OPTION INFORMATIQUE
Devoir surveillé 1

Toutes les fonctions sont à écrire en OCaml. On attachera le plus grand soin à leur lisibilité. On pourra toujours librement utiliser une fonction demandée à une question précédente, même si cette question n'a pas été traitée. Lorsqu'une question impose une contrainte sur le coût en temps d'une fonction, on ne demande pas de démontrer que cette contrainte est respectée.
Pour toute fonction programmée, on précisera le type de cette fonction.

1 Représentation d'ensembles par des listes

Dans cette partie, on représente un ensemble fini par la liste **triée** (dans l'ordre croissant) **sans doublon** de ses éléments. On identifiera alors l'ensemble et la liste le représentant.

1. Écrire une fonction calculant le cardinal d'un ensemble.
2. Écrire une fonction renvoyant l'élément minimal d'un ensemble non vide. Cette fonction devra être de coût constant.
3. Écrire une fonction renvoyant l'élément maximal d'un ensemble non vide.
4. Écrire une fonction testant l'appartenance d'un élément à un ensemble. On prendra garde à stopper le parcours dès que possible.
5. Écrire une fonction `comprehension` prenant en argument un prédicat p et un ensemble l et renvoyant l'ensemble des éléments de l vérifiant p.
6. Écrire une fonction `insertion` prenant en argument un élément x et un ensemble l et renvoyant l'ensemble contenant x et les éléments de l.
7. Écrire une fonction renvoyant l'union de deux ensembles. Chaque liste ne devra être parcourue qu'au plus une fois.
8. Écrire une fonction renvoyant l'intersection de deux ensembles.
9. Écrire une fonction renvoyant la différence ensembliste de deux ensembles.
10. Écrire une fonction testant l'inclusion d'un ensemble dans un autre.
11. On cherche à écrire une fonction `images` renvoyant l'ensemble des images des éléments d'un ensemble par une fonction f. L'enjeu est de s'assurer que ces images vont bien apparaître dans l'ordre croissant et sans doublon dans le résultat.
    (a) Écrire une version de cette fonction `images` utilisant la fonction `insertion`.
    (b) Donner un exemple de fonction f pour laquelle le calcul de la version précédente de `images` a un coût quadratique en la longueur de la liste.
    (c) Pour améliorer le coût dans le pire cas, on se propose d'utiliser un tri fusion. Écrire une fonction `partition` prenant en argument une liste l (non nécessairement triée ou sans doublon) et renvoyant deux listes se partageant les éléments de l et dont la différence de longueur est au plus 1.
    (d) En déduire une fonction récursive `tri_fusion` prenant en argument une liste non triée et renvoyant l'ensemble de ses éléments.
    (e) Utiliser cette fonction `tri_fusion` pour obtenir une nouvelle version de la fonction `images`.
12. Écrire une fonction `produit` réalisant le produit cartésien de deux ensembles. On notera que l'ordre que Caml utilise sur les couples est l'ordre lexicographique, `produit [1; 2; 3] [4; 5]` doit renvoyer `[(1, 4); (1, 5); (2, 4); (2, 5); (3, 4); (3, 5)]`.
13. Écrire une fonction `parties` renvoyant l'ensemble des parties d'un ensemble. Là encore, on notera que Caml utilise l'ordre lexicographique sur les listes, `parties [1; 2; 3]` doit renvoyer `[[]; [1]; [1; 2]; [1; 2; 3]; [1; 3]; [2]; [2; 3]; [3]]`.

2 Expressions arithmétiques

On considère dans cet exercice les expressions arithmétiques définies au sein du type suivant (par simplicité, on n'utilisera que les opérations + et $*$) :

```
type expr_arith =
    Plus of expr_arith * expr_arith |
    Fois of expr_arith * expr_arith |
    Const of float |
    Var of char ;;
```

1. Représenter l'expression $e = 3x^2 + y$ dans ce type.
2. Donner les écritures préfixe et postfixe de l'expression obtenue.
3. Écrire une fonction `afficher_infixe` prenant en argument une telle expression, et l'affichant en notation infixe, en utilisant un parenthésage systématique. Par exemple, l'expression obtenue à partir de e sera affichée sous la forme `((3.*(x*x))+y)`.
4. On note o le nombre d'occurrences des constructeurs `Plus` et `Fois` dans une expression, et v le nombre d'occurrences des constructeurs `Const` et `Var`. Donner une relation entre o et v, puis la démontrer par induction structurelle.
5. Écrire une fonction `valeur` prenant en argument une expression arithmétique et une fonction `f : char -> float` associant à chaque caractère nom de variable une valeur, et renvoyant la valeur de l'expression toute entière.
6. Écrire une fonction `deriv` prenant en argument une expression arithmétique et un caractère correspondant à un nom de variable, et renvoyant l'expression obtenu par dérivation selon cette variable.
   Par exemple, sur l'expression e dérivée selon `'x'`, on obtiendrait une représentation de $6x$.
7. Écrire une fonction `exposant` prenant en argument une expression et un caractère correspondant à un nom de variable, et renvoyant l'exposant maximal de cette variable dans la forme développée de l'expression.
   Par exemple, dans $(x + y^2)x + y^3$, l'exposant de x est 2, celui de y est 3.
8. Écrire une fonction `simplifie` prenant en argument une expression arithmétique et renvoyant l'expression obtenue en appliquant **autant de fois que possible** les simplifications :
$$0 \times e = e \times 0 = 0, \quad 1 \times e = e \times 1 = e, \quad 0 + e = e + 0 = e$$
9. On souhaite à présent **parser** une expression, c'est-à-dire passer d'une liste de caractères à un arbre. On suppose pour simplifier que ces listes de caractères ne contiennent aucune constante, mais uniquement les caractères `'+'`, `'*'` pour les opérations, ou d'autres caractères correspondant alors à des noms de variables.
   (a) Écrire une fonction `parse_pref` prenant une liste de caractères l, et renvoyant un couple (a, d) tel que l est la concaténation de deux listes g et d, g correspondant à la forme préfixe d'une expression dont l'arbre est a. On supposera que l admet une telle décomposition (qui est unique).
   En particulier, si l est une liste correspondant à la forme préfixe d'une expression, `parse_pref l` doit renvoyer `(a, [])`, où `a` est l'arbre de l'expression. Par exemple :
   `parse_pref ['+'; 'a'; 'b']` doit renvoyer `Plus (Var 'a', Var 'b'), []`
   `parse_pref ['+'; 'a'; '*'; 'b'; 'c'; 'd']`
   doit renvoyer `(Plus (Var 'a', Fois (Var 'b', Var 'c')), ['d']).`
   (b) On travaille à présent sur des formes infixes, qu'on suppose systématiquement parenthésées. Les listes contiennent donc également les caractères `'('` et `')'`.
   Écrire une fonction `parse_inf` de spécification similaire à celle de `parse_pref`, en remplaçant l'ordre préfixe par l'ordre infixe parenthésé. Par exemple :
   `parse_inf ['(';'a';'+';'b';')']` doit renvoyer `Plus (Var 'a', Var 'b'), []`
   `parse_inf ['('; '('; 'a'; '+'; 'b'; ')'; '*'; 'c'; ')'; '*'; 'a'; ')']`
   doit renvoyer `(Fois (Plus (Var 'a', Var 'b'), Var 'c'), ['*'; 'a'; ')']).`