函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

http://blog.sina.com.cn/u/3266490431

第六讲

带标志数据的多重表示

- 通用接口：

```
(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))))
```

带标志数据的多重表示

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，很难保证多个模块的名字唯一性，修改程序很麻烦

数据导向的程序设计

●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

数据导向的程序设计

- 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：

		Types: Polar	Types: Rectangular
Operations	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

- 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>)

例如: op = ‘real-part, type = ‘real-part-polar , item 是某个过程

type可以是个tag的列表,也可以是单个 tag

get 取出表格中与<op> 和<type> 关联的项

(get <op> <type>)

例如: op = ‘real-part, type = ‘real-part-polar

type可以是个tag的列表,也可以是单个 tag

假定语言提供了这些功能，实际实现在第3章。实现表格需要做改变状态的程序设计，第3章讨论

- 本例中，加入和取出的都是过程。在表格里亦可存放其他信息（**“表格”也称为关联表或字典**）

数据导向的复数实现

- 需要分别实现复数直角坐标和极坐标计算过程。建立两种表示的实现，并利用表格建立它们与其他部分的接口

  定义好各个过程，并把它们作为项加入表格

- 系统其他部分不需要（也不必）关心们的具体实现，只是通过表格找到所需过程，用于操作相应的复数

□ 直角坐标和极坐标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，就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

●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, 不用担心重名
  (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

```
;; interface to the rest of the system
;下面的put语句在 install-rectangular-package 调用时执行
(define (tag x) (attach-tag 'rectangular x))
(put 'real-part '(rectangular) real-part) ;注意：第二维是列表
(put 'imag-part '(rectangular) imag-part)
(put 'magnitude '(rectangular) magnitude)
(put 'angle '(rectangular) angle)
(put 'make-from-real-imag 'rectangular ;注意：第二维是单项
     (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)
```

回顾:带标志数据的多重表示

●如果要让前面的 add-complex, sub-complex, mul-complex ,div-complex对两种形式的复数都能工作，则需要往复数的表示形式中添加标记，以便使用到一个复数的时候，通过标记可以知道它是哪种表示形式。此时，复数是嵌套对子。

例: ('rectangular . (34 . 56)) ('polar . (45 . 90))

```
(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);求复数的实部虚部，或极角和模
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum -- CONTENTS" datum)))
```

数据导向的复数实现 – 极坐标

```
(define (install-polar-package)
  ;; internal procedures, 不用担心重名
  (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

```
 ;; interface to the rest of the system
;下面的put语句在 install-rectangular-package 调用时执行
 (define (tag x) (attach-tag 'polar x))
 (put 'real-part '(polar) real-part) ;注意：第二维是列表
 (put 'imag-part '(polar) imag-part)
 (put 'magnitude '(polar) magnitude)
 (put 'angle '(polar) angle)
 (put 'make-from-real-imag 'polar ;注意：第二维是单项
      (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)
```

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，它的参数是操作名和类型标签列表，到表格里查找具体操作

```
(define (apply-generic op . args) ;参数个数不限
  ;op对应于表里的第一维,type-tags对应于表里的第二维
  (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))))))
```

复数结构：`('rectangular . (10 . 20))`
`('polar . (10 . 20))`

数据导向的复数实现 – 通用接口过程

apply是scheme的基本过程

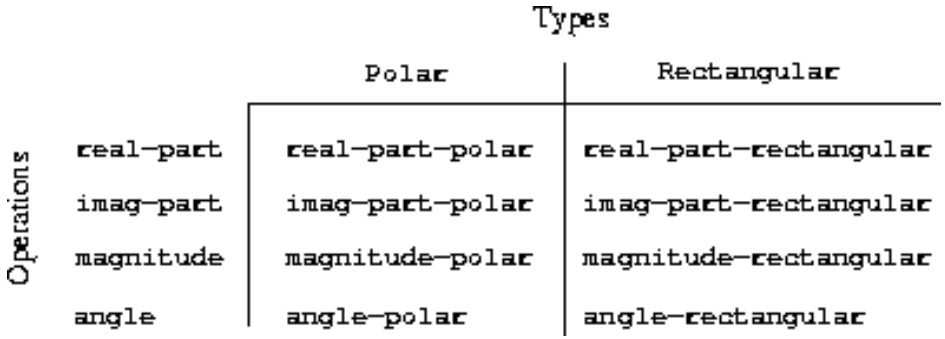
用法：**(apply proc lst)**　**lst是一个列表**

作用：调用 **proc，以lst里的项目作为proc的参数**

数据导向的复数实现 – 选择函数

```
(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))
```

z的结构：
```
('rectangular . (10 . 20))
('polar . (10 . 20))
```




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

对比：带标志数据的多重表示

- 通用接口：

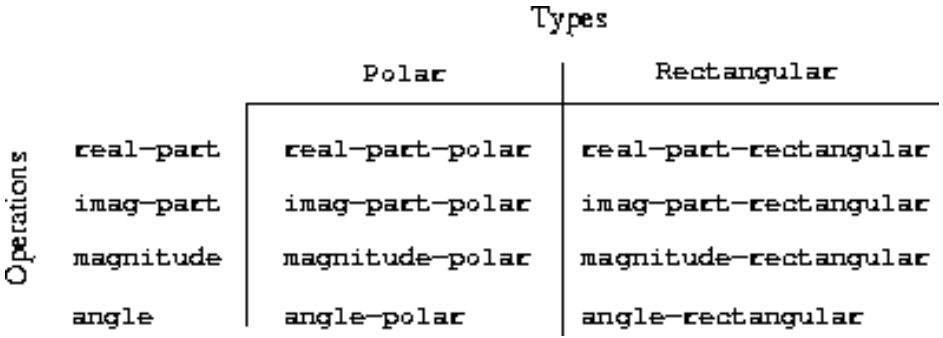
```
(define (real-part z) ;把类型标志写在代码里了，可扩充性差
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type -- REAL-PART" z))))
(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Unknown type -- IMAG-PART" z))))
```

数据导向的复数实现 – 构造函数

```
(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) )
```

;注意：第二维是单项

z的结构： ('rectangular . (10 . 20))
           ('polar . (10 . 20))



数据导向的复数实现

要想增加一种新复数类型，只需要

- 定义实现该类型的install-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有定义了，不需要重写(老办法的构造函数要重写)

已有的所有程序代码都不需要修改

数据导向的复数实现

- 前面的做法是设立表格，用查表的方式选取合适的过程

- 数据导向的另一种实现方式，是采用闭包（相当于对象），对闭包发送消息，以调用合适的过程。

```
(define (make-from-real-imag x y) ;返回一个闭包dispatch
  (define (dispatch 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))))
  dispatch)   ;用法: ((make-from-real-imag 3 4) 'real-part)
```

数据导向的复数实现

●数据导向的另一种实现方式，是采用闭包（相当于对象），对闭包发送消息，以调用合适的过程。

此情况下须重写 apply-generic:

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

选择函数不需要修改

```
(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))
;(angle z) => (z 'angle)
```

数据导向的复数实现

●数据导向的另一种实现方式，是采用闭包（相当于对象），对闭包发送消息，以调用合适的过程。

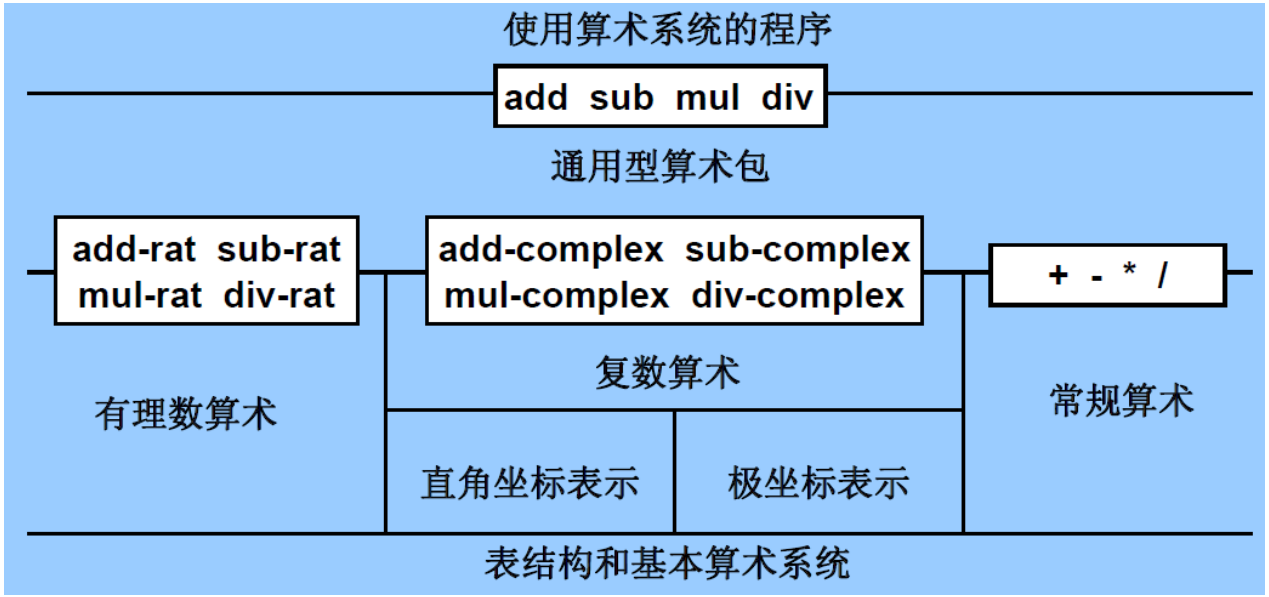
加入一种新类型时，需要定义生成该类数据对象的过程，即构造函数。这个新过程必须能接受与已有类型同样的操作

回顾：

```
(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))
```

包含通用型操作的系统

前面开发过有理数包，复数包，希望将这两种数据类型都整合在一个通用型算术系统，用相同的 add ，sub ，mul ，div 等操作来进行运算。还希望能够容易地往算术系统里添加新的算术包。



通用型算术运算

几个通用型运算过程的定义：

```
(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))
;针对不同类型数据的 add ,sub,mul,div都必须被添加到表格里
```

```
(define (apply-generic op . args) ;参数个数不限
  ;op对应于表里的第一维,type-tags对应于表里的第二维
  (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))))))
```

通用型算术运算

用闭包法实现 apply-generic

```
(define (add x y) (apply-generic 'add x y))

(define (apply-generic op . arg)
    ((apply (car arg) (cons op (cdr arg)))))
```

(add x y) => (x 'add y)

通用型算术运算 – 常规数包

```
(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)
```

构造函数:

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

通用型算术运算 – 有理数包

```
(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))))
```

通用型算术运算 – 有理数包

```
(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))))
;; 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))))
```

通用型算术运算 – 有理数包

```
  (put 'make 'rational
       (lambda (n d) (tag (make-rat n d))))
  'done)

;构造函数:
(define (make-rational n d)
  ((get 'make 'rational) n d))
```

通用型算术运算 – 复数包

```
(define (install-complex-package);安装带两层标记的高级复数包
;下两个过程基于前面实现的(install-polar-package)和
;(install-rectangular-package)
  (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) ;z1 z2是只有一层标记的低级复数。为什么?
    (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))))
```

通用型算术运算 – 复数包

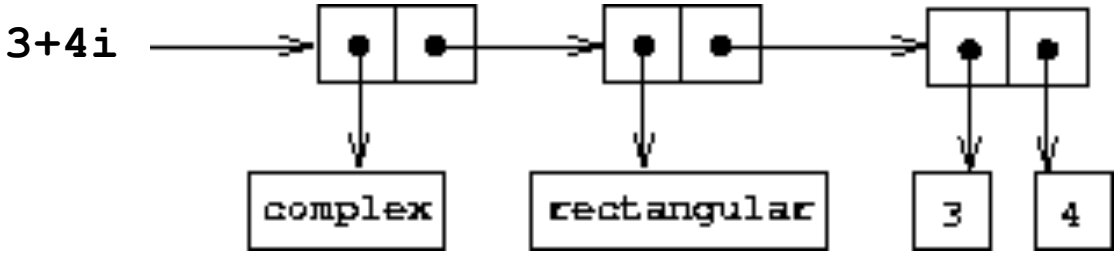
```
(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                     (- (angle z1) (angle z2))))
;; interface to rest of the system
(define (tag z) (attach-tag 'complex z))
(put 'add '(complex complex)
     (lambda (z1 z2) (tag (add-complex z1 z2))))
(put 'sub '(complex complex)
     (lambda (z1 z2) (tag (sub-complex z1 z2))))
(put 'mul '(complex complex)
     (lambda (z1 z2) (tag (mul-complex z1 z2))))
(put 'div '(complex complex)
     (lambda (z1 z2) (tag (div-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)
```

通用型算术运算 – 复数包

;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))
```

于是，一个复数有两重标签，外重说明它是个复数，内重说明该复数的表示方式（直角坐标或极坐标）



嵌套对子: `('complex . ('rectangular . (3 . 4)))`

系统分层

- 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

如何实现复数和普通数，复数和分数，分数和普通数之间的加减乘除?

- 办法一、为每对可以相互操作的类型组合增加两个操作，并放入操作表。例如:

```
;; 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))))

(define (add-schemenum-to-complex x z)
  (make-from-real-imag (+ (real-part z) x)
                       (imag-part z)))
(put 'add '(scheme-number complex)
     (lambda (x z) (tag (add-schemenum-to-complex  x z))))
```

回顾:通用型算术运算

几个通用型运算过程的定义:

```
(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))
```

```
(define (apply-generic op . args) ;参数个数不限
  ;op对应于表里的第一维,type-tags对应于表里的第二维
  (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))))))
```

跨越类型

如何实现复数和普通数，复数和分数，分数和普通数之间的加减乘除?

- 办法一、为每对可以相互操作的类型组合增加两个操作，并放入操作表。

n 种类型m 种操作需要m*n*(n-1) 个混合操作

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

类型强制转换

●如果能将A类型看作是另B类型的特例，B类型的数据也能当A类型的数据来处理，则可以不必再定义一系列A和B之间混合操作。例如，普通数可以看作是虚部为0的复数。

```
;普通数到复数的强制类型转换操作:
(define (scheme-number->complex n)
      (make-complex-from-real-imag (contents n) 0))
;将强制类型转换操作放入表中:
(put-coercion 'scheme-number 'complex scheme-number->complex)
;用put-coercion和get-coercion对该表进行操作
```

类型强制转换

```
(define (add x y) (apply-generic 'add x y))
;修改apply-generic
(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) ;如果找不到类型匹配的过程
              (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)))
; 前面:(put-coercion 'scheme-number 'complex scheme-number->complex)
```

类型强制转换

```
(cond (t1->t2 ;存在t1到t2的转换函数
       (apply-generic op (t1->t2 a1) a2))
      (t2->t1 ;存在t2到t1的转换函数
       (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)))))))
```

通过强制类型转换，每对类型只需要一个或两个互相转换的过程，就能实现两个类型之间的互操作。前提是这两种类型之间可以进行转换。

类型强制转换

■ 如果一组类型具有塔式结构，就可以用强制类型转换，将低类型对象逐步强制转换，直至两对象类型相同

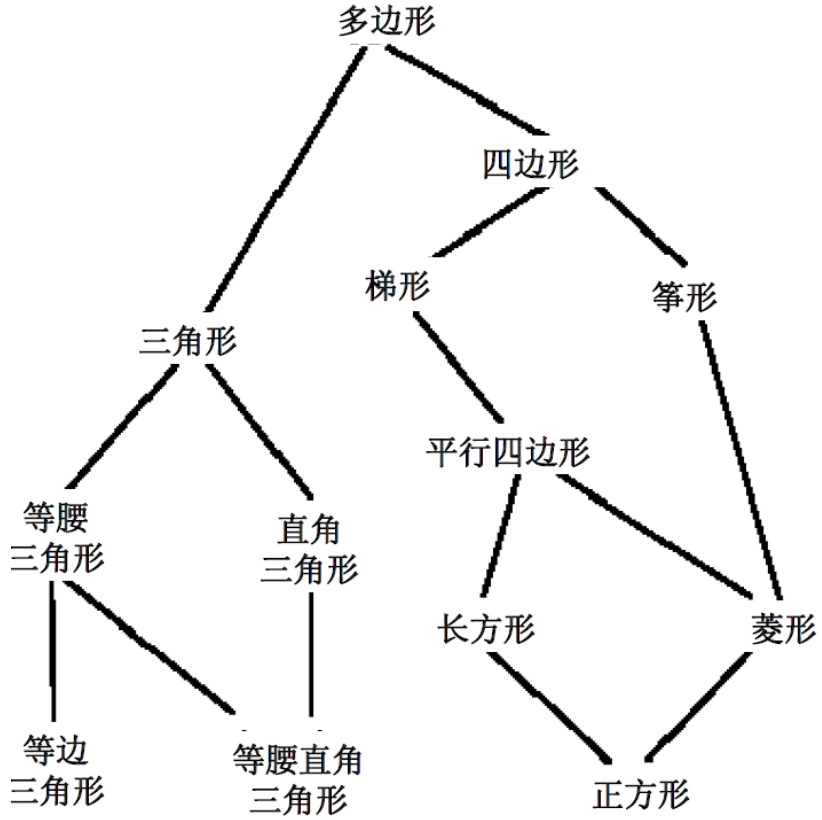
只要有 整数->有理数
有理数->实数
实数->复数

则不需要编写整数->复数，即可完成整数和复数的运算。（作业）

■ 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

■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转换可能性时。比如，正方形和梯形进行运算时，需要向上强制转换。向上强制转换时，可以转换成长方形，也可以转换成菱形。沿哪条路径进行转换更合适，则需要在apply-generic 中遍历不同的转换路径进行尝试。



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

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

符号代数:多项式算术

●多项式的表示:

多项式由两部分构成:变量名（符号)和项表(包含每一项的系数和幂次信息)。

make-poly	从变量和项表构造多项式
variable	提取多项式的变量名
term-list	提取多项式的项表
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。
make-term	构造项
order	取项的次数
coeff	取项的系数

符号代数:多项式算术

- 多项式的加法：

```
(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))))
```

符号代数:多项式算术

- 多项式的乘法：

```
(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))))
```

符号代数:多项式算术

- 把多项式包放入算术系统:

```
(define (install-polynomial-package)
  ;; internal procedures
  ;; 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>
```

符号代数:多项式算术

```
(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:项表加法

- **add-terms** 把未定元幂次相同的项的系数相加，得到和式中各项系数

```
(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 ;将一个最高次项加入项表
                   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)));通用的add
                   (add-terms (rest-terms L1)
                              (rest-terms L2)))))))))
```

```
;(define (add x y) (apply-generic 'add x y))
```

多项式算术:项表乘法

- mul-term-by-all-terms逐个用第一个表的表项，去乘第二个表的每个表项，然后把结果相加

```
(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)));通用的mul
         (mul-term-by-all-terms t1 (rest-terms L))))))

;(define (mul x y) (apply-generic 'mul x y))
```

多项式算术:项表乘法

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类型转换，不同类型的数可以相互运算

□ 为支持多项式运算，需要增加数强制转换到多项式的功能（数看作0次多项式）

□ 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 递归处理。**

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

多项式算术:项表乘法

- 不同变量的多项式之间的关系

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？
解决: 人为规定不同变量符号的优先级。优先级高的符号看作是多项式的自变量

多项式算术:项表的实现

- 方法1： n次多项式就记录0到n次的所有系数（对于稀疏多项式，浪费）

$$x^5 + 3x^4 + x^3 - x + 6$$ => (1 3 1 0 -1 6)

- 方法2： 列表中的每一个元素由两部分组成，系数和次数。

$$x^{1000} - 2x^{10} + 5$$ => ((1000 1) (-2 10) (5 0))

（项表里的项按降幂顺序排列）

多项式算术:项表的实现

- 用方法2的项表操作：

```
(define (adjoin-term term term-list);把最高次项term加入项表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))
```

- 多项式构造函数：

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

补充: call/cc和continuation基础

●一个continuation，可以认为是程序执行的一个现场(断点)，它可以描述成这样：等待一个值v,等到v的值后，就继续做 brabrabra ....(brabrabr里面可能会用到v，也可以不用)

●;contiuation 是first class的，即它可以被赋值给变量，可以当作函数参数...

●一个continuation可以被看作一个单参数的函数,如果contiuation被保存到变量ff里面，那么ff就是一个具有单参数的函数。此时如果执行 (ff y)，发生的事情就是立即用 y的值作为ff这个continuation所代表的现场中被等待的v的值，然后继续做ff里面描述的 brabrabra ....

补充: call/cc和continuation

- 那么如何生成一个 continuation呢？ 有一种表达式，我们叫他call/cc表达式。call/cc表达式被计算的时候，就会生成一个continiuation，这个continiuation代表的就是call/cc被执行前一时刻的现场：等待一个值v(本该是call/cc返回的值），等v的值算出来后，继续做brabrabra .... 这个被等待的 v 的值来自哪里，后面会看到
- call/cc 表达式，一定是调用某个函数f，f必须是有且只有一个参数的函数。call/cc 调用 f时，一定会提供参数x，x就是 call/cc被执行前那一时刻的continuation. x可以被看作是一个有一个参数的函数 ---- 因为一个contiuation可以被看作是一个有一个参数的函数
- call/cc表达式的结束，有两种情况。一种情况，是 f执行的过程中，没有调用 x. 那么 call/cc顺利返回，且返回值就是f顺利执行完毕后的返回值。这种情况下x没有起作用。还有一种情况，就是 f 在执行过程中，调用了 x,比如，执行了 (x p)。那么，整个call/cc表达式在x被调用时就立即停止执行，程序跳转到x这个continuation所代表的现场继续执行。
- x所代表的现场是，等待一个值 v ,然后 brabrabr ...那么，v的值就是调用x时的参数p，然后程序就 brabrabra.....

补充: call/cc和continuation

```
(call/cc (lambda (c) (+ 2 1)))
;=> 3
```

求 call/cc这个表达式的值

call/cc所调用的那个函数 f,就是那个 (lambda (c) (+ 2 1)) 了。调用f时的参数c,就是call/cc表达式被执行前那一刻的continuation。这个continuation代表的现场,当然就是“等待一个值v，然后返回v这个值,然后啥都不干”。

这个被等待的v，在正常情况下应该就是 call/cc表达式的返回值。在f执行过程中，没有调用c，所以 call/cc表达式正常结束，返回值就是f的返回值，即3

整个计算过程中，continuation c没有起作用，就和计算一个普通表达式的过程一样。

补充: call/cc和continuation

```
(call/cc (lambda (c) (+ (c 2) 9)))
;=> 2
```

这个和第一个例子的不同之处在于，在 f 执行过程中，以 2 为参数调用了c这个contiuation。于是f的执行立即停止，不会再做后续的 + 9操作，f也没什么返回之说了。程序挟2这个值直接跳转到c所描述的现场：

这个现场就是 “等待一个值v，然后返回v这个值,然后啥都不干” 。于是v的值就是2，然后返回2

补充: call/cc和continuation

```
(+ 2
  (call/cc
     (lambda (k)
       (* 50 (k 4)))))
```

=> ?

补充: call/cc和continuation

```
(+ 2
  (call/cc
     (lambda (k)
       (* 50 (k 4)))))
;=> 6
```

这里面的 continuation k, 所代表的现场是: “等待一个值 v,然后把 v和2相加的值返回” 。当计算 (k 4)时，程序立即挟值4跳转到k所代表的现场，v即是4。 (+ 2 4) = 6, 于是最终结果就是6

补充: call/cc和continuation

```
(define spot 1234)

(+ 1 (call/cc
        (lambda (k)
          (set! spot k)
          2))) ;=>3

(spot 23)   ;=> ?
(spot 26)   ;=> ?
```

补充: call/cc和continuation

```
(define spot 1234)
(+ 1 (call/cc
        (lambda (k)
          (set! spot k)
          2))) ;=>3
(spot 23)   ;=> 24
(spot 26)   ;=> 27
```

(+ 1 ..)这个表达式执行的时候，k这个continuation 描述的现场是： " 等待一个值 v,然后把 v和1相加的值返回"

在这个表达式执行过程中，没有调用 k,所以表达式正常执行，返回值是3。但是把k这个continuation存到了变量 spot 里面

(spot 23)调用了spot这个continuation,这导致程序挟 23这个值进入spot所代表的现场，即 "等待一个值 v,然后把 v和1相加的值返回".于是 v的值就是23，整个结果就是 24

补充: call/cc和continuation

```
(define c #f)
(define (test2 x)
    (if (call/cc (lambda (k)
            (set! c k) #t))
        (quote ())
        (cdr x)))

(test2 '(1 2)) ;=> ?
(c #f) ;=> ?
```

补充: call/cc和continuation

```
(define c #f)
(define (test2 x)
    (if
        (call/cc (lambda (k)
            (set! c k) #t))
        (quote ()) (cdr x))
)
(test2 '(1 2)) ;=> '()
(c #f) ;=> '(2)
```

continuation可以用来实现 break

```
(define (test element cc)
  (if (zero? element)
      (cc 'found-zero) ; non-local exit
      (void)))
(define (search-zero tst lst)
  (call/cc
   (lambda (return)
     (for-each (lambda (element)
                 (tst element return)
                 (printf "~a~%" element)) ; print
               lst)
     #f))
  (display "end")
  (newline))
(search-zero test '(-3 -2 -1 9 1 2 3)) => ?
(search-zero test '(-3 -2 -1 0 1 2 3)) => ?
```

continuation可以用来实现 break

```
(define (test element cc)
  (if (zero? element)
      (cc 'found-zero) ; non-local exit
      (void)))
(define (search-zero tst lst)
  (call/cc
   (lambda (return)
     (for-each (lambda (element)
                 (tst element return)
                 (printf "~a~%" element)) ; print
               lst)
     #f))
  (display "end")
  (newline))
(search-zero test '(-3 -2 -1 9 1 2 3)) => ?
(search-zero test '(-3 -2 -1 0 1 2 3)) => ?
```

```
-3
-2
-1
9
1
2
3
end
-3
-2
-1
end
```

continuation可以用来实现 return

```
(define (linear-search wanted lst)
  (call/cc (lambda (return)
             (for-each (lambda (k)
                         (if (= k wanted)
                             (return k)
                             (void)))
                       lst)
             #f)))

(linear-search 3 '(1 2 3 4)) =>?
```

continuation可以用来实现 return

```
(define (linear-search wanted lst)
  (call/cc (lambda (return)
             (for-each (lambda (k)
                         (if (= k wanted)
                             (return k)
                             (void)))
                       lst)
             #f)))

(linear-search 3 '(1 2 3 4)) => 3
```