[读书笔记]SICP：7[B]抽象数据的多重表示

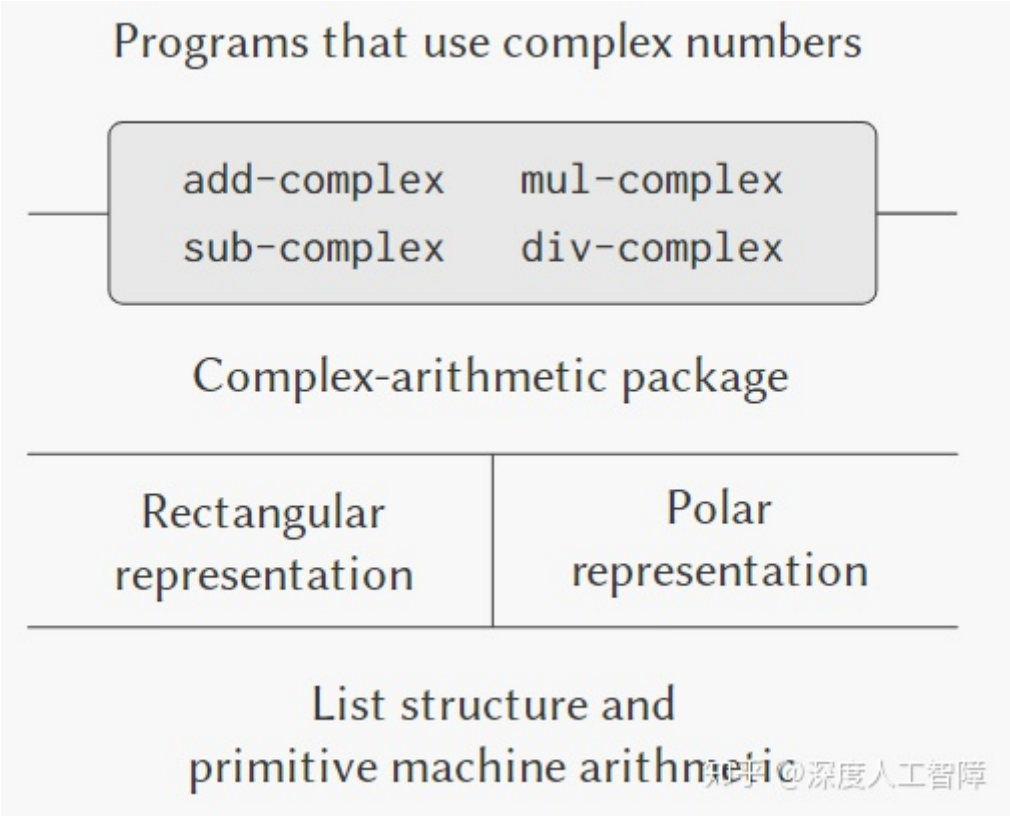
本章对应于书上的2.4。

用来构建在不同表示方式下的通用选择函数。

之前介绍的抽象数据通过设计构造函数和选择函数，从而在抽象数据的使用与具体表示上插入了抽象屏障，也更进一步构造起高阶过程，使得程序的大部分描述能与程序所操作的数据对象的具体表示的选择无关，同时也降低了复杂性。

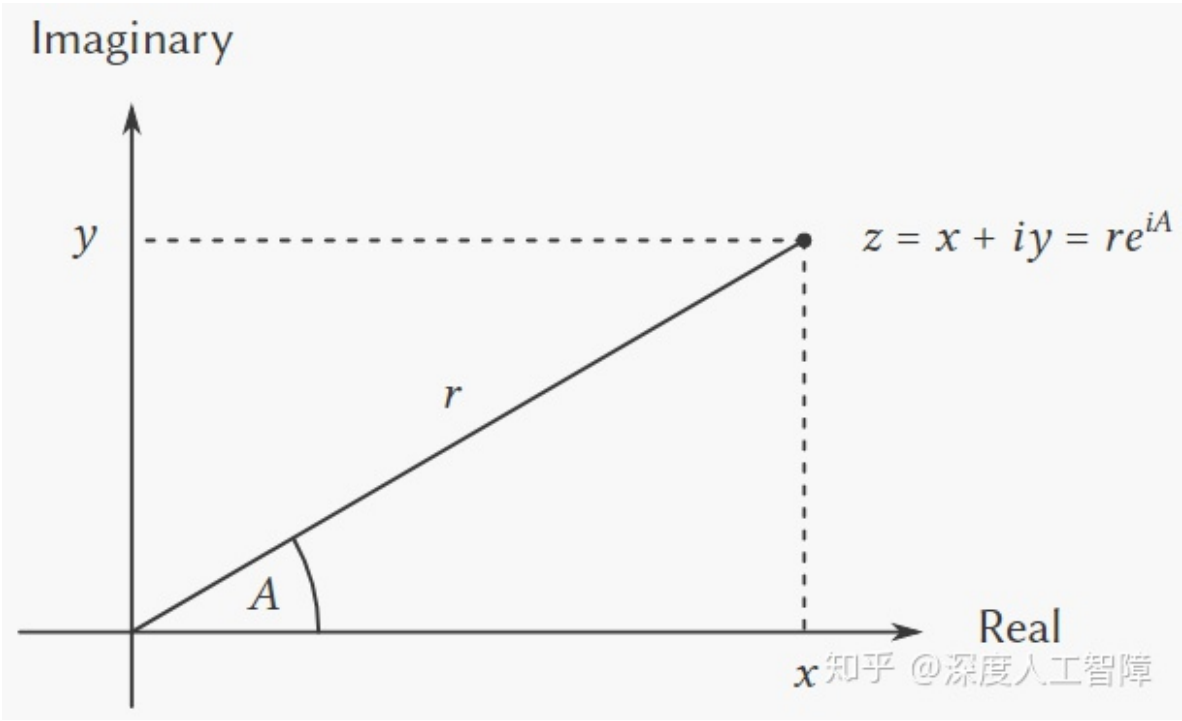
但是这里我们只采用了一种表示方法来表示抽象数据，其实一个数据对象在不同情况下可能需要不同的表示方法，所以希望能构建起同时处理多种表示方法的系统，此时不仅需要将使用和表示隔离开来，还需要隔离开不同的表示方法，使其能在一个系统中共存。这种可以在不止一种表示上操作的过程称为**通用过程**，这里主要是让它们在带有**类型标志**的抽象对象上工作，将采用**数据导向**的方式构建使用了通用过程的系统。

下面将以复数的直角坐标表示和极坐标表示为例来构建一个通用过程



1 复数表示

复数有直角坐标表示和极坐标表示，其中直角坐标表示适合复数加减操作，而极坐标表示适合复数乘除表示，所以对于不同操作，可以采用不同的表示方法。



这里假设已经提供了选择函数`real-part`、`imag-part`、`magnitude`和`angle`，也提供了构造函数`make-from-real-imag`和`make-from-mag-ang`，由此我们可以如下完成对应的复数操作

```
 (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))))
```

接下来就要考虑这两种表示方法的具体实现了。首先，如果在系统中使用直角坐标表示，则对应的选择函数和构造函数如下

```
; 构造函数
(define (make-from-real-imag x y) (cons x y))
;; 由于采用直角坐标表示，所以如果传入模和幅角，要将其转换为实部和虚部
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))
; 选择函数
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
;; 由于采用直角坐标表示，所以可以通过三角关系转换为对应的模和幅角
(define (magnitude z)
  (sqrt (+ (square (real-part z))
           (square (imag-part z)))))
```

```
(define (angle z)
  (atan (imag-part z) (real-part z)))
```

而如果系统使用极坐标表示，则对应的选择函数和构造函数如下

```
; 构造函数
(define (make-from-mag-ang r a) (cons r a))
;; 由于采用极坐标表示，所以如果传入实部和虚部，要将其转换为模和幅角
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
; 选择函数
(define (magnitude z) (car z))
(define (angle z) (cdr z))
;; 由于采用极坐标表示，所以可以通过三角关系转换为对应的实部和虚部
(define (real-part z)
  (* (magnitude z) (cos (angle z))))
(define (imag-part z)
  (* (magnitude z) (sin (angle z))))
```

可以看到，我们可以在内部使用一种表示方法来表示复数，只需要在需要不同表示方法时进行转换。并且借助于选择函数和构造函数构建起的抽象屏障，使得在需要另一种表示方法时才进行变换。

2 带标志的数据

更进一步，这里可以采用**最小允诺原则**，在设计完选择函数和构造函数并决定同时使用直角坐标表示和极坐标表示后，仍维持所用表示方法的不确定性。上一节中如果使用直角坐标表示就将数据全部表示为直角坐标形式，如果使用极坐标表示就将数据全部表示为极坐标表示，而这里的数据所用的表示方法是不确定的。为此需要将极坐标形式的数据和直角坐标形式的数据分隔开来，这里可以在每个复数中都包含一个类型标志部分，用来表示当前复数采用的表示方法，此时就能根据这个类型标志采用对应的选择函数了。

为了操作带类型标志的符号，需要额外引入以下过程

```
; 将类型标志和数据内容组合起来
(define (attach-tag type-tag contents)
  (cons type-tag contents))
; 获得类型标志
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum: TYPE-TAG" datum)))
; 获得数据内容
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum: CONTENTS" datum)))
```

此时我们就能根据复数数据的类型标志来判断对应的标识方式

```
; 判断是否为直角坐标表示
define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
; 判断是否为极坐标表示
(define (polar? z)
  (eq? (type-tag z) 'polar))
```

此时由于当前系统中同时存在两种表示方法的数据，所以需要包含以上两种表示方法的过程，此时它们的过程名都要添加标识来说明适用的表示方法，且在构造函数中需要将类型标志添加进去

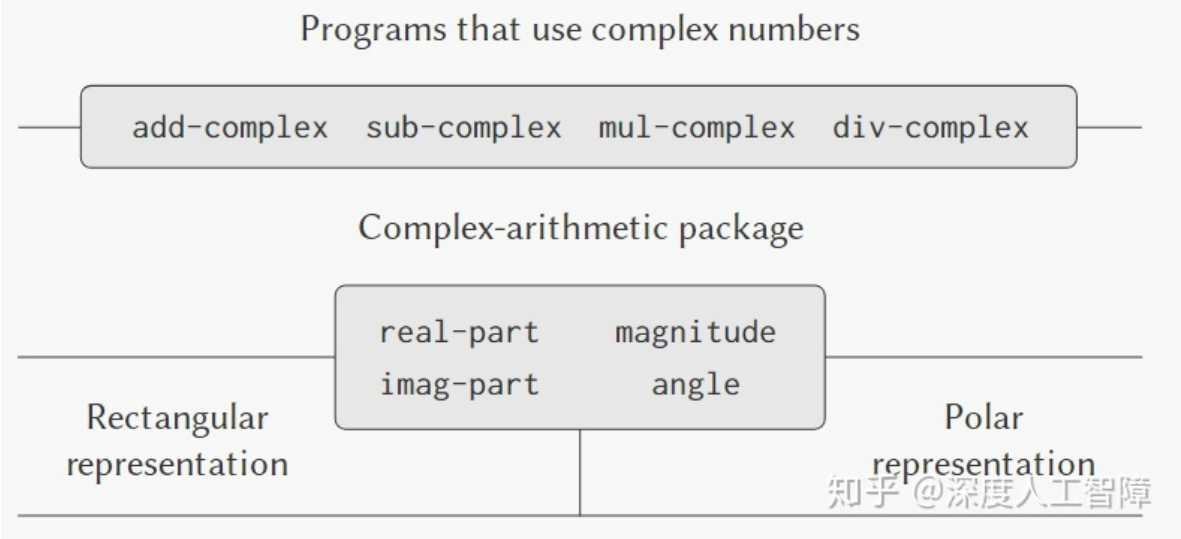
```
; 直角坐标表示
;; 构造函数
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang-rectangular r a)
  (attach-tag
   'rectangular
   (cons (* r (cos a)) (* r (sin a)))))
;; 选择函数
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular 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)))
; 极坐标表示
;; 构造函数
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))
(define (make-from-real-imag-polar x y)
  (attach-tag
   'polar
   (cons (sqrt (+ (square x) (square y)))
         (atan y x))))

;; 选择函数
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (real-part-polar z)
  (* (magnitude-polar z)
     (cos (angle-polar z))))
(define (imag-part-polar z)
  (* (magnitude-polar z)
     (sin (angle-polar z))))
```

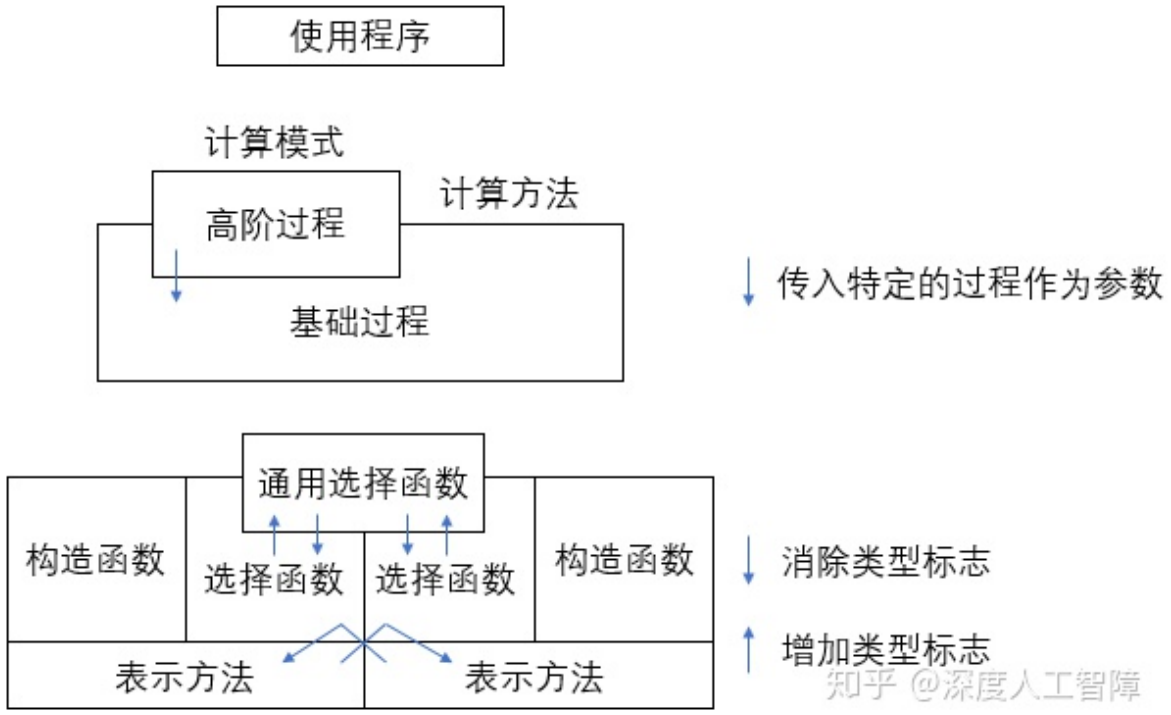
基于以上两种表示方法的过程，可以构建**通用过程**，它会通过数据的类型标志来判断是采用哪种表示方法的过程

```
(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))))
```

此时的`real-part`、`imag-part`、`magnitude`和`angle`就是通用界面过程，基于该选择函数的那些过程就能在不同表示方法的数据上进行操作。并且能根据传入的数据是什么表示类型的，就选择对应表示类型的构造函数。由此构建的复数系统的结构变为如下形式



此时分成了独立的三部分：复数算数运算、极坐标表示和直角坐标表示。我们这里使用了通用选择函数来根据数据的类型标志调用对应表示的选择函数。而这里两者的界面就是类型标志，在底层和高层之间传输时，需要添加和删除类型标志。



抽象屏障

> **综上：**在1中介绍的方法约束了系统中的数据只能有一种表示方法，在需要的时候通过选择函数来转换表示方法。而在2中通过在数据中引入类型标志，提取出了一个通用选择函数，使得系统中的数据能有不同的表示方法。

这种通过检查某个数据项的类型来调用某个适当的过程称为**基于类型的分派**，但是这类方法有两个缺点：

- 通用型界面过程中需要对每种表示方式进行判断，并对这种表示形式选择对应的过程。当对表示方式进行添加删除时，都要修改这些通用界面过程。
- 所有表示方式的过程的名字都不能重复。

这个缺点的根本原因在于实现通用界面过程的技术是**不可加性**的。

3 数据导向的程序设计和可加性

我们可以看到，以上的通用界面过程其实可以统一组织为一个二维表格，其中一维表示各个通用界面操作，另一维为各个表示方法，而表格中的项目就是一系列过程。比如以上构建的复数系统的表格如下

Operations	Types	
	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

知乎@深度人工智障

如果我们能把界面实现为一个过程，然后通过操作明和数据类型的组合在该表格中进行查找，并将查找到的过程应用于参数的内容，此时只需要对表格进行修改就能解决上一节中的问题了。接下来介绍一种能使程序直接利用该表格工作的技术，称为**数据导向的程序设计**，能将系统设计进一步模块化。

为了操作表格，这里假设能使用以下两个操作

```
(put <op> <type> <item>) ; 将<item>放入表中<op>行<type>列
(get <op> <type>)        ; 从表中获得<op>行<type>列的操作
```

此时添加不同的表示方法就类似于通过`put`向表中对应的行列中插入操作，而后通过`get`来获得对应的操作。此时复数的直角坐标表示方法可以表示为以下形式

```
(define (install-rectangular-package)
 ;; 内部构造函数和选择函数
 (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))))
 ;; 插入表中的操作
 (define (tag x) (attach-tag 'rectangular x))
 (put 'real-part '(rectangular) real-part) ; 列名与tag过程中插入的数据标识相同
 (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)
```

首先在`install-rectangular-package`过程中定义直角坐标表示的构造函数和选择函数，由于这些函数过程都是定义在`install-rectangular-package`内部的，所以它们的过程名不会使外部的过程名与其造成冲突，能自由定义。其次在定义完构造函数和选择函数后，需要执行一系列`put`操作将这些过程插入到表格中，注意构造函数要插入对应的数据类型标识，此时获得一个数据时，就能通过它的数据类型标识在表格中进行搜索。

对应的极坐标表示方法的代码为

```
(define (install-polar-package)
```

```
  ;; 内部构造函数和选择函数
  (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)))
  ;; 插入表中的操作
  (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)
```

我们通过执行`install-rectangular-package`和`(install-polar-package)`就能完成通用界面过程的创建，并将其插入二维表中。在使用时，需要通过需要的操作以及传入数据的类型标识来从二维表中提取出对应的过程，再将其应用到输入数据中，可以将其封装为一个统一的界面过程

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

此时就将所有界面过程和全部表示形式都集成到一个二维表中，并可通过一个统一的界面过程`apply-generic`来使用。

同样对上一章中构建的符号求导系统，它的求导过程为

```
(define (deriv exp var)
  (cond ((number? exp) 0) ;第一条求导规则
        ((variable? exp) (if (same-variable? exp var) 1 0)) ;第二条求导规则
        ((sum? exp) (make-sum (deriv (addend exp) var) ;第三条求导规则
                              (deriv (augend exp) var)))
        ((product? exp) (make-sum ;第四条求导规则
                              (make-product (multiplier exp)
                                            (deriv (multiplicand exp) var))
                      (make-product (deriv (multiplier exp) var)
                                    (multiplicand exp))))
        (else (error "unknown expression
                      type: DERIV" exp))))
```

可以发现当添加不同的运算符号时，也需要在该过程中进行修改，这里可以把不同的运算符当做不同的表示方法，将运算符当做类型标识，可以分别将和式和乘式的求导方法插入到二维表中，通过`'deriv`和运算符进行索引。此时的求导过程为

```
 (define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp)) (operands exp) var))))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

而对应的和式和乘式的过程为

```
(define (install-sum-deriv)
  ; 内部函数
  ;; 选择函数
  (define (addend variables) (car variables))
  (define (augend variables) (cadr variables))
  ;; 构造函数
  (define (make-sum a1 a2)
    (cond ((eq? a1 0) a2)
          ((eq? a2 0) a1)
          ((and (number? a1) (number? a2)) (+ a1 a2))
          (else (list '+ a1 a2))))
  ;; 基本过程
  (define (sum-deriv variables var)
    (make-sum (deriv (addend variables) var)
              (deriv (augend variables) var)))
  ; 插入表格
  (put 'deriv '(+) sum-deriv)
  'done)

(define (install-product-deriv)
  ; 内部函数
  ;; 选择函数
  (define (multiplier variables) (car variables))
  (define (multiplicand variables) (cadr variables))
  ;; 构造函数
  (define (make-sum a1 a2)
    (cond ((eq? a1 0) a2)
          ((eq? a2 0) a1)
          ((and (number? a1) (number? a2)) (+ a1 a2))
          (else (list '+ a1 a2))))
  (define (make-product m1 m2)
    (cond ((or (eq? m1 0) (eq? m2 0)) 0)
          ((eq? m1 1) m2)
          ((eq? m2 1) m1)
          ((and (number? m1) (number? m2)) (* m1 m2))
          (else (list '* m1 m2))))
  ;; 基本过程
  (define (product-deriv variables var)
    (make-sum (make-product (multiplier variables)
                            (deriv (multiplicand variables) var))
              (make-product (deriv (multiplier variables) var)
                            (multiplicand variables))))
  ; 插入表格
  (put 'deriv '(*) product-deriv)
  'done)
```

后面需要添加任何运算，只需要将其插入表格中合适的位置就行，无需修改`deriv`过程。

4 信息传递

第2章中通过引入类型标识得到的通用界面过程`real-part`、`imag-part`、`magnitude`和`angle`其实可以看成是对以下表格分解成一行一行，每个通用界面过程适用于所有表示方法的数据。

	Types	
Operations	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

知乎@深度人工智障

其实还有能将该表分解成一列一列，得到能基于操作名获得内容的表示方法。如以下代码

```
(define (make-from-real-imag x y)
  (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)
```

这里的数据使用直接坐标表示，并且将数据表示为一个个过程，此时能将需要的操作作为该数据的参数，就能获得对应的内容。这种方式称为**信息传递**，数据作为实体接收所需操作的名字作为信息。此时的通用过程就变为了

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

注意：这里只能接收一个参数。

同样也能得到用信息传递风格实现的构造函数`make-from-mag-ang`

```
(define (make-from-mag-ang x y)
  (define (dispatch op)
    (cond ((eq? op 'magnitude) x)
          ((eq? op 'angle) y)
          ((eq? op 'real-part) (* x (cos y)))
          ((eq? op 'imag-part) (* x (sin y)))
          (else (error "Unknown op: MAKE-FROM-MAG-ANG" op))))
  dispatch)
```

5 总结

一个带有通用型操作的大型系统中，可能会要加入新的数据对象类型或新的操作，这里介绍了三种策略，对通用界面过程-数据类型二维表格做了不同的处理，来获得不通的通用界面过程：

- **方法一：增加类型标识的通用操作**：对二维表格进行行划分，得到能在不同表示方法中通用的界面选择函数。（在不同表示方法中通用）
- **方法二：信息传递风格**：对二维表格进行列划分，得到能在不同操作上通用的特定表示方法的数据。（在不同操作中通用）
- **方法三：数据导向风格**：将操作插入二维表格中的对应的位置，通过操作名和表示方法从表中获得对应的操作。（在所有操作和表示方法中通用）

当系统需要不断插入新的操作时，方法一需要创建在各个表示方法中通用的新的界面选择函数，无需修改原始代码；而方法二需要在各种表示方法的构建函数中增加一项，需要修改原始代码。当系统需要不断插入行的数据类型时，方法一需要在各个通用界面选择函数中增加一项来处理新的数据类型，需要修改原始代码；而方法二需要创建一个新的数据类型对应的构造函数，无需修改原始代码。

而数据导向风格能在不修改现有操作和数据类型情况下，通过在二维表格中合适的位置插入对应的操作就能添加新的数据和操作了。