Fiche de TD no. 2

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. 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 3. 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 (et expressions lambda)

Exercice 4. Typez et évaluez les expressions 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])
```

Exercice 5. 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, par exemple en réduisant au maximum les parenthèses dans les expressions de type.

Exercice 6. 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.