ソフトウェア技法　第 6 回　201311367　佐藤良祐 2014 年 12 月 1 日

課題 6.1

```
(* 実数float 上の関数 f を数値的に微分する関数 *)
let deriv f x s = (f (x +. s) -. f x) /. s;;

(* 関数f, 整数n に対して f を n 回適用する関数*)
let rec applyn f n x =
  if n = 0 then x else if n = 1 then f x
  else applyn f (n -1) (f x);;
```

実行結果

```
# deriv (fun x -> x *. x) 2.0 0.1;;
- : float = 4.10000000000000142
# deriv (fun x -> x *. x *. x) 2.0 0.1;;
- : float = 12.6100000000000101
# applyn (fun x -> x + 2) 4 3;;
- : int = 11
# applyn (fun x -> x * x) 3 2;;
- : int = 256
```

課題 6.2

```
(* 'a -> bool の関数として表された条件を満たす要素と満たさない要素に分割する *)
let rec split f lst =
  match lst with
    [] -> ([], [])
  | x::rest ->
     let (l1, l2) = split f rest in
     if f x then (x::l1, l2) else (l1, x::l2);;


(* 関数のリスト [f0; f1; f2;...fn]と値v に対しリスト [f0 v; f1 v;...]を返す *)
let rec apply_list fl v =
  match fl with
    [] -> []
  | f::rest -> [f v] @ apply_list rest v;;
```

実行結果

```
# split (fun x -> x mod 2 = 0) [3; 1; 2];;
- : int list * int list = ([2], [3; 1])
# split (fun x -> x mod 2 = 0) [];;
- : int list * int list = ([], [])
```

課題 6.3

```
(* 型'a -> boolの関数として表された条件を満たす要素がリストに存在するかどうかを調べる関数 *)
<<<<<<< HEAD
(* リストのリストを連結する関数をfold_rightを用いて書け *)
let rec flatten l =
  match l with
    [] -> []
  | x::rest -> List.fold_right (fun x y -> x @ y) x (flatten rest);;
=======
let rec exists f lst = List.fold_left (fun x y -> x || f y) false lst;;

(* リストのリストを連結する関数をfold_rightを用いて書け *)
let flatten lst = List.fold_right (fun x y -> x @ y) lst [] ;;
>>>>>>> dev6
```

実行結果

```
# exists (fun x -> x > 1) [0; 3];;
- : bool = true
# exists (fun x -> x < -1) [0; 3];;
- : bool = false
# flatten [[1;2]; [3]; []; [4;5]];;
- : int list = [1; 2; 3; 4; 5]
```

課題 6.4

```
(* 有限分岐の木の定義 *)
type 'a ftree = FBr of 'a * 'a ftree list;;

(* 有限分岐の木に対して木の深さを返す関数 *)
let rec fdepth ftree =
  match ftree with
    FBr (v, []) -> 1
  | FBr (v, ts) -> (List.fold_left (fun x y -> max x (fdepth y)) 0 ts) + 1;;

(* 有限分岐の木を先順で走査する関数 *)
let rec fpreorder ftree =
  match ftree with
    FBr (v, []) -> [v]
  | FBr (v, ts) -> List.fold_left (fun x y -> x @ fpreorder y) [v] ts;;
```

実行結果

```
# let ftre = FBr (1, [FBr (2,[]); FBr (3, [FBr (4, [])])]);;
val ftre : int ftree = FBr (1, [FBr (2, []); FBr (3, [FBr (4, [])])])
# fdepth ftre;;
- : int = 3
# fpreorder ftre;;
- : int list = [1; 2; 3; 4]
```

課題 6.5

```
(* リストのリストの各リストの先頭に要素を加える関数 *)
let addelm v lst = List.map (fun y -> [v] @ y) lst;;

(* リストとして表された集合に対してべき集合を計算する関数 *)
let rec powset lst =
  match lst with
    [] -> [[]]
  | [x] -> [[x]] @ [[]]
  | x::rest -> lst :: [x] :: powset rest;;
```

実行結果

```
# addelm 1 [[2;3]; [4]];;
- : int list list = [[1; 2; 3]; [1; 4]]
# powset [1; 2];;
- : int list list = [[1; 2]; [1]; [2]; []]
# powset [1; 2; 3];;
- : int list list = [[1; 2; 3]; [1]; [2; 3]; [2]; [3]; []]
```

課題 6.6

```
(* 命題論理の論理式のデータ型 *)
type prop = Atom of string
          | Neg of prop (* 否定 *)
          | Conj of prop * prop (* And *)
          | Disj of prop * prop (* OR *)
;;

(* union を計算する関数 *)
let rec union xs ys =
  match xs with
    [] -> ys
  | z::zs -> if List.mem z ys then union zs ys
             else z::union zs ys;;

(* 論理式の中に含まれるアトムの集合をリストとして返す *)
let rec atoms pr =
  match pr with
    Atom str -> [str]
  | Neg v -> atoms v
  | Conj (v1, v2) -> union (atoms v1) (atoms v2)
  | Disj (v1, v2) -> union (atoms v1) (atoms v2);;

(* 真のアトムのリストと論理式が与えられたとき論理式の真偽を返す関数 *)
let rec prop lst pr =
  match pr with
    Atom x -> if List.mem x lst then true else false
  | Neg (x) -> if (prop lst x) then false else true
  | Conj (x, y) -> if (prop lst x && prop lst y) then true else false
  | Disj (x, y) -> if (prop lst x || prop lst y) then true else false;;

(* 論理式が充足可能であるかを調べる関数 *)
```

```
let rec satisfiable pr =
  let rec powset lst =
  match lst with
    [] -> [[]]
  | [x] -> [[x]] @ [[]]
  | x::rest -> lst :: [x] :: powset rest in
  List.exists (fun x -> prop x pr) (powset (atoms pr));;
```

実行結果

```
# atoms (Conj (Atom "a", Disj (Atom "b", Atom "c")));;
- : string list = ["a"; "b"; "c"]
# atoms (Conj (Neg (Atom "b"), Conj (Atom "b", Atom "c")));;
- : string list = ["b"; "c"]
# prop ["a"; "c"] (Conj (Atom "a", Disj (Atom "b", Atom "c")));;
- : bool = true
# prop ["a"; "c"] (Conj (Atom "a" , Disj (Atom "b", Neg (Atom "c"))));;
- : bool = false
# satisfiable (Conj (Atom "a" , Disj (Atom "b", Atom "c")));;
- : bool = true
# satisfiable (Conj (Disj (Atom "a" , Neg (Atom "b")),
                    Conj (Neg (Atom "a") , Atom "b")));;
- : bool = false
```