2. 构造数据抽象(3)

本节讨论

- 数据导向的程序设计
- 消息传递
- 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

前面技术的弱点

- 检查数据的类型，根据类型调用过程，称为基于类型的指派

  这种技术在增强系统模块性方面很有用

  本质：根据数据的类型划分安排处理过程，分解系统功能

- 但是，基于类型的指派有两个重要弱点：
  - 每个通用型过程（如复数的选择函数）必须知道所有类型
    - 只有这样才能完成基于类型的指派
    - 增加一个（带相关操作的）新类型，必须为它选一个新标签，并给每个通用型过程增加一个新分支，完成对新类型的指派
  - 不同类型的表示相互独立，通常分别独立定义
    - 为最后做集成，定义时要准备好，不同类型的定义里采用不同接口过程名；否则集成时可能出现名字冲突
    - 很难保证多个模块的名字唯一性，修改程序很麻烦

前面技术的弱点

- 这两个弱点说明该技术不具有可加性（可加性：可以方便地为已有程序增加新部分。对程序的维护/修改/升级非常重要）
- 修改程序很麻烦，容易引进错误，应尽可能避免
  - 大型系统可能包含成百或更多的类型和表示方式，大量通用型过程。增加一个新类型，工作量极大
  - 如果没有完全了解所有相关程序的程序员，或者使用了没有源代码的库，扩充类型的工作很难完成
- 现在 **scalability** 受到广泛重视
  - **scalability** 指一种技术或方法可以应用于从小型到大型的各种系统，工作负担大致按系统规模增长而成比例增长
  - 如果一种技术不具有 **scalability**，就不可能用于大规模系统

    可加性显然是 **scalability** 的一个方面
- 前面提出的在 **C** 语言里实现多重表示的技术同样存在可加性问题

前面技术的弱点

- 这里出现不希望的情况，根源是：
  - 实际代码里明确写了数据类型或与类型直接相关的信息
  - 一旦类型有变化（如增加了新类型），就必须修改代码
  - 要解决这种问题，必须避免在代码里直接写类型信息，同时还要能建立类型与操作之间的正确联系
- 一种支持系统进一步模块化的技术是数据导向的程序设计
  - 基本想法：用数据结构保存各种类型和与之关联的操作（过程），在需要操作时，通过类型查找相关过程
  - 注意：这种技术要求把操作（过程）的信息存入数据结构
- 下面介绍 **Scheme** 里数据导向程序设计的一种实现方法
  - 在 **C** 语言和其他常规语言里也可以类似实现
  - 但在常规语言里的实现不如在 **Scheme** 里的简单，还要克服具体语言带来的一些具体麻烦

数据导向（数据驱动）的程序设计

- 通过分析可以看出，要处理针对不同类型的一批通用操作，有关信息可以用一个二维表格表示。复数实例的表格：

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

- 对基于类型的指派技术，这种表格藏在通用型过程的代码里
- 数据导向的程序设计里显式表示和处理这种二维表格
  - 前面用一集过程作接口，让它们检查类型，显式指派
  - 数据导向技术用一个通用过程实现接口，它用操作名和类型查找二维表格，找出所需要的过程
  - 增加一种新类型，只需在表格里增加一组新项，不需要修改程序

数据导向的程序设计

- 下面工作基于表格及其两个基本操作

  假定语言提供了这些功能，实际实现在第3章考虑

  实现表格需要做改变状态的程序设计，下一章讨论

- 假定有一个内部表格，其基本操作是 **put** 和 **get**：

  put 把一个项 ***<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>*)

- 显然，加入表格的项和取出后的使用方式要相互协调

  下面例子里，加入和取出的都是过程

  完全可以在表格里存放其他信息（“表格”也称为关联表或字典）

数据导向的复数实现

- 现在考虑用数据导向程序设计技术实现复数系统。这里需要分别实现直角坐标和极坐标计算过程。具体工作：
  - 直角坐标实现
    - 定义好各个过程，并把它们作为项加入表格
    - 这也就是告诉系统应如何处理直角坐标类型的复数
  - 极坐标实现也要做同样的事情
- 这建立了两种表示的实现，并利用表格建立它们与其他部分的接口
  - 系统其他部分不需要（也不必）关心它们的具体实现，只是通过表格找到所需过程，用于操作相应的复数表示
  - 两组操作必须有一些共性：对应的过程，调用方式必须相同
  - 加入一种新数据表示时，只需要把一套过程存入表格。通过表格使用复数功能的程序，很自然地取得了处理这种新类型的能力

数据导向的复数实现：直角坐标部分

- 直角坐标部分定义为一个无参过程，其代码分为两部分
  - 一部分代码定义一批内部过程
  - 另一部分把定义好的过程安装到表格里

```
(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))))
```

;; 过程定义未完，接下页

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)
```

后一半代码把操作安装到表格里的适当位置（注意， **(rectanglar)** 是项的关键码的一部分。）

数据导向的复数实现：极坐标部分

- 极坐标复数部分采用同样实现技术
  - 用一个无参过程创建其过程组
  - 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)) )
  ;; 接下页
```

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)
```

- 把两类复数的过程都安装到表格里，就可以通过表格检索和使用了

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，它基于参数得到操作名和类型标签，到表格里查找具体操作

```
(define (apply-generic op . args) ;; 任意多个参数
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args)) ; 将 proc 应用于... (apply)
          (error
            "No method for these types -- APPLY-GENERIC"
            (list op type-tags))))))
```

- 所有选择函数都基于上面的通用选择过程定义：

```
(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

- **apply** 是一个最重要的系统过程，它把一个过程应用于相应参数

  (apply *proc arg1* ...)　　；可以是多个参数的过程

- 构造函数的实现也基于到表里查找操作：

```
(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) )
```

- 要想增加一种新复数类型，只需要
  - 定义实现该类型的 **package** 过程，基于内部过程和其他已有功能实现该类型的基本操作，并把这些操作安装到操作表格里
  - 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

  已有的所有程序代码都不需要修改

消息传递

- 数据导向的程序设计，关键想法就是显式处理 “类型-操作”表格，在表格里存储程序里的所有通用操作
- 处理这个问题的另外两种方式：
  - 前面介绍的基于通用操作的技术
    - 把操作定义得足够强大（是能理解类型的智能操作），它们能根据被处理的数据的类型决定采用的具体操作
    - 相当于把表格横向切分，每行实现为一个“智能操作”
  - 另一可能性是把表格纵向切分
    - 定义足够聪明的“智能数据对象”，这种对象能根据送来的操作名决定自己要做的工作
    - 面向对象技术里定义的就是这种对象
- 下面介绍在 **Scheme** 里应用后一技术，方法：用过程表示对象（由另一个过程生成），这种过程接受操作名并完成所需的工作

消息传递

- 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)))))
```

- **make-from-mag-ang** 的定义与 **make-from-real-imag** 类似
- 需要重新定义 **apply-generic**
  - 使它能处理上面两个生成过程的需要
  - **apply-generic** 要做的就是把操作名送给相应数据对象

消息传递

- **apply-generic** 的新定义：

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

  其他操作都不改（包括基于 **apply-generic** 定义的选择函数）
- 对象创建操作返回的过程就是 **apply-generic** 调用的过程

  加入一种新类型时，需要定义生成该类数据对象的过程

  这个新过程必须能接受与已有类型同样的操作
- 这种风格的程序设计称为消息传递

  创建对象生成的这种过程应看作一种数据对象，功能强大
  - 能接受一组消息并根据消息决定要做的工作
  - 对消息的响应就是执行特定动作
  - 每个对象有自己的状态

消息传递

- 前面介绍过用过程实现序对，采用的实际上就是这种技术

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- CONS" m)))))
(define (car z) (z 0))
(define (cdr z) (z 1))
```

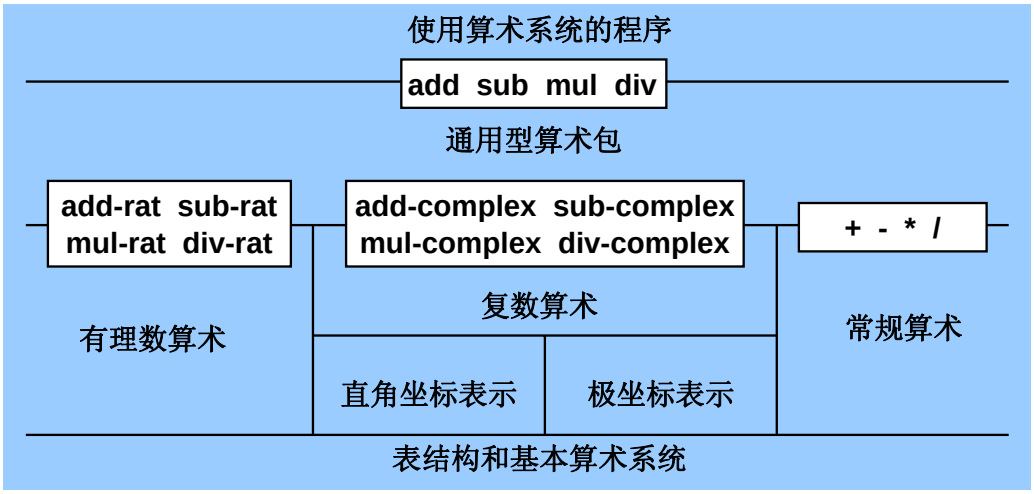
  - cons 是生成数据对象的过程
  - 送给对象的消息是 0 和 1，要求它给出自己的 car 和 cdr 部分
- 这种技术具有实用性。下一章还要继续关注基于消息传递的程序设计

  下面继续讨论数据导向的程序设计
- 有兴趣的同学可以把上面讨论的问题和提出的技术与面向对象程序设计中的相关问题做一些比较

包含通用型操作的系统

- 前面实现过有理数算术包和复数算术包，Scheme 系统提供了常规算术系统。现在考虑如何使用数据导向技术把这些算术系统集成起来。目的是研究定义处理不同参数类型的通用型操作的技术

  下面开发的系统的结构：



- 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

- 例如，整个系统应该只有一个通用加法 **add**
  - 对常规的数，**add** 的行为应该等同于 **+**
  - 对有理数，其行为应等同于 **add-rat**
  - 对复数，其行为等同于 **add-complex**。其他算术运算也类似
- 还用前面技术，每个类型确定（并增加）一个标签，通用型运算 **add** 等根据运算对象的标签完成正确指派
- 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))
```

它们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 数加上标签 **scheme-number**。每个算术运算都有两个参数，检索关键码用 **(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

- 有理数功能可以直接装入表格，不需要修改

```
(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d) (let ((g (gcd n d))) (cons (/ n g) (/ d g))))
  (define (add-rat x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))
  (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))))
```


通用型算术运算：有理数

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

  ;; interface to rest of the system
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational rational)
       (lambda (x y) (tag (add-rat x 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))))
  (put 'make 'rational (lambda (n d) (tag (make-rat n d))))
  'done)

(define (make-rational n d) ((get 'make 'rational) n d))
```

通用型算术运算：复数

- 把类似前面的复数包安装到系统，复数加标签 **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)
```

主要操作都是内部的，采用同样过程名 **add, sub, mul, div** 也互不冲突

通用型算术运算：复数

- 使用时，可以用实部/虚部或模/幅角的方式创建复数

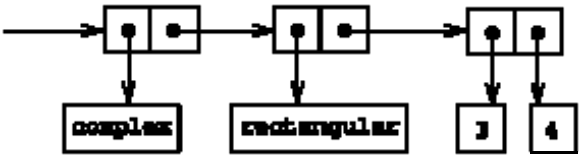
  这要用到直角坐标和极坐标程序包里的过程（构造函数）

  为方便，可以基于这两个函数定义两个全局的构造函数：

```
(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))
```

- 这里的复数有两重标签：外层标签标明不同的数类（有理数/复数/常规数），内层标签标明复数表示方式

  如 **3 + 4i**：



系统分层

- 这种情况很普遍
  - 复杂的系统可能划分为许多不同抽象层次，对应的数据表示也可能包含多个层次
  - 系统的不同层次之间通过一组通用操作相互联系
  - 这些通用操作基于数据的标签区分不同数据类别
  - 为支持一个层次上通用操作，可以用一组数据标签。这使系统的层次性表现为数据的层次性
- 数据在传递和使用中可能历经不同的抽象层次
  - 在“向下”传输的过程中逐步剥离一层层标签，最后实际处理的是没有标签的“裸”数据（实际数据）
  - 在“向上”传输的过程中一层层增加数据标签，以便将来能用于识别数据的实际类型，确定应该用的操作
- 典型情况：网络上的各种传输，例如传递 **Web** 页

跨越类型

- 前面做好的计算系统能处理多种不同数值，但它有一个重大缺点
  - 不同类型的数分属相互独立的不同“世界”
  - 不同世界之间没有任何相互联系
  - 表现：不同类型的数之间不能互操作（对于各种数，就是不能支持不同类型的数之间的混合运算）
- 实际系统中的常见情况：一组类型相互之间有关系，需要做不同类型的数据之间的操作
- 不同数值类型常需要相互运算，如，求一个实数和一个复数之和，求一个复数或实数的倍数或几分之一
  - 在多类型集成算术包里，与之对应的是跨类型操作
  - 如：复数与 **Scheme** 实数的加法，复数或 **Scheme** 实数与 **Scheme** 整数或者自定义的有理数之间的乘法等

跨越类型

- 前面工作中，通过仔细的设计，确立了程序中各部分之间的清晰隔离，追求系统的模块化和可加性

  要考虑实现跨类型操作，必须同时考虑两方面问题：

  - 使扩充后的系统支持跨类型操作
  - 不能严重损害原有的模块分隔
- 实现跨类型操作，一种方式是为每对可以相互操作的类型组合增加一个（或者两个）操作
  - 如果两种类型的相操作具有可交换性（即，操作结果与参数的顺序无关），可以只实现一个混合参数操作
  - 否则就需要定义两个混合参数操作
- 还要考虑如何根据被操作数据和/或结果的类型找到相关操作
  - 同样可以把操作放入数据导向表格
  - 如果有不同类型的三参数或多参数操作，问题会变得更复杂

跨越类型

- 简单考虑：如要在算术系统里增加求复数与常规数之和的操作，可以用标签 **(complex scheme-number)** 在表格里加一个过程：

```
;; 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))))
```

- 这样做很麻烦：n 种类型 m 种操作需要 m*n*(n-1) 个混合操作

  需要定义的操作个数与类型个数之间不是线性关系

  加入一个新类型

  - 需要定义该新类型自身的各种操作，还要定义它与已有各相关类型之间的混合操作
  - 混合操作涉及多个类型，把定义放在那里没有明确的线索

强制

- 如果要考虑的几种数据类型之间没有可利用关系，多种操作之间也没关系，要做它们之间的互操作
  - 只能考虑直接定义所需的跨类型操作，可能很麻烦
  - 应注意：相互无关的类型之间有用的互操作不会很多，需要定义的“混合类型”过程应该 “比较稀疏”
- 如果要考虑的不同类型之间有些关系

  就有可能利用这些关系，把事情做得更好
- 实际中需要互操作的数据类型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（类型集合是子集关系），这种看法称为强制（**coercion**）
  - 例：常规 **Scheme** 数可以看成虚部为 **0** 的复数，它们与复数之间的运算可以转化为复数运算

强制

- 实现这种想法，需要开发一些强制过程，它们从一种类型（参数类型）的数据构造出另一种类型的数据（强制的结果类型）
- 例如

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

  把这个操作安装到一个特殊的强制表格里（设对这个表格的操作过程是 **put-coercion** 和 **get-coercion**）：

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

- 如果某些类型之间不允许强制，就不在强制表中放相关项。例如
  - 可以不允许从复数转换到常规的数（丢失信息）
  - 不允许将实数转换到有理数（实数是近似数而有理数是精确数）
- 安装强制过程后，还需要修改 **apply-generic**，统一处理强制问题

跨越类型：强制

- 新 **apply-generic** 先检查能否直接计算，如果能就直接指派；否则就考虑能否强制（下面定义只考虑了两参数情况）：

```
(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

- 通过强制实现类型间互操作，每对类型只需要一个过程（采用直接混合类型运算，每个通用过程需要有针对每对类型的一个或两个版本）

  能这样做，前提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）

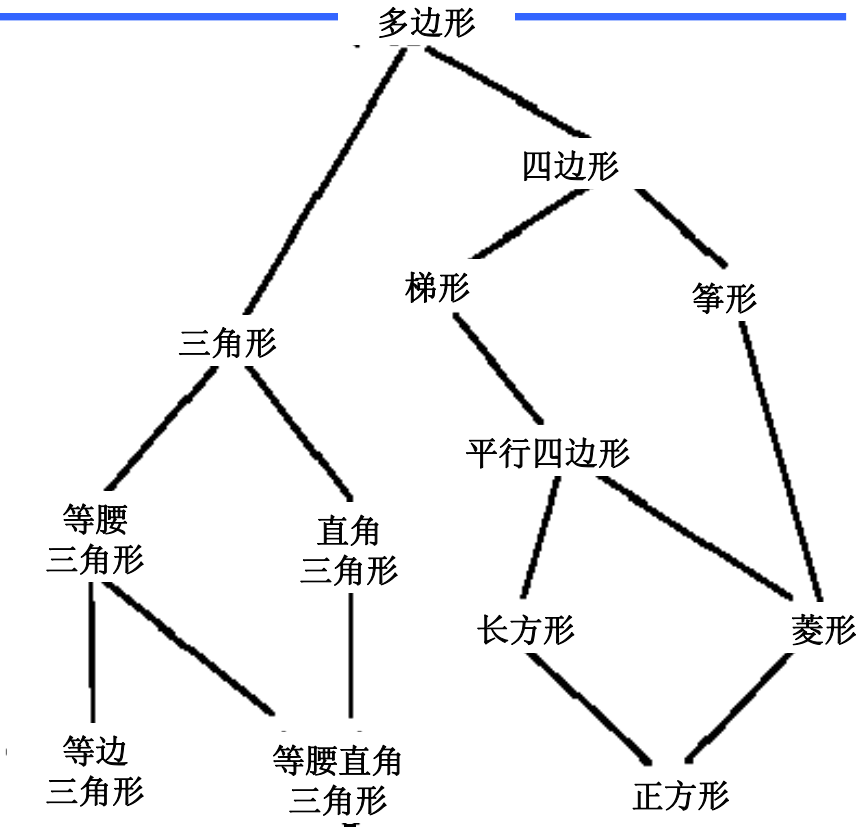
- 简单强制模型要求各对类型之间有某种简单关系

- 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”（见右图，是一个全序）

- 如果一组类型具有塔式结构，就可以用统一实现强制：
  - 将低类型对象逐步强制，直至两对象类型相同
  - 还可能做向下转换。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

- 类型之间的关系可能很复杂，塔式结构可能不足以表达
- 右图：折线段构成的规范的封闭几何图形之间的关系
- 易见，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（请注意面向对象里的多重继承问题）



实例：符号代数

- 现在考虑另一个实例：符号计算系统
  - 这种系统通常都比较复杂
  - 可以看到设计和实现大型系统时可能遇到的许多问题
- 符号计算系统处理的对象是各种代数表达式
  - 代数表达式具有层次结构，一个表达式可以看作从基本运算对象出发，通过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，如各种数和变量
  - 存在一组运算符，如加减乘除。可能更多（如各种函数）
- 现在把代数表达式看作要处理的应用领域的数据对象，它们可以直接映射到 **Scheme** 的符号表达式
  - 基本元素，层次结构，各种基本运算，……
  - 各种运算符和函数可以看作组合表达式的类型

符号代数系统

- 下面考虑开发一个“完整的”符号代数系统

  主要考虑多项式算术的实现问题

- 在系统设计中，将特别考虑
  - 如何充分利用数据抽象和通用型操作的思想
  - 如何把多项式计算与前面的通用算术包集成在一起
  - 目标是构造出一个结构良好，易于扩充的系统
- 基于数据抽象的思想，多项式也看作一种数据抽象
  - 它支持一组基本操作，可以创建，可以访问其成分，……
  - 其具体实现可以推迟到后面考虑
  - 注意：这是本书中始终倡导的设计理念和有效的问题解决途径

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），通过乘法/加法构造出的代数式。下面把多项式定义为某（特定）未定元的项的和式，项可以是
  - 一个系数
  - 该未定元的乘方
  - 一个系数与该未定元的乘方的乘积
  - 这里的系数本身又可以是任意的多项式（可以有自己的未定元），但是它不依赖于当前的未定元
- 例如：$(2y^2 + y - 5)x^3 + (y^4 - 1)x^2 - 4$
- 这里把多项式看作是一种特殊语法形式，而不是它表示的数学对象

  例如，不认为下面是两对分别等价的多项式：

$3x^2 - 2x + 5$ $\quad\quad$ $(y+1)x^2 - 3x + (y+2)$

$3y^2 - 2y + 5$ $\quad\quad$ $(x^2+1)y + (x^2 - 3x + 2)$

多项式算术

- 考虑多项式算术的实现，采用数据抽象的思想
- 首先用名为 **poly** 的数据结构表示多项式，其成分是一个变量（用符号表示）和一组项（项的一个表，项表）
  - 构造函数 **make-poly**，从变量和项表构造多项式
  - 选择函数 **variable** 和 **term-list** 提取表达式的两个部分
- 项表也是数据抽象：
  - **empty-termlist?** 判断项表是否为空
  - **first-term** 取出最高次项
  - **rest-terms** 取得除最高次项之外的其余项的表。
- 项也是数据抽象
  - 构造函数 **make-term**
  - 选择函数 **order** 和 **coeff** 取次数和系数成分

多项式算术

- 现在可以基于这些数据抽象定义各种多项式操作
  - 实现多项式操作仍然采用处理复杂计算的分解技术
  - 多项式计算被归结到一些更基本的过程，推后实现
- 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

- 多项式也放进通用算术系统，以利用已有的算术功能。为此要为多项式定一个标签（下面用 **polynomial**），相关操作也安装到操作表格里：

```
(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 body omitted>
  (define (mul-poly p1 p2) ...) ... ;; <procedures body omitted>
  ;; 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

- 多项式加法中调用了项表加法过程 **add-terms**。实现很直接：把未定元幂次相同的项的系数相加，得到和式中各项系数
- 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** 求系数之和。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用第一个表的项去乘第二个表的各项（用 **mul-term-by-all-terms**），并累加得到的项表。两个项的乘积由其系数之积和次数之和构成。实现：

```
(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

- 所有操作都基于通用算术包的过程（**add** 和 **mul**）实现
  - 使多项式算术系统自动得到处理任何（通用算术包能处理的）系数类型（以及系数为多项式的情况）的能力
  - 由于前面定义的强制，不同类型的数可以相互运算
  - 为支持多项式运算，需要增加数强制到多项式的功能（数看作 **0** 次多项式）

- 例如 $[(y+1)x^2+(y^2+1)x+(y-1)]\cdot[(y-2)x+(y^3+7)]$

  计算系数时，系统通过通用算术包的 **add** 和 **mul** 完成指派。遇到系数也是多项式，就自动调用多项式运算 **add-poly** 和 **mul-poly**

  这样做，其中实际上出现了数据导向的递归

  递归的每一层都根据作为运算对象的数据的类型处理

多项式算术：数据导向

- 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
  - 可能做法：把后一多项式看作 **x** 的 **0** 次多项式，完成运算
- 新问题：
  - 应该把哪个多项式看作另一变量的 **0** 次多项式？
  - 为实现算法（完成计算的过程），必须给出明确规则
  - 理论上说任何变量都不比其他更重要。但现在必须确定一套处理规则，需要人为确定变量的重要性并确定转换规则
- 上面考虑严格化需要给变量排一个序。请大家自己仔细做一下
  - 严格写出计算规则
  - 修改前面给出的程序，实现相关计算

多项式算术：项表的表示

- 现在考虑项表的实现（总是把数据的实现留到后面）
  - 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
  - 计算中需要按降幂顺序使用各项，因此应该用某种排序表示
- 考虑项表表示的一个因素是项的稠密性，看下面两个多项式：

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

- 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，例如 **((1000 1) (10 -2) (0 5))**

  （显然也可以考虑用序对的表）
- 当然可以考虑两种表示共存的系统，为此需要加一层数据封装（参考前面的复数算术系统）
- 下面示例代码采用后一种方式，项表里的项按降幂顺序排列

多项式算术：项表的实现

- 确定了数据表示之后，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。不难将其加入算术包

- 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：虽然对象结构可能很复杂，一种对象可能以多个不同类的对象作为组成部分，但定义它们的通用操作并不困难
  - 首先定义针对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能处理任意复杂的递归结构的数据对象
- 在多项式代数系统里，相关数据类型不能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：
  - 实现多项式转换操作，它能把一个变量的多项式变换为另一个变量的多项式（多项式展开及重新整理）
  - 变量排序，并保证系统里的多项式都是某种“规范形式”（最外变量优先级最高，依次类推）。实际符号计算系统里广泛采用这种技术，但它有时会导致多项式的规模膨胀，影响可读性和计算效率
- 设计大型代数演算系统时，强制问题可能变得很复杂。但也应看到，上面提到的技术足以支持大型复杂系统的开发

本章总结

- 数据抽象的意义
- 构造数据抽象
  - 基本过程
  - 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
- 许多具体的编程实例
  - 展示了上面各种重要技术
  - 说明了其中的考虑，优点和缺点