Université Paris Diderot
L2 Math/Info, L3 Info, M1 Linguistique

Programmation Fonctionnelle
Année 2010-2011

Examen

Mardi 4 janvier 2011

Motivez bien vos réponses. On recommande de *bien lire* l'énoncé d'un exercice avant de commencer à le résoudre.

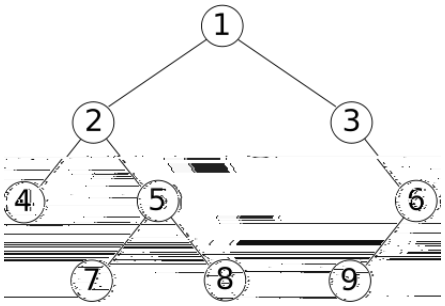
Tout document papier est autorisé. Les ordinateurs, les téléphones portables, comme tout autre moyen de communication vers l'extérieur, doivent être éteints. Le temps à disposition est de 3 heures.

Exercice 1 Pour cet exercice, si OCaml renvoit une erreur, décrivez succintement l'erreur sans donner le message exact. Par exemple "x est de type string mais int était attendu".

1. Donnez la valeur et le type des instructions suivantes :
   (a) `# 3 + 5`
   (b) `# 3 ^ "5"`
   (c) `# "3" ^ "5"`
   (d) `# 3. +. 0`
   (e) `# 3::[4;5;6]`
   (f) `# 1::[2,3]`
   (g) `# ["abc"] @ ["def"]`
   (h) `# 1 @ []`
2. Pour les définitions de fonction (de (a) à (c)), donnez leur type. Pour les appels de fonction (de (d) à (h)), donnez le type et la valeur du résultat. On suppose que l'exception `Empty_list` est précédemment définie.
   (a)
   ```
   # let t l
     match l with
       [] -> raise Empty_list
     | hd :: tl -> hd
   ```
   (b)
   ```
   # let rec c l1 l2 =
       match (l1, l2) with
         ([],_) | (_,[]) -> []
       | (a :: q, b :: p) -> (a, b) :: c q p
   ```
   (c) `# let f l = List.map (fun x -> List.fold_left (+) 0 x) l`
   (d) `# t [1;2]`
   (e) `# t []`
   (f) `# c [1;2] ["un";"deux"]`
   (g) `# c [] [3.1415]`
   (h) `# f [[1;2;3];[4;5;6]]`

Exercice 2 On définit un arbre n-aire comme étant soit une feuille ne comportant aucune information, soit un noeud avec une liste de fils de taille quelconque. Chaque noeud doit pouvoir contenir un élément générique (du même type pour tout l'arbre).

1. Définir en OCaml le type d'un arbre n-aire.
2. Donnez l'expression OCaml correspondant à un arbre n-aire ayant pour sommet 3, et comme fils 2 et 4 (de gauche à droite).
3. Donnez le type et le corps de la fonction qui prend en argument un arbre n-aire et renvoit la liste de ses éléments donnée par un parcours préfixe. On rappelle qu'un parcours préfixe d'un arbre visite d'abord le noeud courant puis récursivement chacun de ses fils, en allant de gauche à droite. Par exemple, pour l'arbre ci-dessous on a : $[1, 2, 4, 5, 7, 8, 3, 6, 9]$



4. Donnez la fonction qui effectue un parcours en largeur de l'arbre. Cette fonction doit retourner une liste avec en premier le sommet, puis tous les noeuds de profondeur 1, et ainsi de suite. Par exemple sur l'arbre au dessus on a : $[1, 2, 3, 4, 5, 6, 7, 8, 9]$
   Indication : Vous pourrez utiliser directement la structure de file (FIFO) vue en cours.

Exercice 3 Soit la définition de fonction suivante :

```
let rec mrg l1 l2 =
  match l1 with
  | [] -> l2
  | h1::r1 -> begin
    match l2 with
      | [] -> l1
      | h2::r2 -> if h1 <= h2 then h1::(mrg r1 l2) else h2::(mrg l1 r2)
  end
```

1. Qu'est-ce que cette fonction calcule si on suppose que les deux listes `l1` et `l2` sont triées dans un ordre croissant ? (une réponse en deux ou trois lignes est suffissante).
2. Expliquez pourquoi cette fonction n'est pas recursive terminale (une réponse en deux ou trois lignes est suffissante).
3. Réécrire cette fonction en une fonction avec le même comportement mais qui est récursive terminale. Si vous utilisez des fonctions auxiliaires il faut qu'elles soient également récursives terminale.

   Indication : Utiliser la fonction `reverse`, dans sa version récursive terminale, vue en cours.
4. Écrire une fonction qui, étant donnée une liste d'entiers $l = [x_1, \ldots, x_n]$, envoie une paire de deux listes (l_1, l_2) telle que
   - l_1 contient exactement les éléments $x_1, x_3, x_5, \ldots$, peu importe l'ordre
   - l_2 contient exactement les éléments $x_2, x_4, x_6, \ldots$, peu importe l'ordre

– la fonction est récursive terminale

5. En utilisant les fonctions obtenues aux questions (**??**) et (**??**), écrire une fonction qui trie une liste d'entiers dans l'ordre croissant.

Exercice 4 Le but de cet exercice est d'écrire une fonction `permutations` qui étant donnée une liste $l = [x_1; \cdots; x_n]$, retourne la liste de ses permutations (peu importe l'ordre). Par exemple, `permutations [1;2;3]` doit renvoyer

$$[[1;2;3];[1;3;2];[2;1;3];[2;3;1];[3;1;2];[3;2;1]]$$

1. Coder la fonction `insert_all` telle que (`insert_all` x $[y_1; ...; y_n]$) retourne

$$[[x; y_1; ...; y_n]; [y_1; x; y_2; ...; y_n]; ...; [y_1; ...; y_n; x]]$$

2. Coder la fonction `concat_all` telle que (`concat_all` $[l_1; ..; l_n]$) retourne l_1`@`...`@`l_n.
3. En déduire un codage de la fonction `permutations`.

Indication : la fonction `List.map` peut servir pour les questions (1) et (3), la fonction `List.fold_left` peut servir pour la question (2).

Exercice 5 Une *formule* est définie comme étant soit une variable (dont le nom est une chaîne de caractères, soit la négation d'une formule, soit la conjonction de deux formules, soit la négation de deux formules. Cela donne lieu à la définition OCaml suivante :

```
type formula =
| Var of string
| Neg of formula
| And of formula * formula
| Or  of formula * formula;;
```

La *forme normale* d'une formule est obtenue par itérant le processus suivant tant que possible :

– remplacer une formule de la forme `Neg (Neg x)` par `x`
– remplacer une formule de la forme `Neg (And (x,y))` par `Or (Neg x, Neg y)`
– remplacer une formule de la forme `Neg (Or (x,y))` par `And (Neg x, Neg y)`

Par exemple, la forme normale de `Neg (And (Neg (Var "x"), Or (Var "y", Neg (Var "z"))))` est obtenue en passant par les étapes suivantes :

```
Neg (And (Neg (Var "x"), Or (Var "y", Neg (Var "z"))))
Or (Neg (Neg (Var "x")), Neg (Or (Var "y", Neg (Var "z"))))
Or (Var "x", Neg (Or (Var "y", Neg (Var "z"))))
Or (Var "x", And (Neg (Var "y"), Neg (Neg (Var "z"))))
Or (Var "x", And (Neg (Var "y"), Var "z"))
```

La forme normale est alors `Or (Var "x", And (Neg (Var "y"), Var "z"))`.

Écrire une fonction OCaml qui prend une formule en argument et qui envoie comme résultat sa forme normale.