Programmation fonctionnelle
TD de soutien

Exercice 1 (Fonctions sur les listes)

1. Réécrivez les fonctions suivantes du module `List` :
   — `List.append`,
   — `List.rev`,
   — `List.fold_right` et
   — `List.fold_left`.
2. Donnez une version récursive terminale de la fonction `fold_right`.
3. Déduisez-en une version récursive terminale de `append`.

Exercice 2 (Notation polonaise inverse)

Usuellement, une expression arithmétique (composée de nombres et des opérations binaires usuelles) est donnée sous forme infixe. Autrement dit, les opérandes apparaissent de part et d'autre de l'opérateur. La composition des expressions nécessite des conventions de priorités sur les opérateur et l'usage de parenthèses pour rendre la lecture des expressions non ambiguës. Les expressions $1 + 2 \times 4 + 3$ et $(1 + 2) \times 4 + 3$ sont différentes.

Des formes d'écriture alternative existent. Nous nous intéressons ici à la notation polonaise inverse. Il s'agit d'une notation postfixe où l'opérateur apparaît après ses opérandes. La lecture des opérations ne nécessite pas de parenthèse. L'expression infixe $1 + 2 \times 4 + 3$ s'écrit $1\ 2\ 4 \times + 3 +$ tandis que l'expression $(1 + 2) \times 4 + 3$ s'écrit $1\ 2 + 4 \times 3 +$.

Le deuxième avantage de cette notation est la facilité de son calcul via une pile. A chaque fois qu'un nombre apparaît on l'empile. Si c'est une opération binaire, on dépile ses deux opérandes puis on empile le résultat. À la fin du calcul, si l'expression est bien formée, la pile est réduite à un élément qui est la valeur de toute l'expression.

On se donne les types suivants pour représenter une expression.

```
type entry = Int of int | Operator of (int -> int -> int)
type expression = entry list
```

1. Définissez une fonction `is_well_formed` qui indique si une expression en notation polonaise inverse est bien formée.
2. Définissez une fonction `eval` qui évalue une expression en notation polonaise inverse.

Exercice 3 (Arbre binaire de recherche)

Un arbre binaire dont les nœuds sont étiquetés par des entiers est un arbre de recherche si :
— c'est une feuille ou
— c'est un nœud d'étiquette x, de fils gauche l et de fils droits r tels que l (resp. r) est un arbre de recherche dont tous les nœuds ont des étiquettes inférieures à x (resp. supérieures à x).

On représente un arbre binaire de recherche par le type :

```
type bst = Node of (int * bst * bst) | Leaf
```

1. Définissez une fonction `mem` qui indique si un élément est présent dans un arbre binaire de recherche.
2. Pour insérer un élément dans un arbre binaire de recherche, on parcours l'arbre jusqu'à trouver une feuille. On la remplace alors par un nouveau nœud contenant uniquement l'élément à insérer.

   Définissez une fonction `insert` qui insère un élément dans un arbre binaire de recherche.
3. Définissez une fonction `min_bst` qui indique si un élément est présent dans un arbre binaire de recherche.
4. Pour supprimer un élément, on cherche sa première occurrence dans l'arbre.
   — Si l'élément n'apparaît pas dans l'arbre, celui-ci est inchangé.
   — Si l'élément est à la racine d'un nœud dont l'un des fils est vide, on le remplace par l'autre.
   — Sinon on remplace l'étiquette du nœud par le minimum y de son sous-arbre droit et on supprime y du sous-arbre droit. On se retrouve alors nécessairement dans le cas précédent.

   Définissez une fonction `remove` qui supprime un élément d'un arbre binaire de recherche.
5. Définissez une fonction `list_to_bst` qui transforme une liste en arbre binaire de recherche.
6. Définissez une fonction `bst_to_list` qui transforme un arbre binaire de recherche en liste triée.
7. Si votre fonction précédente n'est pas en temps linéaire, modifiez la pour atteindre cette complexité.
8. Si votre fonction précédente n'est pas récursive terminale, modifiez la pour qu'elle le devienne. La complexité doit rester linéaire.