2　高階関数とリスト

配列についての補足

```
# let arr = Array.create 5 0;;                    (* 長さ 5, 初期値 0 の配列を作る *)
val arr : int array = [|0; 0; 0; 0; 0|]
# for i = 0 to 4 do arr.(i) <- i+1 done;;                          (* 内容の変更 *)
- : unit = ()
# arr;;
- : int array = [|1; 2; 3; 4; 5|]
# let arr2 = Array.init 5 (fun x -> x*x);;                  (* 作成と同時に初期化 *)
val arr2 : int array = [|0; 1; 4; 9; 16|]
```

とても高階関数

```
# let iter (f : 'a -> unit) (arr : 'a array) =
    for i = 0 to Array.length arr - 1 do
      f arr.(i)
    done;;
val iter : ('a -> unit) -> 'a array -> unit = <fun>
# let local x0 (f : 'a ref -> unit) =
    let r = ref x0 in f r; !r;;
val local : 'a -> ('a ref -> unit) -> 'a = <fun>
# let sum (arr : int array) =
    local 0 (fun r -> iter (fun x -> r := !r + x) arr);;
val sum : int array -> int = <fun>
```

これが先週の for ループによる定義と同値であることを確認するために，各関数を展開すればいい.

$$\textbf{let } f\ x_1 \dots x_n = E$$

の元で，次の展開ができる

$$f\ E_1 \dots E_n \Longrightarrow \textbf{let } x_1 = E_1 \textbf{ and } \dots \textbf{ and } x_n = E_n \textbf{ in } E$$

x の定義である v が値か名前なら，そして x および v(名前なら) が E の中で再定義されていない場合，それを E の中に代入することもできる (E の中の x を全て v に変える)

$$\textbf{let } x = v \textbf{ in } E \Longrightarrow E\{x := v\}$$

では，展開してみよう.

```
local 0 (fun r -> iter (fun x -> r := !r + x) arr)
        ⇓
let x0 = 0 and f r = iter (fun x -> r := !r + x) arr in
let r = ref x0 in f r; !r
        ⇓
```

```
let r = ref 0 in
(let r = r in iter (fun x -> r := !r + x) arr);
!r
```

⇓

```
let r = ref 0 in
let f x = r := !r + x and arr = arr in
for i = 0 to Array.length arr - 1 do f arr.(i) done; !r
```

⇓

```
let r = ref 0 in
for i = 0 to Array.length arr - 1 do r := !r + arr.(i) done; !r
```

組型

```
# let triple = (2, 2.5, 3) ;;                          (* 組の型は要素ごと *)
val triple : int * float * int = (2, 2.5, 3)
# let (a,b,c) = triple ;;
val a : int = 2
val b : float = 2.5
val c : int = 3
# let minmax (x,y : int * int) =                       (* 引数や結果に便利 *)
    if x < y then (x,y) else (y,x) ;;
val minmax : int * int -> int * int = <fun>
```

リストとパターン・マッチング

```
# let l1 = [1;2;3] ;;                    (* 配列と同様，リストの要素は同じ型 *)
val l1 : int list = [1; 2; 3]
# let l2 = 5 :: l1 ;;
val l2 : int list = [5; 1; 2; 3]
# 1 :: 2 :: 3 :: [] ;;                          (* 本来の構造を書き下す *)
- : int list = [1; 2; 3]
# List.hd l1 ;;                                        (* リストの頭部 *)
- : int = 1
# List.tl l1 ;;                                        (* リストの後部 *)
- : int list = [2; 3]
# List.length l1 ;;                                    (* リストの長さ *)
- : int = 3
# match l1 with a :: l -> (a,l) ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
- : int * int list = (1, [2; 3])              (* Warningでも結果が出る *)
# List.tl [] ;;
Exception: Failure "tl".                               (* 実行時エラー *)
# match [] with
    [] -> 0
```

```
  | a :: l -> 1 ;;
- : int = 0
```

反復と再帰

```
# let reverse l =
    let l1 = ref l and l2 = ref [] in
    while !l1 <> [] do                                    (* l1 が空になるまで反復 *)
      l2 := List.hd !l1 :: !l2;
      l1 := List.tl !l1
    done;
    !l2;;
val reverse : 'a list -> 'a list = <fun>
# reverse l1 ;;
- : int list = [3; 2; 1]
# let reverse l =
    let l1 = ref l and l2 = ref [] in
    while match !l1 with                                  (* 分岐と読み出しを同時に行う *)
      [] -> false
    | a :: l -> l1 := l; l2 := a :: !l2; true
    do () done;                                           (* ここですることがない! *)
    !l2;;
val reverse : 'a list -> 'a list = <fun>
# let reverse l =
    let rec rev l1 l2 =                                   (* 再帰関数を使う *)
      match l1 with
        [] -> l2
      | a :: l -> rev l (a::l2)
    in
    rev l [];;
val reverse : 'a list -> 'a list = <fun>
```

末尾再帰とそうでないもの

上の 3 つの reverse の中で，最後のものが最も効率がいい．再帰呼び出しの結果はそのまま関数の結果になるので，呼び出しの入れ子なしに計算できる．

常にそういうわけにはいかない．

```
# let rec append l1 l2 =
    match l1 with
      [] -> l2
    | a :: l -> a :: append l l2 ;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
# let rec rev_append l1 l2 =
    match l1 with
```

```
      [] -> l2
    | a :: l -> append l (a :: l2) ;;
val rev_append : 'a list -> 'a list -> 'a list = <fun>
# rev_append [1;2;3] [4;5;6];;
- : int list = [3; 2; 1; 4; 5; 6]
# let rec append' l1 l2 =
    rev_append (rev_append l1 []) l2 ;;
val append' : 'a list -> 'a list -> 'a list = <fun>
# let rec reverse' l =
    match l with
      [] -> []
    | a :: l -> append (reverse' l) [a] ;;
val reverse' : 'a list -> 'a list = <fun>
```

append の定義は再帰呼び出しの結果をリストの後部として使うので，末尾再帰ではない．実行時にスタックに途中の状態を積んでいく．

しかし，スタックが溢れない限り，プログラムの効率にとって最も重要なのは新しいリスト・セルをどれだけ作ったかである．append と rev_append のどちらでも，l1 の長さ分だが，末尾再帰である append' だとその倍になる．

reverse の直感的な実装である reverse' はやってはいけないことの例である．前節の reverse は皆，l の長さ分のセルで済んでいたのに，今度は append を繰り返し利用する．append の作るセルの数は第一引数に比例するので，reverse' を長さ n のリストに適用すると $\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$ 個のセルを作る．しかも，reverse' 自体も末尾再帰ではない．

実習課題 (2 回目)

1. リストの長さを計算する再帰関数 `length : 'a list -> int` を定義せよ．末尾再帰ですか？そうでなければ，末尾再帰にできますか？

2. リストの要素の合計を計算する再帰関数 `sum : int list -> int` を定義せよ．

3. 配列をリストに変換する関数とその逆関数を定義せよ．
   `val list_of_array : 'a array -> 'a list`
   `val array_of_list : 'a list -> 'a array`
   具体的な値で `list_of_array (array_of_list l) = l` であることを確認せよ．

4. リストと配列のどちらでも行列の転置を定義せよ．
   `val list_transpose : 'a list list -> 'a list list`
   `val array_transpose : 'a array array -> 'a array array`
   リストの場合では再帰関数を使え．配列の場合では `Array.init` を使え．

5. リストと配列のどちらでも行列の積を定義せよ: $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$
   `val list_prod : float list list -> float list list -> float list list`
   `val array_prod : float array array -> float array array -> float array array`
   要素の積をパラメーターに取った定義もできますか？

明日までに 感想や質問のできた分だけを computer-lecture-2005-aw-4@math.nagoya-u .ac.jp 宛に送って下さい．