Fiche de TD no. 3

Fonctions récursives

Exercice 1. Proposez (sans regarder la bibliothèque `Prelude`) une définition récursive pour chacune des fonctions suivantes :

`and :: [Bool] -> Bool`	décider si tous les valeurs logiques d'une liste sont vrais,
`concat :: [[a]] -> [a]`	concaténer une liste de listes,
`replicate :: Int -> a -> [a]`	produire une liste avec n éléments identiques,
`(!!) :: [a] -> Int -> a`	sélectionner le n-ième élément d'une liste,
`elem :: Eq a => a -> [a] -> Bool`	décider si un valeur est un élément d'une liste,
`maximum :: Ord a => [a] -> a`	le maximum d'une liste non vide.

Exercice 2.

1. Définissez, par induction, une fonction qui prend an argument une chaîne de caractères et substitue chaque occurrence d'un caractère c par un chaîne de caractères $f(c)$. La fonction f sera aussi passée en paramètre à la fonction définie.

   Utilisez la méthodologie proposez par Hutton, jusqu'à obtenir un typage polymorphe de la fonction définie.
2. Pouvez vous imaginer une autre définition de la même fonction, qui n'utilise pas la récursion, mais utilise à la place des fonctions connue du `Prelude` ?
3. Pouvez vous donner un exemple d'utilisation non triviale (intéressante) de cette fonction ?

Exercice 3.

1. Donnez une définition récursive de l'opération binaire de concaténation de listes
   `(++) :: [a] -> [a] -> [a]`.
2. Argumentez que, pour n'importe quelle liste `xs`, on a

   `xs ++ [] == xs` et `[] ++ xs == xs`.

   Utilisez l'induction sur la longueur de la liste `xs`.
3. Argumentez que, pour n'importe quel triplet de liste `xs`, `ys` et `zs`, on a

   `(xs ++ ys) ++ zs == xs ++ (ys ++zs)`.

   Utilisez l'induction sur la longueur de la liste `xs`.

Exercice 4. Considérez le script suivant :

```
evenList,oddList :: [a] -> Bool
evenList [] = True
evenList (x:xs) = oddList xs
oddList [] = False
oddList (x:xs) = evenList xs

f xs
  | evenList xs = map (+1) xs
  | oddList xs = f (tail xs)
```

Argumentez (avec précision, donc démontrez) que la fonction `f` y définie est totale. C'est à dire, `f` retourne toujours un valeur, pour n'importe quelle liste passée en paramètre.

Fonctions d'ordre supérieur

Exercice 5. Redéfinissez les fonctions `and`, `concat`, `elem` (voir l'exercice 1) via la fonction d'ordre supérieur `foldr`. Rappel :

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Exercice 6. Voici les types de sept fonctions (dont cinq définies dans `Prelude.hs`) :

```
iterate :: (a -> a) -> a -> [a]
splitAt :: Int -> ([a] -> ([a], [a]))
span :: (a -> Bool) -> ([a] -> ([a], [a]))
eval :: a -> ((a -> b) -> b) -- ... = \x -> \y -> y x
constid :: a -> (a -> (b -> b)) -- ... = \_ -> \_ -> \z -> z
flip :: (a -> b -> c) -> (b -> (a -> c))
until :: (a -> Bool) -> ((a -> a) -> (a -> a))
```

Quelles sont les fonctions d'ordre supérieur ? Argumentez votre réponse.

Exercice 7. Considérez les définitions `Haskell` suivantes :

```
expr1 = map (\x -> x == 0 || x == 1) [0..4]
expr2 = filter (\(_,y) -> even y) (zip [0..4] (tail [0..4]))
expr3 = filter (odd . head) (map (\x -> [x]) [0..4])
expr4 = foldr (\x y -> x == y) True (map even [0..4])
```

1. Typez et évaluez ces quatre expressions.
2. Listez et rappelez le type de toutes les fonctions d'ordre supérieur utilisées dans ces quatre définitions.

`IO`, »= et notation `do`

Exercice 8. Considérez le code `Haskell` suivant :

```
import Data.Char(isAlpha,toUpper)
action =
    getChar >>= \c ->
    putStrLn ['\n',toUpper c] >>= \_ ->
    return (isAlpha c)
```

1. Quelle est, selon vous, la signification de la première ligne ?
2. Expliquez avec précision ce qui se passe si on demande à l'interprète d'évaluer `action`.
3. Écrivez le type de `action`.
4. Réécrivez ce code en utilisant la notation `do`.

Exercice 9. Le programme suivant se sert de l'opérateur `do` pour enchaîner les actions.

```
main = do
  putStr "Quel est votre nom ? \n"
  nom <- getLine
  putStr "Quel est votre age ? \n"
  age <- getLine
  print (nom,age)
```

Réécrivez ce programme en utilisant l'opérateur »= à la place de `do`.