Université Paris Diderot
L3 Info

Programmation Fonctionnelle
Année 2017-2018

Examen – Session 1

lundi 18 décembre 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 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** plus un **appendice**.

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 = "bon jour"in x^x;;`
2. `x^x;;`
3. `let x = 3 in (let x = x + 1 in x)* x;;`
4. `let head_hunter = fun x -> List.map List.hd x;;`
5. `List.fold_right (^)["Hello"; ""; "World"] "!";;`
6. `List.fold_left (^)"!"["Hello"; ""; "World"];;`
7. `let r = ref 0;;`
8.
```
let add_trace m =
  for i = 0 to Array.length m -1 do
    r := m.(i).(i) + !r;
  done;;
```
9. `let a = Array.init 3 (fun i -> Array.init 3 (fun j -> (i+1)*(j+1)));;`
10. `add_trace a;;`
11. `!r;;`
12. `let f()= !r in f (add_trace a);;`
13. `let x = !r in add_trace a; x;;`

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 ! chaque fonction doit avoir exactement le même type et le même comportement.
Indication : Vous pouvez déclarer des fonctions auxiliaires si vous le souhaitez.

```
let exists_perfect l =
  let flag = ref false
  and rl   = ref l in
  try
    while not !flag do
      let n = List.hd !rl
      and s = ref 1 in
      for i = 2 to n/2 do
        if n mod i = 0 then s := !s + i
      done;
      if !s = n
      then flag := true
      else rl := List.tl !rl
    done;
    true
  with Failure "hd" -> false
;;
```

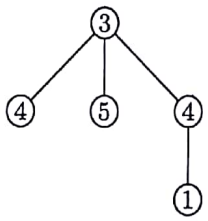
Exercice 3. Nous considérerons dans cet exercice des arbres n-aires non vides, étiquetés par des valeurs de type quelconque. Ces arbres seront représentés à l'aide du type OCaml suivant :

```
type 'a tree = Node of 'a * ('a tree list)
```

Noter qu'un tel type ne permet pas de représenter l'arbre vide : tout arbre a une racine étiquetée par une valeur de type `'a`, ainsi qu'une liste de fils éventuellement vide. Par exemple, la valeur

```
t1 = Node (3, [Node (4, []); Node (5, []); Node (4, [Node (1, [])])])
```

représente l'arbre suivant :



1. Écrire une fonction `size : 'a tree -> int` telle que `size t` renvoie la taille de `t` (*ie.* le nombres de ses nœuds).
2. Écrire une fonction `tree_map : ('a -> 'b)-> 'a tree -> 'b tree` telle que `tree_map f t` renvoie l'arbre obtenu en remplaçant chaque étiquette `a` de `t` par `(f a)`.
3. Écrire une fonction `tree_fold : ('a -> 'b -> 'b)-> 'a tree -> 'b -> 'b` telle que `tree_fold f t b` renvoie la valeur de `(f a1 (f a2 (...(f an b)...)))`, où `a1,...,an` est la suite des étiquettes de `t` rencontrées successivement lors d'un parcours en profondeur. Par exemple, sur l'arbre `t1` ci-dessus, `tree_fold f t1 b` renverra la valeur de `(f 3 (f 4 (f 5 (f 4 (f 1 b)))))`.

Exercice 4. Le but de cette exercice est d'écrire un programme permettant de reformatter un texte en évitant les espaces inutiles et en ne changeant de ligne qu'à chaque fois qu'on dépasse un nombre fixé de caractères. Par exemple, si ce nombre est fixé à 60, le texte suivant

```
OCaml        est l'implémentation la plus avancée du langage de programmation Caml.
Ce langage, de la famille des langages     ML, est un projet open source dirigé et maintenu essentiellement par l'Inria.
```

sera reformatté en :

```
OCaml est l'implémentation la plus avancée du langage de
programmation Caml. Ce langage, de la famille des langages
ML, est un projet open source dirigé et maintenu
essentiellement par l'Inria.
```

Les positions des caractères d'une chaîne sont numérotées de gauche à droite à partir de 0. Nous appellerons *espacement* tout caractère parmi `' '`, `'\n'`, `'\t'`, `'\r'`. Par convention, tout autre caractère sera appelé une *lettre*, quel qu'il soit sa valeur réelle (lettre ou symbole).

Un *mot* d'une chaîne `s` est une suite maximale de caractères de positions consécutives dans `s` et ne contenant aucun espacement. Une *ligne* de `s` est une suite maximale de caractères de positions consécutives dans `s` et ne contenant aucun `'\n'`.

Dans les questions qui suivent, chaque fonction peut, ou même doit se servir des précédentes.

1. Écrire une fonction `is_space : char -> bool` telle que `is_space c` renvoie `true` si et seulement si `c` est un espacement.
2. Écrire une fonction `is_letter : char -> bool` telle que `is_letter c` renvoie `true` si et seulement si `c` *n'est pas* un espacement.
3. Écrire une fonction `first_index : (char -> bool) -> string -> (int * int)` telle que `first_index p s` renvoie un couple `(i, k)` vérifiant la propriété suivante :
   — s'il existe un caractère `c` dans `s` tel `(p c)` soit égal à `true`, alors :
   `i` vaut la première position de ce caractère dans `s`,
   `k` vaut le nombre de positions dans `s` à partir de `i` inclus.
   — sinon `i` est égal au nombre de caractères de `s` et `k` vaut `0`.

   Par exemple :
   — `first_index is_space "Hello Michele !"` vaudra `(5, 11)`,
   — `first_index is_space "HelloMichele!"` vaudra `(13, 0)`,
   — `first_index is_letter "  Hello Michele !"` vaudra `(2, 15)`,
   — `first_index is_letter "  "` vaudra `(2, 0)`.

   Vous pouvez (et même raisonnablement devez) vous servir de `String.length`, *cf.* l'appendice.
4. Écrire une fonction `trim : string -> string`, telle que `trim s` renvoie la chaîne `s` privée de tous ses espacements initiaux. Par exemple, `trim "  Hello Michele !  "` renverra `"Hello Michele !  "`, ou encore, `trim "   "` renverra `""`.
5. Écrire une fonction `cut_first : string -> string * string` telle que, si `s` est une chaîne contenant au moins un mot, `cut_first s` renvoie un couple avec, à gauche, le premier mot non vide de `s` et à droite, la chaîne formée de `s` privée de ce premier mot. Si la chaîne `s` ne contient que des espacements, cette évaluation déclenchera l'exception `Not_found`. Par exemple :
   — `cut_first "  Hello Michele !"` renverra `("Hello", " Michele !")`.
   — `cut_first "  Hello  "` renverra `("Hello", "  ")`.
   — `cut_first "   "` déclenchera une exception.

6. Ecrire une fonction `explode : string -> string list` telle que `explode s` renvoie la liste de tous les mots non vides de `s`, dans l'ordre où ils apparaissent dans `s`. Par exemple, `explode "␣␣Hello␣Michele␣!␣␣␣"` renverra `["Hello"; "Michele"; "!"]`, tandis que `explode "␣"` renverra la liste vide.
7. Écrire enfin une fonction `format : int -> string -> string` telle que `format k s` renvoie la chaîne formée de la suite des mots non vides de `s` , deux mots étant séparés soit par un unique `' '` soit par un unique `'\n'` , chaque ligne étant de longueur maximale, mais sans jamais dépasser `k` caractères.

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`.

— `String.length : string -> int`
Return the length (number of characters) of the given string.

— `String.get : string -> int -> char`
`String.get s n` returns the character at index `n` in string `s`.
Raise `Invalid_argument` if `n` not a valid index in `s`.

— `String.sub : string -> int -> int -> string`
`String.sub s start len` returns a fresh string of length `len`, containing the substring of `s` that starts at position `start` and has length `len`.
Raise `Invalid_argument` if `start` and `len` do not designate a valid substring of `s`.