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/`

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

Un point négatif est réservé pour la présentation.

Vous avez le droit d'utiliser les fonctions définies dans le questions précédentes, même si vous ne les avez pas écrites.

Partie 1 : questions de cours

(4) *Question 1.* Donnez le type des fonctions suivantes

```
let rec f x y z = match x with
    [] -> z
  | [a] -> y+a
  | a::b::x -> f (b::x) y z

let rec f x y z = match x with
    [] -> (y,z)
  | a::x -> f x (y+1) (a@z)

let rec f x y z = match x with
    [] -> y (fst z)
  | a::x -> f x y (snd z,fst a)
```

Correction :

```
#  let rec f x y z = match x with
       [] -> z
     | [a] -> y+a
     | a::b::x -> f (b::x) y z;;
```

On a :

- `y+a` est une valeur de retour de la fonction, c'est forcément un entier, et `y` et `a` sont des entiers,
- `z` est une autre valeur de retour, c'est donc un entier,
- `x` est une liste qui peut contenir `a` tout seul, c'est donc une liste d'entiers.

Au final, on trouve bien `int list -> int -> int -> int`.

```
#  let rec f x y z = match x with
       [] -> (y,z)
     | a::x -> f x (y+1) (a@z)
```

On a :

- `y+1` devient le deuxième argument dans l'appel récursif, c'est donc un entier ; et `y` est donc aussi un entier,

- a@z devient le troisième argument dans l'appel récursif, c'est donc une liste, et a et z sont des listes de même type arbitraire,
- x est une liste et contient des éléments du même type que a, c'est donc une liste de liste,
- (y,z) est une valeur de retour, la fonction renvoie donc un élément de type int * 'a list.

Au final, on obtient donc 'a list list -> int -> 'a list -> int*'a list.

```
#  let rec f x y z = match x with
       [] -> y (fst z)
     | a::x -> f x y (snd z,fst a)
val f : ('a * 'b) list -> ('a -> 'c) -> 'a * 'a -> 'c = <fin>
```

On a, par le premier cas du match :
- x est une liste,
- z est une paire car on fait fst z,
- y est une fonction qui prend en argument le premier composant de x,
- la valeur de retour de f est du même type que la valeur de retour de y.

Par le deuxième cas du match, on a :
- a est une paire car on fait fst a,
- x est donc une liste de paires,
- le troisième argument devient (snd z, fst a), z est donc une paire et le troisième argument est aussi une paire. La première et deuxième composantes sont de même type, car snd z devient la première composante.

On obtient donc ('a*'b) list -> ('a -> 'c) -> ('a*'a) -> 'c.

(4) *Question 2.* Programmez les fonctions suivantes de manière récursive terminale. Si vous pensez que cela n'est pas possible facilement, donnez une justification. (Pas besoin de la programmer.)
- le minimum d'une liste d'entiers,
- l'avant-dernier élément d'une liste,
- la longueur de la branche gauche d'un arbre binaire,
- la profondeur d'un arbre binaire.

Correction : Par exemple :

```
let minListe l =
  let rec aux l a = match l with
      [] -> a
    | b::l -> aux l (min a b)
  in
  match l with
      [] -> raise (Failure "Liste vide...")
    | a::l -> aux l a

let rec avantDernier l = match l with
    [] | [_] -> raise (Failure "Pas d'avant dernier élément...")
  | [a ; _] -> a
  | _::l -> avantDernier l

type 'a arbre = Vide | Noeud of 'a arbre * 'a * 'a arbre
let longueurGauche a =
  let rec aux a l = match a with
      Vide -> l
    | Noeud(gauche,_,_) -> aux gauche (l+1)
  in aux a 0
```

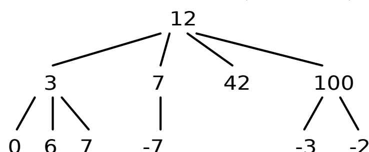
Pour calculer la profondeur d'un arbre binaire, il faut faire deux appels récursifs :
- profondeur du fils gauche,

- profondeur du fils droit.

Il sera donc difficile de programmer cette fonction en récursif terminal car il faudra faire un appel avant l'autre. (C'est quand même possible, mais il faut changer la structure du programme, et cela ne fera rien gagner...)

Partie 2 : Arbres n-aires

Dans un arbre binaire, chaque noeud a exactement deux fils. Nous allons considérer des arbres où chaque noeud peut avoir un nombre quelconque (mais fini) de fils. Par exemple :



La définition du type sera la suivante :

```
type 'a arbre_non_vide = Noeud of 'a * (('a arbre_non_vide) list)
```

L'arbre précédent sera donc représenté par

```
let a = Noeud ( 12 , [ Noeud( 3 , [...]) ;
                       Noeud( 7 , [...]) ;
                       Noeud(42 , [...]) ;
                       Noeud(-1 , [...]) ] )
```

(1) *Question 1.* Donnez une définition du type `'a arbre` qui ajoute l'arbre vide au type `arbre_non_vide`. Ce type sera obtenu à partir du type `arbre_non_vide`.

Correction : On peut utiliser le type `option` :

```
type 'a arbre = 'a arbre_non_vide option
```

ou ajouter un constructeur spécial :

```
type 'a arbre = Non_vide of 'a arbre_non_vide | Vide
```

(2) *Question 2.* Écrivez une fonction `gauche : int arbre_non_vide -> int` qui trouve l'entier tout en bas à gauche dans un arbre. (Dans l'exemple si dessus, il s'agit de l'entier "0".)

Correction : La solution la plus simple est :

```
let rec basGauche a = match a with
    Noeud(x , []) -> x
  | Noeud(_ , a::_) -> basGauche a
```

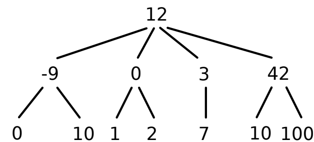
(3) *Question 3.* Écrivez la fonction `profondeur : 'a arbre -> int` qui calcule la profondeur d'un arbre (càd la taille de la plus grande branche).

Correction : Il faut faire deux fonctions intermédiaires : `profondeurNonVide` pour calculer récursivement la profondeur d'un arbre non vide.

```
let rec profondeurAux a = match a with
  | Noeud(_ , []) -> 1
  | Noeud(_ , l) ->
    let lp = List.map profondeurAux l in    (*profondeurs des fils*)
    let maxp = List.fold_left max 0 lp in   (*maximum de la liste lp*)
      maxp + 1

let profondeur a = match a with
    Vide -> 0
  | Non_vide a -> profondeurNonVide a
```

Le but des questions suivantes est d'écrire une fonction `branche_commune : int arbre -> int arbre -> int list` qui cherche si deux arbres ont une branche commune. Par exemple, l'arbre du début de cette partie a une branche commune avec



Il s'agit de la branche `[12 ; 3 ; 7]`.

On suppose que les fils sont triés par ordre croissant de leur racine et qu'il n'y a pas de répétition.

(2) *Question 4.* Écrivez une fonction `commun : int list -> int list -> int` qui cherche un élément commun dans deux listes triées :

```
# commun [0 ; 2 ; 5 ; 7 ; 11 ; 12]  [6 ; 7 ; 12 ; 13] ;;
- : int = 7
# commun [0 ; 2 ; 5 ; 7 ; 11 ; 12]  [6 ; 8 ; 10 ; 13] ;;
Exception: Not_found
```

Remarques :
- on n'utilise pas le type des arbres dans cette question,
- vous pouvez remplacer l'exception par l'utilisation de "`int option`" comme type de retour de votre fonction.

Correction :

```
let rec commun l1 l2 = match l1,l2 with
    [] , _ | _ , [] -> raise Not_found
  | a::l1 , b::_  when a<b  -> commun l1 l2
  | a::_ , b::l2  when a>b  -> commun l1 l2
  | a::_ , b::_  (*when a=b*) -> a
```

(2) *Question 5.* Écrivez une fonction `branche_commune_aux` de type `int arbre_non_vide -> int arbre_non_vide -> bool` qui parcourt les listes en cherchant deux arbres de même racine et
- renvoie `false` s'il n'y en a pas,
- s'appelle récursivement pour essayer de trouver une branche complète commune s'il y en a.

Remarques :
- il ne faut pas utiliser la fonction de la question précédente, seulement s'en inspirer
- une branche commune doit aller jusqu'au bout de l'arbre : on ne peut pas la couper. (Par exemple, "`[12 ; 42]`" n'est pas une branche commune aux deux arbres.)

Correction :

```
(* fonction auxilliaire pour calculer une branche commune dans deux
 * listes d'arbres *)
let rec commun_aux l1 l2 = match l1,l2 with
  | [] , [] -> true
  | [] , _ | _ , [] -> false
  | Noeud(a,_)::l1 , Noeud(b,_)::_  when a<b  -> commun_aux l1 l2
  | Noeud(a,_)::_ , Noeud(b,_)::l2  when a>b  -> commun_aux l1 l2
  | Noeud(a,l1)::l1' , Noeud(b,l2)::l2'  (*when a=b*)  ->
      commun_aux l1 l2 || commun_aux l1' l2'
let branche_commune_aux a1 a2 = commun_aux [a1] [a2]
```

(3) *Question 6.* Modifiez la fonction précédente pour qu'elle puisse renvoyer une branche commune quand elle en trouve une. Quand elle n'en trouve pas, elle pourra soit provoquer une exception, soit renvoyer la liste vide.

Déduisez-en la fonction `branche_commune : int arbre -> int arbre -> int list`.

Correction :

```
let rec commun_aux l1 l2 acc = match l1,l2 with
  | [] , [] -> List.rev acc
  | [] , _ | _ , [] -> raise Not_found
  | Noeud(a,_)::l1 , Noeud(b,_)::_  when a<b  -> commun_aux l1 l2 acc
  | Noeud(a,_)::_ , Noeud(b,_)::l2  when a>b  -> commun_aux l1 l2 acc
  | Noeud(a,l1)::l1' , Noeud(b,l2)::l2'  (*when a=b*)  ->
      try
        commun_aux l1 l2 (a::acc)
      with
        Not_found -> commun_aux l1' l2' acc
let branche_commune a1 a2 = match a1,a2 with
    Vide , Vide -> []
  | Vide , _  |  _ , Vide -> raise Not_found
  | _ , _ -> commun_aux a1 a2 []
```