2. 构造数据抽象(3)

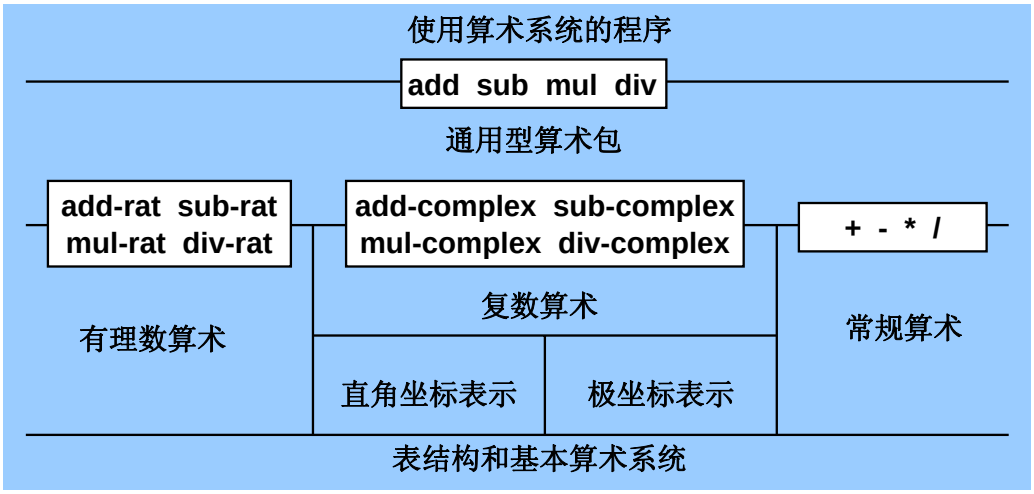
本节讨论

- 分层抽象
- 通用型算术运算
- 不同类型数据的组合
- 符号代数
- 多项式算术

包含通用型操作的系统

- 前面做了有理数算术包和复数算术包，还有系统的常规算术系统。现在考虑如何通过数据导向程序技术把这些算术系统集成到一起，研究定义处理不同参数类型的通用型操作的技术

  完成的系统应具有的结构：



- 还希望系统具有可加性，容易加入其他独立设计的算术包

通用型算术运算

- 整个系统应该只有一个通用加法
  - 对常规的数，行为如同 +
  - 对有理数如同 add-rat
  - 对复数如同 add-complex。其他算术运算也类似
- 下面采用前一节提出的技术，为每个类型加标签，让通用型运算 add 等运算根据运算对象标签完成正确的指派
- 几个通用型运算过程的定义如下：

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
```

这几个过程根据运算对象的类型从表格里提取正确操作

通用型算术运算：常规数

- 常规 Scheme 数加标签 scheme-number。每个算术运算都有两个参数，关键码用 (scheme-number scheme-number)

```
(define (install-scheme-number-package)
  (define (tag x) (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
       (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number (lambda (x) (tag x)))
  'done)
```

- 创建带标志的常规数：

```
(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
```

其他几种类型的数都按同样方式加入这个系统

通用型算术运算：有理数

- 已有的有理数功能可以直接

```
(define (install-rational-
  ;; internal procedures
  (define (numer x) (car
  (define (denom x) (cdr
  (define (make-rat n d)
  (define (add-rat x y)
    (make-rat (+ (* (nume
                 (* (denom
  ... ... ...
  ;; interface to rest of the system
  (define (tag x) (attach-ta
  (put 'add '(rational ratio
  ... ... ...
  (put 'make 'rational (lam
  'done)
(define (make-rational n
```

```
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
```

```
(put 'sub '(rational rational)
     (lambda (x y) (tag (sub-rat x y))))
(put 'mul '(rational rational)
     (lambda (x y) (tag (mul-rat x y))))
(put 'div '(rational rational)
     (lambda (x y) (tag (div-rat x y))))
```


通用型算术运算：复数

- 将类似前面定义的复数包安装到系统里，加标签 complex：

```
(define (install-complex-package)
  ;; imported procedures from rectangular and polar packages
  (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))
  ;; internal procedures
  (define (add-complex z1 z2)
    (make-from-real-imag (+ (real-part z1) (real-part z2))
                         (+ (imag-part z1) (imag-part z2))))
  ... ... ...
  ;; interface to rest of the system
  (define (tag z) (attach-tag 'complex z))
  (put 'add '(complex complex)
       (lambda (z1 z2) (tag (add-complex z1 z2))))
  ... ... ...
  (put 'make-from-real-imag 'complex
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'complex
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

操作都是内部的，包里用同样的过程名 **add, sub, mul, div** 也不会冲突

通用型算术运算：复数

- 外部可以用实部/虚部或模/幅角的方式创建复数

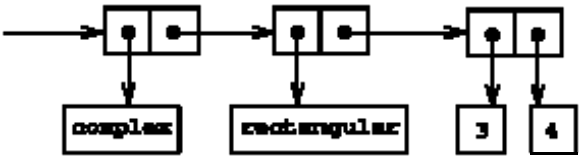
  为此需要从直角坐标和极坐标包里找出相应过程

  基于它们定义相应的全局构造函数：

```
(define (make-complex-from-real-imag x y)
  ((get 'make-from-real-imag 'complex) x y))
(define (make-complex-from-mag-ang r a)
  ((get 'make-from-mag-ang 'complex) r a))
```

- 这样出现了两重标签：外层标签标明数类（有理数/复数/常规数）。得到的复数的内层标签标明其表示方式

  如 **3 + 4i**：



- 复杂系统里可能有许多层次
  - 不同层次之间通过一组通用操作互联
  - 这些操作基于数据的标签区分不同数据类别
- 数据传递中可能跨越不同的抽象层次
  - “向下”传输时逐步剥离一层层标签，最后处理的是无标签数据
  - “向上”传输过程中一层层增加数据标签
  - 典型情况如网络传输，例如传递 **Web** 页
- 前面工作得到了一个支持多种数的计算系统
- 但它有一个重大缺点
  - 不同类型的数属于不同的独立“世界”，不同世界之间相互没有联系，也不能互操作（不能混合运算）
- 实际中常要做不同类型数据之间的操作，跨类型的运算

不同类型的数据的组合

- 前面仔细设计了程序各部分之间的清晰隔离。引进跨类型操作
  - 既要使做出的系统能支持跨类型操作
  - 又不能严重损害原有的模块分隔
- 实现跨类型操作的一种方式是为每对类型组合加一个操作。如复数与常规数求和，用标签 **(complex scheme-number)** 加一个过程：

```
;; to be included in the complex package
(define (add-complex-to-schemenum z x)
     (make-from-real-imag (+ (real-part z) x)
                          (imag-part z)))
(put 'add '(complex scheme-number)
       (lambda (z x) (tag (add-complex-to-schemenum z x))))
```

- 太麻烦。**n** 种类型 **m** 种操作需要 **m*n*(n-1)** 个混合操作
- 加入新类型时不仅要定义其自身的操作，还需定义它与已有的各相关类型之间的混合操作。操作在哪里定义也不好安排

不同类型的数据的组合：强制

- 处理几种相互无关的数据和相互无关的操作时，直接定义跨类型操作最简单，但非常麻烦。如果不同的类型之间有一定关系，就有可能利用，把事情做得更好些
- 不同数据类型之间经常存在一些关系。例如一种类型的对象可看作另一种类型的对象，这种看法称为强制（**coercion**）。例如：常规的数可以看成虚部为 **0** 的复数，它与复数的运算可以转化为复数运算
- 要实现这种想法，需要开发一些强制过程。例如

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

将其安装到特殊的强制表格里（假设有对该表格操作的 **put-coercion** 和 **get-coercion** 过程）：

```
(put-coercion 'scheme-number 'complex
                scheme-number->complex)
```

- 如果某些类型之间不允许自动强制，强制表中就不放相关的项。例如，可以不允许从复数转换到常规数

不同类型的数据的组合：强制

- 安装好所有强制过程后，还需修改 **apply-generic**，统一处理强制问题
- 新 **apply-generic** 先检查能否直接计算，如能就直接指派；否则就考虑能否强制（这里只考虑了两个参数的情况）：

```
(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))
          (if (= (length args) 2)
              (let ((type1 (car type-tags))  (type2 (cadr type-tags))
                    (a1 (car args))   (a2 (cadr args)))
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2 (apply-generic op (t1->t2 a1) a2))
                        (t2->t1 (apply-generic op a1 (t2->t1 a2)))
                        (else (error "No method for these types"
                                     (list op type-tags))))))
              (error "No method for these types"  (list op type-tags)))))))
```


类型的分层结构

- 用强制实现类型互操作的优点：每对类型只需要一个过程（直接做混合类型运算， 每对类型对需要有每个通用过程的一个具体版本）

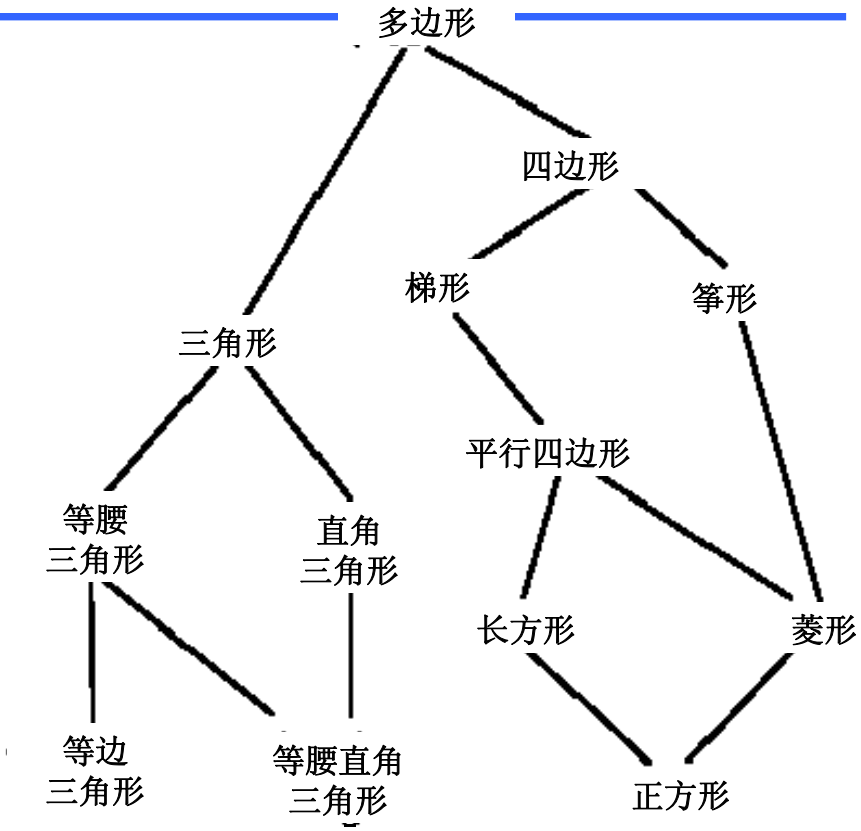
  这样做的条件是所需转换只与类型有关，与具体操作无关

- 更复杂情况可能需要把两个类型转到另一个公共类型（下面讨论）
- 能采用上述简单强制模型，就要求各对类型之间存在某种很简单的关系
- 例：算术系统存在一个类型分层：整数是有理数的子集，有理数是实数的子集，实数是复数的子集。人们称这种结构为“类型塔”（右图，是一种全序）

- 如果一组类型具有塔式结构，就可以用统一的模式实现相关强制：将低类型的对象逐步强制，直至两对象类型相同
- 如果类型具有塔式结构，还可能做向下的转换。例如，计算结果是复数 **4.3 + 0i** 时可以将其转换为实数 **4.3**

分层结构的不足

- 某些类型之间的关系复杂，分层结构可能不足以表达
- 右图表示由折线段表示的各种规范的封闭几何图形之间的关系
- 在复杂的类型关系结构中，向上强制或向下转换都可能很困难
- 特别是存在多种强制可能性时（面向对象的多重继承）



实例：符号代数

- 实例：一个符号计算系统
- 这种系统通常都很复杂
  - 能表现出设计和实现大型系统时可能遇到的许多问题
- 代数表达式具有层次结构，一个代数表达式可以看作从基本运算对象出发，用运算符和函数等构造而成的一棵树
  - 存在若干类基本运算对象，如数和变量
  - 存在一组运算符，如加减乘除。可能更多（如各种函数）
  - 各种运算符和函数可以看作表达式组合的类型
- 下面考虑开发一个完整的符号代数系统，其中集中关注多项式算术
- 设计中将特别考虑
  - 如何利用数据抽象和通用型操作的思想
  - 目标是构造出一个结构良好，易于扩充的系统

符号代数：多项式算术

- 多项式是基于一个或多个未定元（变量），通过乘和加构造起来的代数式。下面将多项式定义为某个未定元的项的和式，一个项可以是
  - 系数
  - 该未定元的乘方
  - 系数与未定元的乘方的乘积
  - 系数本身又可以是任意多项式，但它不依赖于当前未定元
- 例如： $(2y^2 + y - 5)x^3 + (y^4 - 1)x^2 - 4$
- 这里把多项式看作一种特殊语法形式，而不是它表示的数学对象

  例如不认为下面两个多项式等价：

$$3x^2 - 2x + 5 \qquad 3y^2 - 2y + 5$$

符号代数：多项式算术

- 考虑如何实现多项式算术。先把多项式表示为名为 **poly** 的数据结构，它由一个变量（符号）和一组项构成
  - 选择函数 **variable** 和 **term-list** 提取表达式的两个部分
  - 构造函数 **make-poly**，从变量和项表构造多项式
- 多项式加法和乘法过程：

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (add-terms (term-list p1) (term-list p2)))
      (error "Polys not in same var -- ADD-POLY" (list p1 p2))))

(define (mul-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (mul-terms (term-list p1) (term-list p2)))
      (error "Polys not in same var -- MUL-POLY" (list p1 p2))))
```

多项式算术包

- 把多项式放进通用算术系统，需要为它提供标签。下面用标签 **polynomial**，相关操作也安装到操作表里：

```
(define (install-polynomial-package)
  ;; internal procedures. First, representation of poly
  (define (make-poly variable term-list) (cons variable term-list))
  (define (variable p) (car p))
  (define (term-list p) (cdr p))
  <procedures same-variable? and variable? from section 2.3.2>
  ;; representation of terms and term lists
  <procedures adjoin-term ...coeff from text below>
  (define (add-poly p1 p2) ...) ... ;; <procedures used by add-poly>
  (define (mul-poly p1 p2) ...) ... ;; <procedures used by mul-poly>
  ;; interface to rest of the system
  (define (tag p) (attach-tag 'polynomial p))
  (put 'add '(polynomial polynomial)
       (lambda (p1 p2) (tag (add-poly p1 p2))))
  (put 'mul '(polynomial polynomial)
       (lambda (p1 p2) (tag (mul-poly p1 p2))))
  (put 'make 'polynomial
       (lambda (var terms) (tag (make-poly var terms))))
  'done)
```


多项式算术：项表的加法

- 多项式加法通过项表的加法实现，未定元幂次相同的项相加，得到和式中各项系数。项表是数据抽象：谓词 **empty-termlist?** 判断是否空，**first-term** 取出最高次项，**rest-terms** 取得其余项的表。
- 项也是数据抽象，构造函数 **make-term**，选择函数 **order** 和 **coeff**
- 项表求和过程如下：

```
(define (add-terms L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1)) (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term t1 (add-terms (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
                  (adjoin-term t2 (add-terms L1 (rest-terms L2))))
                 (else
                  (adjoin-term
                    (make-term (order t1) (add (coeff t1) (coeff t2)))
                    (add-terms (rest-terms L1) (rest-terms L2)))))))))
```

注意：这里用通用过程 **add** 求两个系数之和。这一做法很重要（后面解释）

多项式算术：项表的乘法

- 求两个项表之积的方法是逐个地用第一个表的项去乘第二个表中的各项（用 **mul-term-by-all-terms**），再把得到的项表累加起来。两个项的乘积由其系数之积和次数之和构成

```
(define (mul-terms L1 L2)
  (if (empty-termlist? L1)
      (the-empty-termlist)
      (add-terms (mul-term-by-all-terms (first-term L1) L2)
                 (mul-terms (rest-terms L1) L2))))
(define (mul-term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let ((t2 (first-term L)))
        (adjoin-term
          (make-term (+ (order t1) (order t2))
                     (mul (coeff t1) (coeff t2)))
          (mul-term-by-all-terms t1 (rest-terms L))))))
```


多项式算术：数据导向

- 由于操作都基于通用算术过程实现，这一多项式算术系统自动地就可以处理任何（算术包能处理的）系数类型
  - 由于有前面定义的强制，不同类型的数可以相互运算
  - 为支持各种多项式的运算，需要增加把数强制到多项式的能力（数就是 0 次多项式）
- 例如 $[(y+1)x^2+(y^2+1)x+(y-1)]\cdot[(y-2)x+(y^3+7)]$

  系统计算系数时通过通用的 **add** 和 **mul** 指派。由于系数也是多项式，因此将自动调用多项式运算 **add-poly** 和 **mul-poly**

  这样产生了数据导向的递归，每层都根据被运算数据的类型处理
- 还可以重新考虑不同变量的多项式之间的关系。例如

  2 x + 4 和 **2 y – 1** 求和，可以把后一多项式看作 **x** 的 **0** 次多项式

  将这种考虑严格化，需要给变量排一个顺序（请自己想想）

多项式算术：项表的表示

- 考虑项表的实现
  - 项表是以次数为键值的系数集合，可采用任何能表示集合的技术
  - 需要按降幂顺序使用表中的项，因此应采用某种排序表示
- 考虑项表表示的一个因素是项的稠密性，看下面多项式：

$$x^5 + 3x^4 + x^3 - x + 6 \qquad x^{1000} - 2x^{10} + 5$$

- 稠密多项式可以直接用项的表表示，如 **(1 3 1 0 -1 6)**

  稀疏多项式最好用两项表的表的形式，例如 **((1000 1) (10 -2) (0 5))**

  （显然也可以考虑用序对的表）
- 实际上也可以考虑两种表示共存的系统（参考前面的复数算术系统），加一层数据封装
- 下面采用后一种方式，项表里的项按降幂顺序排列

多项式算术：项表的实现

- 确定了数据表示，项表的实现很简单：

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))
      term-list
      (cons term term-list)))
(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))
(define (make-term order coeff) (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

注意：**=zero?** 也应为通用型过程。不难在算术包中加入它

- 创建多项式的过程：

```
(define (make-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

符号代数包中类型的分层结构

- 这个系统说明：一种对象的结构可能很复杂，以许多不同类的对象作为组成部分。但对它们定义通用操作不会更困难
- 先定义针对复杂对象中各种成分的操作，安装适当的通用型过程。数据导向的程序技术完全可以处理具有任意复杂递归结构的数据对象
- 在多项式代数系统里，多种相关的数据类型无法安排到一个类型塔里。例如 x 的多项式和 y 的多项式哪个更高？可能解决办法：
  - 实现多项式转换，把一个变量的多项式变换为另一个变量的多项式（展开并重新整理）
  - 变元排序，多项式总保持为某种“规范形式”（最外变元的优先级最高，依次类推）。这种技术使用广泛，但它有时会扩大多项式的规模，并会影响其可读性
- 设计大型代数演算系统时，强制问题可能很复杂，不好控制。但另一方面，已有的技术足以支持大型的复杂系统

本章总结

- 数据抽象的意义
- 构造数据抽象的基本过程和设计原则
- 闭包性质的意义
- 通用型过程
- 消息传递风格的程序设计
- 数据导向的程序设计
- 基于数据导向和通用型操作实现复杂系统的技术