Programmation Fonctionnelle L2 – Examen Partiel 16/3/2021

1 Types

Considérer le programme suivant.

```
1  f1 'a' = []
2  f1 _ = "c"
3  f2 (x:xs) = tail xs
4  f3 x
5   | head x == "b" = "a"
6  f4 x y = x (x y)
7  f5 x = if head x == 'a' then "a"
8  f6 x y = map (x y) y
9  f7 x y = filter (/=[y]) x
10 f8 = zipWith (:)
```

- 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)

Solution :

```
1  f1 'a' = []
2  f1 _ = "c"
3  f1 :: Char -> [Char]
4  f2 (x:xs) = tail xs
5  f2 :: [a] -> [a]
6  f3 x
7   | head x == "b" = "a"
8  f3 :: [String] -> String
9  f4 x y = x (x y)
10 f4 :: (a->a)->a->a
11 --f5 x = if head x == 'a' then "a"
12 -- Erreur, manque le else !
13 f6 x y = map (x y) y
14 f6 :: ([a]->a->b) -> [a] -> [b]
15 f7 x y = filter (/=[y]) x
16 f7 :: Eq a => [[a]] -> a -> [[a]]
17 f8 = zipWith (:)
18 f8 :: [a] -> [[a]] -> [[a]]
```

Barème : $0,5$ points par fonction (pour le bon type ou pour détection d'erreur de type)

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

Solution :

```
1 g1 x = x:"a"
2 g2 x = x>x
3 g3 x [] = tail x
4 g4 = (.)
```

Barème : $0,5$ points par type. Évidemment il y a d'autres solutions possibles, vous pouvez les vérifier avec ghci.

3 Moyenne Glissante

Le but de cet exercice est d'écrire une fonction qui calcule la moyenne glissante d'une liste donnée. Pour rappel, la moyenne glissante est une moyenne qui au lieu d'être calculée sur l'ensemble des n valeurs d'un échantillonnage, est calculée tour à tour sur chaque sous-ensemble de f valeurs consécutives ($f \leq n$). Le sous-ensemble utilisé pour calculer chaque moyenne, parfois appelé "fenêtre", "glisse" sur l'ensemble des données.

Donc, par exemple, si on nous donne la liste $[7, 21, 32, 49, 51]$ et on nous demande de calculer la moyenne glissante avec une fenêtre de taille 3, on considère toutes les sous-listes consécutives de 3 éléments (donc les listes $[7, 21, 32], [21, 32, 49], [32, 49, 51]$) et on retourne une liste qui contient la moyenne de chaque telle sous-liste fenêtre (donc on retourne $[20, 34, 44]$).

On vous demande d'écrire deux fonctions :

1. Une fonction `allWindows` qui prend comme argument une liste `xs` et un entier `f` et retourne une liste de listes qui contient toutes les sous-listes consécutives de `xs` de taille `f`. Quel est le type de cette fonction ?

2. Une fonction `moyenneG :: [Int] -> Int -> [Int]` qui calcule la moyenne glissante d'une liste de `Int` pour une taille de fenêtre donnée. Puisque cette fonction est implémentée spécifiquement pour les `Int`, pensez à utiliser `div` pour faire des divisions éventuelles.

Exemples d'utilisation:

```
*Main> allWindows [1..8] 4
[[1,2,3,4],[2,3,4,5],[3,4,5,6],[4,5,6,7],[5,6,7,8]]
*Main> allWindows "SoMuchFun!" 5
["SoMuc","oMuch","MuchF","uchFu","chFun","hFun!"]
*Main> moyenG [1..10] 5
[3,4,5,6,7,8]
*Main> moyenG [1,3..20] 4
[4,6,8,10,12,14,16]
```

(3 points)

Solution :

```
1 moyenG :: [Int] -> Int -> [Int]
2 moyenG xs f=map (`div` f) $ map sum $ allWindows xs f
3
4 allWindows :: [a] -> Int -> [[a]]
5 allWindows xs f = map (\i -> take f (drop i xs)) [0..length xs-f]
```

4 Sous-Matrices

Écrire une fonction `sousMatrice` qui prend comme arguments deux `Int i, j` et une matrice `mss` donnée en forme de liste de lignes. Vous pouvez supposer que `mss` correspond à une matrice correcte, c'est-à-dire, que `mss` n'est pas vide et que toutes les listes contenues dans `mss` sont non-vides et ont la même taille. L'objectif est que la fonction `sousMatrice` retourne la matrice `mss` en ayant supprimé la ligne numéro `i` et la colonne numéro `j` (donc, si `mss` était une matrice de n lignes et m colonnes, votre fonction doit retourner une matrice de $n-1$ lignes et $m-1$ colonnes). Comme d'habitude, la numérotation commence à 0 (donc si `i=0` on supprime la première ligne de la matrice). Si `i` dépasse le nombre de lignes ou `j` dépasse le nombre de colonnes de la matrice, votre fonction doit retourner une liste vide.

Exemples d'utilisation :

```
*Main> sousMatrice 0 2 [ [0,1,2], [3,4,5], [6,7,8] ]
[[3,4],[6,7]]
*Main> sousMatrice 1 1 [ [0,1,2], [3,4,5], [6,7,8] ]
[[0,2],[6,8]]
*Main> sousMatrice 3 0 [ [0,1,2], [3,4,5], [6,7,8] ]
[]
```

Dans les exemples ci-dessus, pour la matrice $\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$ si on supprime la ligne 0 et la colonne 2 on obtient $\begin{pmatrix} 3 & 4 \\ 6 & 7 \end{pmatrix}$. La ligne 3 n'existe pas (les lignes sont numérotées $0, 1, 2$), donc la dernière évaluation retourne une liste vide.

Indice : Vous trouverez utile d'écrire une fonction auxiliaire qui étant donné une liste et un entier `i` supprime l'élément qui se trouve dans la position `i` et retourne le reste de la liste. (3 points)

Solution :

```
1 dropElem :: Int -> [a] -> [a]
2 dropElem i xs = take i xs ++ drop (i+1) xs
3
4 sousMatrice :: Int -> Int -> [[ a ]] -> [[ a ]]
5 sousMatrice i j m | i>=length m = []
6 sousMatrice i j m | j>=length (head m) = []
7 sousMatrice i j m = map (dropElem j) (dropElem i m)
```

5 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 [ (x,y) | x<-[2..3], y<-[x..4] ]
2 foldl (++) "!" ["IL","OVE","HAS","KELL"]
3 foldr (\y xs -> (y+head xs) : xs) [0] [5,7,1,2]
4 map (head.tail) ["abc","def","gh"]
5 zipWith ($) (map (*) [2..5]) [3..7]
```

Solution :

```
[ (x,y) | x<-[2..3], y<-[x..4] ]
[(2,2),(2,3),(2,4),(3,3),(3,4)]

foldl (++) "!" ["IL","OVE","HAS","KELL"]
"!ILOVEHASKELL"
```

```
foldr (\y xs -> (y+head xs) : xs) [0] [5,7,1,2]
[15,10,3,2,0]

map (head.tail) ["abc","def","gh"]
"beh"

zipWith ($) (map (*) [2..5]) [3..7]
[6,12,20,30]
```

Barème : 1 point par question

6 Probabilités

Pour cet exercice vous devez donner un programme de Haskell qui peut manipuler des distributions de probabilité discrètes. On utilise la déclaration suivante :

```
type Prob = [(Double,Double)]
```

L'idée est qu'on va représenter une distribution de probabilité par une liste de paires de `Double`. Chaque paire `(x,p)` signifie que la probabilité d'obtenir la valeur `x` est égale à `p`. Par exemple :

```
coin = [(0.0,0.5),(1.0,0.5)]
bernoulli p = [(0.0,1-p), (1.0,p)]
dice = [(1.0,1/6), (2.0,1/6), (3.0,1/6), (4.0,1/6), (5.0,1/6), (6.0,1/6) ]
```

Dans l'exemple ci-dessus, la variable `coin` correspond à une variable aléatoire qui a 50% chance d'être 0 et 50% chance d'être 1. Plus généralement, `bernoulli p` donne une variable qui a probabilité `p` d'être 1 et `1-p` d'être 0. La variable `dice` correspond à un dé qui a probabilité $\frac{1}{6}$ de donner chacune des valeurs $1, 2, \ldots, 6$.

On constate qu'il existe des listes de type `[(Double,Double)]` qui ne correspondent pas à des distributions de probabilité. Par exemple, la liste `[(1.0, 0.7), (2.0, 0.8)]` n'est pas une distribution valide (car la somme des probabilités est > 1), et la liste `[(1.0, 0.2), (2.0, 0.5), (1.0, 0.3)]` n'est pas valide car on donne deux probabilités différentes pour la valeur `1.0`. On dit qu'une liste `[(Double,Double)]` correspond à une distribution valide si la somme de toutes les probabilités est égale à 1 et on a pour chaque valeur possible `x` un seul élément `(x,p)` dans la liste.

On vous demande de programmer (au moins) les fonctions suivantes. Dans tous les cas, vous pouvez supposer que la distribution donnée est valide et que la taille de toutes les listes est finie.

1. `esperance` qui étant donné une distribution de probabilité calcule son espérance. Pour rappel, pour une variable aléatoire discrète l'espérance est définie comme suit : $\mathbb{E}[X] = \sum_{x \in Val(X)} x \cdot Pr[X = x]$, où $Val(X)$ est l'ensemble de valeurs possibles de X.

2. `variance` qui étant donné une distribution calcule sa variance, en utilisant la formule suivante qui définit la variance : $\mathbb{V}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2]$.

3. `addition` qui prend comme arguments deux distributions de probabilité P_1, P_2 et retourne la distribution de la variable aléatoire $Y = X_1 + X_2$, où X_1, X_2 sont deux variables indépendantes qui suivent les distributions P_1, P_2 respectivement.

Exemples d'utilisation :

```
*Main> esperance dice   -- NB : (1+2+3+4+5+6)/6
3.5
*Main> variance dice == 35/12
True
```

```
*Main> variance (bernoulli 0.4)   -- NB : p*(1-p)
0.24
*Main> addition coin coin
[(0.0,0.25),(1.0,0.5),(2.0,0.25)]
*Main> addition coin (addition coin coin) -- Proba : (1/8, 3/8, 3/8, 1/8)
[(0.0,0.125),(1.0,0.375),(2.0,0.375),(3.0,0.125)]
*Main> esperance (addition dice dice)
7.0
*Main> variance  (addition dice dice) -- NB : = 2*Var[dice], car indep.
5.833333333333334
*Main>variance (addition (bernoulli 0.25) (bernoulli 0.4)) --0.25*0.75+0.4*0.6
0.4275
```

(4 points)

Solution :

```haskell
type Prob = [(Double,Double)]

dice = zip [1.0..6] (repeat (1/6))
coin = [(0.0,0.5),(1.0,0.5)]
bernoulli p = [(0.0,1-p), (1.0,p)]

esperance :: Prob -> Double
esperance xs = sum $ map (\(x,y)->x*y) xs

variance :: Prob -> Double
variance xs = esperance $ map (\ (x,y) -> ((x-xm)*(x-xm),y) ) xs
 where xm = esperance xs

clean :: Prob -> Prob
clean [] = []
clean ((x,p):xs) = (x,p2) : clean (filter (\ (y,_) -> y/=x) xs)
 where p2 = p + sum [ p' | (y,p')<-xs, y==x ]

addition :: Prob -> Prob -> Prob
addition xs ys = clean [ (x+y,p1*p2)  | (x,p1)<-xs, (y,p2)<-ys ]
```

Barème : 2 points pour addition, 1 point pour esperance, 1 point pour variance

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
(==) :: 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
```

Solution :