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/
wiki : http://www.lama.univ-savoie.fr/wiki

Vous avez le droit d'utiliser les fonctions que vous connaissez dans la librairie `List`.
Par exemple :
- `List.map : ('a -> 'b) -> 'a list -> 'b list`
- `List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
- `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
- `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list`
- ...

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

Un point est réservé pour la présentation.

Partie 1 : questions de cours

(3) *Question 1.* Pour chacune des définitions *indépendantes* `x1`, ..., `x4`, donnez la valeur renvoyée par Caml.

```
let o=0
let i=1
let x1 = let i=2 in o+i

let x2 = let rec aux i = if i=0 then 0 else (aux (i-1)) + 2 in aux 3

let x3 = List.map (fun x -> 1 / x) [2 ; 1 ; 0 ; -1 ; -2]

let d l = match l with _::a::_ -> a
let x4 = try d [1]
         with _ -> 2
```

Correction :
- `val x1 : int = 2`,
- `val x2 : int = 6`,
- `Exception : Division_by_zero.`,
- `val x4 : int = 2`.

(3) *Question 2.* Est-ce que les expressions suivantes sont bien typées, et si oui, quels sont leurs types ?

```
(fun x -> x/2) 2
(fun x -> x/2) 1 2
[0.0] @ [1 ; 2 ; 3]
(1+2 , [(3.14 , fun x -> x+1)] )
```

Correction :
- `(fun x -> x/2) 2` : OK, type `int`,
- `(fun x -> x/2) 1 2` : mal typé, "`This function is applied to too many arguments`",
- `[0.0] @ [1 ; 2 ; 3]` : mal typé, on ne peut pas concaténer une liste de flottants et une liste d'entiers,
- `(1+2,[(3.14,fun x->x+1)])` : OK, type `int * (float * (int->int))list` .

(2) *Question 3.* Quel sont les types et les valeurs de `y1` et `y2` ?

```
let y1 = []@[]
let y2 = []::[]
```

Correction :
- `y1` de type `'a list`, égal à `[]`,
- `y2` de type `'a list list`, égal à `[[]]`.

Partie 2 : programmation

(3) *Question 1.* La fonction `List.map` permet d'appliquer une fonction à tous les éléments d'une liste.

▷ Écrivez une fonction `pam` qui prend en arguments :
- une valeur,
- une liste de fonctions

et qui renvoie la liste des valeurs obtenues en appliquant les fonctions de la liste à la valeur. Par exemple :

```
pam 3 [ (fun x -> x+1) ; (fun x -> x*x) ; (fun x -> 42-x) ]
```

donnera `[4 ; 9 ; 39]`.

▷ Quel est le type de cette fonction ?

▷ *(Bonus)* Écrivez la fonction `pam` uniquement à partir de la fonction `map`.

Correction :

```
let rec pam a l = match l with
    [] -> []
  | f::l -> (f a)::(pam a l)
```

ou bien, avec `fold_right` :

```
let pam a l = List.fold_right (fun f t -> (f a)::t) l []
```

Le type de `pam` est `'a -> ('a -> 'b) list -> 'b list`.

Une version qui n'utilise que `List.map` :

```
let pam a = List.map (fun f -> f a)
```

Remarque : on pourrait faire une version récursive terminale de la manière suivante :

```
let pam a l = List.rev (List.fold_left (fun t f -> (f a)::t) [] l)
```

(3) *Question 2.* Que fait la fonction suivante ?

```
let rec seq i n = if n<i then [] else (seq i (n-1))@[n]
```

▷ Que pensez-vous de la complexité de cette fonction ? (Précisez autant que possible ce que vous affirmez.)

▷ Réécrivez une version plus efficace de cette fonction, en expliquant ce que vous faites.

Correction : Cette fonction permet de générer la liste des entiers consécutifs de `i` jusqu'à `n`.

Cette fonction est particulièrement inefficace car elle concatène un élément en fin de liste. Sa complexité en temps est proportionnelle à $(\mathtt{n}-\mathtt{i})^2$. De plus, sa complexité en mémoire n'est pas bonne non plus car la fonction n'est pas récursive terminale.

Version simple, non récursive terminale :

```
let rec seq i n = if n<i then [] else i::(seq (i+1) n)
```

Version récursive terminale avec un accumulateur :

```
let seq i n =
  let rec aux i n acc =
    if n<i
    then acc
    else aux i (n-1) (n::acc)
  in
  aux i n []
```

Partie 3 : petit problème

(5) *Question 1.* La *suite de Robinson* est une variante de la suite de Conway : d'après Wikipedia

> *Chaque terme de la suite se construit ensuite en comptant le nombre d'apparitions des différents chiffres de 9 à 0 (dans cet ordre) dans le terme précédent. Si un chiffre n'apparaît pas, il n'est pas pris en compte.*

Par exemple, si on commence avec `[2]`, les premiers termes sont :

`[2] [1 ; 2] [1 ; 2 ; 1 ; 1] [1 ; 2 ; 3 ; 1] [1 ; 3 ; 1 ; 2 ; 2 ; 1] ...`

Programmez une fonction Caml `robinson : int list -> int -> int list` qui calcule le n-ème terme de la suite de Robinson à partir d'une liste initiale quelconque.

Remarque : ne vous restreignez pas aux chiffres, mais considérez des entiers. (Autrement dit, la suite peut contenir des nombres supérieurs à 10.)

Correction :

```
(* itération d'une fonction *)
let rec iter f n x = if n=0 then x else iter f (n-1) (f x)

(* la fonction pour passer d'un terme au suivant *)
let suivant u =
  (* fonction auxiliaire qui compte le nombre d'apparitions
     de d dans n *)
  let rec aux u n d =
    match u with
      [] -> n::d::[]
    | a::u when a=d -> aux u (n+1) d
    | a::u -> n::d::(aux u 1 a)
  in
  (* il faut trier la liste par ordre décroissant pour pouvoir
     compter en commençant par 9 *)
  let u = List.sort (fun x y -> y-x) u
  in
  match u with
      [] -> []
    | a::u -> aux u 1 a
```

```
(* fonction finale *)
let robinson u n = iter suivant n u
```

Cette version n'est pas récursive terminale. On pourrait remplacer `suivant` par :

```
let suivant u =
  (* fonction auxiliaire qui compte le nombre d'apparitions
     de d dans n et stocke le résultat dans un accumulateur *)
  let rec aux u n d acc =
    match u with
      [] -> n::d::acc
    | a::u when a=d -> aux u (n+1) d acc
    | a::u -> aux u 1 a (n::d::acc)
  in
  (* comme l'accumulateur renverse le résultat, il faut trier la
     liste par ordre _croissant_ *)
  let u = List.sort (fun x y -> x-y) u
  in
  match u with
      [] -> []
    | a::u -> aux u 1 a []
```

Remarque : cette suite est beaucoup plus simple que la suite de Conway, car elle devient périodique au bout d'un moment.