Programmation Fonctionnelle L2 – Examen Partiel 17/3/2023

1 Types

Considérer le programme suivant.

```
1 f1 x = [x]++"abc"
2 f2 x y = (head x):(tail y)
3 f3 x = x.x
4 f4 x y z = if (tail x) then y else z
5 f5 x y = x:f5 y (tail x)
6 f6 x = filter (/= head x) (tail x)
7 f7 x y = [ z+y | z<-x ]
8 f8 x y = [ (x,y), (y,x) ]
```

- Trouver les lignes qui donnent une erreur de compilation. Expliquer quelle est l'erreur. Pour les expressions `f1,f2,..,f8` pour lesquelles le programme n'a pas d'erreur de compilation, donner le type inféré par le compilateur. (4 points)

2 Types – Part 2

Pour chacun des types suivants, donner une expression **correcte** de Haskell qui a ce type, c'est-à-dire, le compilateur infère le type donné pour votre expression. (2 points)

```
1 Char -> Bool -> [Bool]
2 Ord a => a -> [a] -> [a]
3 [a] -> [a->b] -> [b]
4 (b -> a) -> (a->b) -> a -> a
```

3 Map, Fold, etc.

Quels sont les résultats d'évaluation des lignes suivantes ? Est-ce que toutes les lignes sont correctes ou pouvez vous trouver des erreurs de compilation ? (5 points)

```
1 foldl (\x y -> 2*x+y) 0 [1..5]
2 foldr (.) (+5) [(+3),(*2),(9-)] $ 2
3 filter (odd.length) ["hello","my","name","is","Haskell"]
4 [ x y z | x<-[(+),(*)], y<-[1..3], z<-[1..y] ]
5 zipWith ((*).(+1)) [1..6] [-2..2]
```

4 Powerful Numbers

Un nombre entier positif x est appelé "powerful" s'il existent a, b entiers positifs tels que $x = a^2b^3$. Écrire une fonction de Haskell `powerful` qui prend comme argument un entier n et produit la liste de tous les nombres powerfuls qui sont $\leq$ à n. (2 points)

Exemple :

```
*Main> powerful 100
[1,4,8,9,16,25,27,32,36,49,64,72,81,100]
```

5 Éléments mal placés

Écrire les deux fonctions suivantes : (3 points)

1. La fonction `rank` qui prend comme argument une liste `xs` qui contient des éléments distincts et un élément `x` qui paraît dans `xs` et retourne le *rang* de `x` dans `xs`, c'est-à-dire la position où on devrait placer `x` si la liste `xs` était triée en ordre croissant.

2. La fonction `outOfPlace` qui prend comme argument une liste `xs` qui contient des éléments distincts et retourne le nombre d'éléments de `xs` qui sont mal placés, c'est-à-dire qui ne sont pas dans la position où ils devraient se trouver pour que la liste soit triée en ordre croissant.

Pour les deux fonctions, vous pouvez supposer que les listes données contiennent des éléments distincts (pas de doublons) et que l'élément `x` paraît bien dans `xs`.

Exemples d'utilisation :

```
*Main> rank [1,2,5,4,3,0] 3
3
*Main> rank "acbd" 'c'
2
*Main> outOfPlace [0,5,2,3,1,4]
3
```

Dans le dernier exemple, les éléments $5, 1,$ et 4 sont mal placés (il faut les déplacer pour trier la liste), alors que $0, 2, 3$ se trouvent dans les bonnes positions.

NB: Vous devriez utiliser la fonction `rank` pour programmer la fonction `outOfPlace`. Ce n'est pas nécessaire de trier la liste `xs` pour résoudre ces problèmes. Si vous décidez de le faire quand-même, vous devez implémenter un algorithme de tri et pas simplement évoquer les algorithmes vus en classe.

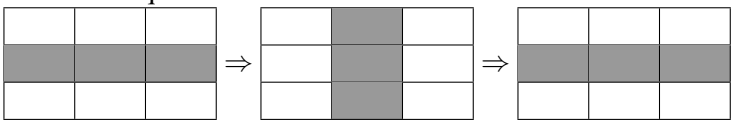
6 Le jeu de la vie

Le jeu de la vie est un automate inventé par le mathématicien John Conway en 1970. Dans ce jeu nous avons une matrice $n \times m$ de cellules qui ont deux états possibles : vivante ou morte. Étant donné une configuration initiale, le jeu se déroule comme suit. Pour chaque cellule c, on regarde son voisinage, à savoir les huit cellules qui se trouvent autour de la cellule c et on applique les règles suivantes :

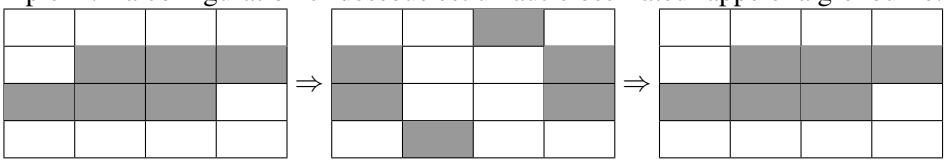
1. Si c est actuellement morte et son voisinage contient exactement 3 cellules actuellement vivantes, alors c devient vivante pour l'étape suivante.
2. Si c est actuellement vivante et son voisinage contient 2 ou 3 cellules actuellement vivantes (sans compter c elle-même), alors c reste vivante pour l'étape suivante.
3. Dans tous les autres cas, c devient (ou reste) morte.

Pour mieux comprendre le déroulement de ce jeu, vous pouvez regarder les exemples ci-dessous, où les cases grises représentent les cellules vivantes.

Exemple 1 : La configuration ci-dessous est un oscillateur, car on répète périodiquement deux configurations. Observez que la cellule au milieu reste toujours vivante (règle 2, cellule vivante avec 2 voisines vivantes), alors que toutes les autres cellules vivantes (qui ont une seule voisine vivante) deviennent mortes. Par contre, les cellules mortes qui avoisinent les trois cellules vivantes deviennent vivantes (règle 1).



Exemple 2 : La configuration ci-dessous est un autre oscillateur appelé la grenouille.



NB: le jeu de la vie est habituellement joué sur une matrice infinie, or pour cet exercice on considère que les dimensions de la matrice sont finies (n lignes et m colonnes). Par conséquent, il y aura des cellules dont le voisinage contient 5 ou 3 voisines (et pas 8), si on regarde par exemple une cellule de la première ligne. Nous allons appliquer les mêmes règles pour ces cellules.

En Haskell nous allons représenter une configuration du jeu de la vie avec une matrice (une liste de listes) dont chaque élément est une booléenne qui signifie si cette cellule est vivante (`True`) ou morte (`False`). Par exemple, les matrices ci-dessous représentent les exemples que nous avons donnés :

```
1  blinker = [
2    [ False, False, False],
3    [ True,  True , True],
4    [ False, False, False]]
5
6  frog = [
7    [ False, False, False, False],
8    [ False, True,  True,  True],
9    [ True,  True,  True,  False],
10   [ False, False, False, False]]
```

Écrire les fonctions suivantes : (4 points)

1. Une fonction `getNeighborhood` qui prend comme arguments la configuration globale actuelle, ainsi que les coordonnées (i, j) d'une cellule. Cette fonction doit retourner une matrice qui contient le voisinage de la cellule (i, j), c'est-à-dire, elle doit retourner une matrice de taille 3×3 qui contient au milieu

la cellule (i, j) et autour d'elle ses voisines dans la configuration donnée. (**NB** si (i, j) est une cellule de la première ou dernière ligne ou colonne, alors la fonction retourne une matrice de taille 2×3 ou 3×2 ou 2×2, selon le cas).

2. Une fonction `nextState` qui prend comme arguments la configuration globale actuelle, ainsi que les coordonnées (i, j) d'une cellule et décide si cette cellule deviendra vivante ou morte à l'étape suivante.

3. Une fonction `oneStep` qui prend comme argument une configuration du jeu et produit la configuration suivante.

Même si vous n'arrivez pas à programmer une fonction, vous pouvez quand-même l'utiliser pour programmer les fonctions suivantes.

Exemples d'utilisation :

```
Main> getNeighborhood frog 1 1
[[False,False,False],[False,True,True],[True,True,True]]
*Main> getNeighborhood frog 3 3
[[True,False],[False,False]]
*Main> nextState frog 2 2
False
*Main> oneStep (oneStep frog) == frog
True
```

Annexe

Consignes

- Essayer de justifier vos réponses aussi bien que possible.
- Pour toutes les fonctions que vous définissez, vous devez préciser une signature de type.
- Pour les exercices de programmation, vous avez le droit de définir d'autres fonctions secondaires en plus des fonctions demandées (mais il faut aussi bien préciser leurs types).
- Vous pouvez utiliser toutes les fonctions de Haskell de la liste suivante. (Pour chaque fonction on vous rappelle aussi son type). Toute autre fonction que vous utilisez doit être implémentée.

Fonctions de base

```
(:) :: a -> [a] -> [a]
head :: [a] -> a
tail :: [a] -> [a]
last :: [a] -> a
(++) :: [a] -> [a] -> [a]
(!!) :: [a] -> Int -> a
length :: [a] -> Int
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
reverse :: [a] -> [a]
repeat :: a -> [a]
(.) :: (b -> c) -> (a -> b) -> a -> c
($) :: (a -> b) -> a -> b
map :: (a -> b) -> [a] -> [b]
zip :: [a] -> [b] -> [(a, b)]
filter :: (a -> Bool) -> [a] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]
foldl :: (b -> a -> b) -> b -> [a] -> b
foldr :: (a -> b -> b) -> b -> [a] -> b
dropWhile :: (a -> Bool) -> [a] -> [a]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
fst :: (a, b) -> a
snd :: (a, b) -> b
(^) :: (Num a, Integral b) => a -> b -> a
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
abs :: Num a => a -> a
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
(/) :: Fractional a => a -> a -> a
div :: Integral a => a -> a -> a
mod :: Integral a => a -> a -> a
max :: Ord a => a -> a -> a
min :: Ord a => a -> a -> a
(<) :: Ord a => a -> a -> Bool
(<=) :: Ord a => a -> a -> Bool
id :: a -> a
```