2. 构造数据抽象(2)

本节讨论

- 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

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)))))

符号数据和符号处理

- 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
- 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

  我写的是“我不写了”
- **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)
```

```
(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?

- 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)))))
```

- 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个代数表达式
  - 代数表达式用 **Scheme** 的符号表达式表示
  - 符号求导程序从一个符号表达式计算出另一个符号表达式
- 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

- 基本符号求导规则：

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

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

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

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

- 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

- 同样按建立数据抽象的方式工作，

  先定义一些构造函数和访问函数

- 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

  一个分支处理被求导代数式的一种（结构）情况

  识别具体结构，基于原表达式的成分构造作为结果的表达式

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

- 使用实例（结果正确，易见其中代数式没化简）：

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

- 实现化简不必修改 deriv，只需修改和式和乘式的构造函数：

=number? 检查是数且相等

```
(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

- 构造复合数据对象，有时表示方式很明显（如前面的例子）

  也存在一些细节问题

  如有理数是否总维持最简形式，是设计选择

- 一般而言，对复杂复合对象，往往存在多种不同表示方式，它们在许多方面表现出不同性质。其中的选择可能不很简单
- 下面以集合为例讨论这方面问题。集合是一些对象的汇集，关键特点是常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，对表示方式没有任何限制
  - 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、有效且易用的方式表示集合
- 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，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）

集合：用排序的表表示

- 提高效率的一种可能是改变表示
- 现考虑用排序表表示集合，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
  - 这些也使集合表示有了更多要求，并不是任何一个表都代表某个集合。例如，(3 4 1 2) 不表示合法的集合
- 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）

数据抽象的多重表示

- 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
- 实现数据抽象的基本方法：
  - 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
- 在一些直接支持数据抽象的语言里，有专门的语言结构
  - 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：
  - 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**）

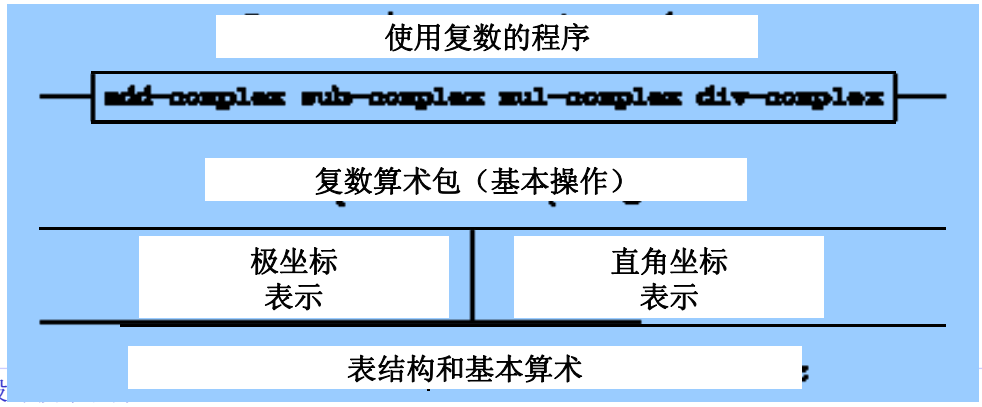
数据抽象的多重表示

- 现在考虑数据抽象实现的重要方面：数据的表示问题
- 要表示某种数据对象，是不是有明确的“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。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，需要开发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，不仅在于允许较容易地在后来改变数据表示

  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，并很好地支持它们之间的互操作

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
- 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
  - 实际用的是什么基础表示并不重要（被抽象屏蔽）
  - 即使实际上采用的是直角坐标表示，也完全可以取它的模；是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
  - 使系统设计具有最大的灵活性
- 如果实际中有需要，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）
  - 下面给给“两种”复数分别加标签 **rectangular** 或 **polar**
  - 通过检查标签，可以确定被使用的实际数据的表示方式，以及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

这一是一个缺点，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))))
```

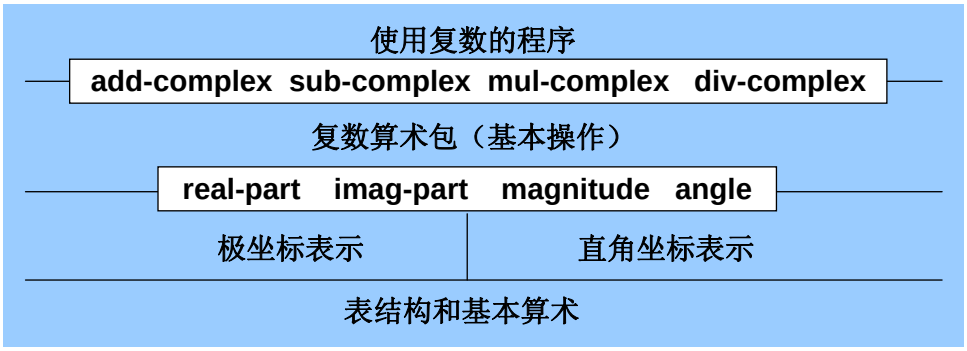
在这些操作之上，实现复数算术的过程都不必修改

带标志复数：构造函数

- 最后的问题：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，说明一下相关技术
- 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 */
    }
}
```

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

- 但采用上面技术实现基于类型的指派有两个重要弱点：
  - 每个通用型过程（如上面的选择函数）都必须知道所有的类型。增加一个新类型时，需要选择一种新标签，并在每个通用型过程里增加一个新分支
  - 即使相互独立的表示可以分别定义，其中也必须采用不同的名字，或者在集成时修改名字（需要修改程序）
- 这两个弱点说明这种技术不具有可加性（即，在已有的程序里增加新的部分是否方便。维护修改升级时常需要这样做）
  - 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
- 现在人们常谈的 **scalability**，可加性是其中的一个方面

前面技术的弱点

- 修改可能很麻烦，很容易引进新错误
  - 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
- 易见，前面在 **C** 语言里实现多重表示的技术，同样有这些问题
- 这里的问题在于：
  - 在代码里明确写出数据类型或与之有关的信息
  - 类型变化了（如增加新类型），代码就必须修改
- 要解决这种问题，就需要避免在代码里写类型
  - 下面介绍 **Scheme** 里的一种处理方法
  - 在 **C** 语言里也可以类似地实现

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，实际上就是在处理一个二维表格

操作 \ 类型	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

- 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

数据导向的程序设计

- 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）
- 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

- 用一个无参过程定义这个部分
  - 过程的前一半定义了一批内部过程
  - 后一半把它们安装到表格里

```
(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))))
```


数据导向的复数实现：直角坐标部分

```
  ;; 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)
```

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

- 实现极坐标的技术与直角坐标一样
  - 用一个无参过程创建相应的过程组
  - 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

```
(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)))
```


数据导向的复数实现：极坐标部分

```
  ;; 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)
```

- 这样就把所有的功能都安装到了表格里

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

  所有的已有程序代码都不需要修改

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：
  - 前面介绍过的的方式是把操作定义得足够聪明，使它们能根据被处理数据的类型决定具体操作，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（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

  一个数据对象是一个有强大功能的数据实体，它能接受消息并根据收到的消息确定去完成某些工作
- 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