Universite Paris Diderot — Programmation Fonctionnelle
L2 Math/Info, L3 Info, M1 Lingustique — Annee 2011-2012

Examen

Jeudi 5 janvier 2012

Motivez bien vos reponses. On recommande de *bien lire* l'enonce d'un exercice avant de commencer a le resoudre.

Tout document papier est autorise. Les ordinateurs, les telephones portables, comme tout autre moyen de communication vers l'exterieur, doivent être eteints. Le temps a disposition est de 3 heures.

Les exercices doivent être rediges en fonctionnel pur : ni references, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements a champs mutables. Chaque fonction ci-dessous peut utiliser les fonctions prede nies (sauf indication contraire), et/ou les fonctions des questions precedentes.

Exercice 1 Donnez le type, et s'il y a lieu, la valeur des expressions suivantes. Si la valeur est une fonction, donnez seulement son type. Si OCaml renvoie une erreur, decrivez succintement l'erreur sans donner le message exact. Par exemple "`x` est de type `string`, mais `int` etait attendu".

1. `let pproj (a,b) = a;;`
2. `List.map (fun x -> -1) ["a";"b";"c"];;`
3. `(+) 2;;`
4. ```
   let double f =
     let two = 2
     in fun x -> two*(f x) ;;
   ```
5. ```
   let retourner l =
     let rec aux acc =
       function
         [] -> acc
       | a::r -> aux (a::acc) r
     in aux [] l ;;
   ```
6. `List.fold_left (fun x y -> y^x) "" ["abc";"def";"ghi"];;`
7. `List.fold_right (fun a b -> a - b) [1;2;3] (-1);;`
8. `if (1-1 = 0.0) then true else false;;`
9. ```
   type abc = A | B | C;;
   (fun x -> match x with _ -> C) A;;
   ```
10. `let f x y z = ((x = y) = z);;`
   ```

Exercice 2 Les quatre fonctions ci-dessous doivent être ecrites sans l'aide des fonctions prede nies.

1. Ecrire l'implementation d'une fonction `for_all` telle que :
   `for_all : ('a -> bool) -> 'a list -> bool`
   `for_all p [x1;...;xn] = (p x1) && ... && (p xn)`
   Autrement dit, la fonction `for_all` considere une fonction `p` et une liste `[x1;...;xn]`, et retourne `true` si et seulement si *chaque* `(p xi)` prend la valeur `true`.
2. Ecrire l'implementation d'une fonction `for_all2` telle que :
   `for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`
   `for_all2 p [x1;...;xn] [y1;...;yn] = (p x1 y1) && ... && (p xn yn)`
3. Ecrire l'implementation d'une fonction `exists` telle que :
   `val exists : ('a -> bool) -> 'a list -> bool`
   `exists p [x1;...;xn] =(p x1) || ... || (p xn)`
   Autrement dit, la fonction `exists` considere une fonction `p` et une liste `[x1;...;xn]`, et retourne `true` si et seulement si *l'un au moins* des `(p xi)` prend la valeur `true`.
4. Ecrire l'implementation d'une fonction `exists_croise` telle que :
   `exists_croise : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`
   `exists_croise p [x1;...;xn][y1;...;yn]` renvoie `true` si et seulement s'il existe $i, j \in \{1, \ldots, n\}$ tel que `(p xi yj)` prenne la valeur `true`.
   Indication : Utilisez la solution de la question precedente.

Exercice 3

1. Appelons *suffixe* de `[x1;...;xn]` chacune des listes
   `[x1;...;xn]`, `[x2;...;xn]` ..., `[xn-1;xn]`, `[xn]`, `[]`
   Ecrire une fonction `suffixes : 'a list -> 'a list list` telle que `suffixes l` renvoie la liste complete des su xes de `l`, tries (comme ci-dessus) par taille decroissante.
2. Appelons *préfixe* de `[x1;...;xn]` chacune des listes
   `[]`, `[x1]`, `[x1;x2]` ...`[x1;...;xn-1]`, `[x1;...;xn]`
   Ecrire une fonction `prefixes : 'a list -> `a list list` telle que `prefixes l` renvoie la liste completes des pre xes, tries (comme ci-dessus) par taille croissante.
3. Appelons *séparation* de `[x1;...;xn]` chacun des couples de listes
   `([],[x1;...;xn])`, `([x1],[x2;...;xn])` ...`([x1;...;xn-1],[xn])`, `([x1;...;xn], [])`
   Ecrire une fonction `separations : 'a list -> `a list list` telle que `separations l` renvoie la liste completes des separations de `l`, tries (comme ci-dessus) par ordre de membre gauche croissant.
4. Appelons *rotation* de `[x1;...;xn]` chacune des listes
   `[x1;...;xn]`, `[x2;...;xn;x1]`, `[x3;...;xn;x1;x2]` ...`[xn;x1;...;xn-1]`
   La liste `[xi+1;...;xn;x1;...;xi]` est appelee *rotation gauche de rang* `i` de `[x1;...;xn]`. Par convention, la rotation gauche de `[x1;...;xn]` de rang `n + i` est egale a sa rotation gauche de rang `i`. La liste vide est l'unique rotation de la liste vide.
   Ecrire une fonction `rotation_gauche : 'a list -> int -> 'a list` telle que `rotation_gauche l i` renvoie la rotation gauche de `l` de rang `i`.

Exercice 4 Un *arbre de traits* est une structure recursive de nie comme suit : un arbre de trait consiste en un n ud contenant une certaine information { appelee *étiquette* de ce n ud { et qui a un nombre quelconque de ls (ce nombre peut être zero). Chacun des ls d'un n ud est adresse par un *nom*. Un n ud peut avoir plusieurs ls avec le même nom. Un arbre de traits sera dit *bien formé* si aucun de ses n uds n'a deux ls avec le même nom.

Voici deux exemples d'arbres de traits. L'arbre a gauche n'est pas bien forme, tandis que