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.

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 simple.

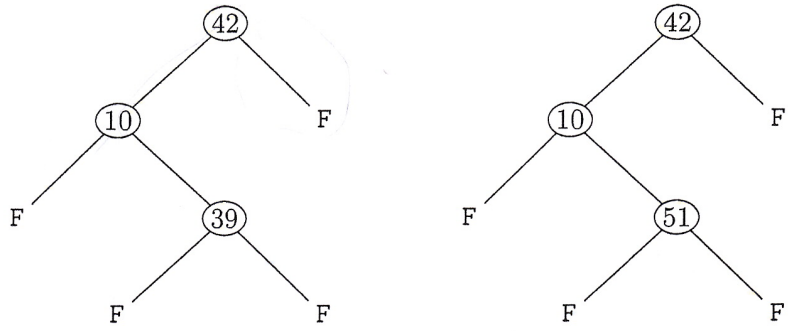


FIGURE 1 – Exemple et contre-exemple d'ABR

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.

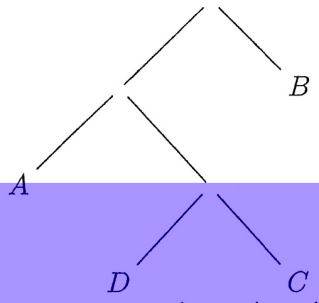
4. Écrire tout d'abord une fonction `les_petits : 'a -> 'a arbre -> 'a arbre` qui prend un élément `x` et un ABR `a`, et fabrique un nouvel ABR constitué de ceux éléments de `a` qui sont strictement inférieurs au pivot `x`.
5. On suppose avoir une fonction similaire `les_grands : 'a -> 'a arbre -> 'a 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'entrée (les « symboles ») par un nombre variable de bits : les symboles les plus fréquents sont codés sur peu de bits, les symboles les moins fréquents sur plus de bits. Les symboles à coder sont placés aux feuilles ; le code associé à un symbole est obtenu en suivant le chemin

type 'a arbre
| Sym of 'a
| Node of 'a*arbre



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 à 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 crée à partir d'un histogramme une forêt où tous 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 iront aux solutions récursives terminales mais restez pragmatiques.