# Programmation Fonctionnelle L2 – Examen Final 27/5/2021

### Consignes

- **Documents autorisés : une feuille manuscrite A4 recto-verso.**
- Vous avez le droit d'utiliser toutes les fonctions de la bibliothèque standard de Haskell. Si vous utilisez des fonctions qui se trouvent dans d'autres modules, comme par exemple `System.IO`, vous devez importer les modules concernés avec `import`.
- Pour les exercices de programmation, vous avez le droit de programmer des fonctions auxiliaires si vous en avez besoin.
- Vous devez préciser les types de vos fonctions.
- La note maximum théorique est 22/20, donc vous pouvez sélectionner de ne pas faire quelques exercices si vous n'avez pas assez de temps.

## 1 Types (2 points)

On considère les fonctions `f1,f2,f3,f4` ci-dessous. Déterminer le type inféré par le compilateur pour chacune de ces fonctions (ou si il y a une erreur de compilation, expliquer quelle est l'erreur).

```
1 f1 x y = [x,y]
2 f2 z = fmap (head) z
3 f3 (Just x) = filter x
4 f4 (x:xs) = [xs]
```

## 2 Que font ces lignes ? (3 points)

Pour cet exercice on considère que les trois fonctions suivantes ont été définies :

```
1 f1 x = [x-1,x,x+1]
2 f2 x = if (x>0) then Just (x-1) else Nothing
3 f3 i xs = drop i xs ++ take i xs
```

Quel est le résultat de l'évaluation de chacune des lignes suivantes ? (Si il y a une erreur de compilation, expliquer quelle est l'erreur)

```
1 (++) <$> ["ab","cd"] <*> ["e","f"]
2 (map pure [1,2,3]) ++ [Nothing]
3 [2] >>= f1 >>= f1
4 f3 2 "abcde"
5 foldr f3 [1..10] [1,2,3]
6 fmap (+1) (f2 2)
```

# 3 Minimum Manquant (3 points)

Écrire une fonction `minMissing` qui prend comme argument une liste d'entiers strictement positifs `xs` et renvoie le plus petit entier strictement positif `x` qui ne paraît pas dans `xs`. Vous pouvez supposer que la liste qui vous est donnée satisfait les conditions ci-dessus (c'est-à-dire, qu'elle ne contient pas d'entier négatif), mais il faut prendre en compte que `xs` pourrait contenir des doublons et qu'elle n'est pas forcément triée.

Exemples :

```
*Main> minMissing [5,2,3,1,3,5]
4
*Main> minMissing [5,2,3,4,1]
6
```

# 4 k-Bonacci (4 points)

Écrire une fonction `kbonacci` qui calcule une généralisation de la suite de Fibonacci qu'on appelle `k`-Bonacci. Dans cette généralisation on a un paramètre `k` donné, les `k` premiers termes de la suite sont égaux à 1 et chaque terme suivant est la somme des `k` termes qui le précédent. (Donc la suite de Fibonacci classique correspond à `k=2`.) Votre fonction doit prendre deux arguments, `k` et `n` et retourner le `n`-ième terme de la suite de `k`-Bonacci.
• Quelle est la complexité de votre programme ? **NB**: C'est important de programmer une fonction de complexité **polynomiale**.

Exemples :

```
> map (kbonacci 2) [1..10]
[1,1,2,3,5,8,13,21,34,55]
> map (kbonacci 3) [1..10]
[1,1,1,3,5,9,17,31,57,105]
> map (kbonacci 4) [1..10]
[1,1,1,1,4,7,13,25,49,94]
```

# 5 Fichiers - Print some lines (3 points)

Écrire un programme de Haskell qui ouvre un fichier appelé `data.txt`. Ce fichier contient du texte dans plusieurs lignes. Par exemple, on pourrait avoir le fichier suivant :

```
Hello
This is the second line!
Bla bla
Line 4 is line 3 since we count from 0
Finito!
```

Vous pouvez supposer que le fichier `data.txt` est valide, existe, et suit toujours ce format.

Le but de votre programme est de lire le contenu du fichier `data.txt` et puis d'afficher quelques-uns de ses lignes sélectionnées par l'utilisateur. Plus précisément, si on suppose qu'on numérote les lignes du fichier à partir du $0$, votre programme doit demander à l'utilisateur de donner une liste de lignes qu'il souhaite afficher et puis afficher ces lignes au terminal. Vous pouvez faire l'hypothèse que l'utilisateur donne toujours une liste valide au terminal (donc des entiers positifs qui ne dépassent pas le nombre de lignes du fichier).

Exemple d'utilisation (avec le fichier ci-dessus) :

```
> runhaskell readfile.hs
Which lines should I print?
0 3 4
Here are the lines you requested:
Hello
Line 4 is line 3 since we count from 0
Finito!
```

# 6 Arbres Équilibrés (3 points)

On a vu la définition suivante pour les arbres binaires.

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Dans un arbre binaire, la hauteur d'un nœud est défini comme suit : la hauteur d'une feuille est $0$; la hauteur d'un nœud interne est égale au maximum des hauteurs de ses deux fils plus $1$.

Un arbre est équilibré si pour chaque nœud $x$ de l'arbre on a la propriété suivante : si $h_1, h_2$ sont les hauteurs des deux fils de $x$, alors $|h_1 - h_2| \leq 1$. Autrement dit, les deux sous–arbres (à gauche et à droite) ont à peu près la même hauteur.

1. Écrire une fonction `height` qui prend comme argument un arbre et retourne la hauteur de sa racine.
2. Écrire une fonction `isBalanced` qui prend comme argument un arbre et retourne `True` si l'arbre est équilibré et `False` sinon.
3. Écrire une fonction `makeTree` qui prend comme argument une liste `xs` et retourne un arbre équilibré qui contient tous les éléments de `xs`.

# 7 Partitions (4 points)

En mathématiques, une partition d'un ensemble $X$ est un ensemble de parties non vides de $X$ deux à deux disjointes et dont l'union est $X$. Par exemple, pour l'ensemble $\{1, 2, 3, 4\}$ l'ensemble $\{\{1, 3\}, \{2\}, \{4\}\}$ est une partition valide, alors que $\{\{1, 2\}, \{2, 3, 4\}\}$ et $\{\{1, 3\}, \{4\}\}$ ne sont pas de partitions valides (dans le premier cas car l'élément 2 paraît dans deux ensembles, dans le deuxième cas car 2 ne paraît dans aucun sous-ensemble.)

En Haskell on va utiliser la définition suivante :

```
type Partition a = [[a]]
```

qui signifie que nous allons utiliser des listes de listes pour représenter les partitions.

1. Écrire une fonction `isPartition` qui prend comme argument une liste `xs` (qui représente l'ensemble $X$), une liste `xss` qui représente une partition de $X$, et renvoie `True` si `xss` est une partition valide de `xs`, c'est-à-dire, si chaque élément de `xs` paraît dans exactement un élément de `xss` et `xss` ne contient pas la liste vide. **NB**: étant donné que nous utilisons les listes pour représenter des ensembles, l'ordre des éléments d'une liste n'a aucune importance et votre fonction devrait retourner le même résultat même si on change l'ordre d'éléments dans une liste, ou l'ordre des listes dans la partition.

2. Écrire une fonction `partitions` qui prend comme argument une liste `xs` et renvoie une liste qui contient toutes les partitions possibles de `xs`.

Pour écrire la fonction `partitions` vous le trouverez utile de programmer une fonction auxiliaire `insert` qui prend comme arguments un élément `x` et une partition `xss` et renvoie la liste de toutes les partitions qu'on peut produire à partir de `xss` si on met `x` dans un sous-ensemble existant de `xss` ou si on met `x` dans son propre sous-ensemble. Donc, `insert` a type `a->Partition a->[Partition a]`.
**Hint**: Si vous programmez `insert`, ce sera possible de programmer `partitions` en utilisant une combinaison de `insert` et l'opération `>>=` (essentiellement, vous allez commencer avec une partition triviale et insérer tous les éléments un par un en utilisant `insert`). Si vous n'arrivez pas à programmer `insert`, expliquez quand-même comment on peut programmer `partitions` sous l'hypothèse que `insert` a déjà été programmée.

Exemples :

```
> isPartition [1,2,3,4] [[3],[2,1],[4]]
True
> isPartition [1,2,3,4] [[3],[2,1],[1,4]]
False
> isPartition "abcd" ["ac","b","d"]
True
> insert 'e' ["ac","b","d"]
[["eac","b","d"],["ac","eb","d"],["ac","b","ed"],["ac","b","d","e"]]
> partitions "abc"
[["abc"],["bc","a"],["ac","b"],["c","ab"],["c","b","a"]]
> partitions "abcd"
[["abcd"],["bcd","a"],["acd","b"],["cd","ab"],["cd","b","a"],
 ["abd","c"],["bd","ac"],["bd","c","a"],["ad","bc"],["d","abc"],
 ["d","bc","a"],["ad","c","b"],["d","ac","b"],["d","c","ab"],["d","c","b","a"]]
```