2. 构造数据抽象(2)

本节进一步讨论数据抽象，包括

- 一个图形语言

  如何应用数据抽象的概念建立图形语言

  使之能很好支持各种复杂的抽象和组合

- 符号数据（与数值数据对应）

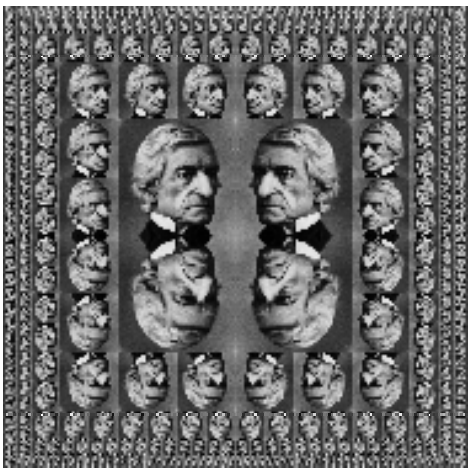
  如何表示符号数据

  符号表达式处理（符号计算，**symbolic computation**）

- 抽象数据的多重表示

一个图形语言

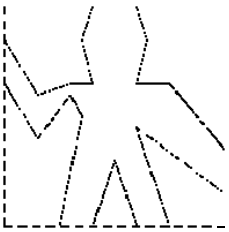
- 以构造一个简单图形语言为例，展示数据抽象和闭包的作用和威力
  - 其中高阶过程起着关键作用
  - 功能：构造重复元素的图形，元素可以按规则改变形状/大小
- 两个这种图形的例子：



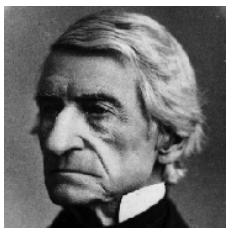
具有 **Escher** 一些画的风格，当然远没有 **Escher** 的画复杂和深刻

图形语言：基本想法

- 基本元素：**painter**。一个 **painter** 的功能是画一种特定图像，它
  - 可以根据要求对所画图像进行变形（改变形状和大小）
  - 图像的变形基于给定的具体框架：



wave 画的图



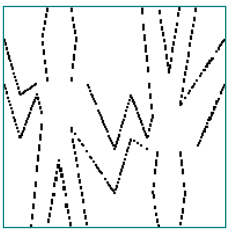
rogers 画的图

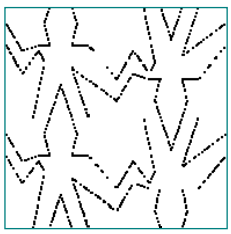
- 图像组合操作：（现假设定义了几种组合操作，具体实现见后）
  - **beside** 使用两个 **painter**，让它们分别在左右两个半区画图
  - **below** 使用两个 **painter**，让它们分别在上下两个半区画图
  - **flip-vert** 使用一个 **painter**，画出上下反转后的图
  - **flip-hozil** 使用一个 **painter**，画出左右反转后的图

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))
```





- 从一个过程中可能抽取出多个不同模式
- 如 **wave4**，也可以把反转方式抽出来作为参数
- 可以有其他考虑

- 可以考虑 **painter** 组合的重要模式，将其实现为 **Scheme** 过程
- 例如，抽象出 **wave4** 里的模式，定义为对图形的操作：

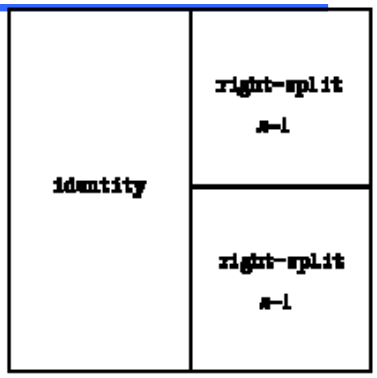
```
(define (flipped-pairs painter)
  (let ((painter2 (beside painter (flip-vert painter))))
    (below painter2 painter2)))

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

这样定义的操作可以用于任何 **painter**

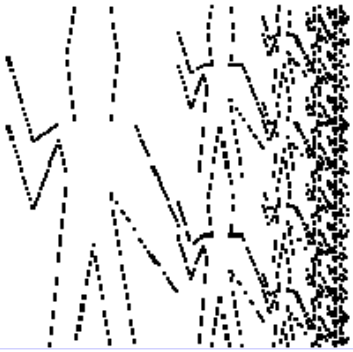
图形语言：组合

■ 组合操作：向右分割和分支：

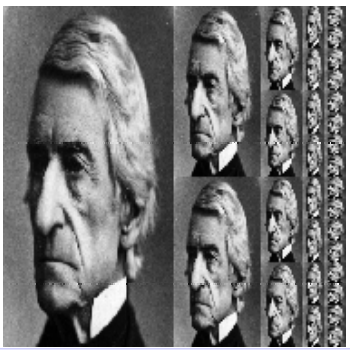
```
(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))
```



(right-split wave 4)



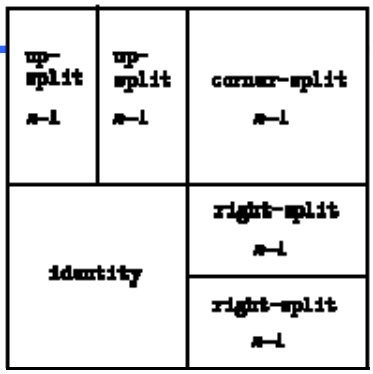
(right-split rogers 4)



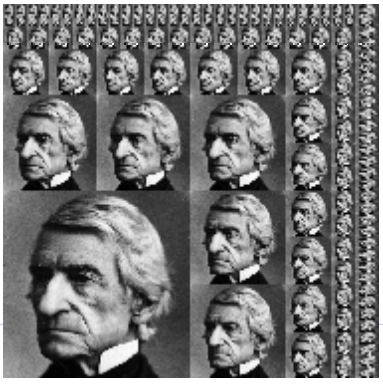
图形语言：组合

■组合操作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)

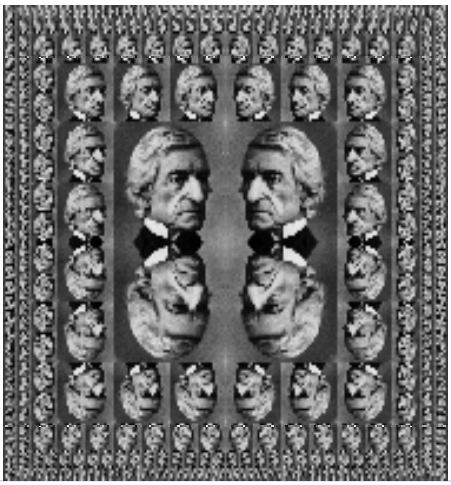
(corner-split rogers 4)

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，就可以定义出 **square-limit**。用它可以生成本节开始展示的两个图形：

```
(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

- 前面对 **painter** 的组合模式进行抽象定义了几个过程

  同样可以对组合操作进行抽象定义各种更高阶的过程

  高阶操作以对 **painter** 的操作作为参数，生成新的 **painter** 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)))
```

其中的 **identity** 是“恒等变换”，直接返回参数

或直接定义

```
(define flipped-pairs
  (combine4 identity flip-vert identity flip-vert) )
```

- 请考虑如何借助 **combine-4** 定义前面的各图形操作

  也可以直接用 **lambda** 构造所需操作

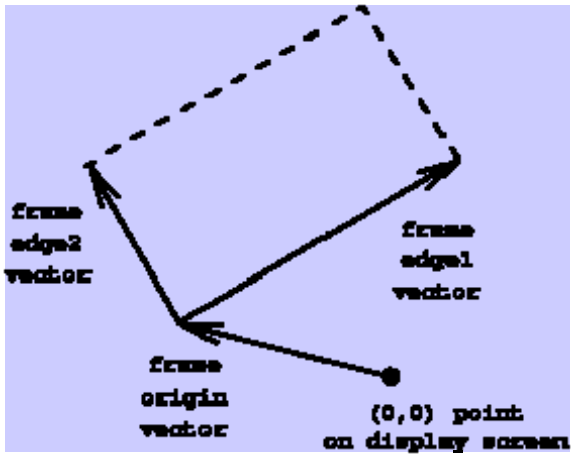
  包括考虑为利用 **conbine-4** 而需要定义的基本操作

- 问题：前面 **4** 个基本操作，对各种变形而言是否足够？（理论问题）

图形语言：框架

考虑 **painter** 的技术基础

- 图像显示基于框架（**frame**）概念，框架用**3**个向量表示：
  - 一个基准向量，作为框架的基点向量
  - 两个角向量描述框架两角顶点相对于基点的位置



- 框架可用于描述单位图形的变换
  - 基点确定图形的平移
  - 两个角向量确定图形的旋转和伸缩
- 把框架 **frame** 作为一种数据抽象，假设有
  - 构造函数 **make-frame**
  - 选择函数 **original-frame, edge1-frame** 和 **edge2-frame**

图形语言：框架

- 用单位正方形 0 ⩽ x, y ⩽1 里的坐标描述图像

  图像的原点总是 (0, 0)，对角点总是 (1, 1), 中点总是 (0.5, 0.5)

- 每个框架关联着一个框架坐标映射
  - 该映射描述如何对图像进行位移和伸缩以适应这一框架
  - 框架映射将图像的每个点映射到框架里的一个点
- 例如，框架 f 描述的图像变换，将图像的点 (x, y) 变换到

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

  易见，被变换图像在变换之后：

  其 (0, 0) 点总位于框架的原点

  其 (1, 1) 点总位于框架原点得到对角点

  其中点总位于框架的中点

图形语言：基本框架

- 下面高阶过程生成对应于参数 frame 的变换过程：

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

- frame-coord-map 生成的过程用给定 frame 去变换其参数向量
  - 向量 v （也是数据抽象）有两个分量，选择函数 (xcor-vect v) 和 (ycor-vect v) 选择其分量，构造函数 (make-vect xcor ycor)
  - 生成的过程用 frame 变换参数向量，包括

    基点移动和两角方向的旋转和比例变换

图形语言：基本框架

- 两个常用的向量操作

```
(define (add-vect v1 v2)
  (make-vect (+ (xcor-vect v1) (xcor-vect v2))
             (+ (ycor-vect v1) (ycor-vect v2)) ) )

(define (scale-vect v factor)
  (make-vect (* factor (xcor-vect v))
             (* factor (ycor-vect v)) ) )
```

- 其他向量操作可以类似定义。如

```
(define (sub-vect v1 v2) (add-vect v1 (scale-vect v2 –1)))

(define (rotate-vect v angle)
   (let ( (c (cos angle)) (s (sin angle)) )
     (make-vect
        (– (* c (xcor-vect v)) (* s (ycor-vect v)))
        (+ (* c (ycor-vect v)) (* s (xcor-vect v)))  )))
```


图形语言：painter

- 每个 **painter** 都是一个过程，以一个框架为参数，根据该框架对自己的图形做位移、旋转和缩放，使之恰好嵌入该框架

  需要一组基本 **painter**，其实现依赖具体图形系统和所需图形类

- 例：假定已有画直线的基本过程 **draw-line**，折线图形中的折线用线段的表表示，下面是一个画折线图的 **painter** 过程：

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each ; 类似 map，对表元素应用过程，但不构造结果表
      (lambda (segment)
        (draw-line
          ((frame-coord-map frame) (start-segment segment))
          ((frame-coord-map frame) (end-segment segment))))
      segment-list) ) )
```

如果实现了线段（也是数据抽象）的表示，并且有了表示 **wave** 图形的线段表，就可以用 **segments->painter** 画出 **wave**

图形语言：painter，变换和组合

- 用过程表示 painter，建立了良好的抽象屏障
  - 任何以框架为参数，能基于它画图的过程都可作为 painter
  - 很容易创建各种基本 painter，也容易通过组合构造复杂 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
- painter 操作基于 transform-painter 定义。该高阶过程以 painter 和框架变换信息为参数，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 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)))))
```

- 有了上面功能强大的基础结构，继续开发可以做出一个丰富的图形系统

语言设计和分层抽象

- 示例总结：语言里的 **painter** 和基本数据抽象都用过程表示。支持
  - 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 **painter** 生成新 **painter**
- 启示：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
  - 通过组合一层次的各种基本元素，得到更高层次的元素
  - 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
  - 基本 **painter** 提供基本图形，**beside** 和 **below** 等操作 **painter**
  - 还实现了图形操作的组合，以 **beside** 和 **below** 等为操作对象

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** 在符号处理方面的情况
  - 讨论的许多问题在各种非数值应用领域里有普遍意义
- 首先考虑如何把处理任意符号表达式的功能引进 **Scheme**。符号计算中处理的对象是表达式（符号表达式），形如：

  (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（表达式）类似：

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

符号数据和符号处理

- 为描述和处理符号表达式，需要有办法写符号（以及符号表达式）本身，而不是说符号（符号表达式）的值
- 自然语言里也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

  我说的是“我不说了”
- **Scheme** 用类似形式描述符号对象

  表达式前加单引号，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)

例：

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

符号数据：eq?

- 基本谓词 **eq?** 判断给它的两个参数是否同一符号（与相等不同，后面讨论）。例

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

- 作为实例，下面考虑一个符号求导程序
  - 符号求导是从一个代数表达式计算出另一代数表达式
  - 代数表达式用 **Scheme** 的符号表达式表示
  - 符号求导程序的工作就是从一个符号表达式算出另一符号表达式
- 符号求导是典型的符号计算
  - 这是最早研究的符号计算
  - 每个支持符号计算的数学软件都提供符号求导功能

例：符号求导

- 符号求导的基本规则（数学）：

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

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

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

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

- 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，对表达式进行分解，最终将达到基础情况
- 要想实现求导，需要确定（设计和实现数据抽象）
  - 一种代数表达式的表示方式，下面将它作为数据抽象
  - 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

- 现在还是按建立数据抽象的标准方式工作

  先设计一批构造函数和选择函数（访问函数）

- 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

  每个分支处理被求导代数式的一种（结构）情况

  做法：识别结构类型，基于原表达式的成分构造结果表达式

符号求导：代数式的表示

- 代数式可用各种合理的方式表示（是数据抽象），最简单方式
  - 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，ax + b 表示为 (+ (* a x) b)
- 代数式的构造函数、选择函数和谓词：

```
(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

- 使用实例（结果正确，也很容易看到这里的代数式没化简）：

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

- 实现化简功能不需要修改 deriv，只需修改和式和乘式的构造函数：

=number? 检查是数而且相等

```
(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

- 复合数据对象该用什么表示形式，有时很明显（如前面一些例子）

  但也存在一些细节

  如有理数是否总维持最简形式，就是一种设计选择

- 一般而言，复杂复合对象往往有多种不同表示方式，它们在许多方面表现出不同性质。不一定容易选择
- 下面以集合为例讨论这方面问题。集合是一些对象的汇集，关键特征是经常被作为整体考虑和处理，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?** 判断是否集合的元素
  - **adjoin-set** 结果是参数集合加上新加入的元素
  - 等等

集合

- 集合是一组对象的无序汇集
  - 显然，可以考虑作为一种数据抽象
  - 集合只要求实现相关操作，对表示方式没有限制
  - 集合操作也没有对实现方式提出特别的倾向
  - 实际实现方式的选择有很大灵活性，可以基于实际需要考虑，可以用任何合理、有效、易编程的方式表示集合
- 最简单的想法是直接用表表示集合，空表对应空集

  是否要求元素唯一出现？这是一种设计选择

- 操作：判断元素是否在集合中（**equal?** 判断两个任意对象是否相等）

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

集合：用任意的表表示

- 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))))
```

- 其他操作的实现类似，都很简单
- 选择表示的一个重要根据是操作效率。用简单的表表示集合：

  判断成员时需要扫描整个表，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）

  如果允许元素重复，操作效率就会不同

集合：用排序的表表示

- 提高效率的一种可能是改变表示
- 考虑用排序的表表示集合，元素按上升序排列
  - 要求存入表中的元素必须能比较大小，假定可以用 **<** 和 **>** 比较
  - 这也使集合表示有了更多要求：不是任何一个表都表示集合，例如，**(3 4 1 2)** 就不是合法的集合表示
- 采用排序的表，判断元素平均只需检查一半元素（仍是**O(n)**操作）

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

  遇到更大元素就可以结束，不需要继续比较

- 但完成一个元素的处理需要做三次比较，单位开销增加。实际效率是否改善并不清楚，需要进一步深入分析

集合：用排序的表表示

- 由于元素排序，求交集操作的效率有本质性的提高
  - 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。改进是显著而清晰的

集合的一些问题

- 集合的并操作与交操作类似

  实现类似，效率也有显著提高
- 书上还讨论了一些问题，请大家自己阅读
  - 用二叉树表示集合（略）
  - 集合与检索（略）
  - Huffman 编码树（略）
- 不同表示带来不同的性质：
  - 销毁的存储
  - 操作效率
  - 编程的复杂程度
  - 等等

数据抽象的多重表示

- 数据抽象可以使程序中的大部分描述与数据对象的具体表示无关
- 实现数据抽象的基本方法（复习）：
  - 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更容易处理的较小的任务
- 有些直接支持数据抽象的语言里提供了专门的语言结构
  - 通过特殊语法结构组合起与一个数据抽象有关的声明定义
  - 提供专门的上下文规则，限制数据抽象内部定义的可用性，达到更好保护数据抽象的目的
- 例如（常规语言）：
  - 面向对象语言（C++, Java）中的类
  - 模块化语言里的包、模块等等（如 Ada 的 package）

数据抽象的多重表示

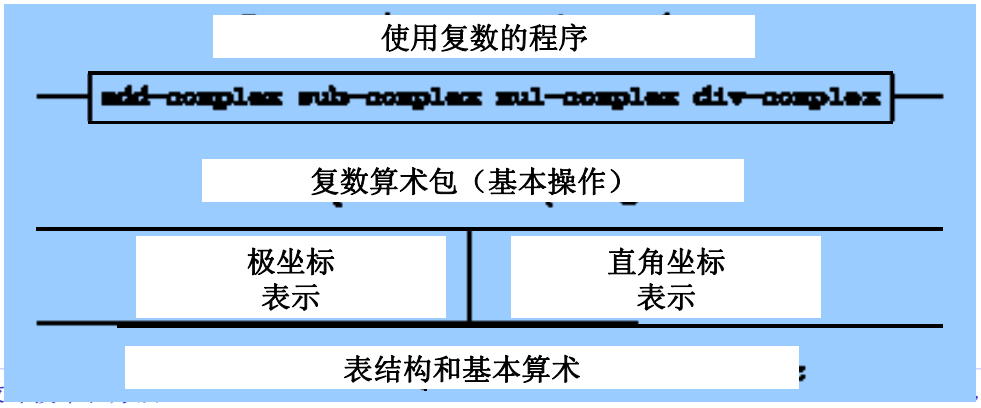
- 现在考虑数据抽象实现的一个重要方面：数据的表示问题

  要表示某种数据对象，是否有明确的“基本表示”方式？
- 未必，实际上
  - 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。一些操作在某种表示下更容易处理。系统里允许同时存在一类数据的两种或多种表示也很有意义
- 复杂系统可能由多人共同完成，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。此时需要组合已有模块的有效技术，而不是重新设计和实现
- 数据抽象的威力，不仅在于允许比较容易地在工作中改变数据表示

  还能支持在一个系统里同时存在同一类数据的多种不同表示，并很好地支持不同表示的数据之间的互操作

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
- 下节课讨论“数据导向”（数据驱动）的程序设计，它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
- 下面以复数为例。要构造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

    $\mathrm{mg}(z_1\cdot z_2)=\mathrm{mg}(z_1)\cdot\mathrm{mg}(z_2)\quad \mathrm{an}(z_1\cdot z_2)=\mathrm{an}(z_1)+\mathrm{an}(z_2)$
- 从开发和使用的角度看，数据抽象支持的是使用复数的各种基本操作
  - 实际用什么基础表示并不重要（被抽象屏蔽）
  - 即使实际用的是直角坐标表示，也完全可以取它的模；对极坐标表示的实数也可以取其实部
- 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）都可以用，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))
```

对这两种实现，已完成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“最小允诺原则”
  - 由于抽象屏障，实际表示形式的选择可以尽量延后
  - 系统设计具有最大的灵活性
- 如果实际中有需要，在设计好构造函数和选择函数之后
  - 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
- 现在考虑如何让一个复数系统里同时允许两种表示形式
  - 为支持这种功能，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）
  - 下面给给“两种”复数分别加标签 **rectangular** 或 **polar**
  - 检查标签就能确定被使用数据的表示方式，使用它们的正确方法

带标志数据和多重表示

- 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)))
```

- 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：
  - 采用不同过程名，以相互区分。构造时需要加类型标签

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)))))
```

修改过程名字是为了避免相互冲突。这是个缺点，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)))
```

- 选择函数需要重新定义为通用型的过程，它们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))))
```

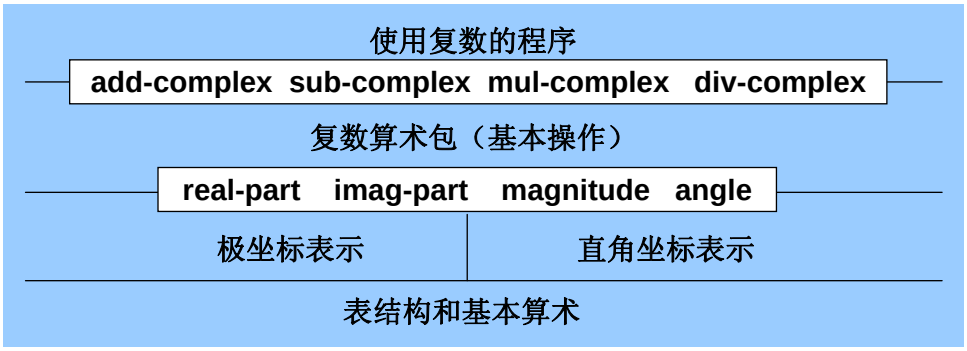
在这些操作之上，实现复数算术的过程都不必修改

带标志复数：构造函数

- 最后问题：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

在 C 语言里实现抽象数据的多重表示

- C 语言没有支持这种需要的直接结构。可以通过技术来实现，其中需要利用 union（联合）的功能（也存在其他技术）
- 数据表示的定义（注意，C 语言要求类型定义）：
  - 定义一个 enum，用一个枚举常量代表数据的一种具体表示
  - 为每种实现表示定义一个结构（如果使用的类型很简单，也可以直接使用具体类型）
  - 定义一个 struct，其中包含一个（上述 enum 的）tag 域和一个 union 域，将其定义为类型
  - 这个 union 是所用的不同表示（的结构）的联合
- 所有接口操作都基于上述类型实现
  - 在操作中检查 struct 里的 tag 域，确定使用的是什么表示
  - 构造操作建立具体的 struct，并设置相应的 tag 域
  - 所有其他操作，都基于接口操作和构造操作实现

C 语言里的多重表示：复数实例

- 以复数的两种实现为例，说明相关技术
- 定义枚举类型（只是为了程序的可读性）：

```
typedef enum {rect, polar} TComp;
```

- 定义复数类型：

```
typedef struct { double re, double im; } CRect;
typedef struct { double mag, double ang; } CPolar;
typedef struct {
   TComp tag;
   union { CRect cr; CPolar cp; } comp;
} GComp, *PGComp;  /* 定义的一般复数类型 */
```

- 应该尽可能定义好各种类型
- 下面操作采用动态存储分配的方式建立复数对象

  使用时需要仔细处理存储管理问题（这里不讨论）

C 语言里的多重表示：复数实例

- 构造操作示例（createPolar 类似）：

```
PGComp creatRect(double re, double im) {
  PGComp p = (PGComp) malloc(size(GComp));
  p ->tag = rect;
  p ->comp.cr.re = re; p->comp.cr.im = im;
  return p;
}
```

- 访问操作示例（其他操作类似）：

```
double realPart( PGComp p ) {
   switch (p->tag) {
     rect: return p->comp.cr.re;
     polar: return p->comp.cp.mag * cos(p->comp.cp.ang);
     default: /* 出错报告和处理，略 */
   }
}
```


总结

- 图形语言
  - 基于过程的图形数据抽象
  - 通过高阶过程进行图形组合和图形变换
  - 基于框架的统一设计
  - 良好的模块化，易于扩充和修改
- 符号表达式和符号处理：符号求导
  - 用嵌套的表结构表示任意的符号表达式，引号表达式
  - 基于表达式类型（首元素）的分情况处理
- 同样数据对象的多种表示：集合。效率和实现复杂性等
- 抽象数据的多重表示：复数
  - 通过数据抽象封装同样数据的多重表示
  - 带标志数据及其处理技术（分情况处理）