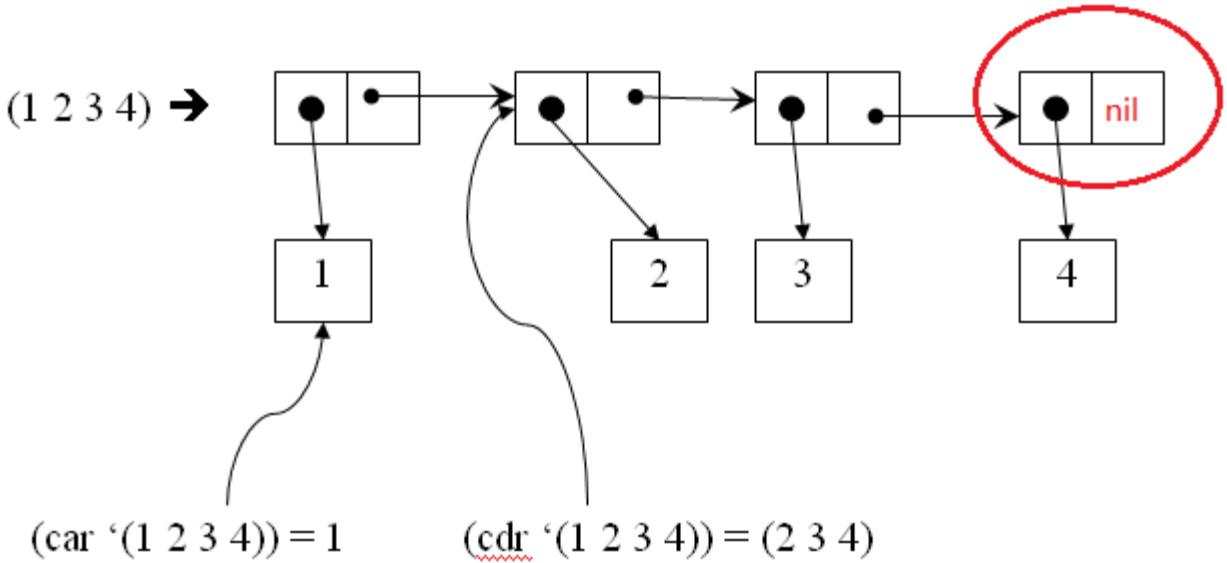


空表不是序对。

```
(pair? '()) ;=> #f
```

列表和序对的关系

列表的最后一个盒子的后半部是个空指针。空表也是一个空指针。



列表和序对的关系

注意序对和表的不同。设

```
(define x (cons 1 2))
(define y (list 1 2))
```

这时有：

```
(car x)       => 1
(cdr x)       => 2
(car y)       => 1
(cdr y)       => '(2)
(cdr (cdr y))  => '()
(cdr (cdr x)) => 出错: cdr: contract violation.
expected: pair? given: 2
```

列表和序对

- 每次cons 都要做一次动态存储分配

- 做表操作时可能导致一些序对单元失去引用。Scheme 系统有内置的废料收集系统，能自动回收这种单元

表操作：取元素

```
(define (list-ref items n)   ;取items的第n个元素,n从0开始
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))


(define squares (list 1 4 9 16 25))
(list-ref squares 3) =>16
```

N太大会导致程序运行出错 ， 例如 (list-ref squares 20) =>

cdr: contract violation
expected: pair?
given: '()

表操作：求表长度

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))


(define odds (list 1 3 5 7))
(length odds)
```

=>4

表操作：拼接表

append是基本过程，实现表的拼接

使用时应使得两个参数都是表，否则拼出来的就不是表了

```
(define squares (list 1 4 9 16 25))
(define odds (list 1 3 5 7))
(append squares odds)
(append odds squares)
```

输出：

'(1 4 9 16 25 1 3 5 7)
'(1 3 5 7 1 4 9 16 25)

表操作：拼接表

append的实现：

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
            (append (cdr list1) list2))))
```

表操作：拼接表

append的实现：

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
            (append (cdr list1) list2))))
```

表操作：拼接表

```
#lang racket
(require r5rs) ;无此就不能用 set-car!
(define s1 (list 1 2))
(define s2 (list 20 30))
(define p (append s1 s2))
(set-car! p 100) ;set-car!就是修改表的car
(car s1)       =>?
```

表操作：拼接表

```
#lang racket
(require r5rs)
(define s1 (list 1 2))
(define s2 (list 20 30))
(define p (append s1 s2))
(set-car! p 100)
(car s1)      => 1
```

表操作：任意多个参数的过程

● 基本过程+、* 和list 等都允许任意多个参数。也可以自定义任意多个参数的过程：

```
(define (f  x y . z) <body>)
```

定义了过程f，f至少有两个参数，且可以有多于2个的任意多个参数

●圆点前后必须有空格。圆点前面的参数是调用时必给的。在必给参数后面可以跟任意多个参数（包括0个）

●在过程中，可变长参数z应被看作一个列表

（f 1 2 3 4 5）=> x = 1 , y = 2, z = (3 4 5)

表操作：任意多个参数的过程

求任意多个数的平方和的过程:

```
(define (square x) (* x x))
(define (square-sum x . args)
  (define (helper sum lst)
    (if (null? lst)
        sum
        (helper (+ sum (square (car lst))) (cdr lst))))
  (helper (square x) args) )
```

如果需要处理的是**0 项或任意多项，参数表用(square-sum . args)**，过程体也需要相应修改

用闭包实现面向对象

```
(define (make-point-2D x y) ;二维点类
  (define (get-x) x)
  (define (get-y) y)
  (define (set-x! new-x) (set! x new-x))
  (define (set-y! new-y) (set! y new-y))
  (lambda (selector . args) ; . args表示后续有任意多个参数，亦可没有
     (case selector
        ((get-x) (apply get-x args)) ;apply是基本过程，以args的元素调用get-x
        ('get-y (apply get-y args)) ;写 ((get-y) (apply get-y args))一样
        ('set-x! (apply set-x! args)) ;case 里面 (get-x)等价于 'getx
        ((set-y!) (apply set-y! args))
        (else (error "don't understand " selector)))))

(define p1 (make-point-2D 10 20))
(p1 'get-x) ;=>10     ;'get-x表示“标识符get-x”,不是字符串"get-x"
(p1 'get-y) ;=>20
(p1 'set-x! 5)
(p1 'get-x) ;=> 5
```

用闭包实现面向对象

```
(define (make-point-3D x y z) ;三维点类，从二维点类派生
  (let ((parent (make-point-2D x y)))
    (define (get-z) z)
    (define (set-z! new-z) (set! z new-z))
    (lambda (selector . args)
      (case selector
        ((get-z) (apply get-z args))
        ((set-z!) (apply set-z! args))
        (else (apply parent (cons selector args)))))))
;parent是一个能接收个数不定的参数的过程
 (define p2 (make-point-3D 10 20 30))
 (p2 'get-x)   ;=>10
 (p2 'get-y)   ;=>20
 (p2 'set-z! 5)
 (p2 'get-z)   ;=>5
 (p2 'set-x! 37)
 (p2 'get-x)   ;=>37
```

C++如何实现表？

scheme中，　表里的元素可以类型各不相同

C++只能实现一个基类指针表，表中元素可以指向各种有共同基类的不同类型的对象。

使用boost库的 any 类可以构建看上去包含不同类型元素的表。

表的映射

●有一类表操作十分常用，即对一个表中的每个元素执行相同操作并算出一个结果，然后用这些结果构造出一个新表。结果在新表中的次序，和原表次序一一对应。

●和C++中STL里的 for_each有一些共同之处
●例：构造把一个表中所有元素等比例放大或缩小的表

```
(define (scale-list items factor)
  (if (null? items)
      '()
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))

(scale-list (list 1 2 3 4 5) 10)
```

=> '(10 20 30 40 50)

表的映射

●可以编写更通用的过程map，完成将函数 proc 运用于一个表中的每个元素x，得到proc(x)，然后用proc(x)构建一张新表。

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))

(map abs (list -10 2.5 -11.6 17))
=> '(10 2.5 11.6 17)
(map (lambda (x) (* x x)) (list 1 2 3 4))
=> '(1 4 9 16)
```

●实际上，map是scheme的基本过程，可以用于列表！

表的映射

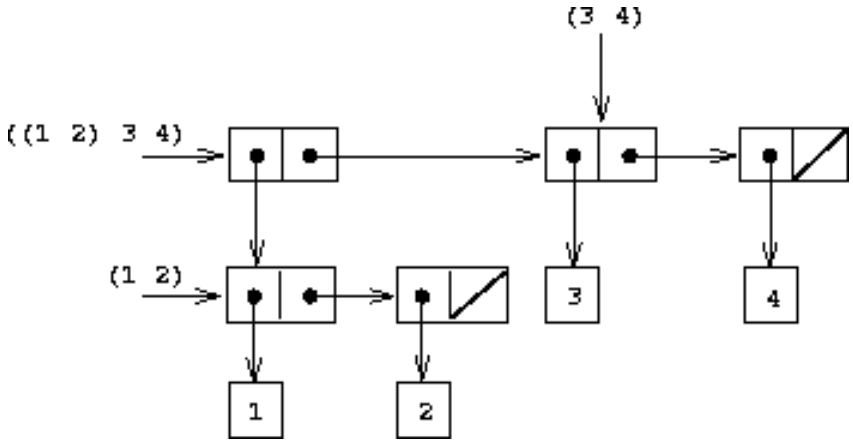
- 用map实现的scale-list :

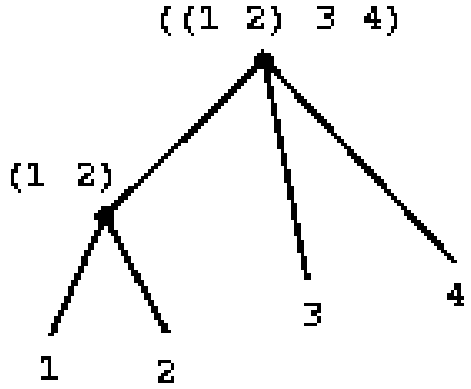
```
(define (scale-list items factor)
  (map (lambda (x) (* x factor))
       items) )

(scale-list (list 1 2 3 4 5) 10)
=> '(10 20 30 40 50)
```

- map实现了对数据的具体操作和对数据的遍历顺序的分离。此种思路具有普遍意义。

树结构

```
(cons (list 1 2) (list 3 4))
=>'((1 2) 3 4)
```

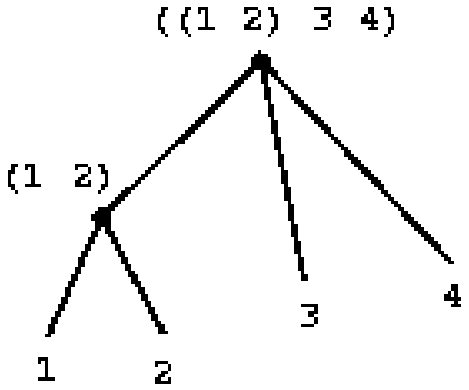




- 这种结构可以看作是树。一个表就是一个子树。非序对的元素(表也是序对），是叶子。

求树的叶子个数

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```




```
(define x (cons (list 1 2) (list 3 4)))
(count-leaves x)   ;=> 4
(count-leaves (list x x))   ;=> 8
(count-leaves (list 1 2 (cons 3 4))) ;=> 4
```

把树叶值乘以factor得到新树

```
(define (scale-tree tree factor)
  (cond ((null? tree) '())
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                    (scale-tree (cdr tree) factor)))))
```

```
(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)) 10)
=>'(10 (20 (30 40) 50) (60 70))
```

用map实现scale-tree

```
(define (scale-tree2 tree factor)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
             (scale-tree2 sub-tree factor)
             (* sub-tree factor)))
       tree))
```

真的和前面的scale-tree等价吗？

用map实现scale-tree

```
(define (scale-tree2 tree factor)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
             (scale-tree2 sub-tree factor)
             (* sub-tree factor)))
       tree))
```

真的和前面的scale-tree等价吗？

```
(define x (list 1 (cons 2 3)))
(scale-tree x 10) ;=> '(10 (20 . 30))
(scale-tree2 x 10) => 错误. map: contract violation
  expected: list?  given: '(2 . 3)  argument position: 2nd
  other arguments...:
   #<procedure>   map的第三个参数必须是 list
```