Programmation Fonctionnelle L2 – Examen Partiel 17/3/2022

1 Types

Considérer le programme suivant.

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

- 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]
2 Eq a => a -> Bool
3 [a] -> (a->b) -> [b]
4 (a -> b) -> [a] -> [(a,b)]
```

3 Nombres de Hamming

Un entier positif x est un **nombre de Hamming** si $x = 2^a 3^b 5^c$, où a, b, c sont des entiers positifs. Par exemple $24 = 2^3 3^1 5^0$ et $60 = 2^2 3^2 5^1$ sont des nombres de Hamming, alors que 14 n'est pas un nombre de Hamming.

Écrire une fonction `firstHamming` qui prend comme argument un entier `n` et renvoie les `n` premiers nombres de Hamming.

Exemple d'utilisation:

```
*Main> firstHamming 25
[1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25,27,30,32,36,40,45,48,50,54]
```

(3 points)

4 Flip

La fonction `flip` de Haskell a comme type `flip::(a -> b -> c) -> b -> a -> c`. Elle prend comme argument une fonction `f` de deux arguments et inverse le rôle de ces deux arguments. Par exemple :

```
*Main> flip (:) [1,2,3] 4
[4,1,2,3]
*Main> flip (-) 5 3
-2
```

Donner une implémentation de la fonction `flip`. (2 points)

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)
NB: la fonction `flip` est celle de l'exercice 4.

```
1 foldl (flip (:)) "abc" "def"
2 foldr ($) 5 [(+3),(*2),(9-)]
3 map (length.tail) ["hello","my","name","is","Haskell"]
4 [ x y y | x<-[(+),(*),(-)], y<-[1..3] ]
5 zipWith (\x y -> if y==0 then Nothing else Just $ x`div`y) [3..7] [-2..2]
```

6 Matchings

Supposons qu'on a deux listes `xs::[a]` et `ys::[b]` **de la même taille**. Un couplage (matching) de ces deux listes est une liste `zs` de type `[(a,b)]` telle que chaque élément `x` de `xs` et chaque élément `y` de `ys` paraît exactement une fois dans un tuple de `zs`.

Exemple : Disons que trois amis, John, Paul, et George, veulent former un groupe musical[1]. On a donc
`xs = ["John", "Paul", "George"]`.

Leurs instruments sont
`ys = ["Guitar", "Piano", "Drums"]`.

Un couplage entre ces deux listes est une affectation qui nous dit quel membre du groupe va jouer quel instrument. Par exemple
`zs1 = [ ("John","Piano"), ("Paul","Guitar"), ("George","Drums")]`
est un couplage valide. Par contre,
`zs2 = [ ("John","Piano"), ("Paul","Guitar"), ("John","Drums")]`
n'est pas valide, car ça donne deux instruments à John et rien à George.

Pour les questions suivantes vous pouvez supposer que les listes `xs,ys` ont la même taille et ne contiennent pas de doublons.

1. Écrire une fonction `isMatching` qui prend comme arguments les listes `xs,ys,zs` et retourne `True` si `zs` est un couplage valide de `xs,ys`.

2. Écrire une fonction `allMatchings` qui prend comme arguments les listes `xs,ys` et retourne une liste `zss` qui contient tous les couplage valides de `xs,ys` (dans n'importe quel ordre).

3. Écrire une fonction `allMatchingsConstrained` qui a le même comportement que `Matchings` sauf qu'elle prend un troisième argument `cs`. La liste `cs` contient des affectations interdites. Par exemple, si John ne veut pas jouer la guitare et Paul ne veut pas jouer la batterie on mettrait

   `cs= [("John","Guitar"),("Paul","Drums")]`

   Votre fonction doit retourner une liste de tous les couplages qui évitent les affectations interdites par `cs`.

Exemples d'utilisation :

```
*Main> isMatching [1,2,3] "abc" (zip [3,2,1] "cab")
True
*Main> isMatching [1,2,3] "abc" (zip [3,2,2] "abc")
False
*Main> allMatchings [1,2] "ab"
[[(1,'a'),(2,'b')],[(1,'b'),(2,'a')]]
*Main> allMatchings [1,2,3] "abc"
[[(1,'a'),(2,'b'),(3,'c')],[(1,'a'),(2,'c'),(3,'b')],[(1,'b'),(2,'a'),(3,'c')],
 [(1,'b'),(2,'c'),(3,'a')],[(1,'c'),(2,'a'),(3,'b')],[(1,'c'),(2,'b'),(3,'a')]]
*Main> allMatchingsConstrained  [1,2,3] "abc" [(1,'a'),(1,'c')]
[[(1,'b'),(2,'a'),(3,'c')],[(1,'b'),(2,'c'),(3,'a')]]
*Main> allMatchingsConstrained  [1,2,3] "abc" [(1,'a'),(2,'b')]
[[(1,'b'),(2,'a'),(3,'c')],[(1,'b'),(2,'c'),(3,'a')],[(1,'c'),(2,'a'),(3,'b')]]
```

(5 points)

[1] Ringo va les rejoindre plus tard. ;-)

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
```