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

Vous avez le droit d'utiliser les fonctions que vous connaissez dans la librairie `List`.

N'oubliez pas de commentez votre code.

Un point est réservé pour la présentation.

Partie 1 : questions de cours (25 min)

(2) *Question 1.* Suite à ce morceau de code Caml, quelles sont les valeurs de `x1`, `x2`, `x3` et `x4` ?

```
let c = ref 0
let a = !c
let d = ref a
let x1 = !c + 1                    (* x1 *)
let x2 =                           (* x2 *)
  c := 42 ;
  a-1
let x3 = (fun x -> c:=x ; x) 12    (* x3 *)
let x4 = !d                        (* x4 *)
```

Correction : On obtient, sans grande surprise :

```
val x1 : int = 1
val x2 : int = -1      (* car a=0 ; on a modifié le contenu de c *)
val x3 : int = 12      (* mais on a modifié le contenu de c *)
val x4 : int = 0       (* le contenu de d n'a pas bougé *)
```

(2) *Question 2.* Quels sont les types des expressions suivantes :

```
- fun a b -> b
- fun l a -> (a+1)::l
- fun n -> if n=0 then true else raise Failure "Erreur"
- (fun x y -> (3.14 , x+y)) 2
```

Correction : les types respectifs sont :

- `'a -> 'b -> 'a`
- `int list -> int -> int list`
- `int -> bool`
- `int -> (float * int)`

Pour la dernière expression, c'est parce que la fonction toute seule est de type `int -> int -> float*int` et qu'on l'applique à un seul argument.

(2) *Question 3.* Qu'est-ce qu'une *référence* en Caml. Expliquez et donnez des exemples.

Correction : une référence peut-être vue comme une *adresse* pour une valeur. (On parle aussi de pointeur.) L'intérêt est qu'il est possible de modifier le contenu d'une référence sans modifier l'adresse elle même. La syntaxe Caml est la suivante :

- `ref` *expr* est une référence vers la valeur donnée par l'expression *expr*. Par exemple, `let p = ref 0` déclare une variable `p` qui est une référence vers la valeur `0`,
- `p :=` *expr* pour modifier le contenu d'une référence. Par exemple, `p := 1`. L'expression "`p := 1`" est de type `unit`.
- `!p` pour obtenir le contenu de la référence `p`. (Si `p` n'est pas une référence, ça provoque une erreur de type.) Par exemple, "`!p`" a la valeur `1`...

(2) *Question 4.* Expliquez brièvement le concept de *mémoization*. Comment peut-on le mettre en œuvre en Caml.

Correction : la mémoization est l'idée de garder en mémoire les valeurs déjà calculées d'une fonction. Le calcul commence par vérifier si la valeur a déjà été calculée, et si c'est le cas, on n'a pas besoin de recommencer. Sinon, on fait le calcul normalement et on rajoute la valeur en mémoire pour les calculs suivants.

Pour une fonction de type `int -> int`, on peut faire ça en rajoutant un argument et une valeur de retour de type `int*int list` (et la fonction devient donc de type `int * (int*int list) -> int * (int*int list)`), ou plus simplement en utilisant une référence de type `int*int list` (ce qui permet de ne pas modifier le type de la fonction).

Le code Caml ressemble à

```
let table = ref []
let rec f x =
  (* on essaie de renvoyer la valeur déjà calculée *)
  try (List.assoc !table x)
  (* sinon, on fait le calcul normalement *)
  with Not_found -> let y = ... in (* le vrai calcul de f *)
                    (* ajout dans la table *)
                    table := (x,y)::!table ;
                    (* on renvoie la valeur *)
                    y
```

Partie 2 : programmation (30 min)

(3) *Question 1.* Le code de Gray permet de générer une énumération des suites de bit de longueur n où un seul bit est modifié à chaque étape. Par exemple, pour les suites de 3 bits :

`000 , 001 , 011 , 010 , 110 , 111 , 101 , 100 .`

Le code de Gray peut être défini de manière récursive : `gray (n+1)` est obtenu en dupliquant `gray n` et en suivant la recette

- on colle un 0 devant tous les éléments de `gray n`,
- on colle un 1 devant tous les éléments de `gray n`, et on renverse le résultat,
- on concatène les deux morceaux.

Écrivez la fonction `gray` correspondante. Quelle est le type de cette fonction ?

Correction :

```
let rec gray n =
  if n<1
  then [[]]
  else let g=gray (n-1) in
       (map (fun l -> 0::l) g) @ (map (fun l -> 1::l) (rev g))
```

Son type est `int -> int list list`.

Remarque : le sujet était un peu ambigu :
- on colle un 1 devant tous les éléments de `gray n`, et on renverse le résultat,

voulait dire qu'on renversait la liste obtenue. Par exemple, `gray 2` donne

`00 , 01 , 11 , 10`

- on colle un `0` devant tout le monde : `000 , 001 , 011 , 010`
- on colle un `1` devant tout le monde : `100 , 101 , 111 , 110`
- on renverse le resultat : `110 , 111 , 101 , 100`,
- on concatène les deux morceaux :

`000 , 001 , 011 , 010 , 110 , 111 , 101 , 100`

J'ai compté tous les points pour ceux qui ont renversé chaque sous-liste...

(3) *Question 2.* Nous avons déjà vu la fonction `map` qui applique une fonction à tous les éléments d'une liste ainsi que la fonction `pam` qui applique toutes les fonctions d'une liste à une valeur.

Écrivez une fonction `mapam : 'a list -> ('a->'b) list -> 'b list` qui applique toutes les fonctions d'une liste à toutes les valeurs d'une autre liste.

Les consignes sont les suivantes :
- l'ordre des valeurs dans le résultat n'est pas important ;
- c'est mieux (1 point) si votre fonction est récursive terminale* ;
- faites attention à la complexité.

Correction : version courte, mais illisible :

```
let mapam lv lf =
  fold_left (fun lr f -> rev_append (rev_map f lv) lr) [] lf
```

Pour ça, il faut savoir que `rev_append` est la version récursive terminale de `append` (`@`) et que `rev_map` est la version récursive terminale de `map`. (Ces deux fonctions donnent un résultat dont l'ordre ne correspond pas à l'ordre initial, mais c'est pas grave.)

On peut les programmer à l'aide de `fold_left` :

```
let rev_map f l = fold_left (fun r a -> (f a)::r) [] l
let rev_append l1 l2 = fold_left (fun r a -> a::r) l2 l1
```

ou bien directement :

```
let rev_map f l =
let rec aux acc = function
  | [] -> acc
  | a::l -> aux (f a :: acc) l
in
aux [] l

let rec rev_append l1 l2 =
match l1 with
  [] -> l2
| a :: l -> rev_append l (a :: l2)
```

Il y a bien sûr de nombreuses autres possibilités...

Partie 3 : petit problème (35 min)

Une méthode simple pour générer la suite des nombres premiers est le *crible d'Ératosthène*.

On commence avec la liste des nombres entiers à partir de `2` jusqu'à `n` :

`2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., n`

et on applique la recette suivante :
- on prend le premier éléments de la liste `p`, il est forcement premier,
- on supprime tous les multiples de `p` dans la liste,
- on recommence...

---

* Attention, je vous rappelle que `List.map` n'est pas récursive terminale.

Quand on tombe sur la liste vide, les nombres qu'on a pris sont exactement les nombres premiers plus petits que `n`.

Pour optimiser un peu, on peut utiliser les deux astuces suivantes :
- on commence directement avec la liste qui ne contient pas les nombres pairs.
- on peut s'arrêter dés que le premier éléments de la liste est plus grand que $\sqrt{n}$, car la fin de la liste ne contient plus que des nombres premiers.

(1) *Question 1.* Écrivez la fonction `init : int -> int list` qui génère la liste des entiers impairs entre `3` et `n`.

Votre fonction devra être récursive terminale...

Correction : Comme on utilise un accumulateur, il faut partir de `n`, mais il faut bien vérifier que `n` est impair :

```
let init n =
  let rec aux i acc =
    if i<3
    then acc
    else aux (i-2) (i::acc)
  in
  let m = if (n mod 2 = 0) then n-1 else n
  in aux m []
```

(2) *Question 2.* Écrivez la fonction `crible : int -> int list` qui renvoie la liste des nombres premiers plus petits que `n`, dans l'ordre croissant.

Par exemple, `crible 50` devra donner :

```
[2 ; 3 ; 5 ; 7 ; 11 ; 13 ; 17 ; 19 ; 23 ; 29 ; 31 ; 37 ; 41 ; 43 ; 47]
```

Essayer de faire une fonction récursive terminale qui évite les problèmes évidents de complexité.

Correction :

```
let crible n =
  let l = init n in
  (* racine carrée de n *)
  let s = int_of_float (sqrt (float_of_int n)) in
  (* le crible pour chercher les nombres premiers
       - l est la liste que l'on examine
       - p est la liste des nombres premiers qu'on a déjà trouvés *)
  let rec pr l p =
    match l with
      (* c'est fini, on renverse p. Ça ne devrait pas arriver souvent
         car on s'arrête avant *)
        [] -> List.rev p
      (* on a atteint la racine carrée de n : on rajoute le contenu
         de l dans p en faisant attention à l'ordre... *)
      | a::_ when a>s -> List.rev (List.rev_append l p)
      (* on filtre la liste en ne gardant que les nombres qui ne sont
         pas multiple de a, et on rajoute a dans p *)
      | a::l -> pr (List.filter (fun x-> x mod a <> 0) l) (a::p)
  in
    pr l [2] (* il ne faut pas oublier 2 dans la liste des nombres
                premiers...  *)
```

(2) *Question 3.* On peut généraliser la première astuce en générant directement la liste des entiers impairs qui ne sont pas multiples de 3 :

`5, 7, 11, 13, 17, 19, 23, 25,...`

Cette suite est obtenue en commençant à 5, et en ajoutant alternativement 2 et 4.

Pour éliminer directement les multiples de 2, 3 et 5, il faut commencer à 7, et ajouter successivement 4, 2, 4, 2, 4, 6, 2 et 6.

Écrivez une nouvelle fonction `init : int -> int` qui prend en argument un entier `max` ; et qui génère la suite obtenue en partant de 7 et en ajoutant successivement 4, 2, 4, 2, 4, 6, 2 et 6 jusqu'à dépasser `max`.

Par exemple, `init 50` devra donner

`[7 ; 11 ; 13 ; 17 ; 19 ; 23 ; 29 ; 31 ; 37 ; 41 ; 43 ; 47 ; 49]` .

Expliquez ce que vous devez modifier dans la fonction `crible` pour prendre cette nouvelle fonction en compte.

Correction :

```
let init n =
(* une fonction plus générale qui marche pour plusieurs listes :
     - l est la liste qui sert à générer le résultat
     - l' est une liste auxiliaire qui sert à parcourir la liste l
     - i est un compteur
     - n est l'élément maximal *)
let rec aux l l' i n =
  if i>n
  then []                        (* on a fini *)
  else match l' with
      []  ->  aux l l i n        (* on réinitialise l' à l *)
    | a::l' -> i::(aux l l' (i+a) n)
in aux [4; 2;  4;  2; 4; 6; 2; 6] [] 7 n
```

Cette version n'est pas récursive terminale. Je vous laisse la transformer en version récursive terminale en rajoutant un accumulateur. (Et en utilisant la fonction `List.rev` dans le cas `i>n`.)

Pour générer les nombres premiers, on fait comme dans la version précédente, mais on utilise cette nouvelle fonction `init` et on appelle `pr l [2 ;5 ;7]`.