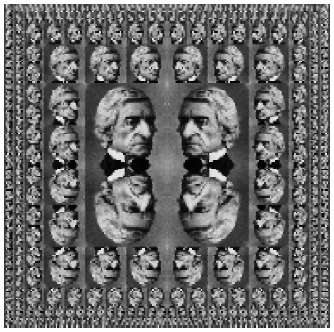
2. 构造数据抽象(2)

本节讨论

- 数据抽象和操作的实例：一个图形语言
- 符号数据（与数值数据对应）
- 符号表达式的处理（计算，符号计算）
- 抽象数据的多重表示
- 数据导向的程序设计
- 消息传递

一个图形语言

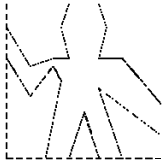
- 以一个构造图形的简单语言为例，展示数据抽象和闭包的作用和威力
  - 其中高阶过程起了关键作用
  - 主要功能：构造重复元素的图形，元素可以按规则变形变大小
- 两个这种图形的例子：

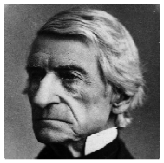



有点像 Escher 画的图（当然，远没那些画复杂、深刻）

图形语言：基本想法

- 基本元素：painter。一个 painter 画出一种特定图像
  - 可根据要求对所画图像进行操作（改变大小和变形）
  - 画出的图像依赖于具体框架。例如：


wave 画的图





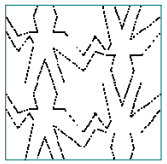
rogers 画的图

- 组合图像的若干方式：（假设定义了各种组合操作，具体实现见后）
  - beside 使用两个 painter，让它们分别在左右半区域画图
  - below 使用两个 painter，让它们分别在上下半区域画图
  - flip-vert 使用一个 painter，画出上下反转后的图
  - flip-hozil 使用一个 painter，画出左右反转后的图

图形语言：组合

- painter 的组合还是 painter，例：

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

- 可能从一个过程抽取出多个不同的模式
- 如对 wave4，可考虑将反转方式抽取出来作为过程参数
- 还可有其他考虑

- 应考虑 painter 的重要组合模式，并将其实现为Scheme过程
- 例如，抽象出 wave4 里的模式：

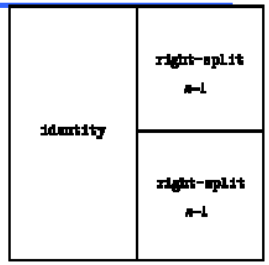
```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))

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

定义好的操作可用于任何 painter

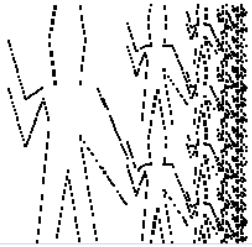
图形语言：组合

- 向右分割和分支：

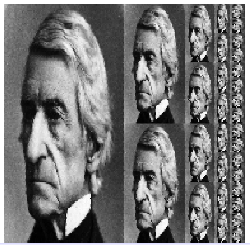
```
(define (right-split painter n)
    (if (= n 0)
        painter
        (let ((smaller (right-split painter (- n 1))))
            (beside painter (below smaller smaller)))))
```



(right-split wave 4)



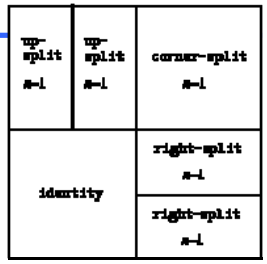
(right-split rogers 4)



图形语言：组合

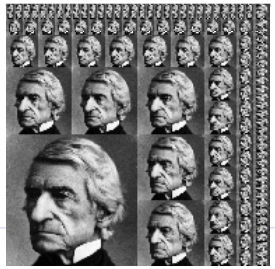
- 向右上角分割和分支：

```
(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))))))
```



up-split 与 right-split 类似

(corner-split wave 4)



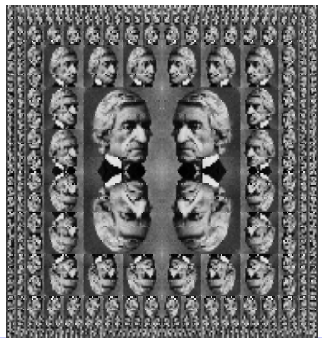
(corner-split rogers 4)

还可以定义更复杂的图形组合过程

图形语言：组合

- 把四个 corner-split 按适当方式组合，定义下面 square-limit，就可以生成本节开始的两个图形：

```
(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

- 前面通过对 painter 的组合模式进行抽象定义了几个过程。同样可以对 painter 的组合操作进行抽象定义高阶过程，它们以对 painter 的操作作为参数，产生对 painter 的新操作
- 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：

```
(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))))
```

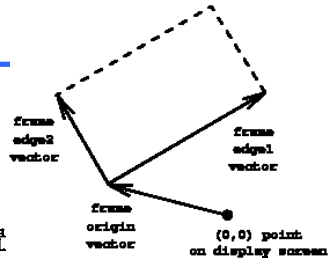
- 重新定义 flipped-pairs：

```
(define (flipped-pairs painter)
    (let  ((combine4 (square-of-four identity flip-vert
                                     identity flip-vert)))
        (combine4 painter)))
```

图形语言：框架

现在考虑 painter 本身及其技术基础

- 图像的显示框架可以用3个向量表示：
  - 基准向量描述框架基准点的位置
  - 两个角向量描述两个相邻角的相对位置



- 设有框架构造函数 make-frame，选择函数 original-frame, edge1-frame 和 edge2-frame

需要一个映射由给定 frame 生成一个过程，它由向量参数生成所需向量：

```
(define (frame-coord-map frame)
    (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))))))
```

- 平面向量 v 有两个分量
- 生成过程用 frame 变换给定向量（移基点+两方向的比例变换）

图形语言：painter

- 每个 painter 都是一个过程，它以一个框架为参数，通过适当的位移和放大缩小，将要画的图形嵌入由参数给定的框架里
- 基本 painter 的实现依赖具体图形系统和被画图像的种类。例如，假定有画直线的基本过程 draw-line，用线段表来表示折线图形中的折线，就可以用下面过程画出各种折线图：

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
      (lambda (segment)
        (draw-line
          ((frame-coord-map frame) (start-segment segment))
          ((frame-coord-map frame) (end-segment segment))))
      segment-list)))
```

实现某种线段表表示（也是数据抽象），给出 wave 图形的线段表，就可以用 segments->painter 定义出 wave

图形语言：painter，变换和组合

- 用过程表示 painter，在图形语言里建立了良好的抽象屏障
  - 容易创建基本 painter，容易通过组合构造复杂的 painter
  - 任何以框架为参数，基于它画图的过程都可作为 painter
- 对 painter 的操作都是创建新 painter，如 flip-vert 和 beside，其中用到作为参数的 painter，还涉及框架变换
- 对 painter 的操作都基于 transform-painter 定义。它以一个 painter 和变换框架的信息为参数，基于变换后的框架调用原 painter。框架变换信息用三个向量描述，分别表示新基准点和两个边向量的终点

```
(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)
                        (sub-vect (m corner2) new-origin)))))))
```


图形语言：变换和组合

- 各种 painter 变换都可以基于过程 transform-painter 定义
- 纵向反转 flip-vert ：

```
(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
```

- 将框架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)))
```

- 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)))
```

图形语言：变换和组合

- 将图像向中心收缩：

```
(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)))
```

- beside 以两个 painter 为参数：

```
(define (beside painter1 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)))))
```


强健设计和语言分层

- 总结：图形语言里的 painter 和基本数据抽象都用过程表示。这样能
  - 以统一方式处理各种本质上完全不同的基本画图功能
  - 组合方式有闭包性质，已有 painter 的组合仍然是 painter
  - 所有过程抽象手段都可以用于组合各种 painter
- 复杂系统应该通过分层设计完成
  - 描述这些层次需要一系列语言
  - 通过组合一个层次中的各种基本元素，得到另一更高层次的元素
  - 每层提供基本元素、组合手段和抽象手段，支持更高层次的构造
- 在图形语言实例中：
  - 基本语言提供基本图形功能，如为 segmant->painter 提供画线段功能，为 rogers 提供画图和着色功能
  - 一组基本 painter 提供基本图形，beside 和 below 等操作 painter
  - 在此之上实现图形操作的组合，以 beside 和 below 等作为操作对象

符号数据和符号处理

- 早期计算机只用于处理数值数据，主要应用是科学和工程计算
- 随着计算机应用发展，人们看到越来越多的非数值计算问题。Lisp语言原本就是要支持非数值计算，数值计算是后加的。下面讨论 Scheme 的这方面情况，许多问题在非数值应用中有普遍意义
- 首先介绍如何把处理任意符号的功能引进 Scheme。符号计算中处理的是下面形式的表达式（符号表达式）：

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

符号表达式的形式与 Scheme 程序类似：

```
(* (+ 23 45) (+ x 9))
(define (fact n) (if (= n 1) 1 (* n (fact (- n 1)))))
```

- 为描述和处理任意符号表达式，需要有办法来说任意符号（以及符号表达式）本身，而不是说符号的值

符号数据：引号

- 自然语言中也需要区分词语本身和词语的意义，如

  我们现在把“我们”写五遍

  他把写了“桌子”的纸条贴在桌子边上

- Scheme 用类似形式描述符号对象，在表达式前加单引号，就表示这个表达式自身
- 引号不仅可用于单个符号，也可用于“组合对象”

```
(car '(a b c))
a
(cdr '(a b c))
(b c)
```

- 为实现符号操作，有基本谓词 eq?，它判断是否同一个符号。例

```
(define (memq item x)
    (cond ((null? x) false)
          ((eq? item (car x)) x)
          (else (memq item (cdr x)))))
```

```
(define a 1)
(define b 2)
(list a b)
(1 2)
(list 'a 'b)
(a b)
(list 'a b)
(a 2)
```

例：符号求导

- 考虑代数表达式的符号求导，得到的应是代数表达式。基本规则：

$$\frac{\mathrm{d}c}{\mathrm{d}x} = 0 \qquad c\text{是常量或与 } x \text{ 不同的变量}$$

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

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

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

- 易见：
  - 需要根据（子）表达式的形式确定适用的求导规则
  - 后两条是递归，分解了表达式，最终将达到基础情况
- 为实现求导，需要（设计和实现数据抽象）
  - 一种代数表达式的表示方式，以它作为数据抽象
  - 一组构造函数和选择函数，包括判断表达式种类的谓词

符号求导：数据抽象

- 假定有如下构造函数、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.

- 基于这些过程，按照求导规则，不难写出完成求导的过程

符号求导：过程定义

```
(define (deriv exp var)
    (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))))
```

- 基本结构是一个 cond 表达式，其中根据被求导代数式的各种情况分别构造出结果代数式

符号求导：代数式的表示

- 代数式可用各种合理方式表示（是一种数据抽象），最简单的方式是直接用符号表示变量，用类似 Scheme 程序的前缀形式表示代数式。例如将 ax + b 表示为 (+ (* a x) b)
- 代数式的构造函数、选择函数和谓词都很容易定义：

```
(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))
```

与乘式有关的几个过程的定义与和式的相应过程类似

符号求导：试验和改进

- 下面是一个使用实例，结果正确，但易见其中代数式没有化简：

```
(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0))
   (* (+ (* x 0) (* 1 y))
      (+  x 3)))
```

- 实现化简不必修改 deriv，只需修改和式和乘式的构造函数：

```
(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))))
```

=number? 判断表达式是不是数

实例：集合

- 前面构造复合数据对象时，表示方式很明显（明确），也存在一些细节问题。如有理数是否总维持最简形式，就是一种设计选择
- 对一种复合对象，往往存在多种不同表示方式，不同表示方式在许多方面表现出不同性质。其中的选择可能不是直截了当的
- 下面以集合为例讨论这方面问题。集合是一些对象的汇集，关键特征是经常作为整体考虑，支持一组集合操作，包括：
  - union-set　　求两个集合的并集
  - intersection-set　　求两个集合的交集
  - element-of-set?　　判断对象是否属于集合
  - adjoin-set　　结果是参数集合加上新加入的元素
  - 等等
- 从数据抽象的观点看，在表示集合的方式选择上有很大自由度，可以用任何合理且易用的方式表示集合

集合：用任意的表表示

- 最简单的想法是直接用表表示集合，元素唯一出现，空集对应空表
- 判断元素是否在集合中（equal? 可用于判断任何对象是否相等）：

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))
```

- 加入元素时需要考虑被加入元素是否已经在集合里：

```
(define (adjoin-set x set)
  (if (element-of-set? x set) set  (cons x set)))
```

- 求交集：

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1) (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```


集合：用排序的表表示

- 选择表示的一个重要根据是操作效率。简单地用表来表示集合，判断成员需要扫描整个表，是 O(n) 操作；加入元素需判断存在性，O(n)；求交集是 O(n*m) 操作（n, m 是两集合元素个数）
- 提高效率的一种可能是改变表示。现考虑用排序表，元素按上升序排列
  - 要求存入表中的元素能比较大小，假定可用 < 和 > 比较
  - 这些也使集合表示有了更多要求，(3 4 1 2) 不再是合法集合
- 采用排序的表，判断元素平均只需检查一半元素：

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)))))
```

  遇到更大的元素就不需要再继续比较了
- 但完成一个元素的处理需要做三次比较，单位开销增加了

集合：用排序的表表示

- 由于元素是排序的，求交集操作的效率将有本质提高
  - 比较两个集合的最小元素，相等则留下
  - 否则丢掉两者中较小的一个，并递归检查
- 过程的实现：

```
(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)))))))
```

操作代价由 O(n*m) 减到 O(n+m)：

每次递归，两个参数表至少减少一个元素。改进是本质性的

集合的一些问题

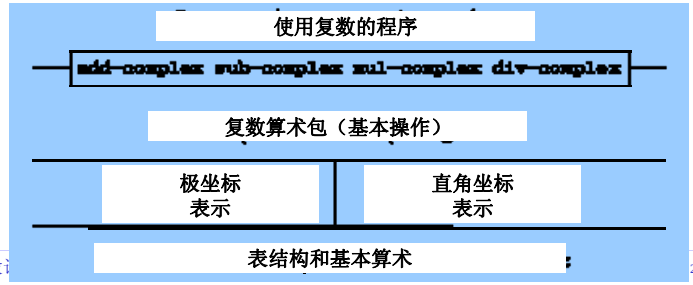
- 用二叉树表示集合（略）
- 集合与检索（略）
- Huffman 编码树（略）

抽象数据的多重表示

- 数据抽象可使程序中的大部分描述与数据对象具体表示无关。方法：
  - 用一组基本操作构筑起抽象屏障（构造函数，选择函数等）
  - 在屏障之外只通过这组基本操作使用数据抽象
  - 通过数据抽象可把大系统分解为一组易于处理的小任务
- 但，数据对象有明确的“基本表示”形式吗？
  - 许多数据对象可以有多种合理的表示形式
  - 各种表示常常互有长短，有时可能希望系统里同时存在多种表示
- 例：复数有极坐标表示和直角坐标表示。有些操作在某种表示下更容易处理。允许两种或多种表示在系统里同时存在是有意义的
  - 复杂系统常由许多人共同完成，可能使用第三方开发的库
  - 同一种数据对象存在多种不同表示的情况不可避免。因此，需要组合已有模块的有效技术，而不是重新设计和实现

抽象数据的多重表示

- 下面研究在一个程序里支持同一种数据的多种表示形式的技术
- 主要研究如何构造通用型操作（可以在不同数据表示上操作的过程）
  - 这里采用的技术是让数据带上特殊标志
  - 通用型（泛型）过程通过检查标志确定如何完成所需操作
- 最后还要讨论“数据导向”（数据驱动）的程序设计，这是一种可用于实现通用型操作的威力强大而且方便易用的技术
- 下面用复数作为例子，构造出的复数系统具有下面结构：



复数的表示

- 复数有两种基本表示方式：
  - 直角坐标表示，将复数表示为实部和虚部，加法很简单：
    $\mathrm{re}(z_1+z_2)=\mathrm{re}(z_1)+\mathrm{re}(z_2)\quad \mathrm{im}(z_1+z_2)=\mathrm{im}(z_1)+\mathrm{im}(z_2)$
  - 极坐标表示，将复数表示为模和幅角，乘法很简单
    $\mathrm{mg}(z_1\cdot z_2)=\mathrm{mg}(z_1)\cdot\mathrm{mg}(z_2)\quad \mathrm{an}(z_1\cdot z_2)=\mathrm{an}(z_1)+\mathrm{an}(z_2)$
- 从开发的角度看，数据抽象只需支持使用复数的各种基本操作，下层采用哪种基础表示并不重要。即使实际上用直角坐标表示，也可以取它的模；用极坐标表示也可以取其实部
- 实现复数包时，用4个选择函数和2个构造函数屏蔽复数的具体表示：
  - 选择函数：real-part, imag-part, magnitude, angle
  - 构造函数：make-from-real-imag 和 make-from-mag-ang

复数运算

- 所有运算都基于基本过程实现，其中的加减运算基于实部和虚部，乘除运算基于模和幅角：

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

- 下面考虑复数的实现。两种具体表示（直角坐标和极坐标）都可用，不同开发者可能做出不同选择

复数的直角坐标和极坐标实现：

- 用序对表示复数，car 和 cdr 分别表示其实部和虚部。基本过程：

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

- 用序对表示复数，car 和 cdr 分别表示其模和幅角。基本过程：

```
(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))
```

对于这两种实现，已实现的复数运算都可以正常工作

带标志数据和多重表示

- 数据抽象支持一种“最小允诺原则”。由于有抽象屏障，实际表示形式的选择时机可以尽量后延，以保证系统设计的最大灵活性
- 实际上，如果需要，设计好构造函数和选择函数后，还可决定同时使用多种不同表示方式，将表示方式的不确定性延续到运行时
- 考虑在一个复数系统里同时允许两种表示形式，为此，选择过程要有办法识别不同表示。解决方法是为数据加标签（自表示数据）。我们给复数加标签 rectangular 或 polar
- 加标签数据抽象（另一层）：选择函数 type-tag 和 contents 取标签和实际数据，构造函数 attach-tag 做出带标签数据：

```
(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)
  (if (pair? datum) (cdr datum)
      (error "Bad tagged datum -- CONTENTS" datum)))
```

带标志数据和多重表示

- 定义判别谓词，确定被处理数据的具体表示类型：

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

- 为支持加标签数据，两种实际表示的实现都需要修改：
  - 采用不同的过程名，以相互区分
  - 给做出的复数加上类型标签

带标志复数：直角坐标表示

- 直角坐标表示的复数的构造函数和选择函数：

```
(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)))
(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)))))
```

这里要改过程的名字，以免相互冲突（这一缺点后面还要讨论）

带标志复数：极坐标表示

- 极坐标表示的复数的构造函数和选择函数：

```
(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)))
```

- 选择函数需重定义为通用型的过程，它们要检查数据的类型标签（数据的类型），根据标签决定怎样操作

带标志复数：通用型选择函数

```
(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))))
(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))))
```

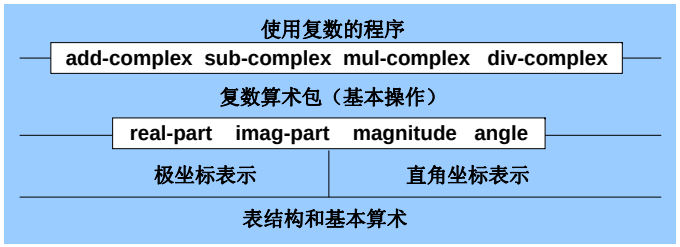
在这些操作之上，实现复数算术的过程都不必修改

带标志复数：构造函数

- 最后的问题是如何构造。一种合理方法是参数为实部和虚部时采用直角坐标表示，模和幅角时用极坐标表示：

```
(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))
```

得到的系统：



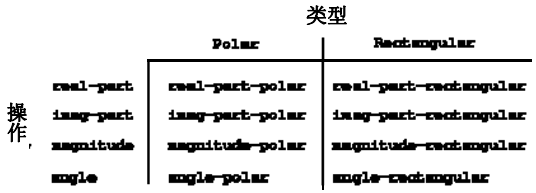
- 注意这里的分层抽象和抽象之间的接口
- 通用型过程识别数据的具体类型，剥离数据标签后传给实际处理过程

前面技术的弱点

- 检查数据的类型并根据类型调用适当过程的操作称为基于类型的指派。这是一种获得系统模块性的强有力技术
- 而用上面技术实现基于类型的指派有两个重要弱点：
  - 通用型过程（上面选择函数）必须知道所有类型。增加一个新类型时需要选一种新标签，并在每个通用型过程里加一个新分支
  - 即使相互独立的表示可以分别定义，也必须采用不同名字，或者在集成时修改名字（这时需要修改程序）
- 这两个弱点说明这种技术不具有可加性（人们常谈 scalability）
  - 增加新类型要求修改每个通用型过程的代码
  - 集成时要修改接口过程的名字避免冲突
- 修改可能很麻烦，容易引进新错误。大型系统里可能有成百不同类型和表示方式，增加一个新类型的工作量巨大。如果没有了解所有程序的程序员，或程序里用了没有源代码的库，事情将变得更困难

数据导向（数据驱动）的程序设计

- 支持系统进一步模块化的一种技术称为数据导向的程序设计。注意，处理针对不同类型的一批通用操作，实际上是处理一个二维表格：



- 数据导向的程序设计直接处理这种二维表格
  - 前面用一集过程作为接口，让它们检查数据类型并显式指派
  - 数据导向技术直接把所需接口实现为一个过程，让它用操作名和类型的组合去查找这个表格，找出所需过程并应用之。增加一种新类型就只需要在表格里增加一组新项，不修改已有程序
- 下面的实现基于两个基本表格操作，现假定语言提供了这两个操作，它们的实现在第3章考虑（需要做改变状态的程序设计）

数据导向的程序设计

- 表格的基本操作 put 和 get：

  put 将一个项 *<item>* 加入表格，使之与 *<op>* 和 *<type>* 关联

  (put *<op>* *<type>* *<item>*)

  get 取出表格中与 *<op>* 和 *<type>* 关联的项

  (get *<op>* *<type>*)

- 用数据导向的程序设计技术实现复数系统
  - 对直角坐标实现，定义相应过程后把它们加入表格作为项，告诉系统如何处理直角坐标类型的复数
  - 对极坐标实现也同样做
  - 以这种方式建立起两种表示与系统其他部分之间的接口

数据导向的复数实现：直角坐标部分

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z)) (square (imag-part z)))))
  (define (angle z) (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a) (cons (* r (cos a)) (* r (sin a))))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

前一半定义了一批内部过程，后一半把它们安装到表格里

数据导向的复数实现：极坐标部分

```
(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z) (* (magnitude z) (cos (angle z))))
  (define (imag-part z) (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y))) (atan y x)))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

所有定义都是内部的，同名过程不会相互冲突，无须重新命名

数据导向的复数实现：通用接口过程

- 复数算术运算的实现基础是一个通用选择过程，它用操作名到表格里查找具体操作

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types -- APPLY-GENERIC"
            (list op type-tags))))))
```

- 选择函数都基于这一通用选择过程定义：

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```


数据导向的复数实现：通用接口过程

- 构造函数也通过提取表中的操作实现：

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

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

- 要增加一种新的复数类型，只需
  - 再写一个实现该类型的 package 过程，用内部过程实现各种基本操作，并将它们安装到操作表格中
  - 实现一个外部的构造函数
  - 外部的选择函数都已经（自然地）有定义了
- 所有已有程序都不必修改

消息传递

- 做数据导向的程序设计，关键想法就是明确处理 “操作-类型”表格，管理程序中各种通用操作。处理同一问题的另外两种方式：
  - 前面的方式是把操作定义得足够聪明，它们能根据数据类型决定具体操作，相当于将表格横向切分，每行实现为一个“智能操作”
  - 另一可能性是将表格纵向切分，定义“智能数据对象”，使它们能根据操作名决定要做的工作，下面讨论这种技术
- 这里的技术是把对象表示为过程，使之对具体操作名能完成所需工作。例如 make-from-real-imag 产生直角坐标对象：

```
(define (make-from-real-imag x y)
    (lambda (op)
      (cond ((eq? op 'real-part) x)
            ((eq? op 'imag-part) y)
            ((eq? op 'magnitude) (sqrt (+ (square x) (square y))))
            ((eq? op 'angle) (atan y x))
            (else
             (error "Unknown op--MAKE-FROM-REAL-IMAG" op)))))
```


消息传递

- make-from-mag-ang 的定义与 make-from-real-imag 类似
- 与之对应的 apply-generic 应该把操作名送给相应的数据对象，将它重新定义为下面形式，

  (define (apply-generic op arg) (arg op))

  其他操作都不必修改了（包括基于 apply-generic 定义的选择函数）

  创建对象的操作返回的过程就是 apply-generic 调用的过程
- 这种风格的程序设计称为消息传递：将一个数据对象想象为一个实体，它能接受消息并根据具体的消息决定完成某些工作
- 前面介绍过用过程实现序对，现在看到这种技术的实用性。下章会关注基于消息传递的程序设计。下面将继续讨论数据导向的程序设计
- 有兴趣的同学可以将本节课讨论的问题与面向对象程序设计中的相关问题做一些比较