TD 10 : IO Monad

1 Exemples de l'utilisation de l'IO Monad

1.1 Médian

Écrire un programme qui demande à l'utilisateur d'entrer quelques entiers au clavier dans une seule ligne. Une fois que l'utilisateur presse Enter, votre programme doit afficher le médian des éléments saisis. Exemple :

```
> runhaskell median.hs
Give me a list of numbers
5 3 9 2 7 11 10
The median is :7
```

NB : Pensez à utiliser les fonctions `getLine`, `words`, `read`

Solution :

```
median :: [Int] -> Int
median [x] = x
median [x,_] = x
median xs = qsort xs !! (length xs `div` 2)
 where
  qsort [] = []
  qsort (x:xs) = qsort (filter (<x) xs) ++ [x] ++ qsort (filter (>=x) xs)

main :: IO ()
main = do
 putStrLn "Give␣me␣a␣list␣of␣numbers"
 stuff <- getLine
 let nums = map read $ words stuff
 let msg = "The␣median␣is␣:" ++ show (median nums)
 putStrLn $ msg
```

1.2 Grep

La commande `grep` de Unix permet à l'utilisateur de chercher pour un mot donné dans un fichier texte. Pour cet exercice vous devez écrire une version plus interactive de cette commande : votre programme doit d'abord demander à l'utilisateur le nom d'un fichier et un mot. Puis, votre programme doit ouvrir le fichier et afficher toutes les lignes qui contient le mot donné.

Solution :

```
import System.IO

subString :: String -> String -> Bool
subString [] _ = True
subString _ [] = False
subString xs ys = or [ prefix xs (drop i ys) | i<-[0..length ys-1] ]
```

```
prefix :: String -> String -> Bool
prefix [] _ = True
prefix _ [] = False
prefix (x:xs) (y:ys)
 | x/=y = False
 | otherwise = prefix xs ys

filterPrint :: String -> [String] -> IO ()
filterPrint w [] = return ()
filterPrint w (l:ls)
 | subString w l =  do
                     putStrLn l
                     filterPrint w ls
 | otherwise = filterPrint w ls

main :: IO ()
main = do
 putStrLn "File␣name␣:"
 fn <- getLine
 hin <- openFile fn ReadMode
 putStrLn "Word␣you␣are␣looking␣for␣:"
 w <- getLine
 stuff <- hGetContents hin
 filterPrint w $ lines stuff
 putStrLn "Thanks"
 hClose hin
```

2 Return, Bind, Join

Les Monads sont une notion, très importante pour Haskell, qui provient du domaine mathématique de la théorie des catégories. Dans cet exercice on regarde de plus près les fondements de la définition des Monads en Haskell. On vous rappelle que la définition minimale est :

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  {-# MINIMAL (>>=) #-}
```

On a expliqué qu'une façon de voir les Monads (et plus largement les Functors) est comme des containers : une liste `[a]` est un container qui contient une collection ordonée d'objets de type `a` ; un `Maybe a` est un calcul qui peut arriver à un objet de type `a` ou à rien ; `IO a` est un "container" qui contient le résultat (de type `a`) d'une interaction avec le monde extérieur.

Dans ce contexte, ce n'est pas trop compliqué de comprendre le rôle de la fonction `return`. Cette fonction prend un élément de type `a` et le met dans le plus simple container du monad donné qui pourrait le contenir. D'ailleurs, on n'utilise le nom `return` que pour des raisons historiques. En réalité, cette fonction est normalement équivalente à la fonction `pure` de la classe `Applicative`.

— Quelle est la définition de `return` pour les Monads `[]` et `Maybe` ?
Solution : Pour les listes on a `return x = [x]`. Pour `Maybe` on a `return x = Just x`.

On regarde maintenant la fonction `(>>=)` (bind). Son type est :

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Pour rappel `x>>=f` veut dire que l'opérateur >>= fait trois choses :

1. Elle extrait les valeurs contenues dans (le Monad) x
2. Elle applique `f` sur chaque telle valeur. Chaque application retourne une collection de résultats de type `b` (un Monad de `b`).
3. Elle récupère toutes les collections les résultats et elle les fusionne dans un Monad qui a le même type que celui retourné par `f`.

Exemple : que donne la ligne suivante ? `[5,7] >>= \x -> [x..2*x]`

Solution : `[5,6,7,8,9,10,7,8,9,10,11,12,13,14]`

On observe maintenant que parmi les trois tâches accomplies par `(>>=)`, on peut déjà simuler les deux premiers en se basant sur le fait que les Monads sont des Functors ! Pour rappel, la fonction `fmap` nous permet d'extraire une par une les valeurs contenues dans un Functor (donc aussi dans un Monad), et d'appliquer une fonction `f` à chacune.

Que fait la ligne suivant ? `fmap (\x -> [x..2*x]) [5,7]`

Solution : `[[5,6,7,8,9,10],[7,8,9,10,11,12,13,14]]`

On comprend donc que la différence principale entre les Monads et les Functors est que les Monads donnent une manière de **combiner** les résultats contenus dans plusieurs monads pour obtenir un seul. Si on va un peu plus loin, on constate qu'en fait, c'est tout à fait possible de définir les Monads sur le seul principe qu'un Monad est un Functor avec `return` et une fonction qui peut combiner plusieurs Monads pour produire un seul. Cette fonction est appelée `join`. En Haskell, vous pouvez trouver une implémentation de cette fonction dans le module `Control.Monad`. On a : `join :: Monad m => m (m a) -> m a`.

Que font les lignes suivantes ?

```
> join [[1,2],[3,4],[5]]
> join [1,2,3]
> join (Just (Just 3))
> join (Just Nothing)
> join [Just 2, Just 3]
```

Solution :

```
> join [[1,2],[3,4],[5]]
[1,2,3,4,5]
> join [1,2,3]
Erreur ! [1,2,3] n'a pas type m (m a), avec Monad m...
> join (Just (Just 3))
Just 3
> join (Just Nothing)
Nothing
> join [Just 2, Just 3]
Erreur ! Dans le type de join, on utilise deux fois le
meme Monad, donc on ne peut pas melanger Maybe et []
```

Pour compléter cet exercice et prouver que la définition des Monads de Haskell (avec `(>>=)`) est équivalent avec celle qui se base sur `join` donner les implémentations suivantes :

1. Programmer la fonction `join` en utilisant `(>>=)`
2. Programmer la fonction `(>>=)` en utilisant `join`

Solution :

On a les deux définitions suivantes :

```
join x = x >>= id
x >>= f = join (fmap f x)
```

Explication : pour définir `join` on se rappelle que la fonction `(>>=)` fait, comme on l'a dit, trois tâches, dont la troisième correspond à la tache de `join` (fusionner les monads retournés par les applications de `f`). Il

suffit donc d'utiliser une fonction `f` qui trivialise les deux premières tâches. Une telle fonction est la fonction `id` qui retourne son argument. Faisons une vérification de type : pour que la fonction `join`, telle qu'on l'a programmée, soit bien typée il faut que (i) l'application de `(>>=)` soit bien typée (ii) son type de retour soit le type de retour de `join`. Selon le type de `(>>=) :: Monad m => m a -> (a -> m b) -> m b` il faut que `x` soit de type `m a`, `id` soit de type `a -> m b`. Donc, en prenant en compte le type de `id`, on a `a=== m b`. Donc, `x::m (m b)`, et `join :: m (m b)-> m b`, comme attendu.

Pour la définition inverse, on se rappelle que les deux premières tâches de `(>>=)` sont aussi accomplies par `fmap`. On utilise donc `join` pour accomplir la troisième. Une vérification de type similaire vous convaincra que cette définition est bien typée.