info401 : Programmation fonctionnelle
Contrôle des connaissances – 2
CORRECTION

Pierre Hyvernat
Laboratoire de mathématiques de l'université de Savoie
bâtiment Chablais, bureau 22, poste : 94 22
email : Pierre.Hyvernat@univ-savoie.fr
www : http://www.lama.univ-savoie.fr/~hyvernat/
wiki : http://www.lama.univ-savoie.fr/wiki

Tout comme pour les TP, n'oubliez pas de commentez votre code pour préciser les points importants.

Un point est réservé pour la présentation.

Si vous savez vous en servir, n'hésitez pas à utiliser les fonctions de la librairie `list` :
- `map : ('a -> 'b) -> 'a list -> 'b list`,
- `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`,
- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`,
- `rev : 'a list -> 'a list`,
- ...

Partie 1 : petits exercices

(2) *Question 1.* Les fonctions suivantes sont-elles récursives terminales ?

```
(* calcule le n-ième élément d'une liste *)
let rec nieme n l =  match l with
      [] -> raise (Failure "nieme")   (* exception *)
    | a::l -> if n = 0 then a else nieme (n-1) l

(* vérifie qu'une liste ne contient que des 0 *)
let rec zero l = match l with
      [] -> true
    | 0::l -> zero l
    | _ -> false

(* sépare une liste de paires en deux listes *)
let rec separe l = match l with
      [] -> ([] , [])
    | (a,b)::l -> let r = separe l in
                  let l1 = fst r in
                  let l2 = snd r in
                    (a::l1 , b::l2)

(* insère un élément à sa place dans une liste triée *)
let rec insere x l = match l with
      [] -> [x]
    | a::l when x<a -> x::a::l
    | a::l -> a::(insere x l)
```

Correction : `nieme` et `zero` sont récursives terminales, mais ni `separe` ni `insere` ne le sont...

(2) *Question 2.* Reprogrammez la fonction `taille : 'a list -> int` qui permet de calculer la taille d'une liste.

(2) *Question 3.* Programmez une fonction `compte : int list -> int*int` qui pour une liste d'entiers, fait la somme des nombres positifs et la somme des nombres négatifs. Par exemple :

```
# compte [ 1 ; 2 ; 0 ; -100 ; 3 ; -42 ; 0 ; 1 ] ;;
- : int * int = (7 , -142)
```

Correction :

```
let compte l =
  let rec aux l acc_pos acc_neg =
    match l with
        [] -> (acc_pos , acc_neg)
      | n::l -> if n<0
                then aux l acc_pos (acc_neg+n)
                else aux l (acc_pos+n) acc_neg
  in
  aux l 0 0
```

(2) *Question 4.* Reprogrammez la fonction `range` de type `int -> int -> int list` du contrôle 1. Cette fonction fait la chose suivante : `range a b` devra renvoyer la liste des entiers entre `a` (compris) et `b` (non compris).

Par exemple :

```
range 1 10   -->  [1; 2; 3; 4; 5; 6; 7; 8; 9]
range 10 11  -->  [10]
range 10 10  -->  []
range 11 4   -->  [11; 10; 9; 8; 7; 6; 5]
```

Consigne : votre fonction devra être récursive terminale...

Correction :

```
let range a b =
  let rec aux a b acc =
         if a<b then aux a (b-1) (b-1::acc)
    else if a>b then aux a (b+1) (b+1::acc)
    else (* a=b *)   acc
  in
  aux a b []
```

Partie 2 : arbres

On utilise le type des arbres :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

Tiré d'une correction imaginaire :

```
let ma_fonction a =
  let rec aux a p = match a with
      Vide -> Vide
    | Noeud (e,g,d) -> Noeud ( (e,p) , aux g (p+1) , aux d (p+1) )
  in
  aux a 0
```

(1) *Question 1.* Est-ce que la fonction `ma_fonction` est récursive terminale ?

(1) *Question 2.* Quel est le type de cette fonction ?

(2) *Question 3.* Quelle était la question ?
(Autrement dit, expliquez avec des mots ce que fait cette fonction.)

Correction :
- cette fonction n'est pas récursive terminale,
- son type est `'a arbre -> ('a * int) arbre`,
- cette fonction rajoute à chaque noeud, sa profondeur dans l'arbre.

Partie 3 : permutations

(1) *Question 1.* Écrivez une fonction `flatten : 'a list list -> 'a list` qui concatène, dans l'ordre, toutes les listes contenues dans une liste de listes. Par exemple :
```
# flatten [ [] ; [1;2;3] ; [11] ; [-12;42;666;0;0;0]] ;;
- : int list = [1; 2; 3; 11; -12; 42; 666; 0; 0; 0]
```

Correction : par exemple :
```
let rec flatten r = match r with
    [] -> []
  | l::r -> l @ flatten r
```
ou
```
let flatten r = List.fold_right (fun l r -> l@r) r []
```

(2) *Question 2.* Écrivez une fonction `insere : 'a -> 'a list -> 'a list list` qui prend un élément et l'insère à tous les endroits possibles dans une liste. Par exemple :
```
# insere "coucou" [ "ici" ; "et" ; "la" ] ;;
- : string list list =
[  ["coucou" ; "ici" ; "et" ; "la"];
   ["ici" ; "coucou" ; "et" ; "la"];
   ["ici" ; "et" ; "coucou" ; "la"];
   ["ici" ; "et" ; "la" ; "coucou"]  ]
```

Pour insérer `x` dans la liste `e::l`, il suffit de prendre :
- la liste `x::e::l`,
- toutes les listes de `insere x l`, auxquelles on rajoute `e` devant.

Correction : par exemple :
```
let rec insere x l = match l with
    [] -> [ [x] ]
  | a::m -> (x::l) :: (List.map (fun y -> a::y) (insere x m))
```

(2) *Question 3.* En utilisant les deux fonctions précédentes `flatten` et `insere`, programmez la fonction `permutation : 'a list -> 'a list list` qui calcule toutes les permutations d'une liste. Par exemple
```
# permutation  [666 ;42; 0] ;
- : int list list =
[  [666 ; 42 ; 0] ; [42 ; 666 ; 0] ; [42 ; 0 ; 666] ;
   [666 ; 0 ; 42] ; [0 ; 666 ; 42] ; [0 ; 42 ; 666]  ]
```
Pour calculer les permutations de la liste `a::l`, il suffit de calculer les permutations de la liste `l`, et pour chacune de ces permutations, d'insérer (au sens de la fonction `insere`) `a` partout.

Correction :

```
let rec permutation l = match l with
    a::m -> flatten (List.map (insere a) (permutation m))
  | _ -> [ [] ]
```

Partie 4 : autre

(2) *Question 1.* Il est facile de transformer une fonction récursive terminale en une boucle `while` dans un langage impératif traditionnel (C, Java, Pascal, ...)

Réfléchissez et essayez d'expliquer comment on fait.