Fonctions du premier ordre

Exercice 1 : Un peu de pliage

Q1. Écrivez une fonction `somme` avec un `foldr`. Puis une autre avec un `foldl`.
Q2. Redéfinissez la fonction `++` avec le `fold` qui va bien.
Q3. Redéfinissez la fonction `concat` avec le `fold` qui va bien.
Q4. Redéfinissez la fonction `map` avec le `fold` qui va bien.
Q5. Redéfinissez la fonction `filter` avec le `fold` qui va bien.

Exercice 2 : Le type Maybe, sortez couvert !

Dans cet exercice nous allons redéfinir des fonctions qui, de base, déclenche des exceptions en cas de mauvaise utilisation, afin de les rendre sûres, c'est-à-dire qu'elles ne font pas planter le programme.
Q1. Redéfinissez la fonction `!!` en une fonction sûre `atSafe`.
Q2. Redéfinissez la fonction `tail` en une fonction sûre `tailSafe`.
Q3. Définissez une fonction `minSafe :: [a] -> Maybe a` qui retourne le minimum d'une liste.

Exercice 3 : Des piles...

Reprenez le type `Pile a` vu en cours.
Q1. Écrivez une fonction `estVide :: Pile a -> Bool`
Q2. Écrivez une fonction `sommet :: Pile a -> Maybe a`
Q3. Écrivez une fonction `depiler :: Pile a -> Pile a`
Q4. Écrivez une fonction `empiler :: a -> Pile a -> Pile a`
Q5. Écrivez une fonction `empilerTout :: [a] -> Pile a -> Pile a` de deux versions différentes, une sans pliage, une autre avec pliage.

Exercice 4 : ... et des files !

Q1. Définissez le type `File a`
Q2. Écrivez une fonction `estVide :: File a -> Bool`
Q3. Écrivez une fonction `tete :: File a -> Maybe a`
Q4. Écrivez une fonction `defiler :: File a -> File a`
Q5. Écrivez une fonction `enfiler :: a -> File a -> File a`
Q6. Écrivez une fonction `enfilerTout :: [a] -> File a -> File a` de deux versions différentes, une sans pliage, une autre avec pliage.

Exercice 5 : Arbre binaire de recherche

Q1. Définissez un type `ArbreBinaire a`
Q2. Écrivez une fonction `ajouterValeur :: ArbreBinaire a -> a -> ArbreBinaire a`
Q3. Écrivez une fonction `supprimerValeur :: ArbreBinaire a -> a -> ArbreBinaire a` qui supprime la première valeur rencontrée uniquement.
Q4. Écrivez une fonction `rechercherValeur :: ArbreBinaire a -> a -> Bool`

Exercice 6 : Encore plus d'arbre

Q1. Définissez la structure `Arbre`. Vous pouvez reprendre celle de l'exercice précédent sur les arbres binaires de recherche.

Q2. Définissez une fonction `hauteur` qui calcule la hauteur de l'arbre.

Q3. Définissez une fonction `nb_noeuds` qui calcule le nombre de nœuds d'un arbre.

Q4. `hauteur` et `nb_noeuds` se ressemble beaucoup. Définissez une fonction `fold_arbre` qui applique un pliage sur un arbre.

Q5. Définissez une fonction `hauteur2` en utilisant la fonction `fold_arbre`.

Q6. Définissez une fonction `nb_noeuds2` en utilisant la fonction `fold_arbre`.

Q7. Définissez une fonction `est_complet` qui indique si un arbre est complet, c'est-à-dire, qu'il possède le même nombre de nœuds à droite et à gauche pour chaque nœud.

Q8. Avez vous utilisé `fold_arbre` pour la question précédente ? Si non, au boulot !

Exercice 7 : Expressions mathématique

Les expressions mathématiques peuvent s'écrire sous la forme d'un arbre.

Par exemple l'expression $((3 * 2) + (7 - 3))/5$ peut se représenter sous la forme de l'arbre suivant :

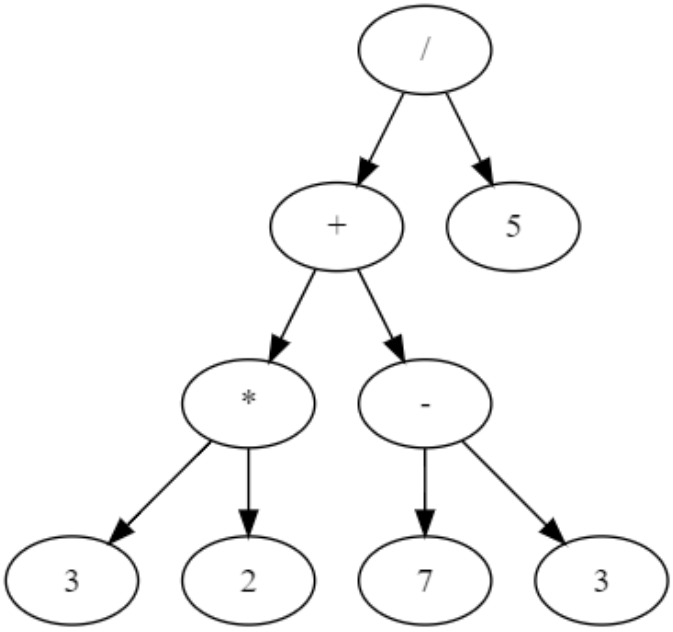

Figure 1: représentation sous la forme d'arbre

La représentation sous forme d'arbre permet d'éviter d'avoir besoin de connaître la priorité des différents opérateurs pour pouvoir interpréter l'expression.

Dans notre représentation arborescente, un nœud sera représenté par un opérateur, et une feuille par un littéral.

Q1. Définissez une structure de données `Expression`.

Q2. Ecrivez une fonction `afficherExp :: Expression -> String` qui affiche une expression mathématique à partir de sa représentation en arbre.

Q3. Ecrivez une fonction `evalExp :: Expression -> Float` qui évalue une expression, c'est-à-dire qui réalise le calcul.