Jacques Garrigue, 2007 年 10 月 31 日

再帰関数プログラミング 2

7　リストと構造的帰納法

リスト

```
# [1; 2; 3];;                                    (* リストは配列のように書く *)
- : int list = [1; 2; 3]
# 1 :: [2; 3];;                          (* cons(コンス) という構成子で作られる *)
- : int list = [1; 2; 3]
# 1 :: (2 :: (3 :: []));;                          (* これが本当の内部構造 *)
- : int list = [1; 2; 3]
# List.hd [1;2;3];;                                          (* リストの頭 *)
- : int = 1
# List.tl [1;2;3];;                                        (* リストの尻尾 *)
- : int list = [2; 3]
```

リストは配列のようなデータ構造だが，簡単に頭に値を追加・削除できるので重宝される．
　リストに対して多くの関数が定義されている．

```
List.length     : 'a list -> int
List.hd         : 'a list -> 'a
List.tl         : 'a list -> 'a list
List.nth        : 'a list -> int -> 'a
List.rev        : 'a list -> 'a list
List.append     : 'a list -> 'a list -> 'a list        (* l1 @ l2 とも書く *)
List.flatten    : 'a list list -> 'a list
List.iter       : ('a -> unit) -> 'a list -> unit
List.map        : ('a -> 'b) -> 'a list -> 'b list
List.fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
List.for_all    : ('a -> bool) -> 'a list -> bool
List.exists     : ('a -> bool) -> 'a list -> bool
List.mem        : 'a -> 'a list -> bool
List.filter     : ('a -> bool) -> 'a list -> 'a list
List.split      : ('a * 'b) list -> 'a list * 'b list
List.combine    : 'a list -> 'b list -> ('a * 'b) list
List.assoc      : 'a -> ('a * 'b) list -> 'b
List.mem_assoc  : 'a -> ('a * 'b) list -> bool
List.remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
...
```

練習問題 7.1 型と名前から，各関数の働きを推理せよ．試しに値を渡してもいい．

パターン・マッチング

```
# let hd l =
    match l with
      []     -> failwith "List.hd"                    (* エラーを起こす *)
    | a :: _ -> a ;;                                    (* 頭部を返す *)
```

```
val hd : 'a list -> 'a
# let tl l =
    match l with
      []     -> failwith "List.tl"
    | _ :: t -> t ;;                                    (* 後部を返す *)
val tl : 'a list -> 'a list
```

パターンマッチングは以下の構造の計算式である.

match 計算式 **with**
  パターン$_1$ ->
    計算式$_1$
| ...
| パターン$_n$ ->
    計算式$_n$

パターンは構成子と名前だけでできた式である．パターンを順番に見て，マッチする値がパターン$_i$と同じ形をしていれば，パターン$_i$の中の名前をマッチする値の対応する部分に束縛して ('_' は束縛されない特別な名前)，計算式$_i$の結果を返す.

リストと再帰　リストに対する関数のほとんどは再帰的に定義されている.

```
# let rec length l =
    match l with
      []     -> 0                                   (* 空リストの長さは0 *)
    | _ :: l' -> 1 + length l' ;;
val length : 'a list -> int
# let rec append l1 l2 =
    match l1 with
      []     -> l2                                  (* l1が空リストならl2を返す *)
    | a :: l' -> a :: append l' l2 ;;
val append : 'a list -> 'a list -> 'a list
# let rec iter (f : 'a -> unit) (l : 'a list) =
    match l with                                    (* リストの先頭からfを適用する *)
      []     -> ()
    | a :: l' -> f a; iter f l' ;;
val iter : ('a -> unit) -> 'a list -> unit
# let rec map (f : 'a -> 'b) (l : 'a list) =
    match l with
      []     -> []          (* 頭にfを適用してからリストの残りにmapをかける *)
    | a :: l' -> let b = f a in b :: map f l' ;;
val map : ('a -> 'b) -> 'a list -> 'b list
```

練習問題 7.2　*1.* リストの和を計算する関数を再帰関数として書きなさい.

```
val sum : int list -> int
```

2. ある条件を満している要素のリストを返す関数 filter を書きなさい.

```
val filter : ('a -> bool) -> 'a list -> 'a list
```

3. 配列に対して定義した fold をリストに対する再帰関数として書きなさい.

```
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

上記の List.fold_left に当る.

4. リストを多項式と見做し *(*頭が定数*)*，ある点で多項式の値を計算する関数を定義せよ.

```
val eval_poly : int list -> int -> int
```

```
# eval_poly [1; 0; 3] 2 ;;
- :  int = 13                                          (* 1 + 0*2 + 3*4 *)
```

構造的帰納法

上の 3 つの関数では，再帰呼び出しがリストの尻尾に対して行われている．こんなように，再帰呼び出しが引数の一部分に対して行われている帰納法を**構造的帰納法**という．自然数に対する帰納法も構造的帰納法の一例である．

```
# type nat = unit list ;;                                  (* 型の略称を定義する *)
type nat = unit list
# let rec nat_of_int n : nat =                              (* 結果を nat にする *)
  if n <= 0 then [] else () :: nat_of_int (n-1) ;;
val nat_of_int : int -> nat
# let int_of_nat : nat -> int = length ;;                 (* length が変換になる *)
val int_of_nat : nat -> int
# let add_nat : nat -> nat -> nat = append ;;                (* append が足算 *)
val add_nat : nat -> nat -> nat
# int_of_nat (add_nat (nat_of_int 2) (nat_of_int 3));;
- : int = 5
```

多くの関数は直接な構造的帰納法で書けるが，効率がよくない場合もある．たとえば，リストを逆順にする rev 何も考えずに書くとこんな感じになる．

```
let rec rev l =
  match l with
    [] -> []
  | a :: l' -> append (rev l') [a] ;;
```

しかし，これで append が呼ばれる回数は l の長さの 2 乗に比例する．補助関数を使うと長さに対して線形な関数が書ける．

```
let rec rev_append l1 l2 =
  match l1 with
    []     -> l2
  | a :: l -> rev_append l (a :: l2)
;;
let rev l = rev_append l [] ;;
```

検索問題

再帰的な関数は検索アルゴリズムに向いている．例えば，有名な SEND + MORE = MONEY 問題は以下のように解ける．(S と M は 1〜9 の数字, E, N, D, M, O, R, Y は 0〜9 の数字に対応しており，SEND と MORE が対応している 10 進の数の足し算が MONEY になっている．)

```
(* 解の型 (略称) を定義する *)
# type dict = (char * int) list ;;

(* 文字と数字の対応が条件を満しいるか判定する関数 *)
val check : dict -> bool = <fun>

(* 解のリストを作る再帰関数 *)
# let rec search (dict:dict) letters numbers =
  match letters with
    [] -> if check dict then [dict] else []
  | a :: letters ->                          (* letters <- List.tl letters *)
```

```
      let rec choose tried numbers =
        match numbers with
          [] -> []
        | n :: numbers ->                    (* numbers <- List.tl numbers *)
            let sols = search ((a,n)::dict) letters (tried @ numbers) in
            sols @ choose (n::tried) numbers
      in
      choose [] numbers ;;
val search : dict -> char list -> int list -> dict list
# let rec interval m n =
    if m > n then [] else m :: interval (m+1) n ;;
val interval : int -> int -> int list = <fun>
# let solve () =
    search [] ['S';'E';'N';'D';'M';'O';'R';'Y'] (interval 0 9) ;;
val solve : unit -> dict list = <fun>
# solve () ;;
- : dict list = [[('Y',2); ...]]
```

練習問題 7.3 上記の check という関数を定義せよ．List.map, List.assoc と前問の eval_poly を使うといい．

解を見付けるのに，check は何回呼ばれたか？search を少し変えてその回数が減らせる．

最適化問題

最適化問題は検索問題の一種として捉えることができる．まず条件を満す解を全て求め，その中から最適なものを選ぶ．

リストに関する有名な最適化問題の一つは「最大連続区間和」(maximum segment sum, 略して mss) である．(負の整数も含む) 整数のリストから，和が最大になるような連続区間を探す．

```
# let rec segments l =                              (* 区間のリストを作る *)
    match l with
      []      -> [[]]
    | _ :: l' -> prefixes l @ segments l'
  and prefixes l =                        (* リストの先頭から始まる区間のリスト *)
    match l with
      []      -> []
    | a :: l' -> [a] :: List.map (fun l -> a :: l) (prefixes l') ;;
val segments : 'a list -> 'a list list
val prefixes : 'a list -> 'a list list
# segments [1; -2; 3];;
- : int list list = [[]; [3]; [-2]; [-2; 3]; [1]; [1; -2]; [1; -2; 3]]
# let sum = List.fold_left (+) 0 ;;
val sum : int list -> int
let mss l = List.fold_left (fun m l -> max m (sum l)) 0 (segments l) ;;
val mss : int list -> int
# mss [1; -2; 3];;
- : int = 3
```

練習問題 7.4 上記の *mss* のアルゴリズムは長さの *4* 乗の足算を必要としている．足算の数が長さに比例するように書き直せ．ヒント：fold_left を使わずに，一個の再帰関数だけで書ける．