情報システム科学実習 II (9)

多相性・総称関数・メソッド

1999 年 12 月 15 日 増原 英彦

1 多相性

異なる種類のデータに対して同じ処理が行えることを多相性(polymorphism) と言う。例として数と行列に対する冪乗の計算を考えてみる。(以下の例は web ページの matrix.scm に定義がある。)

1.1 行列のインターフェイス

行列には次のような構築子・選択子とそれを使った演算があるとする:

- 構築子:
  - `(lists->matrix ((e11 e12 ...) (e21 e22 ...) ...)`: 行のリストから行列を作る。行は要素のリスト。
  - `(vectors->matrix (v1 v2 ...))`: ベクトル (後述) のリストから行列を作る。
- 選択子:
  - `(matrix->lists m)`: 行列の各行を要素のリストにして、それをリストにして返す。`lists->matrix` の逆関数。
  - `(matrix-rows m)`: 行列をベクトルのリストとして返す。
- 述語: `(matrix? x)`: `x` が行列データかどうかを判定する。
- 演算: `(matrix-multiply m1 m2)`: 2 行列の積を求める。

1.2 冪乗のアルゴリズム

冪乗の計算は何度か出てきたが、ここでは次のようなアルゴリズムによって求めるものを定義してみる:

$$x^n = \begin{cases} x & (n = 1) \\ (x^{n/2}) \times (x^{n/2}) & (n \text{ は偶数}) \\ x \times (x^{n-1}) & (\text{それ以外}) \end{cases}$$

これをもとに定義すると以下のようになる:

```
(define (num-square x) (* x x))
(define (num-power x n)
  (cond ((= n 1) x)
        ((even? n) (num-square (num-power x (quotient n 2))))
        (else (* x (num-power x (- n 1))))))
```

行列の冪乗は、乗算の方法が違うだけで、手続きとしてはほとんど同じ形をしている:

```
(define (matrix-square x) (matrix-multiply x x))
(define (matrix-power x n)
  (cond ((= n 1) x)
        ((even? n) (matrix-square (matrix-power x (quotient n 2))))
        (else (matrix-multiply x (matrix-power x (- n 1))))))
```

1.3 関数パラメータによる多相性

数の冪乗も行列の冪乗も積の求め方を除いて同じ定義である。このようなものを共通化する手段として関数パラメータがあった。関数パラメータを使うと次のようになった:

```
(define (fp-square multiply x) (multiply x x))
(define (fp-power multiply x n)
  (cond ((= n 1) x)
        ((even? n)
         (fp-square multiply (fp-power multiply x (quotient n 2))))
        (else (fp-multiply x (fp-power multiply x (- n 1))))))
```

このように定義されたfp-powerのように、パラメータによって同じ関数定義を異なるデータに適用できることをパラメータ多相 (**parametric polymorphism**) という[1]。

2 総称関数

しかし、より複雑な計算を定義しようと思うと、積だけでなく、和や等号、単位元のようなものもパラメータにする必要が出てくるだろう。これは、あまりに繁雑ではないか?

そこで、演算子を引数の種類によって挙動を変えるように定義してみる:

```
(define (multiply x1 x2)
  (cond ((and (matrix? x1) (matrix? x2))
         (matrix-multiply x1 x2))
        ((and (number? x1) (number? x2))
         (* x1 x2))
        (else
         (error "unsupported types:" (list x1 x2)))))
```

これを使って冪乗を書き直せば、関数パラメータが不要になる:

```
(define (square x) (multiply x x))
(define (power x n)
  (cond ((zero? n) x)
        ((even? n)
         (power (square x) (quotient n 2)))
        (else
         (multiply x (power x (- n 1))))))
```

[1]狭い意味では、厳密に型付けされた言語で型の一部分をパラメータにすることによって、同じ定義を異なる型に適用できるようにすることを指す。

このように、引数の種類によって挙動を変える関数のことを総称関数 (**generic functions**) と言う[2]。

3 メソッド

上のようにして総称関数を定義すると、データの種類が増えるたびに定義を変更しなければならない。オブジェクト指向言語では、引数のクラスによって挙動を変える総称関数を自動的に定義する機能がある。STklos では、`define-method` 構文によってメソッド (**methods**) を定義すると、総称関数が自動的に作られる。

例: `matrix-multiply` と`matrix-vector-multiply` を`define-method` を使って書き直すと以下のようになる:

```
;;; Matrix * Matrix = Matrix
(define-method multiply ((m <Matrix>) (n <Matrix>))
  (vectors->matrix
   (map ((curry matrix-vector-multiply) m)
        (matrix-rows (matrix-transpose n)))))

;;; Matrix * Vector = Vector
(define-method multiply ((matrix <Matrix>) (vector <Vector>))
  (list->vector
   (map ((curry vector-dot-product) vector)
        (matrix-rows matrix))))

;;; Number * Number = Number
(define-method multiply ((x <Number>) (y <Number>))
  (* x y))
```

ここでは`multiply` がメソッドの名前、`((m <Matrix>) (n <Matrix>))` が仮引数のリストである。`m` や`n` が仮引数の名前で、`<Matrix>`がクラス名である。

このようにメソッドを定義すると、総称関数`multiply` も自動的に定義される。`multiply` は、引数のオブジェクトのクラスを調べ、適切なメソッドの本体を実行する。Scheme の組み込みデータ型に対してもクラスが用意されているので、数値どうしの積もメソッドとして記述できる。

課題 9-1: (和・スカラー倍)

1. 行列どうしの和、ベクトルどうしの和、数値の和を計算するメソッドをそれぞれ`add` という名前で定義せよ。
2. 行列のスカラー倍、ベクトルのスカラー倍を計算するメソッドをそれぞれ`multiply` という名前で定義せよ。

∎

[2]総称関数と言う場合は、特に引数の全ての種類によって挙動を変えられることを指すこともある。

4　例: 描画エディタ

描画エディタとは、線や円のような図形を編集するプログラムである。このようなエディタには、線・円・四角形などの種々の図形要素をマウスを使って移動したり変形できる機能がある。

このとき、マウス操作によって「変形」という操作を行う部分は図形要素に依らず共通のものである。一方、実際に変形をするためにどのようなことをすべきかは図形要素ごとに異なる。

このような場合、総称関数によって定義を共通化することができる。つまり、図形要素ごとにクラスを定義しておき、変形という操作を総称関数にして、各図形要素ごとに変形をするメソッドを用意すればよい。そうすると、マウス操作を解釈する部分では、図形要素が何であれ共通の定義で変形することができる。

このような描画エディタの簡易版を作ってみよう (web ページ drawing.scm に定義あり)。

4.1　2 次元平面の点

以下の関数があるとする。(これは第 5 回で出てきた。)

```
;;; (make-point x y):    点を作る
;;; (point-x p):         x 座標をとり出す
;;; (point-y p):         y 座標をとり出す
;;; (point-add p1 p2):   点のベクトル和を求める
;;; (point-scale factor p):   座標をスカラー倍する
;;; (point-sub p1 p2):   点のベクトル差を求める
```

4.2　図形要素の外部仕様

図形要素には次のような操作ができるとする:

```
;;; (draw element canvas):
;;;     要素 element を canvas に描く
;;; (move element delta canvas):
;;;     要素 element を delta だけ移動して canvas に描き直す
;;; (resize element delta canvas):
;;;     要素 element を delta だけ変形して canvas に描き直す
```

4.3　描画エディタ

描画エディタは、Tk の canvas オブジェクト、図形要素、マウス操作の開始点を状態として持つようなオブジェクトである:

```
(define-class <DrawingEditor> ()
  ((canvas :init-keyword :canvas        ; Canvas オブジェクト
           :getter canvas)
   (start-point :initform #f            ; ドラッグの開始点
                :getter start-point
                :setter set-start-point)
   (element :initform #f                ; 図形要素
            :getter element
            :setter set-element)))
```

構築子は次のようになる。(bind c "<1>" 関数) は canvas オブジェクトc に対してマウスボタン 1 が押されときに呼び出される関数を登録する。"<B1-Motion>"は、マウスボタン 1 でドラッグしたときに呼びされる関数の登録である。

```
(define (make-editor width height)
  (let* ((c (make <Canvas> :width width :height height))
         (d (make <DrawingEditor> :canvas c)))
    (pack c)
    (bind c "<1>" (click d 1))
    (bind c "<B1-Motion>" (motion d move))
    (bind c "<2>" (click d 2))
    (bind c "<B2-Motion>" (motion d resize))
    d))
```

図形要素を登録すると描画する:

```
(define (add-element drawing-editor element)
  (set-element drawing-editor element)
  (draw element (canvas drawing-editor)))
```

上のbind によって登録される関数は、マウス操作が起きると操作の起きた座標を引数として呼び出される。マウスが最初に押されたときは、その場所を描画エディタオブジェクトの状態に書き込む。click 自身は関数を作る関数であることに注意:

```
(define (click drawing-editor button-number)
  (lambda (x y)
    (set-start-point drawing-editor (make-point x y))))
```

マウスがドラッグされているときは、移動や変形を行う。関数motion は関数を作る関数である。作られた関数が呼び出されると、(1) 移動距離を計算し、(2) 描画エディタ中に登録された図形要素に対して移動・変形といった操作を行う。さらに、(3) 新しいマウスの位置を描画エディタの状態に書き込んで、次の呼び出しに備える。実際に図形要素に何をするかは、押されているマウスボタンによって違うため、関数パラメータaction によって抽象化されている。make-editor の中でmove あるいはresize が与えられている。

```
(define (motion drawing-editor action)
  (lambda (x y)
    (let* ((point (make-point x y))
           (delta (point-sub point (start-point drawing-editor)))
           (element (get-element drawing-editor)))
      (set-start-point drawing-editor point)
      (if element
          (action element delta (canvas drawing-editor))))))
```

4.4 図形要素の内部表現

例として線の内部表現を定義しよう。線は、始点と終点を持つ。また、移動・変形の際に canvas 上に描画された線を消してから描き直すために、描画されたオブジェクトも内部状態になる:

```
(define-class <Stroke> ()
  ((corner1 :init-keyword :corner1 :getter corner1 :setter set-corner1)
   (corner2 :init-keyword :corner2 :getter corner2 :setter set-corner2)
   (drawing-object :getter drawing-object :setter set-drawing-object
                   :initform #f)))
```

描画手続きは、(1) すでに描かれていた場合はまずそれを消す、(2) Tk の<Line>オブジェクトを作る (描画する)、(3) 作られたオブジェクトを覚えておく、の 3 つの操作からなる:

```
(define-method draw ((stroke <Stroke>) canvas)
  (let ((obj (drawing-object stroke)))
    (if obj (destroy obj))              ; erase if drawn before
    (set-drawing-object
     stroke
     (make <Line>
           :coords (points->coords (corner1 stroke) (corner2 stroke))
           :parent canvas))))
```

移動は、始点、終点をそれぞれ与えられた移動距離だけ増やしてやればよい:

```
(define-method move ((stroke <Stroke>) delta canvas)
  (set-corner1 stroke (point-add (corner1 stroke) delta))
  (set-corner2 stroke (point-add (corner2 stroke) delta))
  (draw stroke canvas))
```

変形は、ここでは終点だけを変化させることにする:

```
(define-method resize ((stroke <Stroke>) delta canvas)
  (set-corner2 stroke (point-add (corner2 stroke) delta))
  (draw stroke canvas))
```

課題 9-2: (図形要素)

1. 別の図形要素として四角形のクラス<Box>と、円のクラス<Circle>を定義し、それぞれに適切なメソッドを定義せよ。

   四角・円を Canvas 上に描くには下のようにしてオブジェクトを作る:

   ```
   (make <Rectangle> :parent Canvas :coords (list x1 y1 x2 y2))
   (make <Oval> :parent Canvas :coords (list x1 y1 x2 y2))
   ```

2. 複数の図形要素をグループ化するクラスを定義せよ。例えば移動メソッドは、登録されている全要素をそれぞれ移動させればよいはずである。

5　行列計算の定義

5.1　内部表現

内部表現にはクラスを使い、rows というスロットにベクトルのリストを持っているものとする:

```
;;; 行列の内部表現
(define-class <Matrix> ()       ; クラス定義+選択子
  ((rows :init-keyword :rows :getter matrix-rows)))
(define (vectors->matrix list-of-vectors) ; 構築子
  (make <Matrix> :rows list-of-vectors))
(define (lists->matrix list-of-lists)     ; 構築子
  (vectors->matrix (map list->vector list-of-lists)))
(define (matrix->lists matrix)            ; 選択子
  (map vector->list (matrix-rows matrix)))
(define (matrix? x) (is-a? x <Matrix>))   ; 述語
```

上で使われているlist->vector, vector-elements はそれぞれベクトルの構築子と選択子。また、is-a?は、STklos 組み込みの関数で、あるデータが与えられたクラスオブジェクトかどうかを判定する。

5.2　行列の積

2 つの行列 M, N の積は $M \times N$ は、

$$(M \times N)_i = M(N^{\mathrm{tr}})_i$$

のように定義される。ただし M_i は行列 M の i 行ベクトル、N^{tr} は行列 N の転置行列。これを Scheme で定義すると次のようになる:

```
(define (matrix-multiply m n)
  (vectors->matrix
   (map ((curry matrix-vector-multiply) m)
        (matrix-rows (matrix-transpose n)))))
```

行列 M とベクトル v の積は

$$(Mv)_i = M_i \cdot v$$

のように定義される。ただし、$v \cdot w$ はベクトルの内積。これを Scheme にすると:

```
(define (matrix-vector-multiply matrix vector)
  (list->vector
   (map (lambda (row)
          (vector-dot-product row vector))
        (matrix-rows matrix))))
```

転置行列は、リストの転置によって定義する:

```
(define (matrix-transpose matrix)
  (lists->matrix
   (list-transpose
    (matrix->lists matrix))))
```

リストの転置は、次のように考える:

$$\begin{pmatrix} (e_{00} & e_{01} & \cdots & e_{0N}) \\ (e_{10} & e_{11} & \cdots & e_{1N}) \\ & \vdots & & \\ (e_{M0} & e_{M1} & \cdots & e_{MN}) \end{pmatrix}^{\mathrm{tr}} = \begin{pmatrix} (e_{00}\ e_{10}\ \cdots\ e_{M0}) \\ \begin{pmatrix} (e_{01} & \cdots & e_{0N}) \\ (e_{11} & \cdots & e_{1N}) \\ & \vdots & \\ (e_{M1} & \cdots & e_{MN}) \end{pmatrix}^{\mathrm{tr}} \end{pmatrix}$$

従って Scheme での定義は次のようになる:

```
(define (list-transpose list-of-lists)
  (if (null? (car list-of-lists))
      '()
      (cons (map car list-of-lists)
            (list-transpose (map cdr list-of-lists)))))
```

5.3 ベクトル

- 構築子: `(list->vector (e1 e2 ...))` は要素のリストからベクトルを作る
- 選択子: `(vector->list v)` はベクトルから要素のリストを返す。`list->vector` の逆関数。

  `(vector-element v i)` はベクトル`v` の`i` 番目の要素を返す。

内部表現にはやはりクラスを使う:

```
(define-class <Vector> ()                    ; クラス定義+選択子
  ((elements :init-keyword :elements :getter vector->list)))
(define (list->vector elements)              ; 構築子
  (make <Vector> :elements elements))
(define (vector-element vector index)        ; 選択子
  (list-ref (vector->list vector)))
```

2 ベクトルの内積は、要素どうしの積を足したものなので、各ベクトルをリストにしてから`map` を使って各要素の積を計算し、それを`fold-left` によって和を求める:

```
(define (vector-dot-product v1 v2)
  (fold-left
   + 0 (map * (vector->list v1) (vector->list v2))))
```

課題 9-3: (再び Fibonacci)

フィボナッチ数列

$$\mathrm{Fibonacci(n)} = \begin{cases} 1 & (n = 0, n = 1) \\ \mathrm{Fibonacci}(n-1) + \texttt{Fibonacci}(n-2) & (n \geq 2) \end{cases}$$

は $n \geq 2$ のとき

$$\begin{aligned} \mathrm{Fibonacci}(n+1) &= \mathrm{Fibonacci}(n) + \texttt{Fibonacci}(n-1) \\ &= 2\mathrm{Fibonacci}(n-1) + \texttt{Fibonacci}(n-2) \end{aligned}$$

であるので、

$$\begin{pmatrix} \text{Fibonacci}(n+1) \\ \text{Fibonacci}(n) \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \text{Fibonacci}(n-1) \\ \text{Fibonacci}(n-2) \end{pmatrix}$$

である。従って $n > 1$ のとき:

$$\begin{aligned} \begin{pmatrix} \text{Fibonacci}(2n+1) \\ \text{Fibonacci}(2n) \end{pmatrix} &= \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \text{Fibonacci}(2(n-1)+1) \\ \text{Fibonacci}(2(n-1)) \end{pmatrix} \\ &= \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} \text{Fibonacci}(1) \\ \text{Fibonacci}(0) \end{pmatrix} \end{aligned}$$

である。

これをヒントに、n 番目の Fibonacci 数を計算する関数(`fibonacci n`)を定義せよ。

∎