演習

情報数学の 10 月 22 日（水）Scheme 実習 3 のページ

(http://banon.ueda.info.waseda.ac.jp/oess/IM/Html/
class_rsc/materials/2003-10-22.html)

を Ocaml に直して解いてみる。

let 構文

(let ((x (* 3 3))) (list x x x)) を実際に評価してみよ.

1. Scheme

Scheme で実行すると

```
> (let ((x (* 3 3))) (list x x x))
(9 9 9)
```

2. Ocaml

まず、Scheme を Ocaml に直さないといけない。

```
let x = 3 * 3 in [x;x;x];;
```

実行すると

```
# let x = 3 * 3 in [x;x;x];;
- : int list = [9; 9; 9]
```

となり、Scheme と同じになった。こちらのほうが簡潔そうである。

関数を引数にとる関数

map 関数をそれぞれ自分で作ってみて実行し比較してみる。

1. Scheme

```
(define mymap
  (lambda (f x)
   (cond
     ((null? x) '())
     (else (cons (f (car x)) (mymap f (cdr x)))))))
```

(mymap (lambda (x) (* x x)) '(-2 1 0 1 2 3)) を実行してみる。

```
> (define mymap
     (lambda (f x)
        (cond
          ((null? x) '())
          (else (cons (f (car x)) (mymap f (cdr x)))))))
> (mymap (lambda (x) (* x x)) '(-2 1 0 1 2 3))
(4 1 0 1 4 9)
```

map はリストの中身に対して関数適用を行う。きちんと各要素に 2 乗しているのが見て取れる。

2. Ocaml

x を l に変えておく。taple として引数は渡す。

```
let rec mymap (f, l) =
 match l with
  [] -> [] |
   x :: rest -> f x :: mymap (f, rest) ;;
```

Ocaml でも Scheme で実行したものと同義のものを実行してみる。つまり、
`mymap ((fun x -> x * x), [-2; 1; 0; 1; 2; 3]);;` を実行する。

```
# let rec mymap (f, l) =
   match l with [] -> [] |
     x :: rest -> f x :: mymap (f, rest) ;;
val mymap : ('a -> 'b) * 'a list -> 'b list = <fun>
#   mymap ((fun x -> x * 2), [-2;1;0;1;2;3]);;
- : int list = [-4; 2; 0; 2; 4; 6]
```

Ocaml でも Scheme と同じ結果になった。ただし、それを実現するまでの違いに注意する。

関数を返す関数

cons をカリー化してみる

1. Scheme

```
(define cons1
  (lambda (a)
    (lambda (b) (cons a b))))
```

注意すべき点は、lambda が 2 個あるという点である。これによって、関数を返す関数を実現できる。また、カリー化関数は使うときも注意をしなければならない。

```
> (cons 1 '(2 3))
(1 2 3)
> (cons1 1 '(2 3))
procedure cons1: expects 1 argument, given 2: 1 (2 3)
> ((cons1 1) '(2 3))
(1 2 3)
```

実際使ってみると、cons1 は 2 引数では実行できない事が分かる。カリー化により、1 引数の関数を返す関数になっているからである。これは間違え易いので、カリー化をするときは気をつけること。

2. Ocaml

```
let cons1 a = fun b -> a :: b ;;
```

Ocaml の方が記述する事が少ない。また、括弧もないので分かり易いだろう。Scheme と同様に、実際に使ってみる。

```
# let cons1 a = fun b -> a :: b ;;
val cons1 : 'a -> 'a list -> 'a list = <fun>
# cons1 34 [] ;;
- : int list = [34]
# cons1 45 [32];;
- : int list = [45; 32]
```

Scheme では、カリー化した関数につける括弧に気をつけなければならなかった。しかし、Ocaml ではそんな事を一切気にすることなくカリー化した関数を使用できる。Ocaml の方が、カリー化は使いやすいと言える。

演習 1

1. 数学記法の $\sum$ は，1 引数関数（単項関数）f および 下限 a，上限 b をもらって，$f(a) + f(a+1) + \cdots + f(b)$ を返す「関数」とみることができる.$\sum$ を Scheme で定義せよ.

2. (myexpt 2) が 二乗関数，(myexpt 3) が三乗関数，... （以下同様）として使えるように，myexpt を定義せよ.

Answer

Scheme、Ocaml の双方でプログラムを実行してみる。

1.

1. Scheme の場合

```
(define sigma
  (lambda (f)
    (lambda (a)
      (lambda (b)
        (cond
          ((>= b a) (+ (f a) (((sigma f) (+ a 1)) b)))
          (else 0))))))

(define sigma
  (lambda (f)
    (lambda (a)
      (lambda (b)
        (cond
          ((>= b a) (+ (f a) (sigma f (+ a 1) b)))
          (else 0))))))
```

以上の２つの関数 sigma を良く見比べてもらいたい。最初は下のものだったが、これでは実はうまくいかない。何故なら、再帰部分でカリー化が失敗しているからである。Scheme でカリー化をするときは括弧に出来るだけ気をつける。

上のものは括弧の付け方がカリー化に適したものになっている。実行すると

```
> (define sigma
  (lambda (f)
    (lambda (a)
      (lambda (b)
        (cond
          ((>= b a) (+ (f a) (((sigma f) (+ a 1)) b)))
          (else 0))))))
> (((sigma (lambda (x) (* x x))) 1) 9)
285
> (((sigma (lambda (x) (- x 1))) 0) 5)
9
```

となる。

2. Ocaml の場合

```
let rec sigma f = fun a -> fun b ->
    if b >= a then f a + sigma f (a+1) b else 0 ;;
```

Scheme よりも見やすく、簡単にかける。実行すると、

```
# let rec sigma f = fun a -> fun b ->
   if b >= a then f a + sigma f (a+1) b else 0 ;;
val sigma : (int -> int) -> int -> int -> int = <fun>
```

```
# sigma (fun x -> x * x) 1 9 ;;
- : int = 285
# sigma (fun x -> x - 1) 0 5 ;;
- : int = 9
```

となる。if 内部は逆のほうが良かったかもしれない。

2.

1. Scheme の場合

```
(define myexpt
  (lambda (n)
    (lambda (x)
      (cond
        ((= n 0) 1)
        (else (* ((myexpt (- n 1)) x) x))))))
(define sq
  (myexpt 2))
(define cu
  (myexpt 3))
```

もちろんカリー化を使い、n 乗するプログラムを返す関数 myexpt を作る。ここでは n をうまく使って、再帰で実現している。sq,cu は引数を 2 乗、3 乗するプログラムである。実行してみると、

```
> (sq 4)
16
> (cu 7)
343
```

となった。良さそうである。

2. Ocaml の場合

```
let rec myexpt n =
  fun x -> if n = 0 then 1
    else (myexpt (n-1) x) * x ;;
```

Scheme のプログラムを変えただけである。

```
# let rec myexpt n =
    fun x -> if n = 0 then 1
      else (myexpt (n-1) x) * x ;;
val myexpt : int -> int -> int = <fun>
# let sq = myexpt 2;;
```

```
val sq : int -> int = <fun>
# let cu = myexpt 3;;
val cu : int -> int = <fun>
# sq 4;;
- : int = 16
# cu 7;;
- : int = 343
```

カリー化をすることで、実現できた。

演習２

1. 「与えられた関数を２回適用する」関数，つまり，たとえば ((double list) 777) を評価すると ((777)) が返ってくるような関数 `double` を定義せよ．

2. (double (double f)) は,「関数 f を４回適用する」関数になりそうである．では ((double double) f) は何をする関数だろうか？

Answer

1.

1. Scheme の場合

```
(define double
  (lambda (f)
    (lambda (x)
      (f (f x)))))
```

実際に使ってみる。

```
> ((double list) 777)
((777))
```

これでよさそうだ。

2. Ocaml の場合
資料に載っている。それをそのまま使う。

```
let double f x = f (f x);;
```

```
# let double f x = f (f x);;
val double : ('a -> 'a) -> 'a -> 'a = <fun>
```

定義は出来た。しかし、list 処理を実行しようとすると、

```
# (double (fun x -> x :: []) 777);;
Characters 18-19:
  (double (fun x -> x :: []) 777);;
                    ^
This expression has type 'a list but is here used with type 'a
```

のようにエラーとなった。

これは、引数となっている関数に原因がある。問題は型で、関数を 2 回適用しようとすると、この場合一回目と型が違ってしまう。それによって型エラーが出たのである。これは今は解決できないので保留という事にしておく。

2.

1. Scheme の場合

```
(define four1
  (lambda (f)
    (double (double f))))

(define four2
  (lambda (f)
    ((double double) f)))

> ((four1 list) 777)
((((777))))

> ((four2 list) 777)
((((777))))
```

double を 2 回適用する関数 four を 2 種類作った。その上で、list 処理させてみた。結果、一見同じように見える。そこでもっと違う関数を適用してみる事にする。

```
> ((four1 (lambda (x)
            (* x x))) 2)
65536
> ((four2 (lambda (x)
            (* x x))) 2)
65536
> ((four1 (lambda (x)
            (- x x))) 2)
0
> ((four2 (lambda (x)
            (- x x))) 2)
0
```

どうも、この方法では良く分からない。埒が明かないので trace してみる。

```
> (trace four1)
(four1)
> ((four1 list) 777)
|(four1 #<primitive:list>)
|#<procedure:14:4>
((((777))))

> (trace four2)
(four2)
> ((four2 list) 777)
|(four2 #<primitive:list>)
|#<procedure:14:4>
((((777))))
```

four 関数で見る限り、全く同じ事をしている。仕方がないので double 自体を trace してみる。

```
> (trace double)
(double)

> (double (double list))
|(double #<primitive:list>)
|#<procedure:14:4>
|(double #<procedure:14:4>)
|#<procedure:14:4>
#<procedure:14:4>

> ((double double) list)
|(double #<procedure:traced-double>)
|#<procedure:14:4>
|(double #<primitive:list>)
|#<procedure:14:4>
|(double #<procedure:14:4>)
|#<procedure:14:4>
#<procedure:14:4>
```

(double (double list)) はそのまま定義できているが、((double double) list) は再帰的に定義されている。しかし、結果は同じという珍しいものになっている。仕事量からすると上の定義のほうがよさそうだ。

2. Ocaml の場合

今度は Ocaml でやってみる。型を表示してくれるので面白い結果になるかもしれない。

```
# let four1 x = double (double x);;
val four1 : ('a -> 'a) -> 'a -> 'a = <fun>
```

```
# let four2 x = (double double) x;;
val four2 : ('a -> 'a) -> 'a -> 'a = <fun>
```

trace をかけてみる。

```
# four1 (fun x -> x * x);;
four1 <-- <fun>
four1 --> <fun>
- : int -> int = <fun>

# four2 (fun x -> x * x);;
four2 <-- <fun>
four2 --> <fun>
- : int -> int = <fun>
```

逆に Ocaml では何も分からないということが判明した。結果的に、両者は同じ事をする関数であるといえる事が分かった。

演習 3

(a b c d) および (p q) という 2 本のリストを受け取ると、両者の「直積」 ((a p) (a q) (b p) (b q) (c p) (c q) (d p) (d q)) を返すような関数 product をなるべく簡潔に定義せよ.

Answer

http://www.altum.jp/math/exp2a/prac/12.html より、直積の Scheme のプログラムは

```
(define (direct-product G H)
  (apply append
         (map (lambda (x)
                      (map (lambda (y) (list x y)) H)
                      ) G)
         )
  )
```

となる。

実際に Scheme で実行してみると、

```
> (define (direct-product G H)
  (apply append
         (map (lambda (x)
                      (map (lambda (y) (list x y)) H)
                      ) G)
         )
```

```
  )
> (direct-product '(1 2) '(3 4 5))
((1 3) (1 4) (1 5) (2 3) (2 4) (2 5))
> (direct-product '(2 3 4 5) '(2 3))
((2 2) (2 3) (3 2) (3 3) (4 2) (4 3) (5 2) (5 3))
```

となった。これを Ocaml でも実装したい。

```
let direct_product g =
 fun h -> map (fun x -> map (fun y -> x :: y :: []) h) g;;
# let direct_product g =
   fun h -> map (fun x -> map (fun y -> x :: y :: []) h) g;;
val direct_product : 'a list -> 'a list -> 'a list list list = <fun>
# direct_product [1;2] [3;4];;
- : int list list list = [[[1; 3]; [1; 4]]; [[2; 3]; [2; 4]]]
```

上のプログラムだと、list の中身がおかしくなっている。作ったリストを map でリストに戻すからおかしくなるのだろう。

scheme だと

```
> (append '(1 2) '(3 4) '(5 6))
(1 2 3 4 5 6)
```

が出来るが、Ocaml だと

```
# (@) [1;2;3] [4;5;6] [7;8;9];;
Characters 0-3:
  (@) [1;2;3] [4;5;6] [7;8;9];;
  ^^^
This function is applied to too many arguments,
maybe you forgot a ';'
# (+) 1 2;;
- : int = 3
# (@) [1] [2];;
- : int list = [1; 2]
```

となってしまう。append はそのままでは引数を複数取れないし、apply もない（はず）。しかし、append の定義を少し変えれば出来そうだ。apply と append の機能を併せ持つような関数を定義してみる。

```
let rec apply_append l =
  match l with [] -> [] |
   x :: rest -> append x (apply_append rest);;
```

これを使うと、

```
let direct_product g = fun h ->
 apply_append (map (fun x -> map (fun y -> x :: y :: []) h) g);;
```

が定義できる。実際に使ってみる。

```
# let rec apply_append l =
   match l with [] -> [] |
    x :: rest -> append x (apply_append rest);;
val apply_append : 'a list list -> 'a list = <fun>
# let direct_product g = fun h ->
   apply_append (map (fun x -> map (fun y -> x :: y :: []) h) g);;
val direct_product : 'a list -> 'a list -> 'a list list = <fun>
# direct_product [1;2] [3;4];;
- : int list list = [[1; 3]; [1; 4]; [2; 3]; [2; 4]]
# direct_product [1;2;3] [4;5;6];;
- : int list list =
[[1; 4]; [1; 5]; [1; 6]; [2; 4]; [2; 5]; [2; 6]; [3; 4]; [3; 5]; [3; 6]]
```

となり、出来ていることがわかった。

リスト処理パターンの発見と畳み込み (fold) 関数

```
(define sum-list
  (lambda (x)
    (cond
      ((null? x) 0)
      (else (+ (car x) (sum-list (cdr x)))))))
```

Ocaml に直すと、

```
let rec sum_list l = match l with [] -> 0 | x :: rest -> x + (sum_list rest);;
```

```
# let rec sum_list l =
   match l with [] -> 0
    | x :: rest -> x + (sum_list rest);;
val sum_list : int list -> int = <fun>
# sum_list [1;2;3;4;5;6;-2];;
- : int = 19
# sum_list [-1;-2;-3;-4;-5];;
- : int = -15
# sum_list [2];;
- : int = 2
```

```
(define fold-right
  (lambda (f x unit)
    (cond
      ((null? x) unit)
      (else (f (car x) (fold-right f (cdr x) unit))))))
```

Ocaml では

```
let rec fold_right f l e =
 match l with
  [] -> e
  | x :: rest -> f x (fold_right f rest e);;
```

演習 4

1. この fold-right を使って，これまでに出てきた sum-list, mymax, myappend と同じ機能を実現してみよ．

2. fold-right とは左右逆に (f ... (f (f (f unit a1) a2) a3) ... an) を計算する関数 fold-left は，

```
(define fold-left
  (lambda (f unit x)
  (cond
    ((null? x) unit)
    (else (fold-left f (f unit (car x)) (cdr x))))))
```

と書ける．これを使って sum-list や mymax を定義してみよ．さらに，これを使って myreverse が定義できるかどうか，工夫してみよ．

Answer

1.

1. Scheme の場合

```
> (define sum-list
   (lambda (x)
      (fold-right + x 0)))
> (sum-list '(1 2 3 4 5 6 7 8 9))
45
> (sum-list '(-1 -2 -3 -4))
-10
```

ただ単に fole-right の引数に+を入れるだけで実現できる。x はリスト、unit は 0 である。

mymax は以下のものであった。

```
(define mymax
  (lambda (l)
(cond
  ((null? l) "Not defined")
```

```
  ((null? (cdr l)) (car l))
  ((< (car l) (mymax (cdr l))) (mymax (cdr l)))
  (else (car l))))))
```

これを改良すると

```
(define mymax
  (lambda (l)
    (cond
     ((null? l) "Not defined")
      (else (fold-right (lambda (x y)
        (cond
        ((< x y) y)
        (else x))) l -111)))))
```

fold-right の引数となる関数に cond を使う事でうまく比較を表現した。
この方法の問題点はリストがすべて-111 以下ならばうまくいかないということである。

```
> (mymax '(1 3 5 6 7 ))
7
> (mymax '(5 67 9 33 235 35 67 ))
235
> (mymax '(-123 -567 -222))
-111
> (mymax '())
"Not defined"
> (mymax '(1 2 3 5 9))
9
```

うまくいかせる方法もあるのだろうが、ちょっと分からなかった。

myappend は以下の様なものであった。

```
(define myappend
  (lambda (x y)
    (cond
      ((null? x) y)
      (else (cons (car x) (myappend (cdr x) y))))))
```

これを fold-right を使って定義すると

```
(define myappend
  (lambda (x y)
    (fold-right cons x y)))
```

実際に使ってみると

```
> (myappend '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
> (myappend '(3 5 6) '())
(3 5 6)
```

2. Ocaml の場合

```
let sum_list l = fold_right (fun x y -> x + y) l 0;;
```

使ってみると

```
# let sum_list l =
   fold_right (fun x y -> x + y) l 0;;
val sum_list : int list -> int = <fun>
# sum_list [1;2;3;4;5];;
- : int = 15
# sum_list [-2;3;4;5];;
- : int = 10
# sum_list [];;
- : int = 0
```

mymax の場合は少し難しく

```
let mymax l = if (null l) then "Not defined"
 else fold_right (fun x y -> if (x < y) then y else x) l -111;;
```

これで実行してみようとしたところ

```
# let mymax l = if (null l) then "Not defined"
   else (fold_right (fun x y -> if (x < y) then y else x) l 0);;
Characters 51-101:
  let mymax l = if (null l) then "Not defined"
   else (fold_right (fun x y -> if (x < y) then y else x) l 0);;

This expression has type 'a -> 'a but is here used with type int
```

というエラーが出た。Ocaml の強い型制限によって、Scheme のようにはいかないことが分かる。そこで妥協して

```
#   let mymax l = fold_right (fun x y -> if (x < y) then y else x) l 0;;
val mymax : int list -> int = <fun>
# mymax [1;2;3;4;6];;
- : int = 6
# mymax [3;4;6;9;2;45;7];;
- : int = 45
```

とした。実行もよさそうだ。ただし、例によってリストの要素が全て 0 より小さいときは問題が起こる。

Ocaml では型が違うとエラーが起こってくる。そこが難しい。

```
let myappend l1 l2 = fold_right (fun x y -> x :: y) l1 l2;;
```

myappend を実行すると以下のようになる。

```
# let myappend l1 l2 = fold_right (fun x y -> x :: y) l1 l2;;
val myappend : 'a list -> 'a list -> 'a list = <fun>
# myappend [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
# myappend [3;5;7] [24;667;23];;
- : int list = [3; 5; 7; 24; 667; 23]
```

2. fold-left を OCaml で書くと

```
let rec fold_left f e l =
 match l with [] -> e | x :: rest -> fold_left f (f e x) rest;;
```

1. Scheme の場合

```
> (define sum-list
   (lambda (x)
      (fold-left + 0 x)))
> (sum-list '(1 2 3 4 5 6 7 8 9))
45
> (sum-list '(-1 2 -3 4))
2

(define mymax
  (lambda (l)
    (cond
      ((null? l) "Not defined")
      (else (fold-left (lambda (x y)
        (cond
          ((< x y) y)
          (else x))) -111 l)))))
> (mymax '(1 2 3 4 5))
5
> (mymax '(3 4 7 2 3 98 3 2 ))
98

(define myreverse
```

```
  (lambda (seq)
    (fold-left (lambda (x y)
      (cons y x)) '() seq)))
> (myreverse '(2 4 5))
(5 4 2)
> (myreverse '(1 2 3 4 5 6 7 8 9))
(9 8 7 6 5 4 3 2 1)
```

2. Ocaml の場合

```
let sum_list l = fold_left (fun x y -> x + y) 0 l;;

# let sum_list l = fold_left (fun x y -> x + y) 0 l;;
val sum_list : int list -> int = <fun>
# sum_list [1;2;43;5];;
- : int = 51
# sum_list [1;-2;5;7];;
- : int = 11
# sum_list [-4;0];;
- : int = -4
```

Ocaml で myreverse を定義すると

```
let myreverse l = fold_left (fun x y -> y :: x) [] l;;

# let myreverse l = fold_left (fun x y -> y :: x) [] l;;
val myreverse : 'a list -> 'a list = <fun>
# myreverse [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]
# myreverse [];;
- : 'a list = []
# myreverse [[1;2;4];[2;3;5];[4;7;8]];;
- : int list list = [[4; 7; 8]; [2; 3; 5]; [1; 2; 4]]
```

演習５（汎用ソーター）

1. 整数を要素とするリストを降順に整列（ソーティング）する関数を書け．効率は O(n2) よりも悪くなければよいものとする．

2. 降順にしか整列できないのでは汎用性に欠けるので，上で書いた整列関数の「整数を要素とする」および「降順」という部分を一般化した整列関数を書け．たとえば (sort ¿= '(3 1 4 1 5 9)) を実行すると (9 5 4 3 1 1) が返ってくるようになっていればよい．(順序判定のための ¿= を引数に指定しているところに注目．)

3. それを用いて,「多数の x-y 座標 （リスト (x y) で表現，x, y は整数） のリストをもらって，x 座標の小さな順 （ただし x 座標が同じ点は y 座標の小さな順） に並べ替える」 作業を行なってみよ.

例 ((4 2) (1 3) (3 5) (8 6) (4 7) (8 1)) を与えたら ((1 3) (3 5) (4 2) (4 7) (8 1) (8 6)) が返ってくればよい.

Answer

Ocaml の資料に２種類ソートが載っている。insertion sort で考えてみる。

```
let rec insert (x : float) =
 function [] -> [x] |
  (y :: rest) as l -> if x < y then x :: l
   else y :: (insert x rest);;

let rec insertion_sort =
 function [] -> [] | x :: rest -> insert x (insertion_sort rest);;
```

より

```
(define insert
  (lambda (y)
    (lambda (l)
      (cond
        ((null? l) (cons y '()))
        (else (cond
                ((< y (car l)) (cons y l))
                (else (cons (car l) ((insert y) (cdr l))))))))))

(define insertion_sort
  (lambda (l)
    (cond
     ((null? l) '())
     (else ((insert (car l)) (insertion_sort (cdr l)))))))
```

となる。実行してみると

```
> (insertion_sort '(1 3 5 2 4))
(1 2 3 4 5)
> (insertion_sort '(3 5 67 1 34))
(1 3 5 34 67)
```

となったので大丈夫そうだ。

```
(define insert_op
  (lambda (op x l)
    (cond
      ((null? l) (cons x '()))
      (else
        (cond
          ((op x (car l)) (cons x l))
          (else (cons (car l) (insert_op op x (cdr l)))))))))

(define i_s_op
  (lambda (op l)
    (cond
      ((null? l) '())
      (else (insert_op op (car l) (i_s_op op (cdr l)))))))

> (insert_op < 7 '(1 3 5 8))
(1 3 5 7 8)
> (i_s_op < '(1 3 5 2 4))
(1 2 3 4 5)
> (i_s_op > '(1 3 5 2 4))
(5 4 3 2 1)
```

となったので実装できた。ただし、これは op に何が入っても良くなっているので、注意。
Ocaml でやってみる。

```
# let rec insert_op op (x : float) l =
 match l with [] -> [x] | (y :: rest) as l ->
  if (op x y) then x :: l else y :: (insert_op op x rest);;
val insert_op : (float -> float -> bool) -> float -> float list
 -> float list = <fun>

# insert_op (fun x y -> if x < y then true else false) 2. [3.;5.;1.];;
- : float list = [2.; 3.; 5.; 1.]
```

insert は実装できた。ただ、引数が関数のため指定が長くなってしまう。

```
# let rec i_s_op op =
   function [] -> [] | x :: rest -> insert_op op x (i_s_op op rest);;
val i_s_op : (float -> float -> bool) -> float list -> float list = <fun>
# i_s_op (fun x y -> if x < y then true else false) [1.;3.;5.;2.;4.];;
- : float list = [1.; 2.; 3.; 4.; 5.]
```

多分、出来た。

感想

思った事、Scheme は上部で定義を確認しながらできるところは良い。Scheme だと書き易いものも、Ocaml だと書き易いものも、両方ある。

参考文献

[1] http://www.altum.jp/math/exp2a/prac/12.html
直積を解くときに役立った。

[2] http://www.sampou.org/scheme/sicp/answer/sicp.2.2.html
部分集合はここでは使わなかったが、勉強になった。

[3] http://www.sampou.org/scheme/sicp/mailingList/msg00880.html

[4] http://www.sampou.org/scheme/sicp/mailingList/msg00045.html
畳み込みは難しかったので、大いに役立った。