Fiche de TD no. 2

Exercice 1 : *Type et contraintes de classes des fonctions.* Considérez les définitions suivantes :

```
appl (f,x) = f x
pair x y = (x,y)
mult x y = x * y
double = mult 2
sym (x,y) = x == y
palindrome xs = reverse xs == xs
twice f x = f (f x)
incrAll xs = map (+1) xs
norme xs = sqrt (sum (map f xs)) where f x = x^2
greaters n xs = [ x | x <- xs, x > n]
menu xs = concat (map f (zip [1..length xs] xs))
    where f (n,x) = "("++show n++") " ++ show x ++ "\n"
```

1. Calculez les types de toutes ces fonctions.
2. Énumérez tous les opérateurs et/ou fonctions surchargées qui apparaissent dans ces définitions (à la droite de l'égalité simple), avec leur types et contraintes de classe. Parmi ces fonctions, quelles sont des *méthodes de classe* ?
3. Affinez le calcul des types en ajoutant les *contraintes de classe* au type d'une fonction, lorsque un *méthode* (ou une fonction soumise à des contraintes) apparaît dans le corps de la définition.
4. Quelles sont les fonctions de deux arguments ? Si une telle fonction n'est pas en *forme currifiée*, donnez une définition équivalente qui soit en forme currifiée.
5. Que fait la fonction `map` ? Quel est son type ?
6. Quelles sont les fonctions d'*ordre supérieur* ?
7. Quelles sont les fonctions *polymorphes* ?

Conditionnels, équations avec conditions de garde, filtrage

Exercice 2. Considérez la fonction définie comme suit :

```
safetailtail xs =
    if length xs == 0 then []
    else if length xs == 1 then [] else tail (tail xs)
```

— Définissez la même fonction en utilisant, à la place des conditionnels,

1. d'abord les équations avec conditions de garde,
2. ensuite le filtrage par motifs,
3. le filtrage par motifs par l'expression `case .. of ..`

— Esquissez comment les expressions conditionnelles ci-dessus se transforment en expressions `case .. of ..` lors de la précompilation du code dans le langage noyau.

Exercice 3. Donnez trois possibles définitions de l'opérateur logique `(||)`, en utilisant le filtrage par motifs.

Expressions lambda

Exercice 4. Utilisez les expressions lambda pour réécrire les définitions des fonctions `norme` et `menu`—depuis l'Exercice 1—de façon à éliminer les clauses `where`.

Exercice 5. Considerez les expressions lambda ci-dessous :

```
\n -> n + 2

\f -> \x -> f x

\f -> \xs -> [ f x | x <- xs ]

\(b,x,y) -> if b then x else y

\xs -> let m = moyenne xs in
         moyenne (map (\x -> abs (x - m)) xs)
```

Pour chaque expression lambda

1. expliquez ce que l'expression signifie ;
2. donnez lui un type.

Compréhension sur les listes

Exercice 6. Utilisez la compréhension sur les listes pour définir une fonction qui calcule l'écart moyen d'une liste de flottants.

Exercice 7. Un nombre positif est *parfait* s'il est la somme de tous ses facteurs, sauf lui même. En utilisant la compréhension, définissez [1] une fonction

```
perfects :: Int -> [Int]
```

qui retourne la liste de tous les nombres parfaits, jusqu'à la limite passée en paramètre.

Exercice 8 : *Compréhension sur les listes.* Le produit scalaire de deux listes d'entiers xs et ys de longueur n est la somme des produits des entiers correspondants :

$$xs \cdot ys = \sum_{i=0}^{n-1} xs_i * ys_i \,.$$

1. Utilisez la compréhension et les fonctions vues en cours pour définir une fonction qui retourne le produit scalaire de deux listes.
2. Autrement, utilisez l'induction pour définir cette fonction.

Exercice 9. Un triplet (x, y, z) d'entiers positifs est dit *de Pythagore* si $x^2 + y^2 = z^2$.

1. Définissez, en utilisant la compréhension, une fonction

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

   qui envoie un entier n vers la listes de tous les triplets de Pythagore avec composantes dans `[1..n]`.
2. Si (x, y, z) est un triplet de Pythagore, alors (y, x, z) est aussi un triplet de Pythagore. Proposez une solution à l'exercice précèdent, où seulement un triplet parmi (x, y, z) et (y, x, z) apparaît dans la liste retournée.
3. Si (x, y, z) est un triplet de Pythagore, alors $x, y < z$. Proposez un solution de l'exercice précèdent qui utilise cette observation pour accélérer les calculs.

1. Vous pouvez assumer que la fonction `factors :: Int -> [Int]` (vue en cours, qui calcule les facteurs d'un nombre) est définie, donc l'utiliser pour resoudre l'exercice.