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

Programmation Fonctionnelle
Année 2013-2014

# Examen

## Lundi 6 janvier 2014

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.

Les exercices doivent être rédigés en fonctionnel pur : ni références, ni tableaux, ni boucles `for` ou `while`, pas d'enregistrements à champs mutables. Chaque fonction ci-dessous peut utiliser les fonctions prédéfinies (sauf indication contraire), et/ou les fonctions des questions précédentes.

Cet énoncé a 4 pages.

**Exercice 1** (Expressions). Qu'affiche-t-il quand on entre l'une après l'autre les expressions suivantes dans le toplevel Caml ? Donner le type de chaque expression ou définition (ou le message d'erreur si l'expression est mal typée), ainsi que sa valeur et son effet de bord (ou l'exception levée) le cas échéant. Justifier si nécessaire.

1. `1 +. 3 = 12;;`
2. `2. *. (float_of_int (int_of_float 10.));;`
3. `"foo" ^ "bar" ^ "baz";;`
4. `let f x = x + 1 in let g = fun x -> x * 2 in g (f 17);;`
5. `let rec fact2 = function 0 -> 1 | n -> n * fact2 (n - 2);;`
6. `fact2 5;;`
7. `fact2 (-1);;`
8. `let g = (+) 42 in g 3;;`
9. `(1,2,3)::[(4,5,6)];;`
10. 
```
match [1;2;3] with
  [] -> 17
| [x;y] -> 25
| x::y -> 33
| _ -> 51;;
```
11. 
```
exception Ex;;
try raise Ex
with Ex -> raise Ex;;
```
12. 
```
type mtype = Base of bool | Ind of mtype;;
Ind (Ind (Base true));;
```
13. (Bonus)

```
type 'a ptype = Ground of 'a | Step of ('a ptype);;
Step (Ground (Step (Ground 17)));;
```

**Exercice 2.**

1. Écrire une fonction `un_sur_deux : 'a list -> 'a list`. Cette fonction doit renvoyer la sous-liste d'une liste obtenue en ne prenant qu'un élément sur deux, à partir du premier.

   Exemple : `un_sur_deux ['a';'b';'c';'d';'e';'f';'g'] = ['a';'c';'e';'g']`

2. Écrire une fonction `groupes_de_deux : 'a list -> 'a list list`. Appliquée à une liste `l`, cette fonction doit construire la liste de listes obtenue en regroupant deux par deux les éléments de `l`, en partant des deux premiers. Si `l` contient un nombre impair d'éléments, la dernière liste sera simplement une liste singleton. Exemple :

```
groupes_de_deux ['a';'b';'c';'d';'e';'f';'g']
                = [['a';'b'];['c';'d'];['e';'f'];['g']]
```

3. On souhaite généraliser la fonction précédente, en regroupant cette fois les éléments d'une liste en listes à `k` éléments, où `k` est un entier positif quelconque.

   Écrire `decomp : int -> 'a list -> ('a list * 'a list)`. Appliquée à un entier `n` et une liste `l`, cette fonction doit renvoyer un couple de listes `(lg, ld)` où : `lg` est la liste formée des `n` premieres éléments de `l`, et `ld` est la liste des éléments restants. Si `l` contient moins de `n` éléments, cette fonction doit simplement renvoyer `(l,[])`.

   Exemple : `decomp 4 ['a';'b';'c';'d';'e;'f'] = (['a';'b';'c';'d'], ['e';'f'])`

   Indication : lorsque la valeur d'une expression `expr` est un couple, on peut récupérer directement les composantes de ce couple en écrivant `let (x,y) = expr in ...`

4. En vous servant de la fonction précédente, écrire `groupe_de : int -> 'a list -> 'a list list` telle que `groupes_de k l` renvoie la liste de listes obtenue en regroupant en listes à `k` éléments les éléments de `l`, à partir des `k` premiers. Si le nombre d'éléments de `l` n'est pas un multiple de `k`, la dernière liste contiendra simplement un groupe incomplet. Exemple :

```
groupes_de 3 ['a';'b';'c';'d';'e';'f';'g']
             = [['a';'b';'c'];['d';'e';'f'];['g']];;
```

**Exercice 3.** Dans cet exercice on reprend la représentation en OCaml des arbres binaires (déjà vue en TP) à l'aide de la déclaration de type (polymorphe) suivante :

```
type 'a arbre = F | N of 'a * 'a arbre * 'a arbre ;;
```

où la construction `N(x,g,d)` représente un arbre dont la racine est étiquetée par `x` (de type générique `'a`) et qui a comme fils gauche `g` et fils droit `d` ; les feuilles `F` ne sont pas étiquetées.

Un *arbre binaire de recherche* (ABR) est un arbre binaire dans lequel *tous* les nœuds `N(x,g,d)` vérifient la propriété suivante (la comparaison utilisée ici est la comparaison générique $<$ d'OCaml) :

- toutes les étiquettes apparaissant dans le fils gauche `g` sont strictement inférieures à `x`.
- toutes les étiquettes apparaissant dans le fils droit `d` sont strictement supérieures à `x`.

Par exemple, parmi les deux arbres binaires d'entiers de la Figure 1, le premier est un ABR et le deuxième non.

1. Écrire une fonction `max_abr : 'a arbre -> 'a` qui prend en argument un arbre binaire `t` *supposé être un ABR différent de* `F`, et renvoie le plus grand élément de `t`. Cette fonction devra profiter des propriétés des ABR pour trouver ce plus grand élément sans avoir besoin de parcourir tout l'arbre. Vous pourrez par exemple lancer une exception si l'arbre entier est `F`.

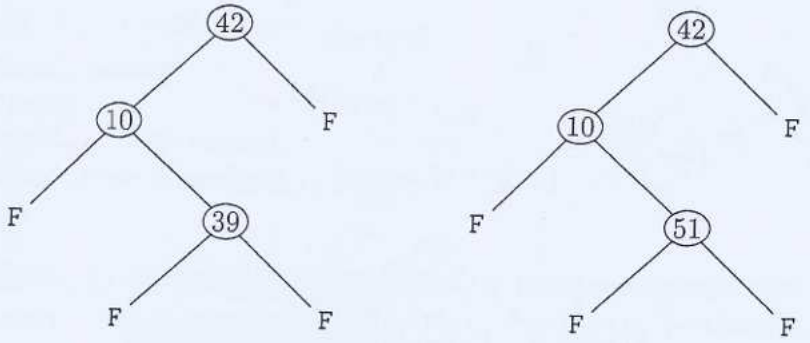



FIGURE 1 – Exemple et contre-exemple d'ABR

2. De façon similaire, écrire une fonction `min_abr : 'a arbre -> 'a` qui prend un arbre binaire `t` *supposé être un ABR différent de* F, et renvoie le plus petit élément de `t`.
3. En utilisant `min_abr` et `max_abr`, écrire une fonction `est_abr : 'a arbre -> bool` renvoyant `true` si l'arbre binaire passé en argument est un ABR, `false` sinon.

Dorénavant, on ne considère que des ABR. On souhaite construire l'*union* de deux ABR, c'est-à-dire un nouvel ABR qui contient exactement les éléments de ces deux ABR d'origine. Cette union se fera en éliminant les redondances : de toute façon, un même élément ne pourrait pas apparaître plusieurs fois dans un véritable ABR.

4. Écrire tout d'abord une fonction `les_petits : 'a -> 'a arbre -> 'a arbre` qui prend un élément `x` et un ABR `a`, et fabrique un nouvel ABR constitué des éléments de `a` qui sont strictement inférieurs au pivot `x`.
5. On suppose avoir une fonction similaire `les_grands : 'a -> 'a arbre -> 'a arbre` qui fabrique l'ABR des éléments strictement supérieurs à un pivot donné (ne pas l'écrire). En utilisant ces deux fonctions `les_petits` et `les_grands`, écrire maintenant une fonction `union : 'a arbre -> 'a arbre -> 'a arbre` qui construit par récurrence un ABR issu de l'union de deux ABR.

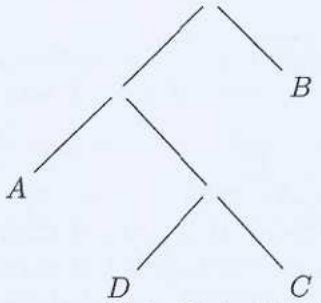
   *Attention* : les fonctions `union` produisants des résultats correctes mais sans respecter la consigne ci-dessus seront pénalisées.

**Exercice 4** (Compression selon Huffman). Le but de ce problème est d'écrire le cœur d'un compresseur de fichier, semblable à *gzip* mais utilisant l'algorithme de Huffman.

L'idée fondamentale de l'algorithme de Huffman est de coder les valeurs possibles des octets du fichier d'entré (les « symboles ») par un nombre variable de bits : les symboles les plus courants seront codés par un petit nombre de bits, tandis que les plus rares par un nombre de bits plus grands.

Le codage que construit l'algoritme de Huffman est un *codage préfixe* : aucun code n'est un préfixe d'un autre. Un codage préfixe peut etre représenté par un arbre binaire où les symboles à coder sont placés aux feuilles ; le code associé à un symbole est obtenu en suivant le chemin menant à ce symbole depuis la racine de l'arbre.

Pour effectivement gagner de l'espace, un pas vers la gauche serait représenté par le bit 0 et un pas vers la droite par le bit 1. Nous allons utiliser ici un type enumeré `type atome = Droite | Gauche` pour les bits et `type parcours = atome list` pour les parcours.




| Symbole | Code |
|---|---|
| A | `[Gauche;Gauche]` |
| B | `[Droite]` |
| C | `[Gauche;Droite;Droite]` |
| D | `[Gauche;Droite;Gauche]` |

```
let ex = Node(Node(Sym('A'),Node(Sym('D'),Sym('C'))),Sym('B'))
```

FIGURE 2 – Exemple d'arbre de Huffman

1. A partir de l'exemple, définir le type `arbre_huffman` polymorphe en fonction du type des symboles.

   La première étape de l'algorithme consiste à construire un histogramme, c'est-à-dire une `('a*int) list` qui a chaque symbole associe le nombre de fois qu'il apparait dans le fichier à compresser (sa « fréquence »). Ne pas écrire cette fonction !

   À partir de cet histogramme, on défini le *poids* d'un arbre comme :
   - Le poids de `Sym(x)` est la fréquence de `x` dans l'histogramme.
   - Le poids de `Node(a,b)` est la somme des poids de `a` et `b`.

2. Écrire la fonction `insert : ('a * int) -> ('a * int) list -> ('a * int) list` telle que si `l` est une liste *triée par ordre croissant selon ses deuxièmes arguments*, `insert e l` est une liste *triée par ordre croissant selon ses deuxièmes arguments* contenant `e`.

3. En déduire une fonction pour trier l'histogramme par fréquence de symbole croissante.

   L'algorithme de Huffman proprement dit travaille sur une forêt (une liste de paires (arbre de Huffman, poids de l'arbre)) triée par ordre croissant de poids.

4. Écrire la fonction `construit_foret` qui fait d'un histogramme une forêt dont les arbres sont `Symb(x)` pour tous les symboles `x` et qui est convenablement triée. Donner son type.

5. Écrire la fonction `construit_huffman : ('a * int) list -> 'a arbre_huffman` qui construit pour un histogramme donné son arbre de Huffman.

   À chaque étape de l'algorithme, on prend les deux arbres de moindre poids (ce sont les premiers éléments de la forêt) que l'on enlève de la forêt et que l'on combine en un seul arbre. On insère ensuite l'arbre obtenu au bon endroit de la forêt.

   L'algorithme se termine lorsque la forêt ne contient qu'un seul arbre.

6. Écrire la fonction `dictionnaire : 'a arbre_huffman -> ('a * parcours) list` qui renvoit la liste des parcours associé aux symboles. Par exemple,

```
dictionnaire ex = [
(A, [Gauche;Gauche]); (D, [Gauche;Droite;Gauche]);
(C, [Gauche;Droite;Droite]); (B, [Droite]);
]
```

**Indication** Vous aurez besoin d'une fonction auxilière qui parcours l'arbre en accumulant l'inverse du parcours réalisé pour arriver jusqu'au nœud en cours de traitement.

Des points bonus irons aux solutions récursives terminales mais restez pragmatiques.