Université Paris Diderot
L3 Info

Programmation Fonctionnelle
Année 2017-2018

Examen – Session 2

lundi 18 juin 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 f = let a = 3 in fun x -> x+a;;`
2. `let a = 1 in f a;;`
3. `List.fold_right (^)["session";"2"] "!";;`
4. `let f = List.fold_right (^);;`
5. `let g = List.map string_of_int [1;2;3] in f g ".";;`
6. `"0":: g;;`
7. ```
   let rec church_of_int n =
     if n = 0 then fun f x -> x
     else fun f x -> church_of_int (n-1) f (f x);;
   ```
8. `let three f x = church_of_int 3 f x;;`
9. `three (fun x -> x+1)0;;`
10. `three three (fun x-> x+1)0;;`

Scanned with CamScanner
   ```

Exercice 2. Pour chacune des fonctions récursives ci-dessous : indiquer si sa récurrence est terminale ; dans le cas contraire, donner une définition équivalente avec une récurrence terminale.

```
(*1*)
let rec f x =
  if x = 0 then ""
  else (^) (f (x-1)) (string_of_int x)
;;

(*2*)
let g x =
  let rec aux x c =
    if c = x then []
    else c::(aux x (c+1))
  in aux x 0
;;

(*3*)
let rec h x =
  if x = 0 then ()
  else (print_int x; h (x-1))
;;

(*4*)
let k () =
  let rec aux c =
    try aux (read_int ()::c) with
      Failure _ -> c
  in aux []
```

Exercice 3. Voici un extrait de la documentation officielle du module `List` de OCaml :

`val 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.

`val combine : 'a list -> 'b list -> ('a * 'b) list`

Transform a pair of lists into a list of pairs : combine `[a1; ...; an] [b1; ...; bn]` is `[(a1,b1); ...; (an,bn)]`. Raise `Invalid_argument` if the two lists have different lengths. Not tail-recursive.

`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 split : ('a * 'b) list -> 'a list * 'b list`

Transform a list of pairs into a pair of lists : `split [(a1,b1); ...; (an,bn)]` is `([a1; ...; an], [b1; ...; bn])`. Not tail-recursive.

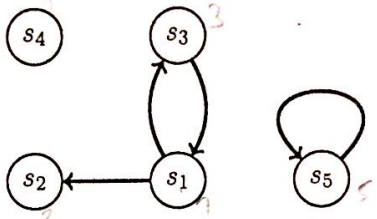
1. Écrire la fonction `map2` comme composition fonctionnelle (sans aucun pattern-matching) de fonctions ci-dessus autre que `map2` et, éventuellement, de fonctions du module `Pervasives`.

Scanned with CamScanner

2. Écrire la fonction `split` comme une composition fonctionnelle (sans aucun pattern-matching) utilisantune seule des fonctions ci-dessus autre que `split` et, éventuellement, de fonctions du module `Pervasives`.

Exercice 4. Rappelons qu'un *graphe orienté* est la donnée d'un ensemble de *sommets* et d'un ensemble d'*arêtes*, chaque arête a étant un couple de sommets du graphe de la forme (s, t) – les sommets s et t sont appelés *source* et *but* de a, on dit aussi que a va *de* s *vers* t, ou que a est une arête *sortante* pour s et *entrante* pour t.

Par exemple, la donnée des ensembles $\{s_1, s_2, s_3, s_4, s_5\}$ et $\{(s_1, s_2), (s_1, s_3), (s_3, s_1), (s_5, s_5)\}$ définit un graphe orienté, représentable de la manière suivante :



Un *chemin* de longueur $k > 0$ dans un graphe orienté est une suite de sommets $(p_0, \ldots, p_k)$ telle que pour chaque i, il existe une arête de p_i vers p_{i+1} – on dit alors que ce chemin va *de* p_0 *vers* p_k. Par exemple, dans le graphe ci-dessus, (s_5, s_5) est un chemin de longueur 1 de s_5 vers lui-même, et (s_1, s_3, s_1, s_2) est un chemin de longueur 3 de s_1 vers s_2.

L'*arité* (respectivement la *co-arité*) d'un sommet est le nombre de ses arêtes sortantes (respectivement entrantes). Par exemple, dans le graphe ci-dessus : s_1 est d'arité 2 et de co-arité 1 ; s_5 est d'arité et de co-arité égales à 1 ; s_4 est d'arité et de co-arité nulles.

1. Donner en Ocaml la définition d'un type d'enregistrement polymorphe `'a graph` contenant deux champs `sommets` et `aretes`. Ce type doit permettre la représentation de tout graphe orienté fini dont les sommets sont d'un type `'a` quelconque.
2. Donner la définition d'un type polymorphe `'a path` permettant de représenter des chemins dans un graphe représenté dans le type `'a graph`.
3. Avec les types que vous aurez choisis et en supposant que s_1, s_2, s_3, s_4, s_5 sont les entiers 1, 2, 3, 4, 5, donner les valeurs représentant le graphe et les deux chemins donnés en exemple ci-dessus.

2. Écrire les fonctions

```
arity   : 'a graph -> 'a -> int
coarity : 'a graph -> 'a -> int
```

telles que `arity g s` (resp. `coarity g s`) renvoie l'arité (resp. la co-arité) du sommet `s` dans le graphe `g`. Si `s` n'est pas un sommet de `g`, une exception devra être levée.

3. Écrire

```
is_path: 'a graph -> 'a path -> bool
```

telle que `is_path g p` renvoie `true` si et seulement si `p` est un chemin dans `g` ("chemin" toujours au sens de la définition ci-dessus, c'est-à-dire allant bien d'un sommet de `g` vers un sommet de `g`).

Scanned with CamScanner

4. Écrire

   `val visit_from: 'a graph -> 'a -> 'a list`

   tel que si `g` représente un graphe, et `s` un sommet, `visit_from g s` renvoie la liste sans répétitions des sommets atteints par tous les chemins partant de `s`.
   Par exemple : `visit_from g 1` et `visit_from g 3` doivent renvoyer une liste contenant les même éléments que `[1; 2; 3]` ; l'appel `visit_from g 5` doit renvoyer `[5]` ; l'appel `visit_from g 4` doit renvoyer `[]`. Attention aux bouclages.

5. A l'aide des fonctions précédentes, écrire la fonction :

   `val cyclic: 'a graph -> boolean`

   tel que `cyclic g` renvoie `true` si et seulement si le graphe `g` contient un *cycle*, c'est à dire un chemin allant d'un sommet vers lui-même.

A Quelques fonctions utiles de la librairie d'OCaml

— `string_of_int : int -> string`
Return the string representation of an integer, in decimal.

— `read_int : unit -> int`
Flush standard output, then read one line from standard input and convert it to an integer. Raise `Failure "int_of_string"` if the line read is not a valid representation of an integer.

— `print_int : int -> unit` Print an integer, in decimal, on standard output.

— `fst : 'a * 'b -> 'a` Return the first component of a pair.

— `snd : 'a * 'b -> 'b` Return the second component of a pair.

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

— `List.mem : 'a -> 'a list -> bool`
`mem a l` is true if and only if `a` is equal to an element of `l`.

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

Scanned with CamScanner