Fiche de TD-TP 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.

Exercice 2. Définissez une fonction récursive

```
fusionner :: [Int] -> [Int] -> [Int]
```

qui fusionne deux listes triées pour produire une seule liste triée. Définissez ensuite une fonction récursive

```
msort :: [Int] -> [Int]
```

qui implémente le tri par fusion (merge sort en anglais). Cet algorithme de tri peut se spécifier par les deux règles suivantes :
– les listes de longueur ≤ 1 sont déjà triées ;
– on peut trier les autres listes peuvent en les découpant en deux morceaux, en triant ces deux morceaux, et en fusionnant les listes résultantes.

Fonctions d'ordre supérieur

Exercice 3.

1. Rappelez ce que veut dire qu'une fonction est en forme curryfiée.
2. Proposez un exemple d'une fonction en forme curryfiée, et d'une autre qui n'est pas en forme curryfiée.
3. Définissez la fonction

   ```
   curry :: ((a,b) -> c) -> a -> b -> c
   ```

   qui prend en paramètre une fonction dont le paramètre est un couple, et transforme cette fonction en sa forme curryfiée.

Exercice 4. Utilisez les fonctions `map` et `filter` pour donner une expression équivalente à la suivante :

```
[f x | x <- xs, p x]
```

Exercice 5. Redéfinissez `map f` et `filter p` à l'aide de `foldr`.

Exercice 6. Proposez une définition de la fonction `foldl` (à lire fold left), où l'on a :

```
foldl x1:x1:...:xn:[] (@) v = (...(x1@x2)...@xn)@v
```

Exercice 7. Considérez le script Haskell suivant :

```
1 type Pixel = Int
2 type Ligne = [Pixel]
3 type Image = [Ligne]
4 type Effet = (Pixel,[Pixel]) -> Pixel
5 type Point = (Int,Int)
```

```
 7 taille_x,taille_y :: Image -> Int
 8 taille_x = length . head
 9 taille_y = length

11 sousliste :: Int -> Int -> [a] -> [a]
12 sousliste i j xs = drop (i-1) (take j xs)

14 voisinage :: Image -> Point -> (Pixel,[Pixel])
15 voisinage img (x,y) = (p,ps)
16     where
17       p = (img !! (y-1)) !! (x-1)
18       ps = concat (map (sousliste (x-1) (x+1)) (sousliste (y-1) (y+1) img))

20 appliquerEffet :: Effet -> Image -> Image
21 appliquerEffet effet image =
22     let
23         (xmax,ymax) = (taille_x image,taille_y image)
24         pts = [ [ (x,y) | x <- [1..xmax] ]  |  y <- [1..ymax] ]
25     in
26       map (map (\p -> effet (voisinage image p))) pts
```

1. Marquez toutes les constructions syntaxiques que vous ne connaissez encore pas.
2. Marquez toutes les fonctions d'ordre supérieure (définies ou utilisées) dans le script. Donnez leur type.
3. Trouvez une expression équivalente à `\p -> effet (voisinage image p)` sans la notation λ.
4. Expliquez, ligne par ligne et puis d'un point de vue globale, le fonctionnement du code et ce qui est achevé par ce script.

Input-output

Exercice 8.

1. Rappelez le type de `putStr`,`return` et `( >>= )`.
2. Expliquez ce qui est achevé par la fonction suivante :

```
afficheEtIncrementeComplexe (x,y) =
    putStr (show x ++" + i"++show y)
    >>=
    \() -> putStr "\n"
    >>=
    \() -> return (x+1,y+1)
```

   et donnez un type à cette fonction.
3. Dans le code ci-dessus, que ce fait la fonction `show` ? Pouvez vous lui donner un type ?
4. Expliquez pourquoi le code similare

```
afficheEtIncrementeComplexe (x,y) =
    putStr (show x ++" + i"++show y)
    >>=
    putStr "\n"
    >>=
    return (x+1,y+1)
```

   produit un erreur de type.

En TP

Exercice 9. Testez le script de l'exercice 7 :

1. Écrivez une fonction `showImage` qui transforme une image dans une chaîne de caractères prête à être imprimée. Par exemple, on pourrait avoir :

```
*Main> putStr (showImage [[0,1,0],[1,0,1],[0,1,0]])
 *
* *
 *
*Main>
```

2. Écrivez un effet `smooth` qui colore en noir (resp. blanc) un pixel si les deux tiers des pixels voisins sont noirs (resp. blancs).
3. Testez cet effet sur les figures suivantes :

```
image1,image2 :: Image
image1 = [
 [1,1,1,0],
 [0,0,1,1],
 [1,0,1,0],
 [0,0,0,1]]
image2 = [
 [0,1,1,0,0,1,0,0,0,1],
 [1,0,0,1,0,1,1,0,1,1],
 [1,0,0,1,0,1,0,1,0,1],
 [0,1,1,0,0,1,0,0,0,1]]
```

Exercice 10. Implémentez, en Haskell, le jeu du *nim*. Les règles de ce jeux sont comme suit :
– L'échiquier contient 5 lignes d'étoiles, chaque ligne contenant un nombre d'étoiles.
– Le jeu demarre de cette position :

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

– Deux joueurs enlèvent – alternativement – une ou plusieurs étoiles d'une seule ligne.
– Le gagnant est le joueur qui enlève la dernière(s) étoile(s) de l'échiquier.
Conseil : Représentez l'échiquier comme une liste de 5 entier qui représente les étoiles encore à enlever. Par exemple, la position initiale est représentée par `[5,4,3,2,1]`.