Université Paris Diderot
L3 Info

Programmation Fonctionnelle
Année 2016-2017

Examen – Session 2

lundi 19 juin 2017

Tout document papier est autorisé. Les ordinateurs, téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints. Le temps à disposition est de **3 heures**.

Les exercices doivent être rédigés **en fonctionnel pur** : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Une fonction écrite en style impératif ne rapportera aucun point.

Les questions sont indépendantes mais il est parfois nécessaire (ou recommandé) d'utiliser les fonctions définies précédemment (même si elles sont non traitées) ou prédéfinies dans la bibliothèque standard (notamment dans le module sur les listes).

Certaines questions, plus difficiles, sont marquées d'une étoile : ★.

On recommande de *bien lire* l'énoncé d'un exercice avant de commencer à le résoudre. Cet énoncé a **3 pages** plus un **appendice**.

Exercice 1 (Expressions). Qu'affiche le toplevel Caml quand on entre l'une après l'autre les expressions suivantes ? Donner pour chaque expression ou définition : son type ; sa valeur ; le cas échéant, son effet de bord ou l'exception levée. Si l'expression est mal typée ou incorrecte, donner le message d'erreur. Justifier si nécessaire.

1. `let x = "bonne "in let x = "chance"in x^x;;`
2. `List.map string_of_int [1;2;3];;`
3. `List.map print_int [1;2;3];;`
4. `List.fold_left (/)12 [1;2;3];;`
5. `List.fold_right (/)[1;2;3] 12;;`
6. `let list = List.map (+)[1;2;3];;`
7. `let comp = fun f g x -> f (g x);;`
8. `List.fold_right comp list (fun x-> 2)0;;`
9.
```
type flux = A of int*(unit -> flux);;
let f =
  let rec aux x = A (x,fun ()->aux (x+1))
  in aux 0;;
```

★ `let next (A (_, f))= f ()in next (next f);;`

Exercice 2. Écrire une version du programme suivant en style fonctionnel pur, c.à-d. qui n'utilise ni boucles, ni références, ni tableaux, ni autres données mutables. Attention ! la fonction doit avoir exactement le même type et le même comportement.
Indication : Vous pouvez déclarer des fonctions auxiliaires si vous le souhaitez.

```
let f x =
  let a = ref 1 in
```

```
let b = ref (read_int x) in
while !b > 0 do
  a:= !b * !a;
  b:= !b - 1;
done;
print_int !a
```

Exercice 3. Voici deux extraits de la documentation officielle OCaml du module `List` :

— **val** `map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`

`List.map2 f [a1; ...; an] [b1; ...; bn]` is `[f a1 b1; ...; f an bn]`.
Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

— **val** `rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`

`List.rev_map2 f [a1; ...; an] [b1; ...; bn]` gives the same result as `List.map2 f [a1; ...; an] [b1; ...; bn]` is `[f a1 b1; ...; f an bn]`, but is tail-recursive and more efficient.

Questions :

1. Programmer `map2`
2. Programmer `rev_map2` de façon récursive terminale

Dans les deux questions ci-dessus, vous ne devez utiliser aucune des fonctions du module `List`.

Exercice 4. Dans cet exercice, nous considérerons des polynômes à une inconnue et à coefficients entiers. On se restreindra aux formes canoniques constituées d'une somme de monômes, comme par exemple :

$$3 - 3x + 25x^2 + 2x^4. \tag{1}$$

Le degré d'un monôme est l'exposant de son inconnue. Le degré d'un polynôme est le degré maximal de ses monômes. Par exemple, le polynôme ci-dessus est de degré 4.

Plus généralement, un polynôme de degré n est de la forme $\sum_{i=0}^{n} a_i x^i$, où a_i est le coefficient entier (éventuellement nul) du monôme de degré i, et $a_n \neq 0$. Nous représenterons en OCaml ce polynôme par la liste $[a_0; \ldots; a_n]$ des coefficients de ses monômes, du degré 0 jusqu'au degré du polynôme. On supposera que lorsque cette liste est non vide, son dernier élément est toujours non nul. La liste vide représente le polynôme nul 0. Par exemple, avec cette convention, le polynôme (1) ci-dessus sera représenté par la liste `[3; -3; 25; 0; 2]`.

L'utilisation judicieuse des fonctions du module `List` de OCaml sera appréciée dans les questions suivantes.

1. Ecrire une fonction `degree : int list -> int` prenant en argument une liste représentant un polynôme, et renvoyant le degré de ce polynôme (attention, la liste peut être vide).
2. Certaines manipulations peuvent faire apparaître des zéros à la fin d'une liste d'entiers. Écrire une fonction `normalize : int list -> int list` qui, étant donnée une liste d'entiers quelconque, renvoie la liste obtenue en supprimant tous les zéros situés à la fin de cette liste. Par exemple, `normalize [2;0;0;1;0;0;0]` s'évaluera en `[2;0;0;1]`.
3. Définir une fonction `coeff_mult : int -> int list -> int list` qui, étant donnés un entier `n` et une liste représentant un polynôme, renvoie la liste représentant le produit de `n` et de ce polynôme. Par exemple, `coeff_mult 3 [3; -3; 25; 0; 2]` renverra `[9; -9; 75; 0; 6]`. Attention : `coeff_mult 0 [3; -3; 25; 0; 2]` doit renvoyer la liste vide.

4. Définir une fonction `sum : int list -> int list -> int list` qui, étant données deux listes représentant deux polynômes, renvoie la liste représentant la somme de ces deux polynômes. Par exemple, `sum [3; -3; 25; 0; 2] [0; 1; 2]` s'évaluera en `[3;-2;27; 0; 2]`. (*Suggestion : utilisez* `normalize` *pour éviter d'avoir des zeros à la fin de la liste resultante*).

5. Définir une fonction `eval: int list -> int -> int` qui, étant donnés une liste représentant un polynôme $P(x)$ et un entier `n`, renvoie la valeur de $P($`n`$)$. Par exemple, l'expression `eval [3; -3; 25; 0; 2] 2` s'évaluera en 129, qui est égal à $3-3\times 2+25\times 2^2+2\times 2^4$. (Suggestion : définissez une fonction auxiliaire `pow : int -> int -> int` permettant de calculer une puissance entière d'un entier donné).

★6. Définir une fonction `mult : int list -> int list -> int list` qui, étant données deux listes représentant deux polynômes, renvoie la liste représentant le produit de ces deux polynômes. Rappelons que l'on a :

$$(\sum_{i=0}^{n} a_i x^i) \times (\sum_{j=0}^{m} b_j x^j) = \sum_{k=0}^{n+m} c_k x^k$$

avec pour chaque k :

$$c_k = \sum_{\substack{i,j \text{ tel que} \\ i+j=k}} a_i + b_j$$

Par exemple `mult [3; -3; 25; 0; 2] [0; 1; 2]` s'évaluera en `[0; 3; 3; 19; 50; 2; 4]`.
(*Suggestion : voici une description détaillée de cette multiplication, exploitant la structure des listes. Soient* $\texttt{l}_1 = [\texttt{a}_0; \ldots; \texttt{a}_n]$ *et* $\texttt{l}_2$ *les listes représentant les polynômes à multiplier.*

(a) *on calcule la liste de listes* $\texttt{l}' = [\texttt{m}_0; \ldots; \texttt{m}_n]$, *où chaque* $\texttt{m}_i$ *représente le produit de* $\texttt{a}_i$ *et du polynôme représenté par* $\texttt{l}_2$.

(b) *on calcule la liste obtenue en ajoutant* `i` *zéros en tête de* $\texttt{m}_i$ *dans* `l'`, *pour chaque* `i` *– ceci revient à incrémenter de* `i` *le degré de chaque monôme du polynôme représenté par* $\texttt{m}_i$ *; on obtient une liste* $\texttt{l}'' = [\texttt{m}_0; 0 :: \texttt{m}_1; 0 :: 0 :: \texttt{m}_2 \ldots; 0 :: \ldots 0 :: \texttt{m}_n]$.

(c) *on calcule la liste représentant la somme de tous les polynômes représentés par les éléments de* `l"` *; le résultat est bien* `mult` $\texttt{l}_1$ $\texttt{l}_2$*).*

A Quelque fonction utile de la librairie d'OCaml

— `List.fold_left : ('a -> 'b -> 'a)-> 'a -> 'b list -> 'a`
`List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1)b2)...)bn`.

— `List.fold_right : ('a -> 'b -> 'b)-> 'a list -> 'b -> 'b`
`List.fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b)...))`. Not tail-recursive.

— `List.length : 'a list -> int`
Return the length (number of elements) of the given list.

— `List.map : ('a -> 'b)-> 'a list -> 'b list`
`List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1 ; ...; f an]` with the results returned by `f`. Not tail-recursive.

— `List.mapi : (int -> 'a -> 'b)-> 'a list -> 'b list`
Same as List.map, but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument.

— `List.rev : 'a list -> 'a list`
List reversal.