Jacques Garrigue, 2011 年 11 月 9 日

再帰的アルゴリズム

9　リストの整列

集合の要素をある順序で並べるという操作は様々なところで必要になる．(可変な) 配列に対して行うこともできるが，不変なリスト構造に対しても同じぐらい効率よくできる．

挿入による整列

人間が自然に行う整列の一つである．元のリストから要素を順に取って行き，それを新しいリストの正しい所に挿入して行く．リストならではのやりかたでもある．

```
let rec insert l x =
  match l with
    [] -> [x]
  | a::l' ->
      if x < a then x :: l else a :: insert l' x ;;
val insert : 'a list -> 'a -> 'a list
let insertion_sort l =
  List.fold_left insert [] l ;;
val insertion_sort : 'a list -> 'a list
insertion_sort [3; 1; 5; 2] ;;
- : int list = [1; 2; 3; 5]
```

とても分かりやすいやりかただが，調べて見ると効率があまりよくない．例えば，比較の回数を数えると，元々整列されていたリストに対して行うと，毎回構築して行くリストの全ての要素と比較しないと行けない．n がリストの長さなら，

$$\text{比較の回数} = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n)$$

作らなければいけないリストのセルの数も同様に $O(n^2)$ である．

とても運がよいときは入力が逆順になっていると，n 回で済むが，全ての順序に対する平均を考えるとやはり $O(n^2)$ になる．

練習問題 9.1 元のリストが整列されていたということはよくあるので，そのときは n 回で済むように定義を簡単に変えられるか？そうすると逆順の場合はどうなるか？

融合による整列　もう一つの基本的な考え方は，要素をリストに挿入するのではなく，既に整列された二つのリストを融合することである．

```
let rec merge l1 l2 =
  match l1, l2 with
    [], _ -> l2
```

```
  | _, [] -> l1
  | a :: l1', b :: l2' ->
      if a < b then a :: merge l1' l2 else b :: merge l1 l2'
;;
val merge : 'a list -> 'a list -> 'a list
let rec merge_one ll =
  match ll with
    l1 :: l2 :: ll' -> merge l1 l2 :: merge_one ll'
  | _               -> ll ;;
val merge_one : 'a list list -> 'a list list
let rec merge_all ll =
  match ll with
    []  -> []
  | [l] -> l
  | _   -> merge_all (merge_one ll) ;;
val merge_all : 'a list list -> 'a list
let merge_sort l =
  merge_all (List.map (fun x -> [x]) l) ;;
val merge_sort : 'a list -> 'a list
# merge_sort [3; 1; 5; 2] ;;
- : int list = [1; 2; 3; 5]
```

こちらの比較の回数を数えると，merge に関して最高で l1 と l2 の長さの合計である (最低でもそのどちらかの長さ)．merge_one では全ての部分リストに対してそれを行うので，元のリストの長さに比例する．merge_all が繰り返される回数は $\log_2 n$ なので，全体では

$$\text{比較の回数} = O(n \log_2 n)$$

挿入整列よりはだいぶ早い．偶然に早くなることはないが，平均ではるかに勝っている．

ちなみに，作られるリストのセルの数は比較の回数に比例するが，最初だけ各要素に対してセルが作られる．ということは，セルの数が $O(n(\log_2 n + 1))$．$\log_2 n$ 自体が小さな数になることが多いので，この差は無視できない．

練習問題 9.2 最初に List.map で一個だけのリストを作る代りに，merge_one みたいに元のリストを *2* 個ずつ見ながら，*2* 個の整列したリストのリストを作る関数 merge_start を書きなさい．

```
val merge_start : 'a list -> 'a list list
```

10　木構造の処理

再帰データ型の定義は線形でなくてもいい．よく知られている例はファイルシステムなどに使われる木構造である．

```
type fs =
    File of string                                  (* ファイルの中身 *)
  | Directory of (string * fs) list ;;
                            (* 名前とファイルまたはフォルダーの連想リスト *)

let desktop = Directory ["test.ml", File "let x = 1;;"] ;;
```

```
val desktop : fs = ...
let home =
  Directory ["Desktop", desktop; ".emacs", File "(require 'caml-font)"] ;;
val home : fs = ...

let rec lookup path fs =
  match path, fs with                    (* path と fs を同時にマッチする *)
  | name :: rem, Directory dir ->          (* ファイルをフォルダーで探す *)
      lookup rem (List.assoc name dir)       (* 連想リストをキーで探す *)
  | [], _ -> fs                    (* ファイルまたはフォルダーが見つかった *)
  | _ -> failwith "lookup"                   (* ファイルの中には入れない *)
;;
val lookup : string list -> fs -> fs = <fun>

lookup ["Desktop"; "test.ml"] home;;
- : fs = File "let x = 1;;"
```

練習問題 10.1 特定の場所にファイルやフォルダーを挿入する関数を書きなさい.

```
val add : string list -> fs -> fs -> fs
```

抽象構文と評価

木構造のもう一つの応用は抽象構文の表現である.
以下のような言語を扱う処理系を作りたい.

$$\text{式} ::= \text{実数} \mid \text{変数} \mid \text{式} + \text{式} \mid \text{式} \times \text{式} \mid \sin(\text{式}) \mid \cos(\text{式}) \mid (\text{式})$$

式の例

$$5 \times 2 + 3 \qquad (3 + y) \times 12$$

抽象構文を表す型をまず定義する.

```
type expr =
    X                                          (* 変数は X だけ *)
  | Cst of float                             (* 定数には実数を使う *)
  | Add of expr * expr
  | Mul of expr * expr
  | CMul of float * expr                       (* 定数の掛け算 *)
  | Sin of expr
  | Cos of expr

let rec eval e x =                             (* 評価は再帰関数 *)
  match e with
    X     -> x
  | Cst a -> a
  | Add (e1, e2) -> eval e1 x +. eval e2 x
  | Mul (e1, e2) -> eval e1 x *. eval e2 x
  | CMul (a, e1) -> a *. eval e1 x
  | Sin e1 -> sin (eval e1 x)
  | Cos e1 -> cos (eval e1 x)
```

```
# eval (Mul (Sin X, Sin X)) 1.;;
- : float = 0.708073418273571176
```

こんな式に対して，微分が簡単に定義できる．

```
let derive_op1 = function                              (* 引数に対する微分 *)
    CMul(a, x) -> Cst a
  | Sin x -> Cos x
  | Cos x -> CMul(-1., Sin x)
  | _ -> failwith "derive_op1"
val derive_op1 : expr -> expr

let get_arg = function                              (* 1引数の構文の引数を返す *)
    CMul (_, e1) | Sin e1 | Cos e1 -> e1
  | _ -> failwith "get_arg"                          (* それ以外の構文でエラー *)

let rec derive = function
    X -> Cst 1.
  | Cst _ -> Cst 0.
  | Add (e1, e2) -> Add(derive e1, derive e2)
  | Mul (e1, e2) -> Add (Mul(derive e1, e2), Mul(e1, derive e2))
  | e -> Mul (derive (get_arg e), derive_op1 e)

# derive (Mul (Sin X, Cos X));;
- : expr =
Add (Mul (Mul (Cst 1., Cos X), Cos X),
 Mul (Sin X, Mul (Cst 1., CMul (-1., Sin X))))
```

ここで**function**というキーワードを使っているが，以下の構文の略になる．

```
fun e -> match e in
```

短かい上に引数の名前を考えなくていいから便利だ．

練習問題 10.2　*1.* 様々な式を微分し，正しさを確かめよ．問題*4.1.2*のderiveとも比較せよ．

2. 新しい基本関数を追加せよ．

式の単純化

微分を正しく定義できたものの，式をそのまま出すと冗長な部分が多い．やはり単純しなければならない．

```
# derive (Sin (Add(X, Cst 1.)));;
- : expr = Mul (Add (Cst 1., Cst 0.), Cos (Add (X, Cst 1.)))
```

まず考えられるのは，変数を含まない部分を完全に計算してしまうことだ．

```
let map f e =                                          (* mapも要る *)
  match e with
```

```
    X | Cst _ -> e
  | Add (e1, e2) -> Add (f e1, f e2)
  | Mul (e1, e2) -> Mul (f e1, f e2)
  | CMul (a, e1) -> CMul (a, f e1)
  | Sin e1 -> Sin (f e1)
  | Cos e1 -> Cos (f e1)

let rec const = ...                                    (* X を含むかどうか *)
val const : expr -> bool

let rec simpl e =
  if const e then Cst (eval e 0.) else map simpl e ;;

# simpl (derive (Sin (Add(X, Cst 1.))));;
- : expr = Mul (Cst 1., Cos (Add (X, Cst 1.)))
```

練習問題 10.3 上の const 関数を定義せよ。

これで余計な部分が少し減ったが，まだまだ残っている．
次の考え方は，単純化に使える方程式を利用する．例えば，上の例では，

$$\mathtt{Mul}(\mathtt{Cst\ 1.}, \mathtt{E}) = \mathtt{E}$$

は知られている．

```
exception Nomatch

let simpl1 = function
  | Add(Cst 0., e1) | Add(e1, Cst 0.)           -> e1
  | Mul(Cst a, e1) | Mul(e1, Cst a)             -> CMul(a, e1)
  | Mul(CMul(a, e1), e2) | Mul(e1, CMul(a, e2)) -> CMul(a, Mul(e1, e2))
  | CMul(1., e1) -> e1
  | CMul(0., e1) -> Cst 0.
  | CMul(a, CMul(b, e1)) -> CMul(a*.b, e1)
  | _ -> raise Nomatch

let rec simpl e =
  if const e then Cst (eval e 0.) else
  let e = map simpl e in
  try simpl (simpl1 e) with Nomatch -> e ;;

# simpl (derive (Sin (Add(X, Cst 1.))));;
- : expr = Cos (Add (X, Cst 1.))
```

多項式と正規化

こんな単純化方法では限界がある．例えば，

```
# simpl (derive (Mul (Sin X, Cos X)));;
- : expr = Add (Mul (Cos X, Cos X), CMul (-1., Mul (Sin X, Sin X)))
```

のような結果を多項式として表示したいが，上の構文には多項式を表す方法がない．
単純化を越えて，式を正規化するために，Add と Mul を多項式に変えるといい．
話をはっきりさせるために，もっと制限された式に戻す．

```
type expr0 =
    PowX of float                                          (* X の a 乗 *)
  | Cst of float
  | Add of expr0 * expr0
  | Mul of expr0 * expr0
```

これだったら，変数 X の多項式を次数と係数のリストと見做せばよい．

```
type poly = (float * float) list
let eval_poly p x =
  List.fold_left (fun r (expn, coeff) -> r +. coeff *. (x ** expn)) 0. p
```

もっとも重要な操作は多項式の和だが，次数の順序で並べられていると仮定すると効率よくできる．

```
let rec add_poly p1 p2 =
  match p1, p2 with
    [], _ -> p2
  | _, [] -> p1
  | (x,a)::p1', (y,b)::p2' ->
      if x = y then
        if a +. b = 0. then add_poly p1' p2'
        else (x,a+.b)::add_poly p1' p2'
      else if x < y then (x,a)::add_poly p1' p2
      else (y,b)::add_poly p1 p2'
```

練習問題 10.4 *1.* この add_poly を使って，多項式の掛け算 mul_poly を定義せよ．

2. add_poly と mul_poly を使って，expr0 を poly に変換する関数 poly_of_expr0 : expr0 -> poly を定義せよ．

変数の数を増やすと，上の定義では足りない．

```
type expr1 =
    Pow of string * float
  | ...
type poly = ((string * float) list * float) list
```

数式で書くと

$$P = \Sigma_i(\Pi_j X_{ij}^{a_{ij}}) \cdot b_i$$

このとき，add_poly の定義は変えなくていい．mul_poly では (string * float) list の掛け算を行わなければならないが，実は add_poly がそこでも使える．

練習問題 10.5 多変数多項式における mul_poly および poly_of_expr1 を定義せよ．