情報システム科学実習 II (5)

# データの抽象化

1999 年 11 月 17 日 増原 英彦

複合データ (compound data) とは、複数のデータを 1 つのデータとしてまとめあげる手段である。(*cf.* C の `struct`, Pascal の `record`.)

例えばグラフィクス・プログラムでは、2 次元座標系上の位置を表わす「点」を多用する。このような点は、x,y 座標という 2 つの数でもって表わしてもよいが、できれば 1 つの点はプログラム上でも 1 つのデータとして扱いたい。

その際に、データの抽象化を行うことは重要である。

- データが実際にはどのようにようにして作られているか、を意識する世界を内部表現という
- 一方、データを使用する世界では、構築子 (constructor) を使ってデータを作り、選択子 (selector) を使って、データの要素をとり出す

この二つの世界を独立させて扱うことが、データの抽象化である。

## 1 データを使用する世界

例えば、2 次元の点の例では、構築子・選択子として次のものがあればよいだろう:

`(make-point` $x$ $y$`)`: 点 $(x, y)$ を作る

`(get-x` $p$`)`: 点 $p$ の $x$ 座標を返す

`(get-y` $p$`)`: 点 $p$ の $y$ 座標を返す

構築子と選択子が定められれば、内部表現とは独立に使用する世界の操作、つまり 2 次元座標の操作を定義することができる。

例えば、ベクトル和は

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$$

なので、

```
(define (point-add p1 p2)
  (let ((x1 (get-x p1)) (y1 (get-y p1))
        (x2 (get-x p2)) (y2 (get-y p2)))
    (make-point (+ x1 x2) (+ y1 y2))))
```

のように定義できる。

練習 **5-1:** (**2** 次元の点)

2 次元の点に対する操作として、以下のものを定義せよ:

$$
\begin{array}{lcll}
|(x,y)| & = & \sqrt{x^2+y^2} & (\text{原点からの距離}) \\
s(x,y) & = & (sx, sy) & (\text{スカラー倍}) \\
p_1 - p_2 & = & p_1 + (-1 \cdot p_2) & (\text{差}) \\
|p_1, p_2| & = & |p_1 - p_2| & (2\ \text{点間の距離}) \\
(x_1,y_1)\cdot(x_2,y_2) & = & x_1x_2 + y_1y_2 & (\text{内積}) \\
\mathrm{rotate}((\mathrm{x},\mathrm{y}),\theta) & = & (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta) & (\text{回転})
\end{array}
$$

∎

2 次元の点を表示する関数も定義しよう。2 次元グラフィクス・ライブラリは以下のように使う:

```
> (load "grapher.scm")          ; web ページに定義あり
> (define g (make <Grapher> :width 300 :height 300))
> (pack g)                      ; 表示する
> (draw-line g 10 10 200 200)   ; 線を引く
> (set-color! g 255 128 0)      ; 色を RGB で指定する
> (draw-circle g 150 150 30)    ; 半径 30 の円を描く
```

```
(define (plot g p)              ; 点をプロット
  (let ((x (get-x p)) (y (get-y p)))
    (draw-circle g x y 5)))

(define (line g p1 p2)          ; 2 点間に線を引く
  (let ((x1 (get-x p1)) (y1 (get-y p1))
        (x2 (get-x p2)) (y2 (get-y p2)))
    (draw-circle g x1 y1 x2 y2)))
```

課題 **5-1:** (公開鍵暗号)

公開鍵暗号 (のまねごと) を作ってみよう。

1. 鍵は冪乗数と「大きな数」の組みからなる。2 次元座標の例を参考にして、構築子・選択子として必要な関数の名前とその働きを決めて書け。

2. 鍵は 2 つの素数 $p, q$ から作る。$n = pq,\ \phi(p,q) = (p-1)(q-1)$ とする。$e$ が $\phi(p,q)$ と互いに素な数であるとき、$e$ と $n$ の組みを公開鍵になる。$p, q$ から公開鍵を作る関数 (`make-encription-key p q`) を定義せよ。

   なお $e$ は以下のような関数を使って(`find-mutually-indivisible phi 3`) として 3 から順に愚直に求めてゆくことにする。

```
(define (find-mutually-indivisible phi n)
  (if (or (divisible? phi n) (divisible? n phi))
      (find-mutually-indivisible phi (+ n 1))
      n))
```

3. 一方、$p, q$ と公開鍵における $e$ が与えられたときに、$\bmod \phi(p,q)$ での $e$ の逆数を $d$ とする。つまり、$ed \bmod \phi(p,q) = 1$ である。秘密鍵は $d$ と $n$ の組みである。$p, q$ と公開鍵から秘密鍵を作る関数(`make-decription-key p q encription-key`)を定義せよ。

   なお $d$ は以下のような関数を使って(`find-inverse-mod e 1 phi`)のようにして 1 から順に求めてゆくことにする。

```
(define (find-inverse-mod e n phi)
  (if (= (modulo (* e n) phi) 1)
      n
      (find-inverse-mod e (+ n 1) phi)))
```

4. ある数 $x$ の鍵 $(e, n)$ による暗号化は $x^e \bmod n$ である。また、暗号化された数 $c$ の復号は対になっている鍵 $(d, n)$ を使って $c^d \bmod n$ である。これらを行う関数(`encript number key`)を定義せよ。

∎

# 2 内部表現 — 対 (pair)

複合データの内部表現を考える。Scheme には、複合データを作るために対 (pair) というものがあり、そのために次のような原始的な関数が用意されている。

(`cons` $x$ $y$): 2 つの引数から対を作り、1 つのデータとして返す

(`car` $p$): 対の 1 つ目の値をとり出す

(`cdr` $p$): 対の 2 つ目の値をとり出す

これらの関数を使うことで、複数のデータを一つの値として扱うことができる。

```
> (define x (cons 1 2))    ; 1 と 2 の対を作る
> (car x)                  ; 対の 1 番目
1
> (cdr x)                  ; 対の 2 番目
2
> (define y (cons 3 x))    ; 3 と x の対を作る
> (car y)                  ; 対の 1 番目
3
> (car (cdr y))            ; 対の 2 番目 (つまり対 x) の 1 番目
1
```

この対を使って、分数の内部表現を定義すると、次のようになる:

```
(define (make-point x y) (cons x y))
(define (get-x p) (car p))
(define (get-y p) (cdr p))
```

これとデータを使用する世界とを組み合わせると、2 次元の点が確かに扱えるようになる:

```
(define p1 (make-point 150 50))
(set-color! g 255 0 0)
(plot g p1)
(define p2 (make-point 50 150))
(set-color! g 0 255 0)
(plot g p2)
(define p3 (point-add p1 p2))
(set-color! g 0 0 255)
(plot g p3)
```

課題 **5-2:** (**RSA** 暗号の内部表現)

1. RSA 暗号の内部表現を決めて、構築子・選択子を定義せよ。
2. 暗号・復号が働いていることを確認せよ。
3. ある文字列を、各文字の文字コードについて暗号化したところ $[2228, 6464, 9366, 3716, 3176, 5072, 402, 7911]$ であった。復号鍵が $(8743, 10403)$ だとして元の文字列を復元せよ。

∎

# 3 抽象化による壁

これまで見てきたように、同じ「点を扱う」と言っても様々な意識のレベルがある。データの抽象化では、「下位の内部表現を意識せずにデータを使うレベル」と「上位から隠されている内部表現のレベル」があり、それらの「壁」の間を構築子・選択子といった関数が橋渡しをしている関係がある。実際に大きなアプリケーションプログラムを書く場合には、こういったレベルが何段にも積み重ってできている。

このように、複数のレベルによってプログラムを構成することの利点は、

- 各レベルが独立している
- 複雑なデータ構造を考える際に、いきなり詳細まで考えずに済む

といったことである。

各レベルが独立していることの利点は、内部表現の定義を変えても、それを使う側の定義を変更する必要がないことである。例えば、2 次元座標上の点を内部的に極座標系で表現していたとしても、使う側 (`add-rat` など) の定義を変更する必要がない。

練習 **5-2:** (極座標)

2 次元座標上の点 $(x, y)$ は、原点からの距離 $r$ と仰角 $\theta$ の組みによっても表わすことができる。内部表現として極座標系を使う`make-point`, `get-x`, `get-y` を定義せよ。さらに、それが `point-add`, `plot` などの定義を変更せずに利用できることを確めよ。

点 $(x, y)$ の仰角 $\theta$ は以下のように計算できる:

$$\theta = \begin{cases} \mathrm{atan}(y/x) & (x > 0) \\ \pi/2 & (x = 0, y > 0) \\ \mathrm{atan}(y/x) + \pi & (x < 0) \\ -\pi/2 & (x = 0, y < 0) \end{cases}$$

∎

# 4 データの意味

点の内部表現として $(x, y)$ によるもの (デカルト座標系) や極座標系があることを見てきた。これらの内部表現を定義するとき、どうやって定義が妥当であると判断するのだろうか?

これには、「使う側」が抽象データに期待しているような性質によって妥当性を判断する。このような性質はまた「仕様」とも呼ばれる。

例えば、2 次元の点であれば、

> 任意の数 $x$, $y$ に対して(`make-point` $x$ $y$) によって作られる値を $p$ としたときに、(`get-x` $p$) $= x$ かつ (`get-y` $p$) $= y$ が成り立つ

ことが期待される性質である。

Scheme が提供している cons 対も抽象データである。これが満たすべき性質は次のようになる:

> 任意のデータ $x$, $y$ に対して(`cons` $x$ $y$) によって作られる値を $p$ としたときに、(`car` $p$) は $x$ かつ (`cdr` $p$) は $y$ になる。

余談　Scheme において、対は原始的な関数によって提供されていると説明してきたが、必ずしもその必要はない。例えば、以下のような定義

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "argument should be 0 or 1" m)))))
(define (car z) (z 0))
(define (cdr z) (z 1))
```

は、対の内部表現として「0 か 1 を受け取って car 部か cdr 部を返す関数」としている。

また、さらにトリッキーな定義として次のようなものも可能である。

```
(define (cons x y) (lambda (f) (f x y)))
(define (car z) (z (lambda (a b) a)))
(define (cdr z) (z (lambda (a b) b)))
```

# 5 おまけ: 座標系の抽象化

より複雑なデータの抽象化として座標系を抽象化することを試みよう。

2 次元グラフィクスでは、座標系は原点と $x, y$ 方向のベクトルによって定義できる。そこでまず、以下のようにして座標系の抽象データを作る:

```
(define (make-coordinates origin x-axis y-axis)
  (cons origin (cons x-axis y-axis)))
(define (get-origin coordinates) (car coordinates))
(define (get-x-axis coordinates) (car (cdr coordinates)))
(define (get-y-axis coordinates) (cdr (cdr coordinates)))
```

3 つ以上のデータを 1 つのデータとして扱うため、cons を 2 重に使っていることに注意。

前出の Grapher が使っている座標系は、$(0,0)$ を原点、$w$ と $h$ をそれぞれ描画領域の幅と高さとすると、$(w,0)$ と $(0,h)$ が $x, y$ 方向のベクトルになるので、次のようにして作ることができる:

```
(define (grapher-coordinates grapher)
  (let ((origin (make-point 0                     0))
        (x-axis (make-point (get-width grapher) 0))
        (y-axis (make-point 0                     (get-height grapher))))
    (make-coordinates origin x-axis y-axis)))
```

画面の大きさに依存せずに絵を描くために、絵を描くプログラムは $1 \times 1$ の領域を使っていると仮定する。例えば以下のような感じである:

```
(define (half-picture liner)    ; 絵の左半分を描く
  (liner 0.5 0.11 0.43 0.0) (liner 0.43 0.0 0.39 0.33)
  (liner 0.39 0.33 0.29 0.31) (liner 0.29 0.31 0.24 0.2)
  (liner 0.24 0.2 0.31 0.06) (liner 0.31 0.06 0.15 0.16)
  (liner 0.15 0.16 0.06 0.29) (liner 0.06 0.29 0.0 0.53)
  (liner 0.0 0.53 0.06 0.8) (liner 0.06 0.8 0.25 1.0)
  (liner 0.25 1.0 0.17 0.76) (liner 0.17 0.76 0.22 0.71)
  (liner 0.22 0.71 0.33 0.82) (liner 0.33 0.82 0.38 0.66)
  (liner 0.38 0.66 0.5 0.93))
```

上の関数が使うliner は、始点と終点の $x, y$ 座標を受け取り、それらを与えられた座標系に変換して描画するものである。そのままdraw-line を使うと、1 ドットにつぶれてしまうため、座標変換をしてから描画する:

```
;; (px,py) を座標系<O,X,Y>へ変換する: p' = O+pxX+pyY
(define (convert-point coordinates p)
  (let ((origin (get-origin coordinates))
        (x-axis (get-x-axis coordinates))
        (y-axis (get-y-axis coordinates)))
    (point-add origin
               (point-add (point-scale (get-x p) x-axis)
                          (point-scale (get-y p) y-axis)))))

;; 座標変換をしてから grapher 上に線を引く関数を作る
(define (make-liner grapher coordinates)
    (lambda (x1 y1 x2 y2)
      (let ((p1 (convert-point coordinates (make-point x1 y1)))
            (p2 (convert-point coordinates (make-point x2 y2))))
        (draw-line grapher
                   (get-x p1) (get-y p1) (get-x p2) (get-y p2))))))
```

これらを用いると、次のようにして半分の絵を描くことができる:

```
> (define (draw-half-picture g c)
>  (half-picture (make-liner g c)))
> (define g (make <Grapher> :wdith 300 :height 300))
> (pack g)
> (draw-half-picture g (grapher-coordinates g))
```

絵を水平方向に反転させるには、座標系を反転させればよいので、次のようになる。

```
;; 座標系の反転: <O,X,Y> は <O+X,-X,Y> に移る
(define (coordinates-flip-horizontally coordinates)
  (let ((origin (get-origin coordinates))
        (x-axis (get-x-axis coordinates))
        (y-axis (get-y-axis coordinates)))
    (let ((new-origin (point-add origin x-axis))
          (new-x-axis (point-scale -1 x-axis)))
      (make-coordinates new-origin new-x-axis y-axis))))
```

これを使えば、以下のようにして完成した絵を描く関数を定義できる。

```
(define (draw-picture g c)
 (half-picture (liner g c))
 (half-picture (liner g (coordinates-flip-horizontally c))))
```

座標系をいろいろと変換してゆくと楽しい図案ができる。以下は、与えられた座標系に対して、下半分と右上 $1/4$ にはそのまま絵を描き、左上 $1/4$ には再帰的にこの描画をくり返すものである。

```
(define (recursive-draw g c)
  (update)
  (let* ((origin (get-origin c))
         (x-axis (get-x-axis c))
         (y-axis (get-y-axis c)))
    (if (< 1 (point-length x-axis))
        (let ((bottom-half-coordinates
               (coordinates-move
                (make-point 0 1)
                (coordinates-scale (make-point 1 0.5) c)))
              (right-upper-coordinates
               (coordinates-move
                (make-point 1 0)
                (coordinates-scale (make-point 0.5 0.5) c)))
              (left-upper-coordinates
               (coordinates-turn-left
                (coordinates-scale (make-point 0.5 0.5) c))))
          (draw-full-picture g bottom-half-coordinates)
          (draw-full-picture g right-upper-coordinates)
          (recursive-draw g left-upper-coordinates)))))
```

(`(coordinates-move p c)`, `(coordinates-scale p c)` はそれぞれ与えられた座標系のもとで座標系を移動したり拡大縮小したりするものである。)

課題 **5-3:** (おまけ)
座標系の変換を用いていろいろな図案を作ってみよ。 ∎