Jacques Garrigue, 2009 年 11 月 18 日

再帰敵アルゴリズム2

9　再帰データ型 (続き)

木構造

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 9.3 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．

$$\text{式} ::= \text{整数} \mid \text{変数} \mid \text{式} + \text{式} \mid \text{式} \times \text{式} \mid (\text{式})$$

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 =
    Num of int
  | Var of string
  | Plus of expr * expr
  | Mult of expr * expr
```

それに対して基本操作を定義する．

```
let map_expr f e =                                    (* 再帰的でない map *)
  match e with
  | Num _ | Var _ -> e
  | Plus (e1, e2) -> Plus (f e1, f e2)
  | Mult (e1, e2) -> Mult (f e1, f e2)
val map_expr : (expr -> expr) -> expr -> expr
let rec subst env e =                                 (* 変数の代入 *)
  match e with
  | Var x when List.mem_assoc x env -> Num (List.assoc x env)
  | e -> map_expr (subst env) e
val subst : (string * int) list -> expr -> expr
```

元の map_expr は再帰的でないが，subst の中では再帰的に使われている．汎関数は役に立つ．

ここで新しいパターンも使われている．map_expr では or-パターンで Num と Var を同時に扱っている．subst で条件付きパターンで x が env に含まれているときだけ，代入を行う．条件は bool 型を返さなければならない．

可能な限り式を評価する関数を定義する．

```
let rec eval e =
  match map_expr eval e with
  | Plus (Num x, Num y) -> Num (x + y)
  | Mult (Num x, Num y) -> Num (x * y)
  | e' -> e'
val eval : expr -> expr = <fun>
let e = subst ["x", 3; "z", 2]
                (Plus (Var "y", Mult (Var "x", Var "z")));;
val e : expr = Plus (Var "y", Mult (Num 3, Num 2))
let e' = eval e;;
val e' : expr = Plus (Var "y", Num 6)
```

練習問題 9.4　*1.* expr に関するコードを入力し，例に対して動かす．

2. 式の構文に ‘–式’ を追加し，それに合わせて定義を修正せよ．

3. expr を文字列に変える関数を定義せよ．括弧を多く使ってもいい．