Lambda calcul et programmation fonctionnelle

TP 2

Exercice 1

La dernière fois, nous avons écrit la fonction fibo. Seulement, nous étions obligés de fournir le paramètre d'appel directement dans le code source. Pour ne plus appeler fibo 30 mais fibo 40, il fallait changer le code source et le recompiler. Ce n'est pas acceptable.

On souhaite plutôt demander à l'utilisateur d'entrer une valeur au clavier, et ensuite, d'appliquer la fonction fibo sur cette valeur.

En Haskell, ce n'est pas si simple que ça en a l'air ! Les entrées-sorties ne sont pas des processus purement fonctionnels. Mais heureusement, il existe évidemment une façon de le faire.

Votre programme principal était :

```
main :: IO ()
main = print (fibo 30)
```

Nous allons maintenant utiliser l'affectation pour lire une valeur au clavier et l'affecter à une variable. Attention : one ne peut faire ça que dans les fonctions qui utilisent IO. N'essayez pas de mettre un getLine ou une flèche d'affectation ailleurs !

```
main :: IO ()
main = do
      putStrLn "Entrez une valeur numérique"
      n <- getLine
      print (fibo n)
```

Compilez le programme. Vous aurez une erreur de type : getLine renvoie un String mais fibo attend un entier. On doit donc convertir la chaîne de caractère en entier. Pour cela, on utilisera la fonction read :

```
main :: IO ()
main = do
      putStrLn "Entrez une valeur numérique"
      saisie <- getLine -- saisit une chaîne au clavier
      let n = (read saisie) in --  convertit la saisie en Int
            print (fibo n)
```

Comment read sait-il qu'il faut convertir en entier plutôt qu'en caractère ou qu'en booléen ou qu'en liste ou autre ? Il fait de l'inférence de type (il déduit le type lui-même) : vu qu'on utilise n comme paramètre à fibo, et que fibo attend un entier, ça veut dire que n est forcément un entier.

Attention ! On n'utilise la flèche <- que quand on reçoit une entrée/sortie. Convertir saisie en entier n'est pas une entrée-sortie : on n'utilise donc pas la flèche !

Compilez votre programme, et essayez d'entrer plusieurs valeurs différentes. Que se passe-t-il si vous entrez « toto » ?

Exercice 2

Écrivez un programme qui demande à l'utilisateur de saisir une liste d'entiers et qui affiche cette liste.

Écrivez un programme qui demande à l'utilisateur de saisir une liste d'entiers et affiche la longueur de cette liste. Attention : la fonction length qui est prédéfinie en Haskell ne fonctionne pas comme vous l'attendez (on verra pourquoi plus tard en cours). Vous devez importer une autre fonction length disponible dans un autre module. Ajoutez les lignes suivantes tout en haut de votre fichier :

```
import Prelude hiding (length)
import Data.List hiding (length)
length = genericLength
```

(Oui, nous vous cachons des choses...)

Exercice 3

Créez un nouveau fichier appelé moyennes.hs et recopiez-y les imports vus ci-dessus.

Recopiez la fonction moyenne ci-dessous qui prend une liste de Double en paramètres et qui renvoie la valeur moyenne. Si la liste est vide, on renverra la valeur 0.

```
moyenne :: [Double] -> Double
moyenne [] = 0
moyenne l  = (sum l) / (length l)
```

Écrivez ensuite la fonction main qui demande à l'utilisateur une liste de Double, et qui affiche à l'écran la moyenne des valeurs de cette liste.

Exercice 4

Créez un nouveau fichier appelé minimal.hs.

Écrivez la fonction minRec qui prend une liste de Int et renvoie la valeur la plus petite. Si la liste est vide, on renverra 0. Utilisez une écriture récursive.

Écrivez la fonction minFoldr qui fait la même chose, mais cette fois en utilisant la fonction foldr.

Écrivez ensuite la fonction main qui demande à l'utilisateur une liste de Int, et qui affiche à l'écran le minimum des valeurs de cette liste.

Exercice 5

Créez un nouveau fichier appelé stayPositive.hs.

Écrivez la fonction positivesRec qui prend une liste de Int et renvoie la même liste mais sans les valeurs négatives ou nulles. Par exemple, pour la liste [3, 0, -1, 2, -5], on renverra [3, 2].

Écrivez la fonction positivesFilter qui , cette fois, utilise la fonction filter.

Écrivez ensuite la fonction main qui demande à l'utilisateur une liste de Int, et qui affiche à l'écran les valeurs positives de cette liste.

Exercice 6

Créez un nouveau fichier upcase.hs.

Nous allons utiliser la fonction toUpper définie dans le module Data.Char pour mettre des mots en majuscules. Tout d'abord, importez le module Data.Char au début de votre programme :

```
import Data.Char
```

La fonction toUpper a la signature suivante :

```
toUpper :: Char -> Char
```

Elle va convertir un caractère en son équivalent en majuscule.

Écrivez une fonction toUpperString qui a la signature suivante :

```
toUpperString :: [Char] -> [Char]
```

Rappel : une chaîne de caractères est équivalente à une liste de Char.

Comme son nom l'indique, cette fonction va convertir une chaîne de caractères en majuscules. En d'autres termes, nous avons une liste de caractères, et nous souhaitons renvoyer une liste de caractères de même taille, mais transformés. Quelle fonction d'ordre supérieur devez-vous utiliser ?

Écrivez la fonction main qui demande à l'utilisateur de saisir une chaîne de caractères et l'affiche en majuscules.

Écrivez une fonction toUpperStrings qui a la signature suivante :

```
toUpperStrings :: [[Char]] -> [[Char]]
```

Cette fonction convertit toutes les chaînes d'une liste de chaînes en majuscules.

Écrivez une fonction removeEmpty qui supprimes les chaînes vides d'une liste :

```
removeEmpty :: [[Char]] -> [[Char]]
```

Écrivez une fonction toUpperNonEmpty qui supprime les chaînes vides d'une liste et les convertit en majuscules.

```
ToUpperNonEmpty :: [[Char]] -> [[Char]]
```

Par exemple, on auraient

```
toUpperNonEmpty ["", "Toto", "Bonjour c’est moi", "", "12345"]
```

qui vaudra

```
["TOTO", "BONJOUR C’EST MOI", "12345"]
```

Modifiez votre fonction main pour qu’elle demande à l’utilisateur d’entrer plusieurs chaînes de caractères et qu’elle affiche à l’écran les chaînes non vides en majuscules.