Université Paris Diderot
L3 Info

Programmation Fonctionnelle
Année 2018-2019

Examen – Session 1

lundi 17 décembre 2018

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 dans les questions précédentes (même si elles sont non traitées) ou prédéfinies dans la bibliothèque standard (notamment dans le module sur les listes).

Il est recommandé de *bien lire* l'énoncé d'un exercice avant de commencer à le résoudre. Cet énoncé a **4 pages**.

Exercice 1 (Expressions)**.** Qu'affiche l'interpréteur OCaml lorsque l'on évalue l'une après l'autre les lignes ci-dessous ? Pour chaque expression ou définition, indiquer : son type ; sa valeur (ou **<fun>** s'il s'agit d'une valeur de fonction) ; le cas échéant, son effet de bord ou l'exception levée. Si l'expression est mal typée ou incorrecte, indiquer le message d'erreur. Justifier si nécessaire.

1. ```
   let x = 2010 + let x = 4 in 4 + x;;
   ```
2. ```
   x;;
   ```
3. ```
   exception Invalid_argument of int;;
   ```
4. ```
   let f x =
       if x < 0 then raise (Invalid_argument x)
       else x ;;
   ```
5. ```
   let f x =
       if x < 0 then print_endline "Invalid argument"
       else x ;;
   ```
6. ```
   let f x =
       if x < 0 then print_endline "Invalid argument"
       else raise (Invalid_argument x) ;;
   ```
7. ```
   f (-1); f 1 ;;
   ```
8. ```
   try f (-1); f 1 with Invalid_argument x -> print_int x ;;
   ```
9. ```
   try f 1 ; f (-1)with Invalid_argument x -> print_int x ;;
   ```
10. ```
    let rec loop () =
      try f (-1) ; f 1  with Invalid_argument x -> loop ()
        in loop () ;;
    ```

Exercice 2. Réécrire le programme ci-dessous en style fonctionnel pur, *ie.* en n'utilisant ni boucles, ni références, ni tableaux, ni d'autres données mutables. Attention ! Les deux fonctions doivent avoir exactement le même type et donner les mêmes sorties sur les mêmes entrées.
Indication : Vous pouvez déclarer des fonctions auxiliaires si vous le souhaitez.

```
let erat n =
  let a = Array.init (n-1) (fun i -> i+2)
  in
  let r = ref []
  in
  for i = 0 to n-2 do
    if a.(i) > 0 then
      let x = a.(i)
      in
      begin
        r := x::!r ;
        for j = i+1 to n-2 do
          if a.(j) mod x = 0 then a.(j) <- 0
        done
      end
  done ;
    !r
```

Exercice 3. Etant donnée une liste `l = [a1; ... ; an]`, les *préfixes* de `l` sont les listes de la forme `[a1; ... ; ai]` pour $i \in [0, \ldots, n]$. Programmer les fonctions suivantes. L'utilisation de la récursion terminale sera appréciée :

Question 1. `find_prefix : ('a list -> bool) -> 'a list -> 'a list * 'a list`. Etant donnée une propriété `p` sur les listes et une liste `l`, `find_prefix p l` doit renvoyer le couple `(l1,l2)` où `l1` est le plus petit préfixe de `l` satisfaisant `p` et `l1@l2 = l`. Si un tel préfixe n'existe pas, la fonction lancera l'exception `Not_found`.

Question 2. `forall_prefix : ('a list -> bool) -> 'a list -> bool` qui étant donnée une propriété `p` sur les listes et une liste `l` vérifie que tous les préfixes de `l` satisfont `p`.

Question 3. `cartesian_map : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list` telle que `cartesian_map f [a1; ...; an] [b1; ..; bk]` s'évalue en :

```
[f a1 b1; ...; f an b1; ....; f a1 bk;... ; f an bk],
```

où toute autre liste contenant les mêmes éléments.

Exercice 4. Appelons *mot sur l'alphabet* $\{-1, 1\}$ toute suite $u = \langle u_1, \ldots, u_n \rangle$ telle que $u_i \in \{-1, 1\}$ pour chaque $i \in [1, \ldots, n]$. La longueur n de ce mot se note $|u|$. On dira que u est un *mot de pile* s'il vérifie les deux propriétés suivantes :

1. la somme des éléments de u vaut -1 :
   $\sum_{i=1}^{n} u_i = -1$.
2. la somme des éléments de tout préfixe strict de u est positive ou nulle :
   $\sum_{i=1}^{k} u_i \geq 0$ pour tout $k \in \{1, \ldots, n-1\}$,

On note par un point $\_ \cdot \_$ la concaténation des mots. Par exemple, si $a = \langle a_1, \ldots, a_n \rangle$ et $b = \langle b_1, \ldots, b_m \rangle$, on a $a \cdot b = \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$.

Les mots de pile seront représentés en OCaml par des listes d'entiers, i.e à l'aide du type `int list`. Les fonctions de l'exercice 3 sont librement utilisables dans les questions, même si vous n'avez pas réussi à les écrire.

Question 1. Donner toutes les valeurs en OCaml correspondant aux mots de pile de longueur 1 et 3 et 5. Notez qu'un mot de pile est nécessairement de longueur impaire.

Question 2. Écrire une fonction `est_pile : int list -> bool` vérifiant si un mot est un mot de pile.

Il est facile de voir que si v et w sont des mots de pile, alors $\langle 1 \rangle \cdot v \cdot w$ est aussi un mot de pile. Inversement, tout mot de pile de longueur supérieure ou égale à 3 se décompose de manière unique sous la forme $\langle 1 \rangle \cdot v \cdot w$, où v et w sont des mots de pile et v est de longueur minimale.

Question 3. Pour chacun des mots suivants, trouver les mots v, w tels que $\langle 1 \rangle \cdot v \cdot w$ soit égal à ce mot :

— $\langle 1, 1, 1, -1, -1, -1, -1 \rangle$
— $\langle 1, -1, 1, -1, 1, -1, -1 \rangle$

Question 4. Écrire une fonction `decompose : int list -> (int list * int list)` qui, étant donné un mot de pile `u` de longueur supérieure ou égale à 3, renvoie le couple `(v, w)` correspondant à sa décomposition. La fonction devra lancer une exception si le mot fourni en entrée n'est pas un mot de pile.

Question 5 On souhaite à présent calculer *tous* les mots de pile d'une longueur donnée.

a. Écrire une fonction
   `ajoute : int list list -> int list list -> int list list -> int list list`
   telle que `ajoute l1 l2 l` ajoute en tête de la liste `l` tous les éléments de la forme `((1::t1)@t2)`, où `t1` est un élément de `l1`, et `t1` un élément de `l2`.
b. En déduire une fonction `pile : int -> int list` qui, pour tout entier `n`, renvoie la liste des mots de pile de longueur inférieure ou égale à $2 \times \mathtt{n} + 1$.

A Quelques fonctions utiles de la librairie d'OCaml

— `(mod): int -> int -> int`
Integer remainder. If `y` is not zero, the result of `x mod y` satisfies the following properties : `x = (x / y)* y + x mod y` and `abs(x mod y)<= abs(y)- 1`. If `y = 0`, `x mod y` raises `Division_by_zero`. Note that `x mod y` is negative only if `x < 0`.
Raise `Division_by_zero` if `y` is zero.

— `List.hd : 'a list -> 'a`
Return the first element of the given list.
Raise `Failure "hd"` if the list is empty.

— `List.tl : 'a list -> 'a list`
Return the given list without its first element.
Raise `Failure "tl"` if the list is empty.

— `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)...)))`.

— `List.exists : ('a -> bool)-> 'a list -> bool`
`List.exists p [a1; ...; an]` checks if at least one element of the list satisfies the predicate `p`. That is, it returns `(p a1)|| (p a2)|| ... || (p an)`.

— `List.map : ('a -> 'b)-> 'a list -> 'b list`
`List.map f [a1; ...; an]` is `[f a1; ...; f an]` with the results returned by `f`.

— `(List.filter : ('a -> bool)-> 'a list -> 'a list`
`filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.

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

— `List.init : int -> (int -> 'a)-> 'a list`
`List.init len f` is `[f 0; f 1; ...; f (len-1)]`, evaluated left to right.
Raises `Invalid_argument` if `len < 0`.