システム理論演習 II

(6) オブジェクトと状態

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

1997 年 6 月 6 日

- より大きなシステムを作る場合には、何を部品として選ぶかが重要になるが、実世界のモノを選ぶのが自然である。
- 実世界のモノを計算機の中のオブジェクトに対応させることを、モデル化という。
- システムの変更というのは、少数のオブジェクトに関する変更で済むことが多い ⇒ 部品化の利点
- モノには状態がある。

  状態を扱う方法:

  - オブジェクトの集合: identity を保ったまま、振舞が変化してゆく
  - ストリーム: 情報がシステムの中を流れてゆく

1 代入と内部状態

オブジェクトの集合によって実世界をモデル化する場合には、個々のオブジェクトは identity を保ちながら、内部状態が変化してゆくのが自然である。例えば、銀行口座をオブジェクトとする場合、同じ口座の残高が変化してゆく。

Scheme では内部状態を、局所的な状態変数 *(local state variables)*として扱う。状態変数はいままでの変数と同じであるが、代入演算子 *(assignment operators)*によって状態を変化させることができる。

1.1 局所的な状態変数

銀行口座の例: 口座には最初 1 万円があるとする。手続 (`withdraw x`) は口座から `x` 円引き出して、残高を返すものとする。もし残高よりも沢山引き出そうとした場合は "insufficient funds" というメッセージを返す。使用例は以下のようなものになる。

```
> (withdraw 2500)
7500
> (withdraw 2500)
5000
> (withdraw 6000)
"Insufficient funds"
> (withdraw 1500)
3500
```

この手続 `withdraw` を定義するためには、残高を表わす変数 `balance` というものを設け、それに対する代入を行う。

```
(define balance 10000)

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

この定義に使われているもののうち、 `set!` は、以下のような形式を持ち、⟨名前⟩ の値を ⟨新しい値⟩ の値に変化させる。

(set! ⟨名前⟩ ⟨新しい値⟩)

また、 `begin` は、以下のような形式で、⟨式 1⟩ から ⟨式 k⟩ までを順に評価して、⟨式 k⟩ の値を `begin` 形式の値とするものである。

(begin ⟨式 1⟩ ⟨式 2⟩ ... ⟨式 k⟩)

上の `withdraw` の定義だと、残高 `balance` の値は誰でも触ることができてしまう。そこで、以下のような定義に変更する。

```
(define new-withdraw
  (let ((balance 10000))
    (lambda (amount)
      (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))))
```

この定義では、 `balance` は `let` 形式の中にあるので、手続 `new-widthdraw` を通してしか触れなくなる。このように、内部状態を外部から触れなくすることを一般に「カプセル化 (*encapsulation*)」と言う。

このままでは、世界中に 1 つしか口座がない。以下のように定義を変更すると、「引き出しをする手続」を返す手続になる。

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

これを用いると、`(make-withdraw x)` が評価される度に新しく口座ができることになる。

```
> (define W1 (make-withdraw 10000))
> (define W2 (make-withdraw 10000))
> (W1 5000)
5000
> (W2 7000)
3000
> (W2 4000)
"Insufficient funds"
> (W1 4000)
1000
```

引き出ししかできない口座だと、残高は減る一方なので、預金もできるようにする。そのためには、図 1のような message passing の方法をとる。ここでは、このようにすることをオブジェクト化と呼ぶことにする。

課題 6-1: (集計機)

集計機 (accumulator) は 1 引数の手続きで、引数の数を内部状態の和に足してゆく。また、呼び出される度に現時点での和を返す。このような集計機を作る手続 `make-accumulator` を定義せよ。手続 `make-accumulator` の引数は、内部状態の和の初期値とする。

```
;;; オブジェクト化された銀行口座
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)

;; 使用例
> (define acc (make-account 10000))
> ((acc 'withdraw) 5000)
5000
> ((acc 'withdraw) 6000)
"Insufficient funds"
> ((acc 'deposit) 4000)
9000
> ((acc 'withdraw) 6000)
3000
```

図 1: Message passing による銀行口座の表現

```
> (define A (make-accumulator 5))
> (A 10)
15
> (A 10)
25
```

■

課題 6-2: (呼出回数の記録)

プログラムの実行中に、ある手続が何回呼び出されたかを調べることができると、プログラムの検査の際に便利である。そこで、手続の呼出回数を記録するための手続 `make-monitored` を定義せよ:

- 手続 `make-monitored` は、手続 (`f` とする) を受け取って、手続き (`mf` とする) を作って返す。
- 手続 `f` は、数を受け取るような手続き。
- 手続 `mf` は、数を受け取った場合は、呼び出し回数を増やした後に、手続 `f` を呼び出して、その答えを返す。
- 手続 `mf` が、引数として `how-many-calls?` というシンボルを受け取った場合は、ここまでの `mf` の呼び出し回数を返す。
- 手続 `mf` が、引数として `reset-count` というシンボルを受け取った場合は、記録している呼び出し回数を 0 に戻す。

```
> (define s (make-monitored sqrt))
> (s 100)
10
> (s 'how-many-calls?)
1
```

また、フィボナッチ数の計算の際の呼び出し回数を、以下のようにして測ってみよ。

```
(define monitored-fib
  (make-monitored
    (lambda (n)
      (cond ((= n 0) 0)
            ((= n 1) 1)
            (else (+ (monitored-fib (- n 2))
                     (monitored-fib (- n 1)))))))))
```

∎

課題 6-3: (暗証番号付き口座)

手続 make-account に、暗証番号を付け加えてみよ。

- make-account 自身は、残高の初期値に加えて、暗証番号に相当するシンボルを受け取って新しい口座を作る
- さらに、 withdraw, deposit の手続きには、引き出し / 預金額に加えて暗証番号を受け取り、口座を作ったときと同じ暗証番号であった場合のみ操作ができるようにする
- 暗証番号が間違っていた場合は、 “Incorrect password” というメッセージを返すものとする

```
> (define acc (make-account 10000 'secret-password))
> ((acc 'secret-password 'withdraw) 4000)
6000
> ((acc 'some-other-password 'deposit) 5000)
"Incorrect password"
```

∎

1.2 代入の利害得失

利点: 乱数を発生させることを考える。通常 (疑似) 乱数は、適当な関数 (rand-update) と、適当な初期値 (x_1 とする) をもとにして、

$$\begin{aligned} x_2 &= (\texttt{rand-update } x_1) \\ x_3 &= (\texttt{rand-update } x_2) \\ &\vdots \end{aligned}$$

のような系列として求める。

このままでは、乱数を使う人は

- 前回の乱数を覚えておく
- 新しい乱数を rand-update を使って作る
- 新しい乱数を次回ために覚えておく

という操作が必要になる。内部状態を用いれば、以下のような手続きを定義することによって前回の乱数を明示的に覚えておく必要がなくなる。

```
(define rand
  (let ((x rand-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

これは、乱数を発生させるモノを、内部状態 x を持つオブジェクトとして扱えることによる利点である。

問題点:

式の意味: 一方、代入の存在によって、いままで簡単であったプログラムの意味が複雑になってくる。

ある式がどのように評価されるか、を考えるとき、いままでは、「手続き呼出を手続き本体で置き換える」「変数を値で置き換える」ことで説明できた。ところが、代入があるとそうはいかない。

「同じ」であることの意味: また、値が同じである、という概念にも修正が必要になる。いままでは、「同じ手続きに同じ引数を与えた場合、同じ結果が返ってくる」、「ある式の一部を、同じ値の別のものと置き換えても、全体の値は変わらない」という、いわゆる参照透明性 *(referential transparency)*があったのだが、もはやれも成り立たない。

命令型 (imperative) プログラミングの落し穴: `set!` と `begin` があると、 C や Pascal のような、命令型のプログラミングが行える。しかし、命令型プログラミングでは代入の順序に非常に気を使わなければいけない。例えば下の `factorial` は命令型プログラミングによって階乗を定義したつもりのものだが、これの間違いを発見するのは容易でない。関数的なスタイルと比較すると、その差は歴然である。

```
;;; 注意! この手続は正しくない
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! counter (+ counter 1))
                 (set! product (* counter product)))))
    (iter)))
```

2 書き変え可能データ

第 4 回の資料では、複雑なデータを一つのデータとしてまとめて扱う構造データについて説明した。構造データも、内部状態を持つオブジェクトとして扱うことができる。このようなデータを書き変え可能 *(mutable)*データと呼び、内部状態を変化させる手続きのことを書き変え操作子 *(mutator)*と呼ぶ。

2.1 書き変え可能なリスト構造

対 (pairs) に対する書き変え操作としては、 `set-car!`, `set-cdr!` がある。

```
> (define x (list (list 'a 'b) 'c 'd))
> (define y (list 'e 'f))
> (set-car! x y)
> x
((e f) c d)                ;; x の car 部が書き変わる
```

課題 6-4: (append!)

2 つのリストをつなげる手続き `append` は以下のようなものだった。

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

これは、リスト `y` の手前にリスト `x` の要素を `cons` でつないでいる。

リストをつなぐ別の方法を考える。リスト `x` の最後の cons セルの cdr 部は空リストになっている。ここにリスト `y` をつなげる手続き `append!` を定義せよ。ただし、リストの最後の cons セルを得る手続き `last-pair` は以下のように定義されているとする。

```
(define (last-pair)
  (if (null? (cdr x)) x (last-pair (cdr x))))
```

定義された append! と append の違いはどのようなものか? 以下の例で確かめよ。

```
> (define x (list 'a 'b))
> (define y (list 'c 'd))
> (define z (append x y))
> z
⟨??⟩
> (cdr x)
⟨??⟩
> (define w (append! x y))
> w
⟨??⟩
> (cdr x)
⟨??⟩
```

∎

2.2 共有と identity

代入によってあるモノの状態を変化させた場合、この状態の変更は同じモノの状態は同じような変化しているはずである。別の言い方をすれば、同じモノを共有している 2 つのモノがあった場合、書き変えの影響が観測できる。

```
> (define x (list 'a 'b))
> (define z1 (cons x x))
> (define z2 (cons (list 'a 'b) (list 'a 'b)))
> (define (set-to-wow! x) (set-car! (car x) 'wow) x)
> z1
((a b) a b)
> (set-to-wow! z1)
((wow b) wow b)
> z2
((a b) a b)
> (set-to-wow! z2)
((wow b) a b)
```

手続 eq? は、ここで言う同じモノかどうかを調べる。つまり、(eq? (car z1) (cdr z1)) は真になるし、(eq? (car z2) (cdr z2)) は偽になる。

課題 6-5: (count-pairs:1)

リスト構造の中にある対の数を数える手続き count-pairs を考える。単純には、対があった場合には car 部と cdr 部の対の数の和に 1 を足すような、以下のようなものになりそうだが、これは正しくない。

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

この定義が間違った答えを返す例を挙げよ。 ∎

課題 6-6: (count-pairs:2)

正しい count-pairs を定義せよ。

ヒント: すでに数えた対を覚えておく補助データを用意して、リスト構造を順に調べてゆけばよい。 ∎

2.3　キュー構造を作る

set-car!, set-cdr! を使うと、いわゆるキュー (queue) 構造を作ることができる。キュー構造とは、最初に入れたデータが最初に出てくるようなものである。

```
;;; キューの長さに依存せずに、データの追加・削除をするために、キューの先頭
;;; と最後の対を  front-ptr,  rear-ptr という変数で管理する。
(define front-ptr '())
(define rear-ptr '())

;;; キューにデータを追加するには、 rear-ptr の cdr 部に新しく対を書き込
;;; めばよい。
(define (insert-queue! item)
  (let ((new-pair (cons item '())))
    (set-cdr! rear-ptr new-pair)
    (set! rear-ptr new-pair)))

;;; 一方、キューの先頭データは front-ptr の対の中にあるので、先頭要素
;;; を得るには、以下のようになる。
(define (front-queue) (car front-ptr))

;;; また、先頭要素を削除するには、 front-ptr が二番目の対を指ように書
;;; き変えればよいので、次のようになる。
(define (delete-queue!) (set! front-ptr (cdr front-ptr)))

;;; キューが空のときは、 front-ptr が空リストになっている
(define (empty-queue?) (null? front-ptr))
```

このままでは、キューが空だったときの front-queue や delete-queue! の動作がおかしくなる。そのような場合の処理も加えたものが queue.scm にある (ホームページ参照)。

課題 6-7: (キューのオブジェクト化)

上に挙げたキューの定義では、同時に一つのキューしか存在し得ない。(ファイル queue.scm の定義でも同じである。) 複数のキューが扱えるように、キューの定義をオブジェクト化せよ。具体的には、以下のようにする。

```
(define (make-queue)
  (let ((front-ptr ...) (rear-ptr ...))
    ⟨内部関数の定義をここに書く⟩
    (define (dispatch m) ...)
    dispatch))
(define (insert-queue! queue item) ...)
            ⋮
```

■

2.4　表を作る

表とは、値がキーによって検索できるように登録されているものである。要素の登録には insert-table!、検索には lookup-table という手続があるものとする。

```
> (insert-table! 'may 31)
> (insert-table! 'june 30)
> (lookup-table 'may)
31
> (lookup-table 'january)
#f
```

表の内部表現としては、キーと値で対を作り、それをリストにして管理する。表はとりあえず大域変数 table にあるものとする。リスト中から値を検索するために、以下のような手続きを定義する。

```
(define table '())
(define (lookup-table key)
  (let ((record (assoc key table)))
    (if record
        (cdr record)
        #f)))
(define (assoc key records)
  (cond ((null? records) #f)  ;; records が空リストなら偽を返す
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```

ただし、#f は「偽」を表わす記号である。

表に要素を登録する手続 insert-table! は、まず同じキーの要素を検索して、すでに登録されていた場合は、その値を書き換え、そうでない場合はリストに追加をする。

```
(define (insert-table! key value)
  (let ((record (assoc key table)))
    (if record
        (set-cdr! record value)
        (set! table (cons (cons key value) table)))))
```

課題 6-8: (表のオブジェクト化)

上の表の定義では、大域変数を使っているので、 make-table という手続きで作られるオブジェクトとして定義せよ。

```
> (define days (make-table))
> (insert-table! 'may 31 days)
> (insert-table! 'june 30 days)
> (lookup-table 'may days)
31
> (lookup-table 'january days)
#f
```

■

課題 6-9: (Memoization)

前回計算しておいた値を記録しておくことを memoization という。関数が呼び出される度に、引数とその値を記録しておき、もし以前に同じ引数で呼び出された記録があった場合には、実際に関数の計算をせずに値を返すようにすることで、実行を高速化できる場合がある。

ここでは一引数の関数に限って memoize する手続を定義する。

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result (lookup-table x table)))
        (if previously-computed-result
            previously-computed-result
            (let ((result (f x)))
              (insert-table! x result table)
              result))))))
```

これを使ってフィボナッチ数を定義し直すと、以下のようになる。

```
(define memo-fib
  (memoize (lambda (x)
             (cond ((= n 0) 0)
                   ((= n 1) 1)
                   (else (+ (memo-fib (- n 2)) (memo-fib (- n 1))))))))
```

さらに、 make-monitored を使って、 memo-fib の呼出回数を測って、 memoize しない場合と比較せよ。 ■