info401 : Programmation fonctionnelle
Contrôle des connaissances – 1
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

(3) *Question 1.* Pour chacune des expressions suivantes, dites :
- si l'expression est bien typée (auquel cas vous donnerez sa valeur et son type),
- si l'expression est mal typée (auquel cas vous expliquerez pourquoi).

```
let exp1 = 1/2+3
let exp2 = 1 + (fst [2;3])
let exp3 = (1,2)::[]
let exp4 = [ [1;2;3] ] @ [ [] ]
let exp5 = (fun x->x+1) []
let exp6 = (fun f a b -> f a a) (fun n m -> n+m) 10 100.0
```

Correction :
- `exp1` de type `int` et de valeur `3`,
- `exp2` non définie : on ne peut pas prendre le `fst` d'une liste,
- `exp3` de type `(int*int) list` et de valeur `[ (1,2) ]`,
- `exp4` de type `int list list` et de valeur `[ [1 ;2 ;3] ; [] ]`,
- `exp5` non définie : la fonction est de type `int -> int`,on ne peut pas l'appliquer à une liste,
- `exp6` de type `int` et de valeur `20`.

(3) *Question 2.* Dans le morceau de code suivant, donnez la liste des variables de l'environnement aux endroits `(*1*)`, `(*2*)`,`(*3*)`, `(*4*)`, `(*5*)`, `(*6*)` et `(*7*)`.

```
(* on suppose que l'environnement est vide *)
let z = 0;;
(*1*)
let rec somme l = (*2*) match l with
          []   ->  (*3*) 0
        | [a]  ->  (*4*) a
        | b::l ->  (*5*) let r = somme l in
                   (*6*) r+b  ;;
(*7*)
```

Correction :
- `(*1*)` l'environnement contient `z`
- `(*2*)` l'environnement contient `z`, `somme` et `l`,
- `(*3*)` l'environnement contient `z`, `somme` et `l`,
- `(*4*)` l'environnement contient `z`, `somme` et `l` et `a`,
- `(*5*)` l'environnement contient `z`, `somme`, `b` et `l`,
- `(*6*)` l'environnement contient `z`, `somme`, `b`, `l` et `r`,
- `(*7*)` l'environnement contient `z` et `somme`.

Partie 2 : récursion

(2) *Question 1.* Programmez une fonction `filtre` qui permet de ne conserver que les éléments d'une liste qui vérifient une condition passée en argument. Par exemple

```
let ok n = n > 10
let resultat = filtre ok [2; 4; 12; 14; 7; 10; 11; 11; 9]
```

donnera un `resultat` égal à `[12 ; 14 ; 11 ; 11]`.

Correction :

```
let rec filtre p l = match l with
    [] -> []
  | a::l -> if (p a)
            then a::filtre p l
            else filtre p l
```

(1) *Question 2.* Quelle est le type de votre fonction ?

Correction :

```
...
val filtre : ('a -> bool) -> 'a list -> 'a list = <fun>
```

(1) *Question 3.* Programmez la même fonction en utilisant la fonction `List.fold_right` de type `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`. (C'est presque la même que la fonction `applique` vue en TD.)

(Si vous l'avez déjà écrite avec `List.fold_right`, écrivez maintenant une version sans `List.fold_right`.)

Correction :

```
let filtre c l =
  List.fold_right (fun a r -> if (c a) then a::r else r) l []
```

(3) *Question 4.* Plutôt que de ne garder que les éléments qui vérifient la condition, on veut maintenant séparer les éléments : d'un coté ceux qui vérifient la condition, de l'autre coté, ceux qui ne la vérifient pas. Par exemple :

```
let ok n = n > 10
let resultat = separe ok [2; 4; 12; 14; 7; 10; 11; 11; 9]
```

donnera un `resultat` égal à `( [12 ;14 ;11 ;11] , [2 ;4 ;7 ;10 ;9] )`.

Programmez la fonction `separe` et donnez son type.

Correction :

```
let rec separe c l = match l with
    [] -> ( [],[] )
  | a::l ->
      let r = separe c l in
      let r1 = fst r in
      let r2 = snd r in
```

```
if (c a)
then (a::r1 , r2)
else (r1 , a::r2)
```

(3) *Question 5.* Le tri rapide (*quick-sort*) fonctionne de la manière suivante :
- les listes à zéro ou un élément sont déjà triées
- pour trier une liste qui commence par l'élément `a` :
  . on divise la queue de la liste en deux parties : celle des éléments plus petits que `a` et celle des éléments plus grands que `a`.
  . on trie (récursivement) chacune de ces parties,
  . on concatène les deux listes triées, en mettant `a` au milieu.

Programmez la fonction `qsort`.

Correction :
```
let rec qsort l = match l with
    [] -> []
  | a::l ->
    let p = separe (fun x -> x<a) l in  (* on sépare la liste *)
    let l1 = fst p in (* éléments plus petits que a *)
    let l2 = snd p in (* éléments plus grands que a *)
      (qsort l1) @ [a] @ (qsort l2)
```

Partie 3 : constructeurs

Rappel : le type `'a option` est défini comme suit :
```
type 'a option  =  None  |  Some of 'a
```

(2) *Question 1.* Écrivez une fonction de type `int option list -> int` qui fait la somme de tous les éléments d'une liste, en comptant `0` pour `None` et `n+1` pour `Some(n)`.

Par exemple, "`somme [None ; Some 1 ; Some 3 ; None]`" donnera la valeur `6`.

Correction :
```
let rec somme l = match l with
    [] -> 0
  | None::l -> somme l
  | Some n::l -> n+1+somme l
```
ou
```
let somme l =
  List.fold_right
    (fun a r -> match a with None -> r | Some n -> n+1+r)
    l
    0
```
ou encore mieux (on verra pourquoi par la suite)
```
let somme l =
  List.fold_left
    (fun r a -> match a with None -> r | Some n -> n+1+r)
    0
    l
```

(2) *Question 2.* On veut maintenant faire la somme des éléments d'une liste mais en comptant `n` pour les éléments de la forme `Some n` et $+\infty$ pour les éléments de la forme `None`. Autrement dit, `None` $+ x =$ `None` pour n'importe quel x.

Par exemple, "`somme [Some 0 ; Some 3 ; None]`" donnera la valeur `None` et "`somme [Some 7 ; Some 3 ; Some 2]`" donnera la valeur `Some 12`.

Programmez la fonction correspondante de type `int option list -> int option`.

Correction :

```
let rec somme l = match l with
    [] -> Some 0
  | None::l -> None
  | Some(n)::l ->
      begin
        match (somme l) with
            None -> None
          | Some r -> Some (r+n)
      end
```

ou encore mieux :

```
let somme l =
  let rec aux l acc = match l with
      [] -> Some acc
    | None::_ -> None
    | Some n::l -> aux l (acc+n)
  in
  aux l 0
```