情報システム科学実習 II (7)

階層データ

1999 年 12 月 1 日 増原 英彦

第 5 回の訂正　課題 5-1 の 2 で、$\phi(p,q)$ と互いに素な e を求めるには、最大公約数が 1 となるような e であるべきなので、例えば以下のようにして求めるのが正しい。

```
;; phi と互素な最小の n を求める
(define (find-mutually-indivisible phi n)
  (if (= (gcd phi n) 1)
      n
      (find-mutually-indivisible phi (+ n 1))))

;; ユークリッドの互除法による最大公約数
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

階層データ

Scheme では対 (pair) を使用するための構築子・選択子として cons, car, cdr が提供されていた。これを用いれば、2 次元の座標のように 2 つのデータを組みにして扱うことができた。

(cons 1 (cons 2 3)) のように対の中に別の対を入れることで、3 つのデータを扱うことができる。(cons 1 (cons 2 (cons 3 4))) とすれば 4 つ…… というようにしてゆけば、いわゆる可変長データが扱えるだろう。

そのようなデータを一般化した考え方として列(sequence) と、階層的な(hierachical) データについて見てみる。列はまたリスト(list) とも呼ばれる。Scheme のような言語の総称である 'Lisp" は "LISt Processor" にちなんで[1]名付けられていることが示すように、これらの言語ではリストを容易に扱えるようにしていることが特徴である。

1　列 (リスト)

何個かのデータから成る列 (sequence) を作るには、cons によって:

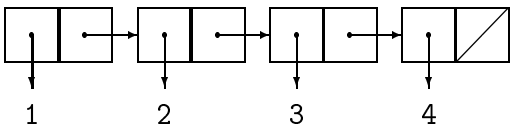
- 1 個目のデータがcar 部分に
- 2 個目以降のデータを列にしたものがcdr 部分に

[1]といっても、こじつけ半分だろうが。

入っているような対を作る。このようにして作った列はリストと呼ばれる。例えば、[1,2,3,4] という列は、

```
(cons 1
      (cons 2
            (cons 3
                  (cons 4 '()))))
```

である。ここで、'() は「空のリスト」を作る式である。作られるデータを図示すると次のようになる。右端の箱の斜線は「空のリスト」を示している:



1.1 組み込み関数

Scheme にはリストを操作するための関数が沢山用意されている。いくつかの例を見てみよう (詳しくは言語仕様書を参照して欲しい):

```
STk> (define a                      ; リストを作る
       (cons 1 (cons 2 (cons 3 (cons 4 '()))))
a
STk> a
(1 2 3 4)
STk> (define b (list 1 2 3 4))  ; 別の作り方
b
STk> b
(1 2 3 4)
STk> (define c (list 5 6 7 8))
c
STk> (car a)                        ; 先頭要素をとり出す
1
STk> (cdr a)                        ; 先頭をとり除いたリスト
(2 3 4)
STk> (list-ref a 3)                 ; 3 番目の要素
4
STk> (length a)                     ; 長さを求める
4
STk> (equal? a b)                   ; 比較
#t
STk> (equal? a c)
#f
STk> (append a c)                   ; 結合
(1 2 3 4 5 6 7 8)
STk> (reverse (append a c))     ; 反転
(8 7 6 5 4 3 2 1)
STk> (last-pair a)                  ; 最後の対をとり出す
(4)
STk> (map (lambda (x) (* x x)) a) ; 各要素を自乗する
(1 4 9 16)
STk>
```

1.2 リストを作る

[1,2,3,4] という列をリストにするときには以下のようにした:

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
```

作られたリストをcdr 部に持つようにcons をすると、リストの先頭に値を追加できる:

```
STk> (define a (cons 1 (cons 2 (cons 3 (cons 4 '())))))
a
STk> (cons 99 a)
(99 1 2 3 4)
STk>
```

例として $[a, a+1, a+2, \ldots, b]$ というリストを作る関数(make-integers-list a b) を作ってみよう。

- もし $b < a$ なら、リストは空 ('()) である
- それ以外の場合、car 部に a、cdr 部に $[a+1, a+2, \ldots, b]$ というリストを持つような対を作ればよい

ので、結局、次のようになる:

```
STk> (define (make-integers-list a b)
       (if (< b a)
           '()
           (cons a (make-integers-list (+ a 1) b))))
make-integers-list
STk> (make-integers-list 4 10)
(4 5 6 7 8 9 10)
```

1.3 リストを使う

例えば整数のリストの合計を求める(list-sum l) を定義してみよう。この関数は次のようになる:

- l が空のリストだったら合計は 0。
- そうでなければ、(先頭要素の値) と (次要素以降の合計) の和が合計。

これを Scheme にすると以下のようになる。関のリストかどうかを判定するのはnull?という関数である:

```
STk> (define (list-sum l)
       (if (null? l)
           0
           (+ (car l) (list-sum (cdr l)))))
list-sum
;; 1 から 10 までの合計
STk> (list-sum (make-integers-list 1 10))
55
```

練習 7-1: (リストを使う)

組み込み関数として示したlist-ref, length, append, reverse, last-pair を自分で定義せよ。 ∎

練習 7-2: (fold)
リストを使う関数の多くは、リストが $[x_1, x_2, \ldots, x_n]$ だったとしたときに、初期値 i と演算 @ を使って

$$x_1@(\cdots(x_{n-1}@(x_n@i))\cdots)$$

あるいは

$$(\cdots((i@x_1)@x_2)@\cdots@x_n)$$

のどちらかに一般化できる。

1. 前者の計算をする関数(`fold-right operator initial elements`) を定義せよ。
2. 後者の計算をする関数(`fold-left operator initial elements`) を定義せよ。
   ヒント: $(\texttt{fold-left}\ @\ i\ [x_1, x_2, \ldots, x_n]) = (\texttt{fold-left}\ @\ (i@x_1)\ [x_2, \ldots, x_n])$ である。
3. `fold-left`, `fold-right` を用いて`list-sum`, `length`, `append`, `reverse` を再定義せよ。

∎

1.4 リストに対する写像

リストを用いた計算の典型は、以下のものに大別できる。

1. 作る
2. 各要素について変換したリストを作る
3. 条件に合致する要素のみをとり出す
4. 全要素を用いた計算をする (例: 合計をとる)

このうち、2 番目と 3 番目について考えてみよう。
まず各要素を変換する場合であるが、これは高階関数の考え方を使って、変換する関数`f` をパラメータとして受け取り、リスト`l` の各要素を変換する関数(`map f l`) があると便利である。実際、Scheme には`map` が予め用意されている:

```
STk> (define (square x) (* x x))
STk> (define a (list 1 2 3 4 5 6 7 8 9 10))
STk> (map square a)              ; 各要素を 2 乗する
(1 4 9 16 25 36 49 64 81 100)
;; 文字列をリストに変換して、各文字を大文字にする
STk> (map char-upcase (string->list "Welcome to the STk"))
(#\W #\E #\L #\C #\O #\M #\E #\space #\T #\O #\space #\T
#\H #\E #\space #\S #\T #\K)
```

次に条件に合致する要素の抜き出しであるが、これも高階関数の考え方を使って、リスト`l` のうち条件`p` を満たすもののみをとり出したリストを作る関数(`filter p l`) があると便利である。こちらは予め用意されていないが、以下のように定義・使用できる:

```
STk> (define (filter predicate elements)
       (cond ((null? elements) '())      ; 空リストの場合
             ((predicate (car elements)); 先頭要素が条件を
              (cons (car elements)       ; 満たす場合
                    (filter predicate (cdr elements))))
             ;; 先頭要素が条件を満たさない場合
             (else (filter predicate (cdr elements)))))
STk> (filter even? a)            ; 偶数のみを取り出す
(2 4 6 8 10)
```

課題 **7-1:** (素数のリスト)

エラトステネスのふるいは、(1) 2 から始まる数列を作り、(2) 先頭から順に数列を見てゆき、消されていなければその数は素数なので、以降の数列からその数の倍数を消す、というものである。これによって素数のリストを求める関数を定義せよ。

1. x がn の倍数のとき、かつそのときのみ((not-divisible? n) x) が#f となる関数を定義せよ。

2. filter とnot-divisible?を使って、数列l からx の倍数をとり除く関数(remove-multiples x l) を定義せよ。

3. 関数(sieve l) の受け取るl が、(1) 先頭x が素数であり、(2) 数列の残りr の要素全てがx 未満の素数で割り切れないものだったとする。このとき、x 以上の素数のリストは、x と (r からx の倍数をとり除いたものから作った素数の列) となるので、次のように定義できる。

```
(define (sieve numbers)
  (if (null? numbers)
      '()
      (let ((prime (car numbers)))
        (cons prime
              (sieve (remove-multiples prime (cdr numbers)))))))
```

この定義を使って 100 以下の素数のリストを作れ。

∎

2 階層的構造

データの中にデータがあり、その中にまた別のデータがある..... というようなものを階層的なデータと言う。このようなデータを表現するにはリストの要素としてリストをとればよい。

例として、数値の二分木を考えよう。二分木の節点 (node) は、数値と左側の枝、右側の枝を持つので、次のような構築子・選択子によって定義できる (以下、web ページ tree.scm に定義あり):

```
(define (make-node num left right) (list num left right))
(define (value tree) (car tree))
(define (left tree) (car (cdr tree)))
(define (right tree) (car (cdr (cdr tree))))
;; また、特殊ケースとして空の節点を#f で表わすことにする:
(define empty-tree #f)
(define (empty? tree) (not tree))
;; 末端の節点は左右の枝が空のものとして定義できる:
(define (make-leaf num) (make-node num empty-tree empty-tree))
```

木に数値を追加する場合を考えよう:

```
(define (add-value tree num)
  ;; 空の木に追加した場合は、末端の節点を作ればよい
  (cond ((empty? tree) (make-leaf num))
        ;; 空でない場合、その節点の数値と比較して、それより小さ
        ;; い場合には、左側の枝に追加する
        ((< num (value tree))
         (make-node (value tree) (add-value (left tree) num)
                    (right tree)))
        ;; それ以外の場合、右側の枝に追加する
        (else
         (make-node (value tree) (left tree)
                    (add-value (right tree) num)))))
```

このようにして木を作れば、各節点の数値は、左側の枝に現われる数値よりも大きく、右側の枝に現われる数値と等しいか小さい、という性質が満たされる。

木に対する計算は再帰的に行う:

```
;; 数値の個数を求めるには、左右の枝に現われる数値の個数に 1 を加
;; えたものを求めればよい
(define (count-leaves tree)
  (if (empty? tree)
      0
      (+ 1 (count-leaves (left tree) (right tree)))))

;; 左の枝をリストにしたもの・その節点の数値・右の枝をリストにし
;; たもの、を順に結合すれば、整列されたリストが得られる
(define (list-values tree)
  (if (empty? tree)
      '()
      (append (list-values (left tree))
              (cons (value tree)
                    (list-values (right tree))))))
```

例:

```
;; リストから木を作る関数を定義
STk> (define (add-values tree numbers)
       (if (null? numbers)
           tree
           (add-values (add-value tree (car numbers))
                       (cdr numbers))))
;; サンプルの木を作る
STk> (define t (add-values empty-tree '(3 1 4 1 5 9 2)))
;; 数値の個数を求める
STk> (count-leaves t)
7
;; 順に値をとり出してリストにすると整列されている
STk> (list-values t)
(1 1 2 3 4 5 9)
```

課題 **7-2: (Q&A)**

「はい」か「いいえ」で答えるような質問によって枝分かれをして、最終的に適当なメッセージを表示するシステムを作ろう。(構築子・選択子を除いて Web ページ q-and-a.scm に定義・例あり)

1. システムの状態は (1) 質問中か (2) 答えのどちらかである。

   質問中の状態は、(a) 質問文の文字列 (b) 「はい」と答えた場合の次の状態 (c) 「いいえ」と答えた場合の次の状態とする。質問中の状態のための構築子(`make-question-node question yes-branch no-branch`)・選択子(`question-text q-node`), (`question-yes q-node`), (`question-no q-node`) を定義せよ。

2. 答えは、答えの文字列だけを持つ。構築子(`make-answer-node answer`)・選択子(`answer-text a-node`) を定義せよ。

3. 状態が (1) か (2) のどちらかを判定するため、(`question-node? object`) を定義せよ。

4. システムは現在の状態に応じて適切な表示を行う。(1) の場合は、問題文と「yes」「no」のボタンを表示する。ボタンがクリックされた場合は、選ばれた状態の表示を行う。(2) の場合は、メッセージを表示する。これらは、次のように定義できる:

```
;;; 状態に応じて適切な表示を行う
(define (display-dialog node)
  (if (answer-node? node)
      (display-answer-node (answer-text node))
      (display-question-node
       (question-text node) (question-yes node) (question-no node))))

;;; 質問の場合、質問文とボタンを表示する
(define (display-question-node text yes-node no-node)
  (pack (make <Label> :text text)
        (make <Button> :text "Yes"
              :command (lambda () (display-dialog yes-node)))
        (make <Button> :text "No"
              :command (lambda () (display-dialog no-node)))))

;;; 答えの場合、メッセージを表示する
(define (display-answer-node message)
  (pack (make <Label> :text message)))
```

5. 適当な質問・答えを作り、先頭の質問を`display-dialog` に適用しよう。