Fiche de TP no. 3

Caveats : *le temps passe vite ! Si, après vingt minutes, vous êtes encore au premier exercice, cela est un bon indice que vous ne vous donnez pas les moyens d'apprendre.* **Objectifs** : *Écrire des premiers scripts relativement complexes ; se familiariser avec la récursion et avec les actions en Haskell. Donnez priorité aux premier 3 exercices ; faites les autres exercices seulement s'il vous reste du temps.*
Vous pouvez trouver les scripts ici : http://pageperso.lif.univ-mrs.fr/~luigi.santocanale/teaching/PF/code/TP3/

Récursion

Exercice 1.

1. Définissez, dans un script, une fonction récursive

```
fusionner :: [Int] -> [Int] -> [Int]
```

   qui fusionne deux listes triées pour produire une seule liste triée ; par exemple :

```
> fusionner [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```

2. Ajoutez au script la définition d'une fonction récursive

```
msort :: [Int] -> [Int]
```

   qui implémente le tri par fusion (merge sort en anglais). Cet algorithme de tri peut se spécifier par les deux règles suivantes :
   — les listes de longueur ≤ 1 sont déjà triées ;
   — on peut trier les autres listes en les découpant en deux morceaux, en triant ces deux morceaux, et en fusionnant les listes résultantes.
3. Modifiez le script de façon qu'on ait le type

```
msort :: Ord a => [a] -> [a]
```

4. Définissez dans le script 5 listes (non triées) de types diffèrents et utilisez ces listes pour tester la correction de la fonction `msort`.

Input et Output

Exercice 2. Écrivez la définition d'une fonction `doMenu :: [(String,IO a)] -> IO a` qui prend en paramètre une liste de couples `(prompt,action):: (String,IO a)`, affiche à l'écran un menu bien presenté avec les prompts un par ligne, demande à l'utilisateur de faire un son choix, et ensuite exécute l'action choisie

Exercice 3. Implémentez, en `Haskell`, le jeu du *nim*. Les règles du jeu sont les suivantes :
— L'échiquier contient 5 lignes d'étoiles, chaque ligne contenant un nombre d'étoiles.
— Le jeu demarre de cette position :

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

— Deux joueurs enlèvent – alternativement – une ou plusieurs étoiles d'une seule ligne.
— Le gagnant est le joueur qui enlève la (ou les) dernière(s) étoile(s) de l'échiquier.
Conseils :

— Représentez l'échiquier comme une liste de 5 entiers qui représentent les étoiles encore à enlever. Par exemple, la position initiale sera représentée par la liste `[5,4,3,2,1]`.
Un choix d'un joueur sera donc (déterminé par) un couple d'entiers : la ligne au-quelle on veut retirer des étoiles, et le nombre d'étoiles à retirer.

— Écrivez une fonction qui vérifie si, par rapport à un echiquier, le choix d'un joueur est valide :

```
estChoixValide :: [Int] -> (Int,Int) -> Bool
```

— Écrivez une fonction qui met à jour l'échiquier, étant donné le choix d'un joueur :

```
mettreAJour :: [Int] -> (Int,Int) -> [Int]
```

— En prenant le jeu du pendu (vu en cours) comme exemple, ecrivez la boucle principale du jeu.

Fonctions d'ordre supérieur

Exercice 4. Considérez le script suivant :

```
factors :: Int -> [Int]
factors n = [ x | x <- [1..n],  n `mod` x == 0]
perfect :: Int -> Bool
perfect n = sum (factors n) - n  == n
perfects :: Int -> [Int]
perfects n = [ x | x <- [1..n], perfect x]
subList1 :: [Int] -> [a] -> [a]
subList1 ns xs = [ xs !! i | i <- ns ]
subList2 :: [Int] -> [a] -> [a]
subList2 ns xs = [ x | (i,x) <- zip [0..length xs -1] xs, elem i ns ]
```

1. En utilisant le fonctions d'ordre supérieur `map` et `filter`, remplacez chaque expression de compréhension sur les listes par une expression équivalente qui n'utilise pas la compréhension.
2. Montrez avec un exemple que les deux fonctions `subList1` et `subList2` ne sont pas équivalentes (c'est-à-dire , elles ne calculent pas toujours le mêmes résultats, etant donné entrées égales). Pouvez vous deviner une condition suffisante sur la liste `ns` qui garantie l'équivalence ?

Exercice 5. Considérez le script suivant :

```
composeAll :: [a -> a] -> a -> a
composeAll [] = id
composeAll (f:fs) = f . (composeAll fs)
composeMix1 :: (a->a)->[a -> a] -> a -> a
composeMix1 g fs = composeAll (map (.g) fs)
composeMix2 :: (a->a)->[a -> a] -> a -> a
composeMix2 g fs = foldr (\f h -> f.g.h) id fs
```

1. Utilisez les fonctions `map` et `composeAll` pour ajouter au script la définition d'une fonction `sumUpTo :: Int -> Int` qui, étant donné en entré un entier n retourne la somme des tous les nombres de 1 jusqu'à n.
2. Ajoutez au script la définition d'une fonction `iter :: Int -> (a->a)->(a->a)` qui prend en paramètre un entier n et une fonction f pour retourner la fonction f^n (le itéré n-fois de f). Cette définition n'utilisera pas l'induction, ni un schéma d'induction tel que `foldr` (pensez à la fonction `replicate`).
3. Modifiez la définition de la fonction `composeAll` en utilisant la fonction d'ordre supérieur `foldr`.
4. Donnez un exemple où les fonctions `composeMix1` et `composeMix2` ne sont pas équivalentes ;-)