システム理論演習 II

(8) レコードとオブジェクト指向

増原英彦 *(masuhara@graco.c.u-tokyo.ac.jp)*

1997 年 6 月 20 日

1 レコードデータ

抽象データ構造には、(1) 複数の状態をひとまとめにする、(2) 内部表現の世界と使う側の世界を分ける、といった役割があった。この中の (1) に注目すると、レコードデータというものになる。

レコードデータは、以下のように、ラベルと値を並べて書くのが一般的である。

$$\{\langle ラベル_1\rangle = \langle 値_1\rangle, \langle ラベル_2\rangle = \langle 値_2\rangle, \ldots\}$$

レコードに関する操作としては、以下のものがあるとする。

- 新しくレコードを作る — `(make-record (list` $\langle ラベル_1\rangle$ $\langle 値_1\rangle$ $\langle ラベル_2\rangle$ $\langle 値_2\rangle$ `...))`
- レコードの中のあるフィールドの値を得る — `(record-ref` ⟨レコード⟩ ⟨ラベル名⟩`)`
- レコードの中のあるフィールドの値を変更する — `(record-set!` ⟨レコード⟩ ⟨ラベル名⟩ ⟨値⟩`)`

乗り物 例えば、位置と速度 (ベクトル) を内部状態として持つような「乗り物」を表わすレコードを作りたかったら、以下のような定義になる。

```
(define (make-vehicle pos vel)
  (make-record (list 'position pos 'velocity vel)))
(define (vehicle-position v) (record-ref v 'position))
(define (vehicle-velocity v) (record-ref v 'velocity))
(define (vehicle-move! v time)
  (let ((pos (vehicle-position v))
        (vel (vehicle-velocity v)))
    (record-set! v 'position
                 (point-add pos (point-scale vel time)))))
(define (vehicle-set-velocity! v vel)
  (record-set! v 'velocity vel))
(define (vehicle-print-location v)
  (point-print (vehicle-position v)))
```

ただし、フィールド position や velocity に入っているデータは、座標を表わす抽象データであるとする。

このようなレコードデータは、「表」データ構造によって簡単に作ることができ、以下のように定義される。

```
(define (make-record label-and-values)
  (let ((r (make-table)))
    (define (iter! pairs)
      (if (not (null? pairs))
          (let ((label (car pairs)) (value (cadr pairs)))
            (insert-table! label value r)
            (iter! (cddr pairs)))))
    (iter! label-and-values)
    r))
(define (record-ref record label)
  (lookup-table label record))
(define (record-set! record label value)
  (insert-table! label value record))
```

実際には、レコード中のフィールドを参照したり変更する度に `record-ref` や `record-set!` を書かなければいけないため、不便である。そこで、レコードのフィールドがあたかも変数であるかのようにするための新しい構文 `with-slots` を用意する[1]。

```
(with-slots ((⟨ラベル1⟩ ... ⟨ラベルm⟩)
             ⟨レコード⟩)
  ;; このスコープ中ではラベル名がフィールドの値に対応
  ⟨式1⟩ ... ⟨式n⟩)

                 ⇓

;; フィールドの値をとり出して変数に束縛
(let ((⟨ラベル1⟩ (record-ref ⟨レコード⟩ '⟨ラベル1⟩))
                        ⋮
      (⟨ラベルm⟩ (record-ref ⟨レコード⟩ '⟨ラベルm⟩)))
  (let ((result (begin ⟨式1⟩ ... ⟨式n⟩))) ;; 式本体を評価
    ;; 変数の値をフィールドに書き戻す
    (record-set! ⟨レコード⟩ '⟨ラベル1⟩ ⟨ラベル1⟩)
                 ⋮
    (record-set! ⟨レコード⟩ '⟨ラベルm⟩ ⟨ラベルm⟩)
    result))
```

構文 `with-slots` を使って「乗り物」の定義を書き直すと、以下のようになる。

```
(define (make-vehicle pos vel)
  (make-record (list 'position pos 'velocity vel)))
(define (vehicle-position v) (with-slots ((position) v) position))
(define (vehicle-position v) (with-slots ((velocity) v) velocity))
(define (vehicle-move! v time)
  (with-slots ((position velocity) v)
    (set! position
          (point-add position (point-scale velocity time)))))
(define (vehicle-set-velocity! v vel)
  (with-slots ((velocity) v) (set! velocity vel)))
```

2 次元の点　次に、2 次元の点を表わす抽象データをレコードを使って定義する。内部表現としては、x, y 座標によるデカルト座標系を使うことにする。

- `(make-cartesian` x y`)` — 点 (x, y) を作る
- `(cartesian-x` p`)` — 点 p の x 座標をとり出す
- `(cartesian-y` p`)` — 点 p の y 座標をとり出す
- `(cartesian-add` p_1 p_2`)` — 点の平行移動、つまり $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$ なる新しい点を作る
- `(cartesian-scale` p s`)` — スカラー倍、つまり $s(x, y) = (sx, sy)$ なる新しい点を作る
- `(cartesian-print` p`)` — 座標を表示する。ここでは "`C(`x`,`y`)`" のように表示されるものとする。

定義は長くなるので省略[2]。

この 2 次元の点の定義を利用すると、「乗り物」の例を動かすことができるようになる。ただし、手続きの名前が違うので、注意が必要である。

[1] この演習のホームページに「レコードアクセス用構文 (with-slots-1.scm) SCM 用」というファイルがあるので、それをダウンロードし、SCM で `(load "with-slots-1.scm")` とタイプする。

[2] この演習のホームページの「2 次元の点 (デカルト座標系) (cartesian-1.scm)」に定義がある。

```
> (define point-scale cartesian-scale)
> (define point-add cartesian-add)
     ...(略)...
> (define my-car (make-vehicle (make-point 0 0) (make-point 30 0)))
> (vehicle-print-location my-car)
C(0,0)
> (vehicle-move! my-car 2)
> (vehicle-print-location my-car)
C(60,0)
> (vehicle-set-velocity! my-car (make-point 20 10))
> (vehicle-move! my-car 1)
> (vehicle-print-location my-car)
C(80,10)
```

課題 8-1: (極座標系)

2 次元の点として、極座標系による抽象データを定義せよ。具体的には、以下のような手続きがあるとする。

- (make-polar r θ) — 点 (r, θ) を作る
- (polar-x p)— 点 p の x 座標つまり $r\cos\theta$ を求める
- (polar-y p)— 点 p の x 座標つまり $r\sin\theta$ を求める
- (polar-radius p)— 原点からの距離、つまり $|(r,\theta)| = r$ を求める
- (polar-angle p)— 角度、つまり $A(r,\theta) = \theta$ を求める
- (polar-scale p s)— 点のスカラー倍
- (polar-print p)— 座標を表示する。ここでは “P(r,θ)” のように表示されるものとする。

∎

2 データと手続の一体化

2 次元の点 cartesian と polar は、それぞれが同じような動作をする手続きを持っている。これらは内部の動作が違うため、別々の名前を持っている。そのため、例えば「点をスカラー倍したときの x 座標を求める」のに、下のように別々の手続きを定義してやらなければならない。

```
(define (cartesian-scaled-x p s)
  (cartesian-x (cartesian-scale p s)))
(define (polar-scaled-x p s)
  (polar-x (polar-scale p s)))
```

そこで、データ中に手続きを埋め込むことで、異なる抽象データを同じ方法で操作できるようにする。ここでは、データを作る際に、手続きをデータのレコード中に登録しておく。手続きを呼び出すときには、該当するフィールドから実際の手続きをとり出し、それを適用すればよくなる。

このアイデアに基づくと、2 次元の点は、それぞれ次のような定義になる。

```
;;; デカルト座標系による 2 次元の点
(define (make-cartesian x y)
  (make-record
   (list 'x-coord cartesian-x
         'y-coord cartesian-y
         'add     cartesian-add
         'scale   cartesian-scale
         'print   cartesian-print
         'x       x
         'y       y)))
```

```
;;; 極座標系による 2 次元の点
(define (make-polar radius angle)
  (make-record
   (list 'x-coord polar-x
         'y-coord polar-y
         'scale   polar-scale
         'radius  polar-radius
         'angle   polar-angle
         'print   polar-print
         'r       radius
         'a       angle)))
```

先程の「点をスカラー倍したときの x 座標を求める」は、下のようになり、 cartesian と polar の両者に共通な使える。このような、異なる種類のデータに対して実行できるような手続きを多相的 *(polymorphic)*な手続きであると言う。

```
(define (scaled-x p s)
  (let ((scale (record-ref p 'scale)))
    (let ((sp (scale p s)))
      (let ((x-coord (record-ref sp 'x-coord)))
        (x-coord sp)))))
```

さらに、レコードから手続きをとり出しては適用する部分を抽象化するために、次の send を定義する。

```
;;; (第 1 版)
(define (send message)
  (lambda arguments ;; = (cons receiver parameters)
    (let ((receiver (car arguments))
          (parameters (cdr arguments)))
      (let ((proc (record-ref receiver message)))
        (if proc
            (apply proc (cons receiver parameters))
            (error "SEND: no method is found" message))))))
```

ただし、

$$\texttt{(lambda}\ \langle\text{名前}\rangle\ \langle\text{式}\rangle\texttt{)}$$

という手続きを適用すると、 ⟨名前⟩ に全引数がリストとなって束縛される。また、 apply はリストになっている引数を手続きに渡すような関数であり、以下のような関係になっている。

$$\texttt{(apply}\ f\ \texttt{(list}\ a_1\ \ldots\ a_n\texttt{))} = \texttt{(}f\ a_1\ \ldots\ a_n\texttt{)}$$

これを用いると、 scaled-x は次のように簡略化される。

```
(define (scaled-x p s)
  ((send 'x-coord)
   ((send 'scale) p s)))
```

また、 send の返り値はデータの種類によらない手続きなので、以下のように普通の名前で定義しておくと便利である。 2 次元の点であれば、

```
(define x-coord (send 'x-coord))
(define y-coord (send 'y-coord))
(define add     (send 'add))
(define scale   (send 'scale))
(define print   (send 'print))
```

のように手続を定義して、下のような使い方ができる。

```
> (define (scaled-x p s)
    (x-coord (scale p s)))
> (define p1 (make-cartesian 10 20))
> (define p2 (make-polar 5 (/ pi 4)))
> (scaled-x p1 0.1)
1.0
> (scaled-x p2 0.1)
353.55339532999e-3
> (print (add p1 p2))
C(13.5355339533,23.535533858566)
```

このように、複数の種類のデータに対して定義されているような手続 / 関数のことを総称関数 *(generic function)*という。他方、総称関数から呼び出される「個々のデータ向けに定義された手続」のことをメソッド *(method)*と呼ぶ。

課題 **8-2:** (図形データ)

2 次元図形のうち、長方形と円を抽象データとしてレコードを使って定義せよ。それぞれ、

- `(make-rectangle p1 p2)` — 点 `p1` から点 `p2` が対角線となるような長方形を作る
- `(make-rectangle p r)` — 点 `p` を中心として半径 `r` の円を作る

また、それぞれのメソッドとして、以下のものを定義せよ。

- `(move! obj p)`— 図形 `obj` を `p` だけ平行移動する
- `(scale! obj s)`— 図形 `obj` の大きさを `s` 倍にする
- `(area obj)`— 図形 `obj` の面積を求める

∎

課題 **8-3:** (合成図形)

複数の図形データをグループ化した、合成図形を定義せよ。`(make-composite (list obj1 obj2 ...))` によって作られるものとする。また、 `move!`, `scale!`, `area` といったメソッドもそれぞれ定義せよ。 ∎

3 クラス

3.1 クラス・メソッド・インスタンス

ここまでは、 1 つのオブジェクトの中にメソッドも内部状態もまとめて放り込まれていた。しかし実用上は、オブジェクトの挙動 (メソッド) は、例えばデカルト座標の点にの間では共通している。このように共通している挙動を、内部表現においても共有させてしまうことができると、将来拡張する場合などには有利となる。

状態変数の名前やメソッドが同じオブジェクトの集まりのことをクラスと言う。そして、あるクラスに属しているオブジェクトのことをインスタンス *(instance)*と言う。状態変数は、インスタンス毎に異なる値を持つものなので、インスタンス変数と呼ぶことがある。

クラス内でメソッド定義を共有するため、オブジェクトの内部表現を「メソッド表」と「状態表」の 2 つに分割した抽象データとして定義する。

- `(make-object` ⟨メソッド表⟩ ⟨状態表⟩`)`— 新しくオブジェクト (インスタンス) を作る
- `(object-method-table` ⟨オブジェクト⟩`)`— メソッド表をとり出す
- `(object-state-table` ⟨オブジェクト⟩`)`— 状態表をとり出す

3.2 メソッド表

また、メソッド表は、以下のような手続きを持つ抽象データになる。

- `(make-method-table (list` ⟨名前⟩ ⟨手続⟩ `...))`— メソッド表を作る
- `(method-table-lookup` ⟨メソッド表⟩ ⟨名前⟩`)`— ⟨名前⟩ に対応する手続をとり出す。もし見つからなかった場合は偽 (`#f`) を返す

手続きを取り出しては適用する `send` は、このメソッド表を調べるように変更されるので、以下のようになる。

```
;;; (第２版)
(define (send message)
  (lambda arguments ;; = (cons receiver parameters)
    (let ((receiver (car arguments))
          (parameters (cdr arguments)))
      (let ((proc (method-lookup
                   (object-method-table receiver)
                   message)))
        (if proc
            (apply proc (cons receiver parameters))
            (error "SEND: no method is defined for the message" message))))))
```

課題 8-4: (メソッド表)

手続 make-method-table, method-table-lookups を定義せよ。 ∎

3.3 状態表

状態表は、いままで通りレコードを用いて表現する。ただし、オブジェクトの状態変数にアクセスするための構文、 with-slots は、以下のように修正される必要がある[3]。

```
(with-slots ((⟨ラベル_1⟩ ... ⟨ラベル_m⟩)
              ⟨オブジェクト⟩)
  ⟨式_1⟩ ... ⟨式_n⟩)

               ⇓

(let ((state-table (object-state-table ⟨オブジェクト⟩)))
  (let ((⟨ラベル_1⟩ (record-ref state-table '⟨ラベル_1⟩))
                        ⋮
        (⟨ラベル_m⟩ (record-ref state-table '⟨ラベル_m⟩)))
    (let ((result (begin ⟨式_1⟩ ... ⟨式_n⟩)))
      (record-set! state-table '⟨ラベル_1⟩ ⟨ラベル_1⟩)
                  ⋮
      (record-set! state-table '⟨ラベル_m⟩ ⟨ラベル_m⟩)
      result)))
```

結局、 2 次元座標の点の定義は、図 1 のようになる。

3.4 「クラス」の抽象データ化

結局のところ、クラスの定義というのは、メソッド表と、インスタンスを作成する手続きにつきる。そこで、この 2 つをまとめて抽象データとしたものをクラスとする。クラスに用意される手続きは、

- (make-class ⟨状態表作成手続⟩ ⟨メソッドリスト⟩)— ⟨状態表作成手続⟩ は、引数から「ラベル名と値のリスト」を作って返すような手続き、 ⟨メソッドリスト⟩ は、「ラベル名とメソッドのリスト」そのものであるとする
- (new ⟨クラス⟩ ⟨引数⟩...)—⟨クラス⟩ のインスタンスを作る。内部状態は、 ⟨引数⟩... をもとに作る。

これを使うと、クラスを 1 つのデータとして扱えるようになる:

[3]定義はホームページの「レコードアクセス用構文 (修正版) (with-slots-2.scm)」にある。この定義を Scheme に読み込んだ場合、with-slots を使っている手続きの定義を全て読み込み直す必要があるので注意せよ。

```
;;; デカルト座標系
(define *cartesian-method-table*
  (make-method-table
   (list 'x-coord cartesian-x
         'y-coord cartesian-y
         'add     add
         'scale   scale
         'print   print)))

(define (make-cartesian x y)
  (let ((state-table (make-record (list 'x x 'y y))))
    (make-object *cartesian-method-table* state-table)))

;;; 極座標系
(define *polar-method-table*
  (make-method-table
   (list 'x-coord polar-x
         'y-coord polar-y
         'add     polar-add
         'scale   polar-scale
         'print   polar-print
         'r       radius
         'a       angle)))

(define (make-polar r a)
  (let ((state-table (make-record (list 'r r 'a a))))
    (make-object *polar-method-table* state-table)))
```

図 1: メソッド表と状態表を分離した 2 次元座標の点

```
(define cartesian
  (make-class
   (lambda (x y) (list 'x x 'y y))
   (list 'x-coord cartesian-x
         'y-coord cartesian-y
         'add     add
         'scale   scale
         'print   print)))
;;; 使用例
> (define p (new cartesian 10 20))
> ((send 'print) p)
C(10 20)
```

課題 8-5: (クラス)

クラスの定義 (`make-class`, `new`) を完成させよ。

さらに、デカルト座標・極座標のクラスを定義して、動作を確めよ。 ∎

課題 8-6: (状態変数の初期値)

手続 `make-class` の別の定義方法としては、次のように、状態変数のラベル名だけを与えておき、手続 `new` の中でレコードを作るようにすることも可能である。

```
(define cartesian
  (make-class
   (list 'x 'y)
   (list 'x-coord cartesian-x
         'y-coord cartesian-y
         'add     add
         'scale   scale
         'print   print)))
```

この方法と、最初に示したような `(lambda (x y) (list 'x x 'y y))` を与える方法の違いについて考えよ。

例えば、極座標の点の内部状態として、角度 (フィールド a) が 0 から 2π におさまるような正規化、つまり、「(new polar $\langle r\rangle$ $\langle 2n\pi+\theta\rangle$) に対して、内部状態 a を θ にする」場合を考えよ。 ▮

課題 8-7: (メソッドの追加)

クラスが定義された後からメソッドを追加 (変更) する方法を考えよ。具体的には、(add-method! ⟨クラス⟩ ⟨名前⟩ ⟨メソッド⟩) という手続を定義して、

```
(add-method cartesian 'radius
  (lambda (p)
    (with-slots ((x y) p)
      (sqrt (+ (square x) (square y))))))
```

のように用いる。実際に、インスタンスを作った後からメソッド追加できることを確めよ。

▮