函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

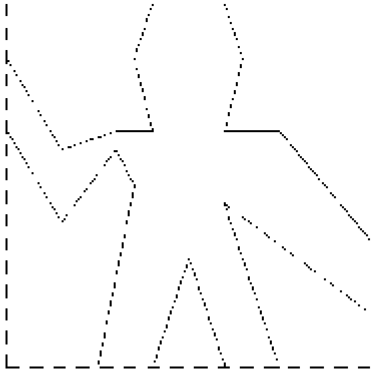
http://blog.sina.com.cn/u/3266490431

第五讲

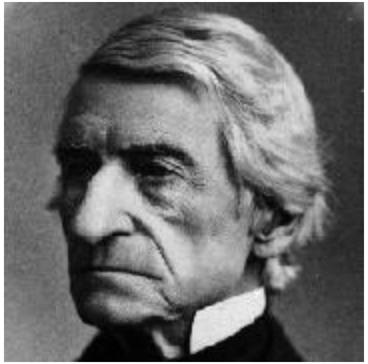
图形语言

●假设有一种闭包，叫painter，painter能接受一个frame作为参数，在frame上画出特定图形的变形(翻转，旋转，拉伸……)。

●这个特定图形，是由和painter闭包相联系的一些列变量来描述，比如，这些变量可以是一个列表，列表里面的每一项是一个线段，整个列表就描述了一个图形。Painter里描述的图形称之为painter的原图形。原图形中所有点的坐标，范围都是[0,1]



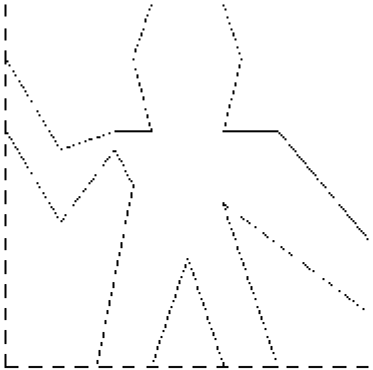
painter **wave** 的原图形



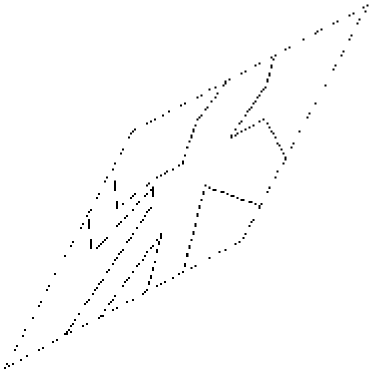
painter **rogers** 的原图形

图形语言

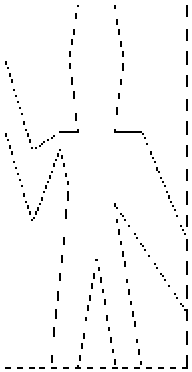
●**frame**代表**painter**绘图的场所(归根到底是窗口上个一个区域）　。**painter**接受一个**frame**作为参数后，就能在该**frame**上画出原图形的变形(翻转，旋转，拉伸……)。



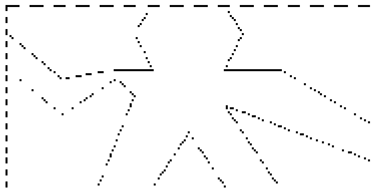
Painter **wave** 的原图形



(wave frame1)



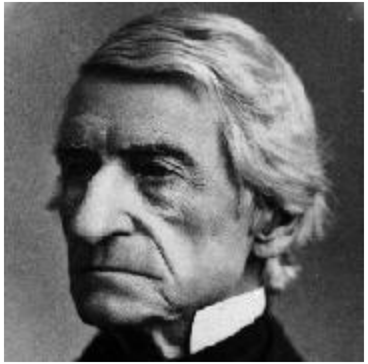
(wave frame2)



(wave frame3)

图形语言

●frame代表painter绘图的场所(归根到底是窗口上个一个区域)。painter接受一个frame作为参数后，就能在该frame上画出原图形的变形(翻转，旋转，拉伸……)。



painter **rogers** 的原图形

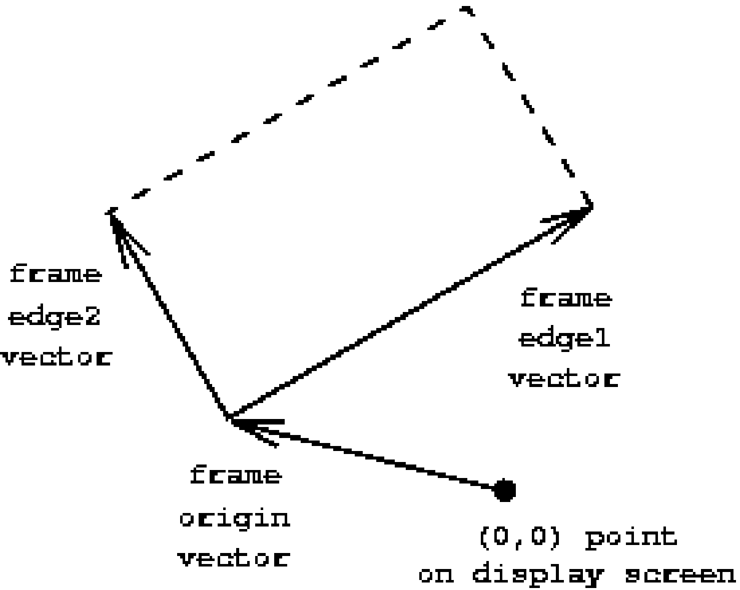
(rogers frame1)

(rogers frame2)

(rogers frame3)

图形语言

●一个**frame**由一个原点和两个向量描述



● frame的原点坐标是相对于显示设备的绝对原点(0,0)的。(显示设备：例如窗口)

●edge1相当于frame的x轴

●edge2相当于frame的y轴

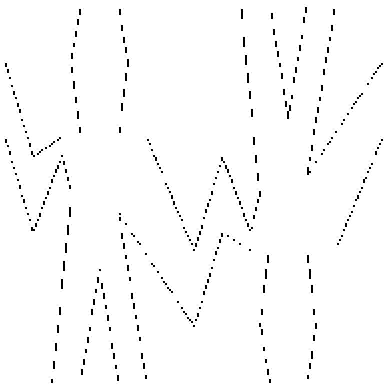
●edge1和edge2的坐标都是相对于frame的原点的

(一个向量可以用一个相对于给定原点的坐标来表示)

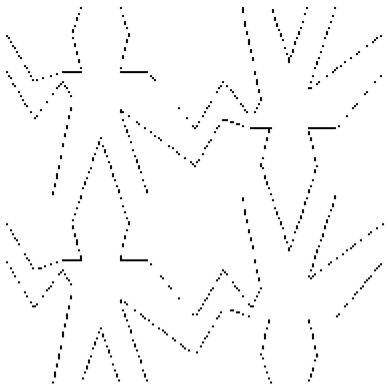
图形语言

- painter可以通过一些方式组合起来，形成新的 painter

```
(define wave2 (beside wave (flip-vert wave)))
(define wave4 (below wave2 wave2))
```



wave2



wave4

beside,flip-vert, below 都需要自己编写还可以写 flip-horiz

这些函数以painter为参数，返回新painter

图形语言

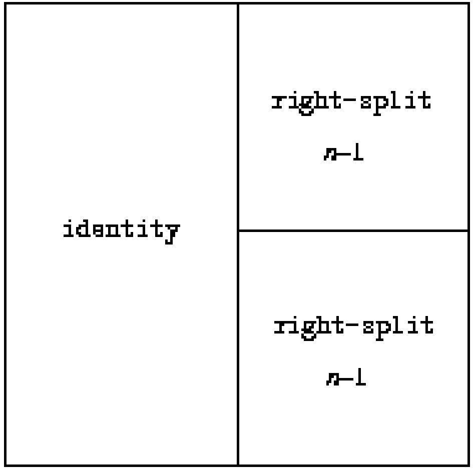
- **wave4的另一种写法：**

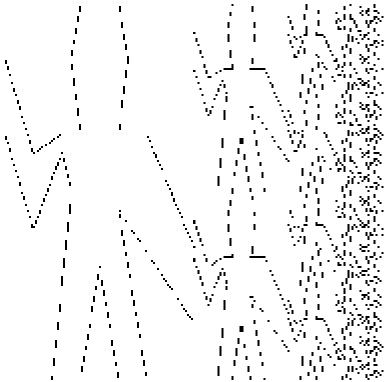
```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))

(define wave4 (flipped-pairs wave))
```

图形语言

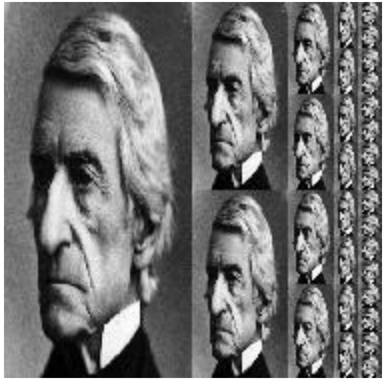
- 递归的painter组合 right-split:

```
(define (right-split painter n) ;生成新painter，右分n次
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))
```





(right-split wave 4)

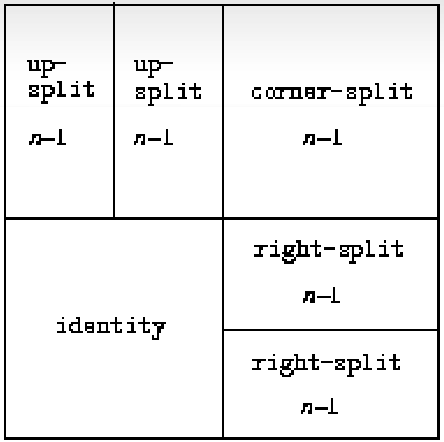


(right-split rogers 4)

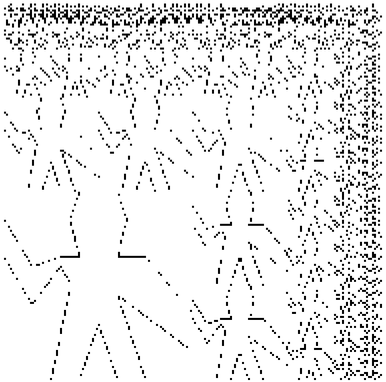
图形语言

- 递归的painter组合 conner-split:

```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter (- n 1))))
        (let ((top-left (beside up up))
              (bottom-right (below right right))
              (corner (corner-split painter (- n 1))))
          (beside (below painter top-left)
                  (below bottom-right corner))))))
```



(corner-split rogers 4)

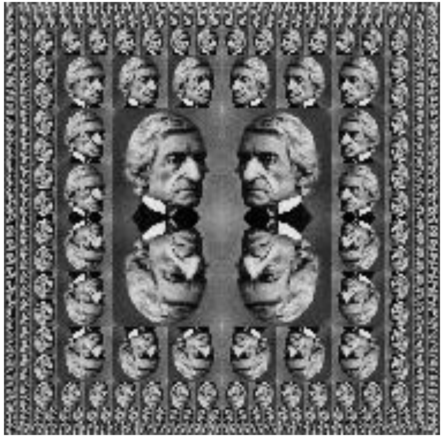


(corner-split wave 4)

图形语言

- 更复杂的painter 组合square-limit ：

```
(define (square-limit painter n)
  (let ((quarter (corner-split painter n)))
    (let ((half (beside (flip-horiz quarter) quarter)))
      (below (flip-vert half) half))))
```



(square-limit rogers 4)

图形语言:高阶操作

- 高阶操作以**对painter 的操作**作为参数，生成**新的对painter 的操作**

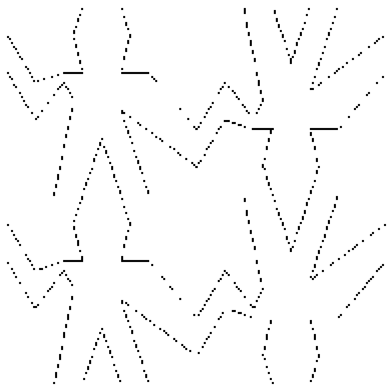
例：**flipped-pairs** 和**square-limit** 都是将原区域分为**4**块而后按不同变换方式摆放四个部分的图像。把这**4** 个变换抽象为过程参数，就得到：

```
(define (square-of-four tl tr bl br)
  (lambda (painter)
    (let ((top (beside (tl painter) (tr painter)))
          (bottom (beside (bl painter) (br painter))))
      (below bottom top))))
```

图形语言:高阶操作

●利用**square-of-four** 重新定义**flipped-pairs**：

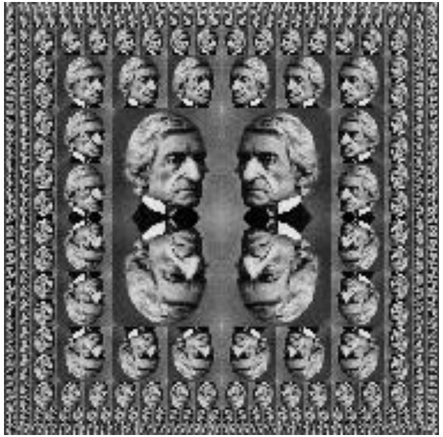
```
(define (flipped-pairs painter)
  (let ((combine4 (square-of-four identity flip-vert
                                  identity flip-vert)))
    (combine4 painter)))
; identity 是“恒等变换”，直接返回参数
```



图形语言:高阶操作

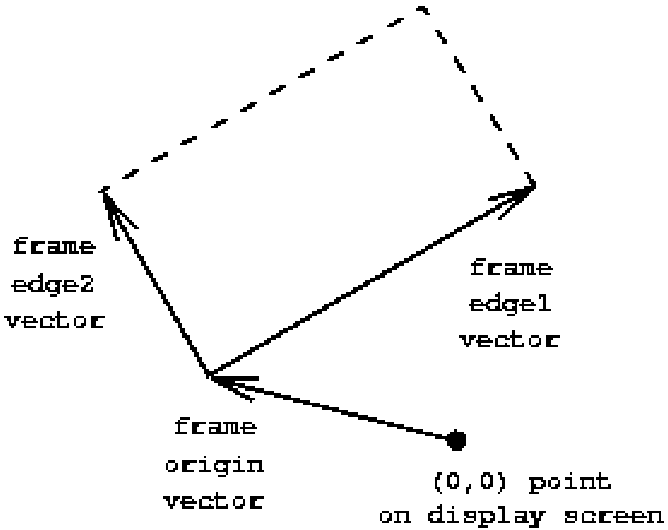
●利用**square-of-four** 重新定义 **square-limit**：

```
(define (square-limit painter n)
  (let ((combine4 (square-of-four flip-horiz identity
                                  rotate180 flip-vert)))
    (combine4 (corner-split painter n))));
```



图形语言：框架

- 一个frame由一个原点和两个向量描述



- frame的原点坐标是相对于显示设备的绝对原点(0,0)的。(显示设备：例如窗口)
- edge1相当于frame的x轴
- edge2相当于frame的y轴
- 用相对于frame的原点的办法记录edge1和edge2的坐标

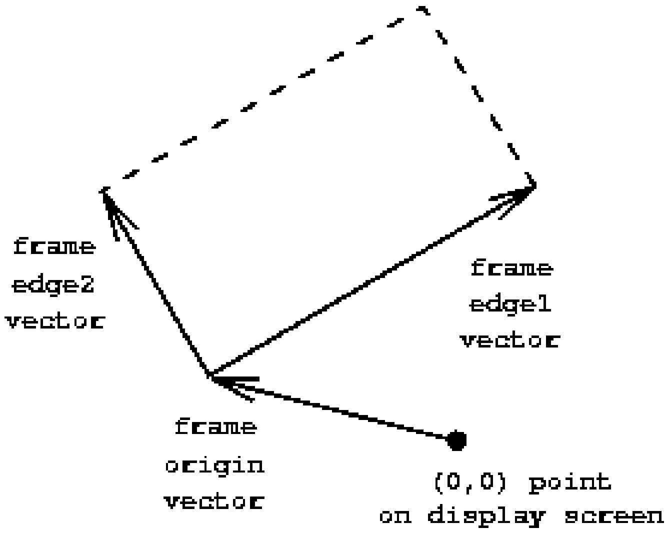
(一个向量可以用一个相对于给定原点的坐标来表示）

painter里的原图形是在单位正方里的。将原图形映射到frame里后，对于原图形里的点 (x,y)，映射点的坐标(相对于绝对原点）是：

origin(frame) + x * edge1(frame) + y * edge2(frame)

图形语言：框架

- 一个**frame**由一个原点和两个向量描述



- frame的原点坐标是相对于显示设备的绝对原点(0,0)的。(显示设备：例如窗口)
- edge1相当于frame的x轴
- edge2相当于frame的y轴
- 用相对于frame的原点的办法记录edge1和edge2的坐标

(x,y)➔
origin(frame) + x * edge1(frame) + y * edge2(frame)

被变换图形在变换之后：
其(0, 0) 点总位于frame的原点
其(1, 1) 点总位于frame原点的对角点
其中点总位于frame的中点

图形语言：框架

- 原图形到frame的变换过程：

(x,y)➔ origin(frame) + x * edge1(frame) + y * edge2(frame)

```
(define (frame-coord-map frame);向量转换器
  (lambda (v) ;v是单位正方形中的向量(点)
    (add-vect
     (origin-frame frame) ;求frame的原点(相对于绝对原点)
     (add-vect (scale-vect (xcor-vect v)
                           (edge1-frame frame))
               (scale-vect (ycor-vect v)
                           (edge2-frame frame))))))
```

(frame-coord-map frame)的返回值是个闭包，该闭包接受向量v作为参数，返回v在frame中对应点的坐标（相对于绝对原点）

图形语言：向量操作

```
(define (make-vect x y) (cons x y))
(define (xcor-vect v)(car v))
(define (ycor-vect v)(cdr v))
(define (add-vect v1 v2)
  (make-vect (+ (xcor-vect v1)
                (xcor-vect v2))
             (+ (ycor-vect v1)
                (ycor-vect v2))))
(define (sub-vect v1 v2)
  (make-vect (- (xcor-vect v1)
                (xcor-vect v2))
             (- (ycor-vect v1)
                (ycor-vect v2))))
(define (scale-vect s v)
  (make-vect (* s (xcor-vect v))
             (* s (ycor-vect v))))
```

图形语言：框架操作

```
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2)) ;edge1和edge2也可以看做点，其坐标是相对于frame原点的

(define (origin-frame f)  (car f))
(define (edge1-frame f)  (cadr f))
(define (edge2-frame f)  (caddr f))

(define (frame-coord-map frame) ;向量转换器
   (lambda (v)
    (add-vect
     (origin-frame frame)
     (add-vect (scale-vect (xcor-vect v)
                           (edge1-frame frame))
               (scale-vect (ycor-vect v)
                           (edge2-frame frame))))))
```

图形语言：painter示例

● 定义painter：

```
(define (segments->painter segment-list) ;segment-list是线段列表
  (lambda (frame)
    (for-each
     (lambda (segment)
       (draw-line ;假定draw-line可以画线（以绝对原点作为原点）
        ((frame-coord-map frame) (start-segment segment))
        ((frame-coord-map frame) (end-segment segment))))
     segment-list))) ;本过程生成一个painter，其原图形是一系列线段

;for segments
(define (make-segment start end)  (cons start end))
(define (start-segment seg)  (car seg))
(define (end-segment seg)  (cdr seg))
;start, end都是线段端点（向量），坐标相对于绝对原点
```

图形语言：painter示例

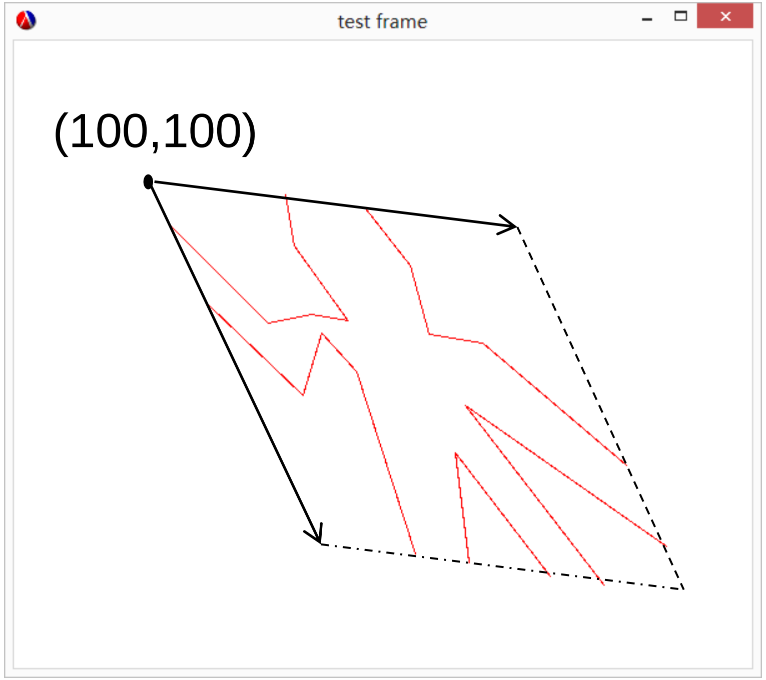
构造出合适的 segment-list后，就可以构造出wave

```
(define (wave frame) ;wave是一个painter，用 frame作为参数
  ((segments->painter waveShape ) frame)) ;waveShape是线段列表
```

;waveShape里的坐标必须都在[0,1]范围内!!!!

图形语言：painter示例

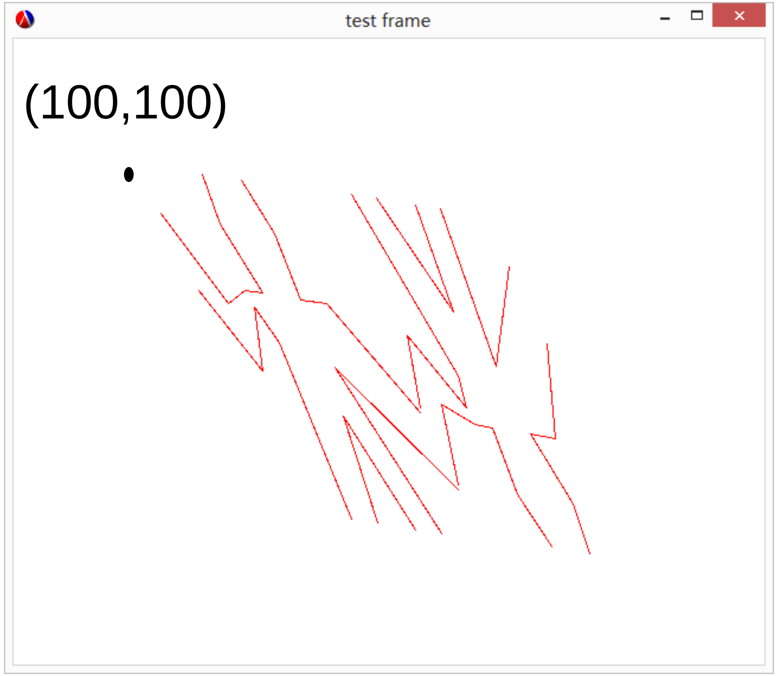
```
(define testFrame
  (make-frame (make-vect 100 100) (make-vect 299 50 )
      (make-vect 150 299)))

(wave testFrame) =>
```



Racket中，单位正方形的左上角坐标是（0，0），右下角坐标是(1,1)。课本上则是左下角坐标为(0,0),右上角坐标为(1,1)

图形语言：painter示例

```
(define testFrame
  (make-frame (make-vect 100 100) (make-vect 299 50 )
      (make-vect 150 299)))

(wave2 testFrame) =>
```




```
 (define wave2
        (beside wave
        (flip-vert wave)))
```

图形语言：变换和组合

beside,flip-vert,below等变换可以用下面的**transform-painter**来实现：

```
;返回值是个painter
(define (transform-painter painter origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
         (make-frame new-origin
                     (sub-vect (m corner1) new-origin);frame的两条边是
                                                       相对于frame原点的
                     (sub-vect (m corner2) new-origin)))))))
```

origin corner1 corner2都是相对于单位正方形的原点(0,0)，坐标范围都是[0，1]

先对原图形做个变换，　　(0,0) -> origin

(1,0) -> corner1

(0,1) -> conrner2

得到新的原图形，然后把新的原图形映射到 **frame**

图形语言：变换和组合

`beside,flip-vert,below`等变换可以用下面的transform-painter来实现：

```
(define (transform-painter painter origin corner1 corner2)…)
```

以一个 frame f为参数，把 (origin corner1 corner2)这个frame映射到 f中去 。它从painter 生成一个新 painter，叫 newpainter 如果 painter f 在 f上绘图, newpainter f 就在 (origin corner1 corner2) 对应到f中的那部分,假设叫 f‘ 上绘图 。这里要求painter要画的东西都是在 (0,0)-(1,1) 范围内的。 (origin corner1 corner2)必须是 在((0,0) (0 1) (1 0))这个单位frame范围内的一个frame, 它和单位frame之间的关系，就是 f' 和 f的关系

图形语言：变换和组合

beside,flip-vert,below等变换可以用下面的transform-painter来实现：

```
;纵向翻转：
(define (flip-vert painter) ;
   (transform-painter painter
                      (make-vect 0.0 1.0)    ; new origin
                      (make-vect 1.0 1.0)    ; new end of edge1
                      (make-vect 0.0 0.0)))  ; new end of edge2

;将图形收缩到原区域的右上四分之一区域：
(define (shrink-to-upper-right painter)
   (transform-painter painter
                      (make-vect 0.5 0.5)
                      (make-vect 1.0 0.5)
                      (make-vect 0.5 1.0)))
```

```
(0,0) -> origin
(1,0) -> corner1
(0,1) -> conrner2
```

图形语言：变换和组合

将图形逆时针旋转90 度:

```
(define (rotate90 painter)
  (transform-painter painter
                     (make-vect 1.0 0.0)
                     (make-vect 1.0 1.0)
                     (make-vect 0.0 0.0)))
```

;将图形向中心收缩:

```
(define (squash-inwards painter)
  (transform-painter painter
                     (make-vect 0.0 0.0)
                     (make-vect 0.65 0.35)
                     (make-vect 0.35 0.65)))
```

图形语言：变换和组合

;beside:

```
(define (beside painter1 painter2) ;在frame左边画painter1,右边画painter2
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
           (transform-painter painter1
                              (make-vect 0.0 0.0)
                              split-point
                              (make-vect 0.0 1.0)))
          (paint-right
           (transform-painter painter2
                              split-point
                              (make-vect 1.0 0.0)
                              (make-vect 0.5 1.0))))
      (lambda (frame)
        (paint-left frame)
        (paint-right frame)))))
```

符号数据和符号处理

- Lisp一大强项是对符号处理，例如函数式求导，多项式运算（非数值计算）等的支持

- 符号计算处理的表达式：

(a b c d)
(23 45 17)
((Norah 12) (Molly 9) (Anna 7) (Lauren 6) (Charlotte 4))

不把 **a b c d**以及 **norah Molly**当变量看，也不当字符串看，就是符号。**a**的值就是**a**，**Norah**的值就是 **Norah**

符号数据和符号处理

- Scheme 用单引号表示后面的东西是符号

```
(define a 1)
(define b 2)
(list a b)
=>'(1 2)
(list 'a 'b)
=>'(a b)
(list 'a b)
=>'(a 2)
(car '(a b c))
=>'a
(cdr '(a b c))
=>'(b c)
```

符号数据和符号处理

- Scheme 用单引号表示后面的东西是符号

在解释器看来，单引号等价于 (quote …)

```
(quote ba)    等价于  'ba

(quote ba)
=>'ba
(quote (a b c d)) 等价于 '(a b c d)
=>'(a b c d)
(car '(list 1 2 3)) => 'list
(car (quote (list 1 2 3))) => 'list
(cadr ''abracadabra) =>?
```

符号数据和符号处理

- **Scheme** 用单引号表示后面的东西是符号

在解释器看来，单引号等价于 (quote …)

```
(quote ba)    等价于  'ba

(quote ba)
=>'ba
(quote (a b c d)) 等价于 `(a b c d)
=>'(a b c d)
(car '(list 1 2 3)) => 'list
(car (quote (list 1 2 3))) => 'list
(car ''abracadabra) => 'quote
(cadr ''abracadabra) => ?
```

符号数据和符号处理

- Scheme 用单引号表示后面的东西是符号

在解释器看来，单引号等价于 (quote …)

```
(quote ba)    等价于  'ba

(quote ba)
=>'ba
(quote (a b c d)) 等价于 `(a b c d)
=>'(a b c d)
(car '(list 1 2 3)) => 'list
(car (quote (list 1 2 3))) => 'list
(car ''abracadabra) => 'quote
(cadr ''abracadabra) => 'abracadabra
```

符号数据和符号处理

- 谓词eq?可以用于判断两个参数是否是同一个符号

```
(define a 3)
(eq? a 3)
=> #t
(define b 3)
(eq? a b)
=> #t
(eq? 'a 'b)
=> #f
(define x 'kk)
(eq? x 'kk)
=> #t
```

实例：符号求导

$$\frac{\mathrm{d}c}{\mathrm{d}x} = 0$$ c 是常量或者与 x 不同的变量

$$\frac{\mathrm{d}x}{\mathrm{d}x} = 1$$

$$\frac{\mathrm{d}u + v}{\mathrm{d}x} = \frac{\mathrm{d}u}{\mathrm{d}x} + \frac{\mathrm{d}v}{\mathrm{d}x}$$

$$\frac{\mathrm{d}uv}{\mathrm{d}x} = u\left(\frac{\mathrm{d}v}{\mathrm{d}x}\right) + v\left(\frac{\mathrm{d}u}{\mathrm{d}x}\right)$$

须用递归处理后两条

实例：符号求导

- 代数式的表示方法：前缀式。各种变量用符号表示。

  例如： `3x+2 => (+ (* 3 x) 2)`

实例：符号求导

须规定表达式的形式（如何构造表达式），以及判断表达式的格式。为此需要以下构造函数，选择函数和谓词：

(variable? e)	**e 是个变量?**
(same-variable? v1 v2)	**v1 和v2 是同一个变量?**
(sum? e)	**e 是和式?**
(addend e)	**和式e 的被加数.**
(augend e)	**和式e 的加数.**
(make-sum a1 a2)	**构造a1 和a2 的和式.**
(product? e)	**e 是乘式?**
(multiplier e)	**乘式e 的被乘数.**
(multiplicand e)	**乘式e 的乘数.**
(make-product m1 m2)	**构造m1 和m2 的乘式**

实例：符号求导

```
(define (deriv exp var) ;对表达式exp求导，var是自变量（符号）
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                         (deriv (multiplicand exp) var))
           (make-product (deriv (multiplier exp) var)
                         (multiplicand exp))))
        (else
         (error "unknown expression type -- DERIV" exp))))
```

实例：符号求导

```
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (product? x)
  (and (pair? x) (eq? (car x) '*)))

(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
```

实例：符号求导

使用实例（结果正确，但是没化简）：

```
(deriv '(* (* x y) (+ x 3)) 'x)

=>
'(+ (* (* x y) (+ 1 0)) (* (+ (* x 0) (* 1 y)) (+ x 3)))
```

实例：符号求导

修改和式和乘式以化简：

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))

(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

实例：集合

用列表来表示集合，并且假定元素不许重复。需要以下集合操作：

union-set	求两个集合的并集
intersection-set	求两个集合的交集
element-of-set?	判断是否集合的元素
adjoin-set	求加入新元素后形成的新集合

实例：用排序的列表实现的集合

```
(define (element-of-set? x set) ;假设set从小倒大排序
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)))))
```

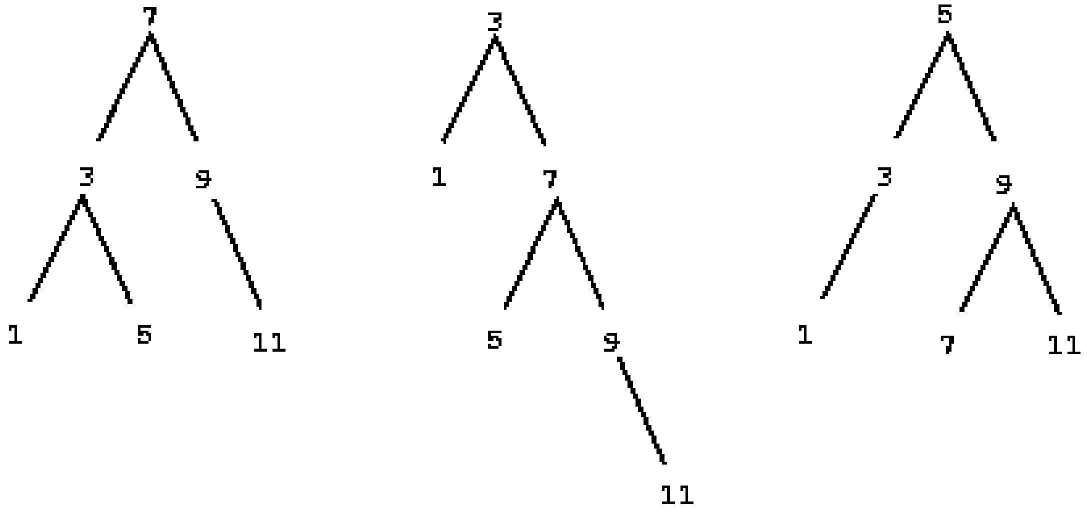
实例：用排序的列表实现的集合

```
;假设set1 set2从小倒大排序
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1
                     (intersection-set (cdr set1)
                                       (cdr set2))))
              ((< x1 x2)
               (intersection-set (cdr set1) set2))
              ((< x2 x1)
               (intersection-set set1 (cdr set2)))))))
```

O(m+n)

实例：用二叉树实现排序的集合

二叉排序树，左儿子总是小，右儿子总是大



实例：用二叉树实现排序的集合

一个二叉排序树是一个有三个元素的列表：

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right)
  (list entry left right))
```

实例：用二叉树实现排序的集合

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))
```

实例：用二叉树实现排序的集合

```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> x (entry set))
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin-set x (right-branch set))))))
```

实例：哈夫曼编码树

需要对信息中用到的每个字符进行编码。

定长编码方案：每个字符编码的比特数都相同。比如**ASCII**编码方案。

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

BACADAEAFABBAAAGAH

被编码为以下54个bits:

001000010000011000100000101000001001000000000110000111

实例：哈夫曼编码树

熵编码：使用频率高的字符，给予较短编码，使用频率低的字符，给予较长编码，如哈夫曼编码。

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

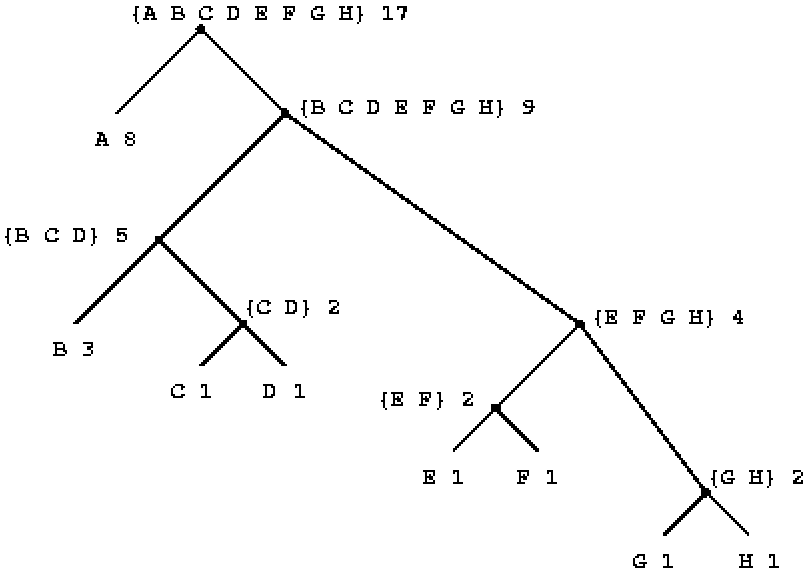
BACADAEAFABBAAAGAH

被编码为以下42个bits:

100010100101101100011010100100000111001111

实例：哈夫曼编码树

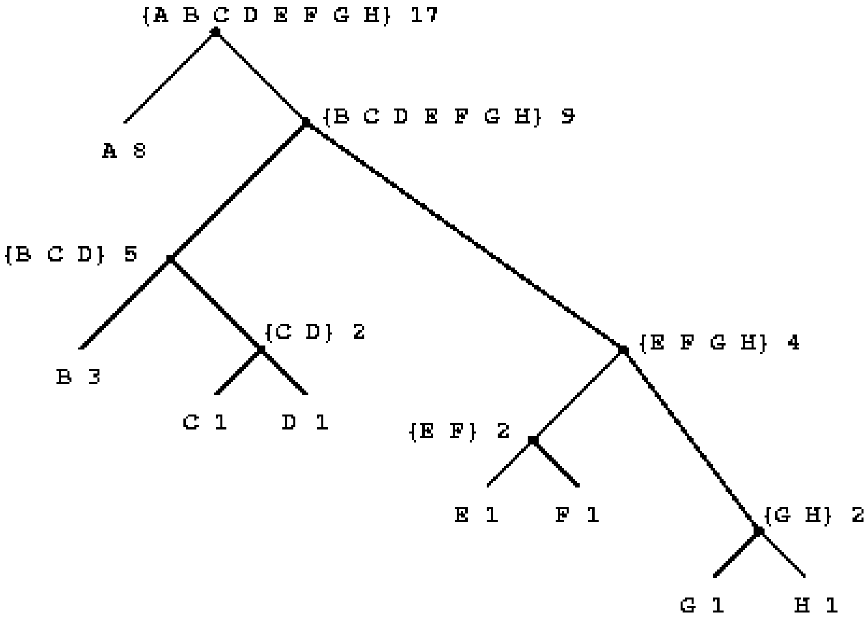
使用可变长编码，需要解决的问题是：如何区分一个编码是一个字符的完整编码，还是另一个字符的编码的前缀。解决办法之一就是采用**前缀编码**:任何一个字符的编码，都不会是其他字符编码的前缀。



哈夫曼编码树:

- 二叉树
- 叶子代表字符，且每个叶子节点有个权值，权值即该字符的出现频率
- 非叶子节点里存放着以它为根的子树中的所有字符，以及这些字符的权值之和
- 权值仅用来建树，对于字符串的解码和编码没有用处

实例：哈夫曼编码树

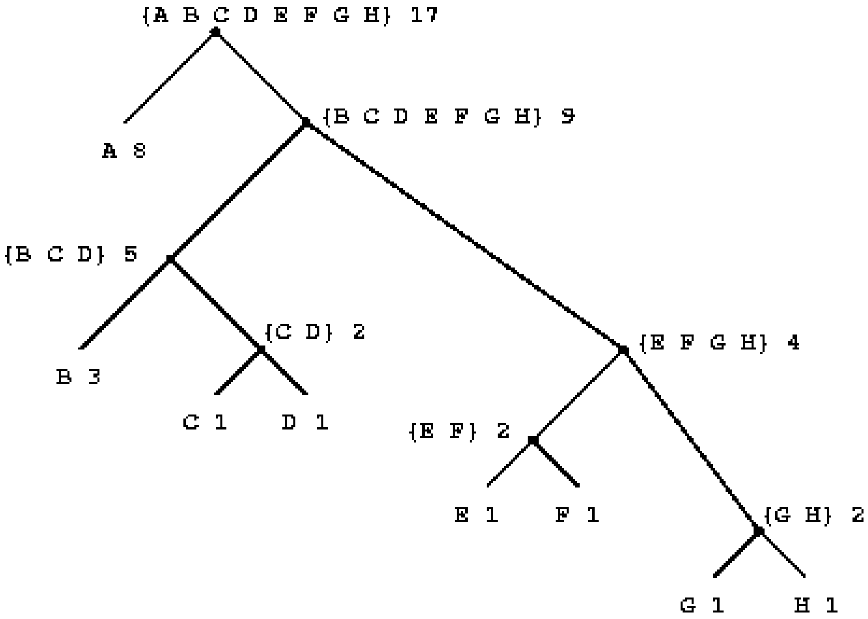


字符的编码过程：

从树根开始，每次往包含该字符的子树走。往左子树走，则编码加上比特1，往右子树走，则编码加上比特0

A	0
B	100
C	1010
G	1110
H	1111

实例：哈夫曼编码树



字符串编码的解码过程：

从树根开始，在字符串编码中碰到一个0，就往左子树走，碰到1，就往右子树走。走到叶子，即解码出一个字符。然后回到树根重复前面的过程。

10001010

BAC

哈夫曼编码树的构造

基本思想：使用频率越高的字符，离树根越近。

过程：

1． 开始时，若有n个字符，则就有n个节点。每个节点的权值就是字符的频率，每个节点的字符集就是一个字符。

2． 取出权值最小的两个节点，合并为一棵子树。子树的树根的权值为两个节点的权值之和，字符集为两个节点字符集之并。在节点集合中删除取出的两个节点，加入新生成的树根。

3． 如果节点集合中只有一个节点，则建树结束。否则，goto 2

哈夫曼编码树的构造

```
Initial leaves
{(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)}Merge
{(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)}Merge
{(A 8) (B 3) ({C D} 2) ({E F G H} 4)}Merge
{(A 8) ({B C D} 5) ({E F G H} 4)} Merge
{(A 8) ({B C D E F G H} 9)}Final merge
{({A B C D E F G H} 17)}
```

哈夫曼编码树不唯一

哈夫曼编码树的构造

代码实现：

;树叶的构造函数和选择函数：

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))
(define (leaf? object)
  (eq? (car object) 'leaf))
(define (symbol-leaf x) (cadr x))
(define (weight-leaf x) (caddr x))
```

哈夫曼编码树的构造

```
;树的构造函数和选择函数:
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right))))

(define (left-branch tree) (car tree))
(define (right-branch tree) (cadr tree))
(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
```

哈夫曼编码树的解码过程

```
(define (decode bits tree) ;bits是字符串的编码01串
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch
               (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch)))))
  (decode-1 bits tree))

(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "bad bit -- CHOOSE-BRANCH" bit))))
```

哈夫曼编码树构造初始叶子集合

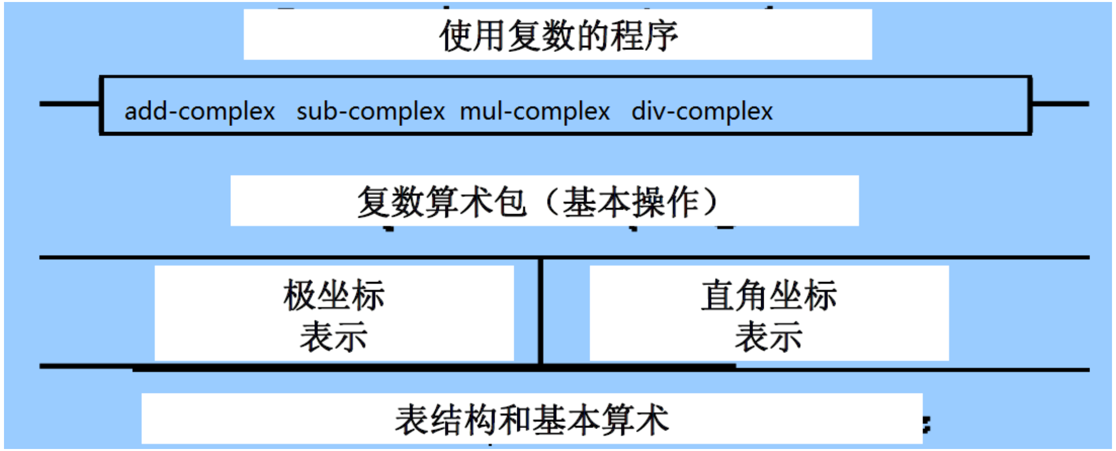
```
(define (adjoin-set x set) ;将x加入有序的set,并保持从小到大的顺序
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))))))


(define (make-leaf-set pairs);从pairs构建Haffman树的初始叶子集合
;pairs 形如: ((A 4) (B 2) (C 1) (D 1))
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set (make-leaf (car pair)     ; symbol
                               (cadr pair))   ; frequency
                    (make-leaf-set (cdr pairs))))))
```

数据抽象的多重表示

有时，在一个程序里，一种数据可能有多种表示方式。但是希望，使用复数的程序，在对复数进行操作的的时候，不必关心复数到底使用哪种方式表示的。

例如，复数有直角坐标和极坐标两种表示方式，但是复数的加减乘除这些操作，只要写一套，就应能用于两种表示方式。

做到这一点的关键是复数的两种的表示方式应对外提供一致的接口。



数据抽象的多重表示

- 复数的直角坐标表示法:

```
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z) ;求模
  (sqrt (+ (square (real-part z)) (square (imag-part z)))))
(define (angle z) ;求极角
  (atan (imag-part z) (real-part z)))
(define (make-from-real-imag x y) (cons x y))
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))
```

数据抽象的多重表示

●复数的极坐标表示法：

```
(define (real-part z)
  (* (magnitude z) (cos (angle z))))
(define (imag-part z)
  (* (magnitude z) (sin (angle z))))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
(define (make-from-mag-ang r a) (cons r a))
```

数据抽象的多重表示

- 复数的操作：

```
(define (add-complex z1 z2) ;直角坐标式
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
(define (sub-complex z1 z2) ;直角坐标式
  (make-from-real-imag (- (real-part z1) (real-part z2))
                       (- (imag-part z1) (imag-part z2))))
(define (mul-complex z1 z2) ;极坐标式
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                     (+ (angle z1) (angle z2))))
(define (div-complex z1 z2) ;极坐标式
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                     (- (angle z1) (angle z2))))
```

带标志数据的多重表示

●如果要让前面的 add-complex, sub-complex, mul-complex ,div-complex对两种形式的复数都能工作，则需要往复数的表示形式中添加标记，以便使用到一个复数的时候，通过标记可以知道它是哪种表示形式。此时，复数是一个嵌套对子。

例：('rectangular . (34 . 56)) ('polar . (45 . 90))

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error “Bad tagged datum -- TYPE-TAG” datum)))
(define (contents datum) ;求复数的实部虚部，或极角和模
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum -- CONTENTS" datum)))
```

带标志数据的多重表示

```
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
(define (polar? z)
  (eq? (type-tag z) 'polar))
```

- 直角坐标表示法:

```
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
           (square (imag-part-rectangular z)))))
(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))
```

带标志数据的多重表示

```
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
              (cons (* r (cos a)) (* r (sin a)))))
```

复数结构：`('rectangular .  (10 . 20))`

带标志数据的多重表示

- 极坐标表示法：

```
(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z))))
(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z))))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
              (cons (sqrt (+ (square x) (square y)))
                    (atan y x))))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))
```

复数结构： `('polar . (10 . 20))`

带标志数据的多重表示

●通用接口：

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type -- REAL-PART" z))))
(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Unknown type -- IMAG-PART" z))))
```

带标志数据的多重表示

- 通用接口：

```
(define (magnitude z)
   (cond ((rectangular? z)
           (magnitude-rectangular (contents z)))
          ((polar? z)
           (magnitude-polar (contents z)))
          (else (error "Unknown type -- MAGNITUDE" z))))
(define (angle z)
   (cond ((rectangular? z)
           (angle-rectangular (contents z)))
          ((polar? z)
           (angle-polar (contents z)))
          (else (error "Unknown type -- ANGLE" z))))
```

带标志数据的多重表示

- 构造函数:

```
(define (make-from-real-imag x y)
  (make-from-real-imag-rectangular x y))

(define (make-from-mag-ang r a)
  (make-from-mag-ang-polar r a))
```