Programmation Fonctionnelle L2 – Examen Final 24/5/2023

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 = map x y ++ map y x
2 f2 (Just x) = [ y++"a" | y<-x ]
3 f3 [x] y = x <*> [y]
4 f4 x y z = if (x>y) then [z] else [x]
```

2 Que font ces lignes ? (3 points)

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

```
1 f1 i xs = take i xs ++ drop (i+1) xs
2 f2 x = [ i*i | i<-[1..x] ]
3 f3 x = if x^2-5*x<0 then Just (x+1) else Nothing
```

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 foldr f1 [0..9] [8,6..2]
2 [2] >>= f2 >>= f2
3 f3 2 >>= f3 >>= f3
4 filter (odd.length.f2) [1..5]
5 fmap (+1) $ f3 5
6 zipWith (^) (f2 4) [1..3]
```

3 Fichiers (3 points)

Écrire un programme de Haskell qui ouvre un fichier appelé `data.txt`. Ce fichier contient plusieurs lignes de texte. Dans chaque ligne il est écrit un prénom, un nom, et un nombre entier qui signifie l'age de la personne concernée, séparés par des espaces. Par exemple, on pourrait avoir le fichier suivant :

```
Ringo Star  29
Mick Jagger 32
Eric Clapton 47
John Lennon 27
Joan Baez 52
Paul Simon 16
Roger Waters 40
```

Vous pouvez supposer que le fichier `data.txt` est valide, existe, et toutes ses lignes suivent 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, votre programme doit demander à l'utilisateur de donner deux nombres entiers : un age minimum a_1 et un age maximum a_2. Le programme doit alors afficher toutes les lignes qui satisfont la condition que l'age de la personne concernée appartient à l'intervalle $[a_1, a_2]$.

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

```
> runhaskell readfile.hs
Minimum age?
20
Maximum age?
40
Here are the people between these ages:
Ringo Star  29
Mick Jagger 32
John Lennon 27
Roger Waters 40
```

4 Majority (3 points)

Étant donné une liste `xs`, on dit qu'un élément `x` est un **élément majoritaire** si le nombre d'apparences de `x` dans la liste est strictement supérieur à la taille de la liste divisée par 2. Écrire une fonction qui prend comme argument une liste `xs`. Si `xs` contient un élément majoritaire `x`, votre fonction doit renvoyer `Just x`, sinon elle doit renvoyer `Nothing`.

Exemples d'utilisation :

```
*> majority "aabc"
Nothing
*> majority "aabca"
Just 'a'
*> majority [1,2,3,1,1]
Just 1
```

5 Pancake Flip (3 points)

Étant donné une liste `xs` et un entier `i` entre 1 et `length xs`, on définit une opération qu'on appelle **pancake flip** comme suit : on prend les `i` premiers éléments de la liste et on calcule la liste inverse, puis on concatène le résultat avec le reste de la liste. Par exemple :

```
*Main> pancakeFlip [0..5] 2
[1,0,2,3,4,5]
*Main> pancakeFlip [0..5] 4
[3,2,1,0,4,5]
*Main> pancakeFlip "abcdef" 4
"dcbaef"
```

- Écrire une fonction `pancakeFlip` qui implémente l'opération ci-dessus.
- Écrire une fonction `kFlips` qui prend comme arguments un entier `k` et une liste `xs`. La fonction `kFlips` doit retourner une liste de toutes les listes qu'on peut obtenir si on applique l'opération `pancakeFlip` sur la liste `xs`, `k` fois.

Exemple :

```
*> kFlips 1 [0..3]
[[0,1,2,3],[1,0,2,3],[2,1,0,3],[3,2,1,0]]
*Main> kFlips 2 [0..3]
[[0,1,2,3],[1,0,2,3],[1,2,0,3],[1,2,3,0],[2,0,1,3],[2,1,0,3],[2,3,1,0],
  [3,0,1,2],[3,2,0,1],[3,2,1,0]]
```

Explication : si on applique l'opération `pancakeFlip` une fois sur la liste `[0..3]`, on peut inverser soit le premier élément, soit les deux premiers éléments, soit les trois premiers, soit toute la liste. Pour obtenir le résultat pour `kFlips 2` on applique le même raisonnement pour chaque liste renvoyée par `kFlips 1`, et on élimine les doublons. Par exemple, `[3,0,1,2]` peut effectivement être obtenue à partir de `[0..3]` si on fait `[0,1,2,3] -> [2,1,0,3] -> [3,0,1,2]`.

6 Nombres de Catalan (3 points)

Les nombres de Catalan C_n sont définis comme suit : $C_1 = 1$ et pour $n > 1$ on met $C_n = \sum_{i=1}^{n-1} C_i \cdot C_{n-i}$. On a donc que les premiers termes de cette suite sont : $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862$. Autrement dit, pour calculer un nombre de Catalan, on prend la liste de tous les nombres précédents, on multiplie le premier avec le dernier, le deuxième avec l'avant-dernier, ...et puis on somme tous les résultats. Ainsi, $14 = 1 \times 5 + 1 \times 2 + 2 \times 1 + 5 \times 1$ et $42 = 1 \times 14 + 1 \times 5 + 2 \times 2 + 5 \times 1 + 14 \times 1$.

Écrire une fonction `catalan` qui prend comme argument un entier `n` et renvoie le n-ième nombre de Catalan C_n. Quelle est la complexité de votre fonction ? (**Indice** : essayez d'utiliser la programmation dynamique pour que la complexité de votre fonction soit polynomiale. Même si vous connaissez d'autres définitions pour la suite de Catalan, on vous demande d'utiliser la définition donnée ci-dessus.)

Exemple :

```
*> map catalan [1..15]
[1,1,2,5,14,42,132,429,1430,4862,16796,58786,208012,742900,2674440]
```

7 Arbres Binaires (5 points)

On a vu la définition suivante pour les arbres binaires.

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

NB: Pour cet exercice on ne va considérer que des arbres binaires qui contiennent des éléments distincts. Par conséquent, pour toutes les questions qui suivent, vous pouvez supposer que l'arbre binaire qui vous est donné ne contient pas de doublons.

Définitions Dans un arbre binaire, la **profondeur** d'un nœud est défini comme suit : la profondeur de la racine est 0; la profondeur d'un nœud interne est égale à la profondeur de son parent plus 1.

On vous rappelle que quand on dit qu'un nœud `x` est un **ancêtre** d'un nœud `y` cela veut dire que tous les chemins entre la racine de l'arbre et `y` contiennent `x`. On peut aussi donner la définition récursive : (i) ∀`x`, `x` est un ancêtre de `x` (ii) si `z` et le parent de `x` et `y` est un ancêtre de `z`, alors `y` est un ancêtre de `x`.

Questions :

- Écrire une fonction `isAncestor` qui prend comme arguments un arbre binaire `t` et deux éléments `x,y`. Le fonction doit renvoyer `True` si `x,y` paraissent dans `t` et `x` est un ancêtre de `y`, et `False` sinon.
- Écrire une fonction `lca` (pour "least common ancestor" – ancêtre minimum en commun), qui prend comme arguments un arbre `t` et deux éléments `x,y`. **Vous pouvez supposer que** `x,y` **paraissent dans** `t`. La fonction doit renvoyer l'élément `z` qui paraît dans `t`, tel que `z` est un ancêtre de `x` et de `y` et `z` est de profondeur maximum.
- Écrire une fonction `depth` qui prend comme argument un arbre binaire `t` et un élément `x` et renvoie la profondeur de `x`. **Vous pouvez supposer que** `x` **paraît dans** `t`.
- Écrire une fonction `distance` qui prend comme argument un arbre binaire `t` et deux éléments `x,y` et renvoie la distance entre `x` et `y` dans l'arbre, où la distance est la longueur du plus court chemin entre les nœuds qui contiennent `x` et `y`. **Vous pouvez supposer que** `x,y` **paraissent dans** `t`. Vous pouvez aussi utiliser les fonctions précédentes (même si vous ne les avez pas programmées).

Pour donner quelques exemples, on considère l'arbre donné par l'expression suivante (voir aussi la Figure 1) :

```
tree1 = Node (Node (Node Leaf 1 Leaf) 2 Leaf)
             5
             (Node (Node Leaf 6 Leaf) 7 (Node Leaf 9 Leaf))
```

Si on utilise les fonctions ci-dessus on a :

```
*Main> isAncestor tree1 5 9
True
*Main> isAncestor tree1 2 9
False
*Main> isAncestor tree1 2 15
False
*Main> lca tree1 6 9
7
*Main> lca tree1 6 2
5
*Main> lca tree1 7 9
7
*Main> depth tree1 9
```

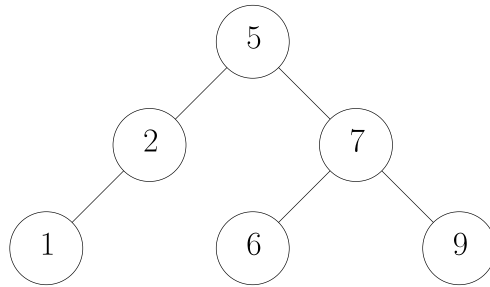

Figure 1: Exemple pour l'exercice 7.

```
2
*Main> depth tree1 2
1
*Main> distance tree1 5 9
2
*Main> distance tree1 1 9
4
```