

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`

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 contiennent le mot donné.

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 ordonnée 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` ?

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]`

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 suivante? `fmap (\x -> [x..2*x]) [5, 7]`

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]
```

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`