Programmation fonctionnelle en Haskell - TP 4 : fonctions d'ordre supérieur

> Une fonction d'ordre supérieur est une fonction prenant comme argument au moins une fonction et/ou renvoyant une fonction.

Exercice 1. Calcul de sommes

1. Écrire une fonction `sumf` qui, étant donné un entier `n` et une fonction `f` à un argument, calcule

$$\sum_{i=0}^{n} f(i)$$

2. Écrire une fonction `sumSquare` qui, étant donné un entier `n` calcule

$$\sum_{i=0}^{n} i^2$$

Exercice 2. Fonctions symétriques

1. Écrire une fonction `sym` qui étant une fonction binaire `f` renvoie une fonction binaire `g` telle que :

$$g(x, y) = f(y, x)$$

2. Écrire une fonction `symf` qui, étant donné une fonction unaire `f` définie sur des nombres positifs retourne une fonction `g` telle que :

$$g(x) = \begin{cases} f(x) & \text{si } x \geq 0 \\ -f(-x) & \text{sinon} \end{cases}$$

Exercice 3. `map`, `fold*`, `filter`

Mappons...

1. Créer à l'aide de `map` une fonction `maj` permettant de mettre en majuscule une chaîne de caractère.
NB : la fonction `Data.Char.toUpper` permet de mettre un caractère en majuscule.

```
Prelude> maj "abc"
"ABC"
```

2. Créer à l'aide de `map` une fonction `reverseAll` permettant d'inverser l'ensemble des lettres d'une liste de mots.

```
Prelude> reverseAll ["Totor","le","Castor"]
["rotoT","el","rotsaC"]
```

3. Créer toujours à l'aide de `map` une fonction `pair` prenant comme argument une liste d'entier et renvoyant une liste de booléens permettant de savoir si le ième élément le la liste initiale est pair ou non.

```
Prelude> pair [1..5]
[False,True,False,True,False]
```

Foldons...

1. Écrire une fonction permettant de calculer la somme des nombres d'une liste en utilisant `foldl` puis `foldl1`.
2. Écrire une fonction permettant de calculer la somme des carrés d'une liste de nombres
3. Redéfinir les fonctions `length`, `and`, `concat` et `elem` via la fonction d'ordre supérieur `foldl`.

Et filtrons...

1. Écrire un prédicat `sup10` qui renvoi vrai si son argument est un entier supérieur à 10, faux sinon.
2. Écrire une fonction `selecSup10` qui prend comme argument une liste `l` et qui retourne une liste composée des éléments supérieurs à 10. Ne pas utiliser `filter`.
3. Réécrire la fonction précédente avec `filter`
4. Écrire une fonction `snsVlls` supprimant l'ensemble des voyelles d'une chaîne de caractères.
5. Écrire une fonction renvoyant uniquement les nombres d'une liste qui sont premiers. On définira pour cela la fonction `prim`, prédicat précisant si un nombre est premier ou non.

Exercice 4. Fonctions `my*`

1. Écrire une fonction `myMap` qui applique une fonction `f`(unaire) à tous les éléments d'une liste et qui retourne la liste des résultats. Écrire une fonction qui prend en entrée une liste et qui renvoie la même liste de nombre à laquelle on aura ajouter `1` à tous les éléments.
2. Écrire une fonction `myFoldl` qui réduit les élements d'une liste suivant la fonction d'accumulation et la valeur initiale de l'accumulateur. Écrire `myFoldl1` en réutilisant myFoldl. Idem avec `myFoldr` et `myFoldr1`.
3. Écrire une fonction `myFilter` qui filtre les éléments d'une liste `l` ne respectant pas le prédicat (unaire) `p`. Enlever l'ensemble des nombres pairs d'une liste.

Problème : *lost in translation*

Le but de cet exercice est de créer un "générateur de traducteurs". Celui-ci prend comme paramètres deux listes de mots que le traducteur connaît, chaque liste étant dans une langue différente. L'élément `n` de la première liste correspondant à l'élément `n` de la seconde liste.

Écrire une fonction `traducteur` qui prend deux listes de mots et qui génère un traducteur. Exemple :

```
Prelude> t = traducteur ["bonjour", "merci", "s'il vous plaît"] ["hello", "thank you", "please"]
Prelude> t "bonjour"
"hello"
```