[读书笔记]SICP：8[B]带有通用型操作的系统

本章对应于书中的2.5。

数据导向的思路：

- 确定需要哪些通用操作和数据类型，构建起对应的二维表格`操作名-数据类型`
- 为你想要添加的内容用一个过程包裹，避免其他过程与其名字冲突，然后在过程内完成对应想要注册的过程
- 放置的表格的行是对应的通用操作名，表格的列是预计会接收到的数据标识。
- 使用`put`将其放到表格中对应的位置。**注意：**产生数据的过程要在结果添加类型标识。

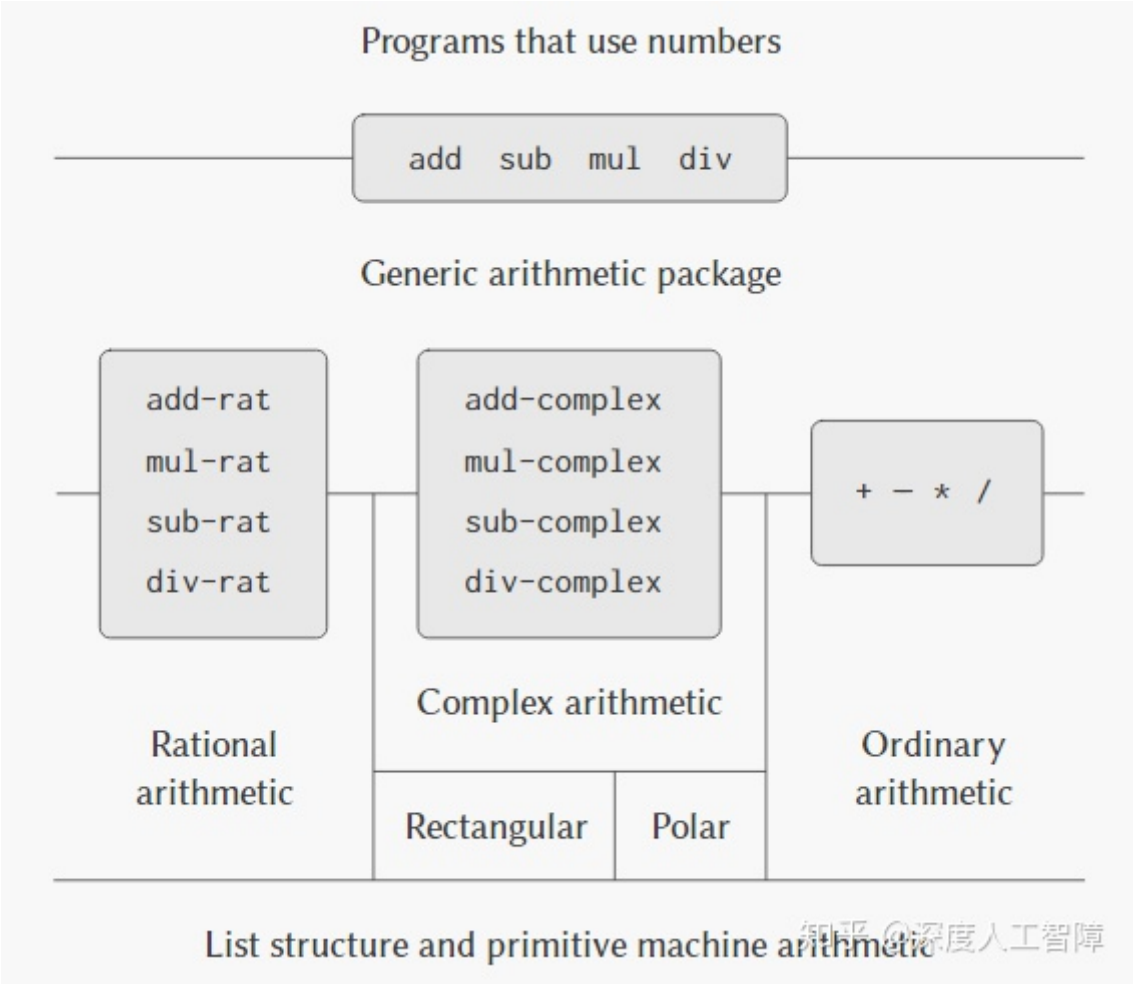
数据导向主要函数：

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args))) ; 根据传入数据获得类型标识符
    (let ((proc (get op type-tags))) ; 从二维表中获得对应的操作
      (if proc ; 如果存在操作
          (apply proc (map contents args)) ; 需要获取数据内容
          (error "No method for these types: APPLY-GENERIC"
            (list op type-tags))))))
```

该函数根据`op`参数选择表格中对应的行，然后根据`args`中的数据类型表来在表格中选择对应的操作，然后将操作应用到数据内容上。是对标签剥离的过程，且操作过程接收到的内容是不包含数据类型的，所以可以直接用原始的过程注册。

- 要注意类型标识的修改
- 使用`apply-generic`调用内部过程时已将类型标识剥离，所以内部过程只考虑数据内容
- 想要实现数据对象的不同表示形式，只需要在安装包中实现对应的选择函数和构造函数，然后将其配置到表格中，其余与表示形式无关的过程，可以不配置到表格中。

上一节中介绍的多重表示的数据关键思想就是通过通用界面过程，将描述数据操作的代码连接到数据的不同表示上。本章将用数据导向技术构造一个算数运算包，将前面构造出来的有理数算数、复数算数和常规算数包含进来，得到在不同表示上的通用操作，且能定义针对不同参数种类的通用型操作。用户通过通用界面`add`、`sub`、`mul`和`div`就能进行算数运算，并且该系统具有可加性



1 通用型算数运算符

我们使用数据导向技术来是像通用型算数运算符`add`、`sub`、`mul`和`div`，当注册好表格中的过程后，我们可以定义下一通用型算数运算符

```
(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))
```

其中`apply-generic`就是上一章中定义的过程，可以根据传入的操作名以及数据的类型标识在表格中找到合适的操作应用于数据内容。

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args))) ; 根据传入数据获得类型标识符
    (let ((proc (get op type-tags))) ; 从二维表中获得对应的操作
      (if proc ; 如果存在操作
          (apply proc (map contents args)) ; 需要获取数据内容
          (error "No method for these types: APPLY-GENERIC"
            (list op type-tags))))))
```

通过上述定义，意味着我们要构建以下二维表格的内容

	scheme-number	rational	complex
add			
sub			
mul			
div			

首先，对于常规数的包为

```
; 常规数
(define (install-scheme-number-package)
  ;;直接注册
  (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)))
  'done)
```

其中可以修改`attach-tag`、`type-tag`和`contents`为如下形式

```
(define (attach-tag type-tag contents)
  (if (eq? 'scheme-number type-tag)
      contents
      (cons type-tag contents)))
(define (type-tag datum)
  (cond ((number? datum) 'scheme-number)
        ((pair? datum) (car datum))
        (else (error "Wrong datum -- TYPE-TAG" datum))))
(define (contents datum)
  (cond ((number? datum) datum)
        ((pair? datum) (cdr datum))
        (else (error "Wrong datum -- CONTENTS" datum))))
```

此时常规数就不会表示成`('scheme-number x)`的形式，就能直接用Scheme中对常规数的操作了。其次可以将有理数的操作注册到表格中

```
; 有理数运算操作
(define (install-rational-package)
  ;; 内部过程
  ;;; 构造函数
  (define (make-rat n d)
    (let ((g (gcd n d)))
      (cons (/ n g) (/ d g))))
  ;;; 选择函数
  (define (numer rat) (car rat))
  (define (denom rat) (cdr rat))
  ;;; 基本过程
  (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))))
  (define (div-rat x y)
    (make-rat (* (numer x) (denom y))
              (* (numer y) (denom x))))
  ;; 注册
  ;;; 添加标签
  (define (tag x) (cons 'rational x))
  ;;; 注册
  (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)
```

最终可以将复数操作注册到表格中。这里复数操作含有两种表示方法，我们先根据上一章注册好通用过程`real-part`、`imag-part`、`magnitude`和`angle`，此时就能通过这四个通用过程来访问任意表示的数据的内容，对应的代码为

```
 ; 复数操作
;; 首先定义不同表示上的通用操作real-part、imag-part、magnitude和angle，对应表格为
;;           rectangular   polar
;;real-part
;;imag-part
;;magnitude
;;angle
;; 为了避免名字冲突，用两个函数来分别注册
(define (install-rectangular-package)
  ;; 构造函数
  (define (make-from-real-imag x y) (cons x y))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))
  ;; 选择函数
  (define (real-part x) (car x))
  (define (imag-part x) (cdr x))
  (define (magnitude x)
    (sqrt (+ (square (real-part x))
             (square (imag-part x)))))
  (define (angle x)
    (atan (imag-part x) (real-part x)))
  ;; 注册
  (define (tag x) (cons 'rectangular x)) ;用rectangular表示直角坐标表示的负数
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (x y) (tag (make-from-mag-ang x y))))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
```

```
  'done)
(define (install-polar-package)
  ;; 构造函数
  (define (make-from-mag-ang r a) (cons r a))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y))) (atan y x)))
  ;; 选择函数
  (define (magnitude x) (car x))
  (define (angle x) (cdr x))
  (define (real-part x)
    (* (magnitude x) (cos (angle x))))
  (define (imag-part x)
    (* (magnitude x) (sin (angle x))))
  ;; 注册
  (define (tag x) (cons 'polar x)) ;用polar表示直角坐标表示的负数
  (put 'make-from-mag-ang 'polar
       (lambda (x y) (tag (make-from-mag-ang x y))))
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  'done)
(install-rectangular-package)
(install-polar-package)
(define (magnitude x)
  (apply-generic 'magnitude x))
(define (angle x)
  (apply-generic 'angle x))
(define (real-part x)
  (apply-generic 'real-part x))
(define (imag-part x)
  (apply-generic 'imag-part x))
```

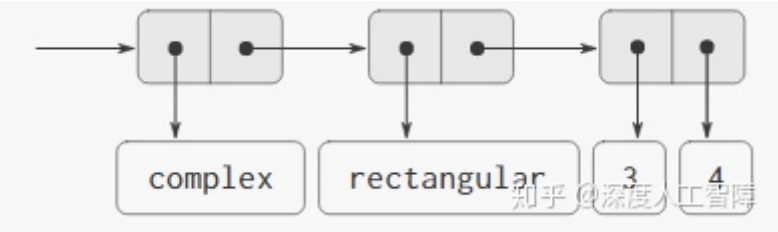
在注册完在不同类型上的通用操作后，就能注册在复数上算数运算了

```
;; 注册在复数上的算数运算
(define (install-complex-package)
  ;;; 构造函数 可以直接获得注册好的函数
  (define (make-from-real-imag x y)
    ((get 'make-from-real-imag 'rectangular) x y)) ;这里还没加上标签，只能用get
  (define (make-from-mag-ang x y)
    ((get 'make-from-mag-ang 'polar) x y))
  ;;; 基本过程
  (define (add-complex x y)
    (make-from-real-imag (+ (real-part x) (real-part y))
                         (+ (imag-part x) (imag-part y))))
  (define (sub-complex x y)
    (make-from-real-imag (- (real-part x) (real-part y))
                         (- (imag-part x) (imag-part y))))
  (define (mul-complex x y)
    (make-from-mag-ang (* (magnitude x) (magnitude y))
                       (* (angle x) (angle y))))
  (define (div-complex x y)
    (make-from-mag-ang (/ (magnitude x) (magnitude y))
                       (/ (angle x) (angle y))))
  ;;; 注册
  (define (tag x) (cons 'complex x))
  (put 'add '(complex complex)
```

```
       (lambda (x y) (tag (add-complex x y))))
  (put 'sub '(complex complex)
       (lambda (x y) (tag (sub-complex x y))))
  (put 'mul '(complex complex)
       (lambda (x y) (tag (mul-complex x y))))
  (put 'div '(complex complex)
       (lambda (x y) (tag (div-complex x y))))
  (put 'make-from-real-imag 'complex
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'complex
       (lambda (x y) (tag make-from-mag-ang x y)))
  )

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

可以发现，这里的一个复数包含了两个类型标识，比如`3+4i`表示为



注意：在较为复杂的系统中会有多个类型标识，需要注意当前最外侧的类型标识是什么。

2 类型之间的关系

我们上面的操作将所有类型相互隔离开来，使得常规数只能与常规数计算、有理数只能与有理数计算，以及复数只能与复数计算。但是其实这些类型的数据能够相互计算，此时就需要考虑跨过类型界限的操作。

2.1 强制类型转换

我们不可能在每个类型下注册与其他类型在所有操作下的计算，另一种方法是考虑**强制类型转换**，比如要计算常规数与复数的混合算数，可以将常规数转换成复数，就转换成了复数的运算问题了。基于这种想法，只需要注册类型之间的强制类型转换方法，无需注册具体的操作，简化了很多工作。

假设这里有一个特殊的强制表格，我们可以将上述的强制类型转换过程注册到该表格中

```
(define (scheme-number->complex x)
  (make-complex-from-real-imag (content n) 0))
(put-coercion 'scheme-number 'complex scheme-number->complex)
```

然后可以修改上面的`apply-generic`过程

```
 (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))
```

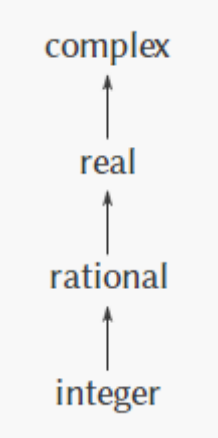
```
                    (t2->t1 (get-coercion type2 type1)))
                (cond (t1->t2 (apply-generic op (t1->t2 a1) a2))
                      (t2->t1 (apply-generic op a1 (t2->t1 a2)))
                      (else (error "No method for these types" (list op type-
tags))))))
            (error "No method for these types" (list op type-tags)))))))
```

它首先根据传入的数据的类型标志`type-tags`判断表格中是否存在对应的操作，如果不存在，则获得两个数据的标志`type1`和`type2`，然后通过`get-coercion`在强制表格中搜索是否存在对应的两个方向的强制类型转换过程，如果有，则对数据进行强制类型转换后，再执行`apply-generic`。

2.2 类型的层次结构

以上方法强制类型转换方法只尝试了类型1转换为类型2，以及类型2转换为类型1，可能都不存在对应的强制类型转换方法，但是可能存在将两个类型都转换到类型3的方法，所以需要建立系统时利用类型之间的进一步结构。

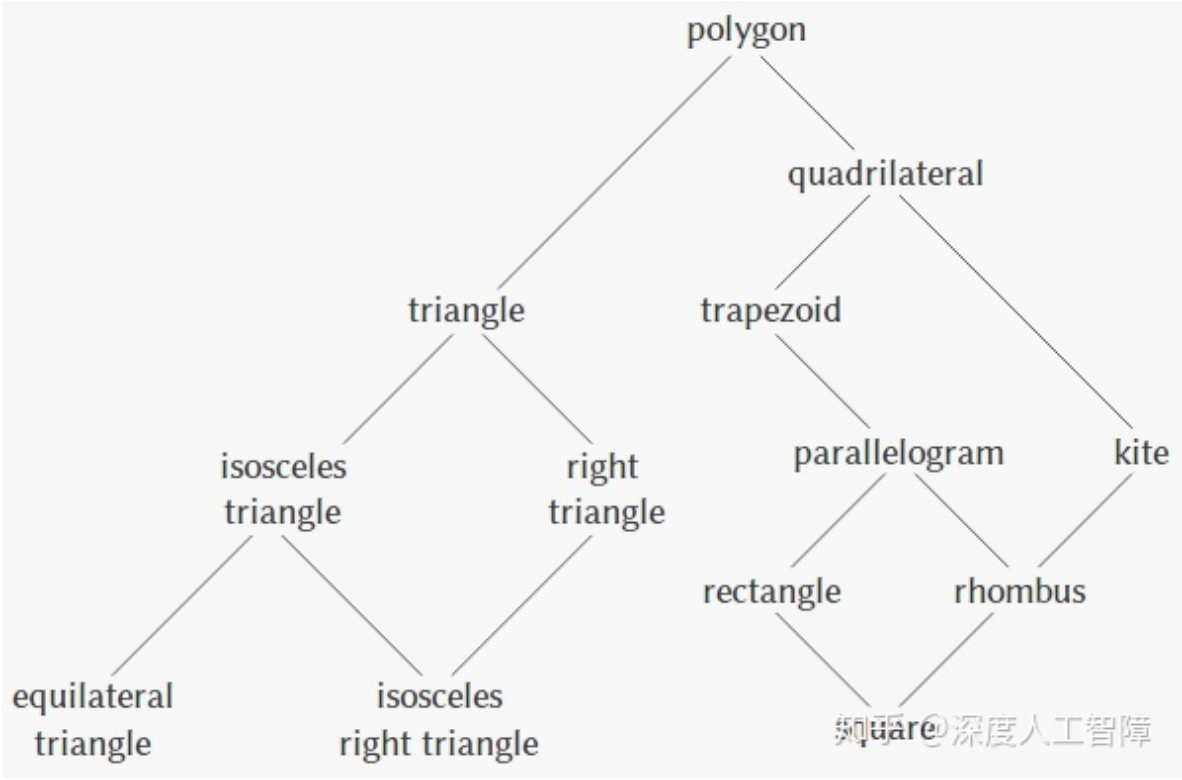
假设我们想要构建一个通用型的算数系统，其中存在整数、有理数、实数和复数，其实这些数据存在一个**类型的层次结构**，整数是特殊的有理数、有理数是特殊的实数、实数是特殊的复数，此时下一层的类型称为上一层类型的**子类型**，上一层类型称为下一层类型的**超类型**，构建起来的层次结构也是最简单的**类型塔**。



类型塔具有以下**优点**:

- 将一个新类型加入层次结构十分简单，只要刻画清楚该类型如何嵌入到它的超类型，以及如何作为子类型的超类型
- 为每个类型都注册一个`raise`过程来将当前类型提升到超类型，此时就能修改`apply-generic`过程，当系统遇到需要对两个不同类型的运算时，就逐步提升较低类型直到所有对象都处于同一层次
- 每个类型都能集成其超类型中定义的所有操作，此时修改`apply-generic`过程，当某个类型中没有注册对应的操作时，就对其进行提升直到有对应的操作。如果到达塔顶还是没有对应的操作，就放弃。
- 可以通过下降类型来得到最优的表达形式

但是在大型系统中的类型层次结构就不是简答的类型塔了，可能是以下形式



此时如何进行类型提升以及类型下降，就是比较困难的任务了。

3 实例

3.1 多项式算数

这里考虑代数演算系统中的多项式算数，其中将多项式定义为项的和式，而每个项是一个系数、变量的乘方或系数与变量的乘积，而系数也可以定义为一个代数表达式，但不依赖于该多项式的变量。比如 $5x^2+3x+7$ 为x的多项式，$(y^2+1)x^3+(2y)x+1$ 为x的多项式，但系数是y的多项式。我们这里只考虑加法和乘法，并要求参与运算的两个多项式具有相同的变量。

接下来**按照数据抽象的方法构建该系统**：

- **步骤一**：假设已经获得多项式的构造函数`make-poly`和选择函数`variable`和`term-list`，其中`variable`获得多项式的变量，`term-list`获得多项式的项。（由此来独立数据对象的具体实现）

此时可以定义多项式的加法和乘法

```
(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))))

(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))))
```

- **步骤二**：对于会产生结果的过程要添加类型标志，将其`put`到表格中，才能通过通用操作来处理多项式运算。

```
(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⟩

  (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)) ;类型标志
  (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)
```

此外，对于项表的操作，也需要先假设一系列过程，由此独立项表的具体实现，这里假设具有以下过程：

- `the-empty-termlist`：返回空项表
- `adjoin-term`：构造函数，将新项加入项表中
- `empty-termlist?`：谓词，检查项表是否为空
- `first-term`：选择函数，提取向表中最高次的项
- `rest-terms`：选择函数，返回除最高次以外的其他项
- `make-terms`：构造函数，通过给定的次数和系数构建出一个项
- `order`：选择函数，返回项的次数
- `coeff`：选择函数，返回项的系数

由此我们就可以实现上面的`add-terms`和`mul-terms`过程了

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

注意：这里`add-terms`和`mul-terms`中都是用了通用操作`add`和`mul`来处理系数的计算，**具有以下好处：**

- 该系数就可以是运算系统中实现的任何数据类型，比如有理数、复数等等，其计算过程都包含在通用操作`add`和`mul`中了。
- 当我们将`add-poly`和`mul-poly`配置到表格中时，`add`和`mul`也就支持多项式的加法和乘法了，此时就能处理变量的系数本身就可以是一个多项式的情况了，比如 $(y^2+1)x^3+(2y)x+1$ ，这种称为**递归数据抽象。**

- **步骤三：**实现完基础过程后，需要考虑数据的表示方式，由此实现对应的构造函数和选择函数。如果数据对象需要多种表示方式，则需要构建通用选择函数。

由于这里可以将项表看成以项的次数作为键值的系数集合，且需要按照次数进行排序，所以可以通过有序集合的形式来实现。但是多项式有两种表示形式：稠密的和稀疏的。 $x^{100}+2x^2+1$ 以稀疏的表示形式为`((100 1) (2 2) (0 1))`，需要配置其选择函数和构造函数：

```
;这里只要保证调用adjoin-term的过程，比如add-terms或mul-terms，总用比表中的项次数更高的项进行调用，就是排序好的了
; 稀疏表示
(define (install-sparse-package)
  ;; 构造函数
  (define (adjoin-term term term-list)
    (if (=zero? (coeff term))
        term-list
        (cons term term-list)))
  ;; 选择函数
  (define (first-term term-list) (car term-list))
  (define (rest-terms term-list) (cdr term-list))
  ;; 注册
  (define (tag x) (attach-tag 'sparse x))
  (put 'adjoin-term 'sparse
       (lambda (term term-list) (tag (adjoin-term term term-list))))
  (put 'first-term '(sparse) first-term)
  (put 'rest-terms '(sparse) rest-terms)
  'done)
```

这里对于每个项，使用通用的结构`（系数 次数）`，无关具体实现，所以项的构造函数和选择函数为

```
(define (make-term order coeff)
  (list order coeff))
(define (order term) (car term))
(define (coeff term) (cadr term))
```

同时也需要配置以稠密形式表示的构造函数和选择函数

```
; 稠密表示
(define (install-dense-term-list)
  ;; 构造函数
  (define (adjoin-term term term-list)
    (cond ((=zero? (coeff term)) term-list)
          ((=equ? (order term) (length term-list)) (cons (coeff term) term-list))
          (else (adjoin-term term (cons 0 term-list)))))
  (define (first-term term-list)
```

```
    (make-term (- (length term-list) 1) (car term-list)))
  (define (rest-terms term-list) (cdr term-list))
  ;; 注册
  (define (tag x) (attach-tag 'dense x))
  (put 'adjoin-term 'dense
       (lambda (term term-list) (tag (adjoin-term term term-list))))
  (put 'first-term '(dense) first-term)
  (put 'rest-term '(dense) rest-terms)
  'done)
```

由此就能得到无关数据表示方式的通用选择函数和构造函数

```
; 通用构造函数和选择函数
(define (adjoin-term term term-list)
  ((get 'adjoin-term (type-tag term-list)) term (contents term-list)))
(define (first-term term-list) (apply-generic 'first-term term-list))
(define (rest-term term-list) (apply-generic 'rest-term term-list))
```