情報システム科学実習 II (8)

リストの利用・オブジェクトと状態

1999 年 12 月 8 日 増原 英彦

1 リストの利用:

リストにはいろいろな応用がある。例として「n 人の女王問題」(n-Queens problem) を解くプログラムを考える。

(ここに示した定義は web ページにある。実行には queen.scm, queen-common.scm, queen-gui.scm の 3 つが必要。(load "queen.scm") として、(queen 4) とせよ。)

愚直な解法は、全ての配置を作ってから、条件を満たしている配置だけをとり出すもので、いわゆる generate-and-test と呼ばれているものである。まず、女王の駒を置く場所を表わすposition の構築子と選択子を定義する。

```
(define (make-position row column) (list row column))
(define (position-row position) (car position))
(define (position-column position) (cadr position))
```

次に、盤上のすべての場所を生成してリストにする (以前定義したmake-integers-list を使っている)。これは、大きさsize の盤であれば、$(0,0),(0,1),\ldots,(1,0),(1,1),\ldots,(\mathtt{size}-1,\mathtt{size}-1)$ の位置をリストにするだけであるので、map を 2 重に使えばよい。

```
(define (generate-positions size)
  (let ((grids (make-integers-list 0 (- size 1))))
    (flatten
     (map (lambda (row)
            (map ((curry make-position) row) grids))
          grids))))
```

なお、flatten は 2 重のリストを 1 重のリストにする関数である。

盤の大きさがsize のとき、size 人の女王が置かれた場所をリストで表わし、これを配置 (placement) と呼ぶことにする。positions 中の場所を使ったnum-queens 人の女王の配置を全て求めるには、次のようになる。(1)positions から場所を一つとり除いて (num-queens − 1) 人分の配置を作り、それにとり除いた場所を加える。(2) positions から場所を一つとり除いてnum-queens 人分の配置を作る:

```
(define (generate-placements num-queens positions)
  (cond ((= num-queens 0) (list '()))
        ((null? positions) '())
        (else
         (append
          (map ((curry cons) (car positions))
               (generate-placements (- num-queens 1) (cdr positions)))
          (generate-placements num-queens (cdr positions))))))
```

女王がとり合う位置にあるかどうかは、行または列が等しい、あるいは斜めに見て同じ列に並んでいることなので、次のように定義できる:

```
(define (check? position1 position2)
  (let ((row1 (position-row position1))
        (column1 (position-column position1))
        (row2 (position-row position2))
        (column2 (position-column position2)))
    (or (= row1 row2)
        (= column1 column2)
        (= (abs (- row1 row2)) (abs (- column1 column2))))))
```

配置 (女王が置かれた場所のリスト) が正しい、つまり、置かれた女王どうしは全てとりあわない位置にあることを調べるには、一人目の女王が残りの女王と全てとりあわないことと、残りの女王がどうして全てとりあわないことによって確認できる:

```
(define (valid-placement? placement)
  (or (null? placement)
      (let ((rest-positions (cdr placement)))
        (and ;; 一人めが他の女王ととりあう位置にあるか
         (not (some ((curry check?) (car placement)) rest-positions))
         ;; 残りの女王達がとりあう位置にあるか
         (valid-placement? rest-positions)))))
```

これらを全て組み合わせると、

1. `generate-positions` で場所のリストを作り
2. `generate-placements` で可能な配置を全て作り
3. `(filter valid-placement? ...)` で正しい配置のものだけをとり出す

というものになる。なお、`make-display-function` は STk のウィンドウとして配置を表示する関数を作る関数である。

```
(define (queens size)
  (make-display-function size
   (lambda (display-function)
     (for-each display-function
               (filter valid-placement?
                       (generate-placements
                        size
                        (generate-positions size)))))))
```

課題 **8-1:** (効率化)

(余力があれば) ここで示した定義では、n 人の女王問題を解くために、${}_{n^2}C_8$ 通りの配置を作らなければいけないため、恐ろしいほどに効率が悪い。次のような効率化を考えてみよ:

1. 配置を作る途中で、とりあう位置かどうかを調べ、とりあう場合にはその先の配置を作らない。こうすれば、2 人目の女王を置いた時点で正しくない配置になった場合、その先の配置を作ることがなくなる。

2. 「全ての場所のリスト」を列単位でまとめ、その中から 1 つ選ぶようにする。各列にはちょうど 1 人の女王が置かれなければいけないが、その条件が自然に満たされる。

3. 新たに r 行目に女王を置くときに、それが他の女王ととりあわない位置にあるかどうかを、隣の列の女王のいる行は $\{r-1, r, r+1\}$ のどれでもない、その隣の列の女王のいる行は $\{r-2, r, r+2\}$ のどれでもない... ... のようにして調べる。

   また、女王の配置を右の列から行番号だけで表わす。例えば、$\{(0,0),(1,3),(2,5)\}$ という配置は (5 3 0) というリストにする。

   こうすれば、とりあわないかどうかを調べる際に、複雑なリストの操作が減し、全体で使われるデータ量も減る。

4. 別のやり方として、「ここまで女王を配置していった結果置けない位置」というものをビットマップで表わせば、論理演算だけでとりあう位置かどうかを調べることができる。このような方法をとり入れてみよ。

∎

2 オブジェクトと状態

- 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。
- 実世界と対応がとれていると、実世界にあわせた変更をするときに対応したオブジェクトを局所的に変更するだけでよくなる ⇒ 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

2.1 オブジェクト: 状態を持つデータ

STk では状態を持つデータを表わすためにオブジェクトを用意している。オブジェクトはその雛形であるクラスから作られる。最も簡単なクラスの定義は次のようになる:

```
(define-class <クラス名> ()
  (スロット名1 スロット名2 ... スロット名n))
```

これで、<クラス名>という名前のクラスが定義される。このクラスからオブジェクトを作るにはmake という手続を使う:

```
(make <クラス名>)
```

作られたオブジェクトには、スロット名 1 からスロット名 n までの n 個のスロット (インスタンス変数とかメンバ変数とかフィールドと呼ばれているものと同じ) がある。オブジェクトのスロットは slot-ref と slot-set! を使って読み書きできる:

```
(slot-ref オブジェクト 'スロット名)      ; スロットを読む
(slot-set! オブジェクト 'スロット名 値) ; スロットに書く
```

例えば、銀行口座をオブジェクトにするには、次のようにする (web ページ account.scm):

```
(require "stklos")                ; おまじない

(define-class <Account> ()        ; クラス定義
  (balance))

;; スロットの読み書き
(define (get-balance account) (slot-ref account 'balance))
(define (set-balance! acccount new-balance)
  (slot-set! account 'balance new-balance))

;; 口座の開設
(define (make-account initial-balance)
  (let ((a (make <Account>)))
    (set-balance! a initial-balance)
    a))

;; 預金引出: 返り値は残高
(define (withdraw account amount)
  (let ((balance (get-balance account)))
    (if (>= balance amount)
        (let ((new-balance (- balance amount)))
          (set-balance! account new-balance)
          new-balance)
        (begin (display "Insufficient funds")
               (newline)
               balance))))
```

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。

```
STk> (define a (make-account 10000))
a
STk> (withdraw a 2500)
7500
STk> (withdraw a 2500)
5000
STk> (withdraw a 6000)
Insufficient funds
5000
STk> (withdraw a 1500)
3500
STk>
```

get-balance、set-balance! や make-account のような定義を毎回書くのはわずらわしい。STk には、そのような関数を自動的に定義するオプションがある。

```
(define-class <Account> ()
  ((balance :init-keyword :balance ; 初期化用キーワード
            :getter get-balance    ; 読み出し
            :setter set-balance!))); 書き込み
```

上のようにスロットを定義すると、(make <Account> :balance 10000) のようにしてオブジェクトを作る時にスロットの中身を決めることができ、get-balance, set-balance! という関数によってスロットの読み書きができるようになる。(スロット定義のところの括弧の個数が増えているのに注意。)

2.2 状態の手続化

オブジェクトは状態を持つことができるが、それを手続きの中で使うと、呼び出す渡に挙動が変わる関数が作れる。いままで状態を使わずに関数を定義していた場合は、同じ関数を同じ引数で呼び出していたら、必ず同じ結果が返ってきたことと比べて、注意が必要である。

```
STk> (define (make-withdrawer initial-balance)
       (let ((account (make <Account> :balance initial-balance)))
         (lambda (amount)
           (withdraw account amount))))
make-withdrawer
STk> (define w (make-withdrawer 10000))
w
STk> (w 6000)
4000
STk> (w 8000)
Insufficient funds
4000
```

課題 8-2: (電卓)

電卓をオブジェクトとしてモデル化しよう。電卓の持っている状態には、(1) いま入力中の数値、(2) 前回入力された演算子、(3) 前回の演算の結果、の 3 つがある。このうち、(2) と (3) は「次の数値を受け取って演算をする関数」のようにして 1 つにまとめることができる。

例えば「1,2,+,3,4,-,5,6」の順に打った場合を考えると、いま入力中の数値は 56 で、前回入力された演算子はマイナス、前回の演算結果は $36(= 12 + 34)$ である。後 2 者をまとめたものは(lambda (next) (- 36 next)) のような関数である。

1. クラス<Calc>を定義せよ。ただし、
   - 現在入力中の値を状態として持つ。読み書きはget-current, set-current! でできる、初期値は 0 とする。
   - 前回入力された演算に対応する関数を状態として持つ。読み書きはget-operator, set-operator! とする。初期値は(lambda (x) x) とする。
2. 数字キーと演算子キーは、次のような関数として定義する。(Web ページcalc-part.scm)

```
(define (push-number-key calc number) ; 数字キー
  (let ((new-current (+ (* (get-current calc) 10) number)))
    (set-current! calc new-current)
    new-current))

(define (push-operator-key calc operator) ; 演算子キー
  (let ((result ((get-operator calc) (get-current calc))))
    (set-current! calc 0)
    (set-operator! calc (lambda (next-value)
  (operator result next-value)))
    result))
```

次の実行はどうなるか。

```
STk> (define c (make <Calc>))
c
STk> (push-number-key c 3)
???
STk> (push-number-key c 4)
???
STk> (push-operator-key c *)
???
STk> (push-number-key c 2)
???
STk> (push-number-key c 3)
???
STk> (push-operator-key c +)
???
```

3. (余力があれば) クリアキー、「＝」キーに相当する関数を定義せよ。
4. (もっと余力があれば) ボタンで数字や演算子が入力できるような GUI を作れ。