Interrogation Écrite 1

Consignes et objectifs

Le but de cette interrogation est d'évaluer vos capacités à :
- implémenter des fonctions récursives ;
- utiliser du filtrage de motif et des itérateurs ;
- utiliser des fonctions à partir de leur documentation ;
- typer correctement (en utilisant les types les plus génériques et les classes de type).

Pour chaque implémentation, vous devez préciser le type le plus général. En cas de doute, écrivez ce que vous avez essayé.

Exercice 1 : Classification de triangles

Écrivez une fonction `typeTriangle` qui prend en argument un triplet représentant les longueurs des 3 côtés d'un triangle et qui renvoie une chaîne de caractères donnant le type du triangle : *équilatéral*, *isocèle* ou *quelconque*.

Éléments de correction

Attention à ne pas oublier les classes sur les types. Ici, il faut que les éléments, qui pourraient être entiers ou non, soient comparables par égalité.

Code

```haskell
typeTriangle :: (Num a, Eq a) => (a, a, a) -> String
typeTriangle (a, b, c) | a == c && b == c           = "Équilatéral"
                       | a == b || b == c || a == c = "Isocèle"
                       | otherwise                  = "Quelconque"
```

Exercice 2 : Utilisation des itérateurs

Utilisez les fonctions `map`, `filter`, et `sum` pour implémenter la fonction `doubleOddSum` qui calcule la somme des nombres impairs d'une liste après avoir doublé chaque élément.

Exemple

`doubleOddSum [1, 2, 3, 4, 5]` doit renvoyer 18 (soit 2 + 6 + 10).

Éléments de correction

Code

```haskell
doubleOddSum :: [Int] -> Int
doubleOddSum = sum . map (*2) . filter odd
```

Exercice 3 : Produit scalaire

La fonction `zip :: [a] -> [b] -> [(a, b)]` prend en argument deux listes et retourne la liste des paires d'éléments correspondants.

La fonction `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` généralise la fonction `zip` en appairant les paires à l'aide de la fonction donnée en premier argument.

Un *vecteur* peut être vu comme une liste de nombres, entiers ou non. En vous servant de fonctions parmi `zip`, `zipWith`, `sum`, `map`, `filter`, `tail` ou `head`, implémentez une fonction `produitScalaire`, qui renvoie le produit scalaire de deux vecteurs.

Exemple

`produitScalaire [1, 2, 3] [4, 5, 6]` renverra $(1 \times 4) + (2 \times 5) + (3 \times 8)$ soit 32.

Éléments de correction

Code

```haskell
produitScalaire :: Num a => [a] -> [a] -> a
produitScalaire v1 v2 = sum (zipWith (*) v1 v2)
```

Exercice 4 : Pour progresser, il faut (se) répéter

Q. 1 Écrivez une fonction récursive qui, étant donnée un entier *n* et un élément, répète l'élément *n* fois.

Éléments de correction

Code

```haskell
repete :: Int -> a -> [a]
repete 0 _ = []
repete n x = x : repete (n - 1) x
```

Q. 2 En vous servant de cette fonction, écrivez maintenant une fonction récursive `repeteChaque` qui prend en paramètre un entier n et une liste et qui répète chaque élément de la liste n fois.

Exemple

Code

```haskell
repeteChaque 2 [1, 2, 3] -- Résultat attendu : [1, 1, 2, 2, 3, 3]
repeteChaque 3 ["a", "b"] -- Résultat attendu : ["a", "a", "a", "b", "b",
↪  "b"]
repeteChaque 0 [1, 2, 3] -- Résultat attendu : []
```

Éléments de correction

Code

```haskell
repeteChaque :: Int -> [a] -> [a]
repeteChaque = concatMap . repete
```

Q. 3 Votre proposition est-elle récursive terminale ? Pourquoi ?

Exercice 5 : Compression d'information

Q. 1 Écrivez une fonction récursive terminale `compress` qui prend en argument une liste et élimine les éléments redondants consécutifs, en conservant le compte.

Exemple

`compress [2, 2, 2, 2, 4, 5, 5]` s'évaluera en `[(2, 4), (4, 1), (5, 2)]`
`compress ['a','a','a','a','a','a']` s'évaluera en `[('a', 6)]`

Éléments de correction

Code

```haskell
compress :: (Eq a, Num b) => [a] -> [(a, b)]
compress [] = []
compress (x:xs)  = go xs x 1
  where
    go [] x n      = [(x, n)]
    go (y:ys) x n | x /= y    = (x, n) : go ys y 1
                  | otherwise = go ys x (n + 1)
```

Q. 2 Écrivez la fonction `decompress`, qui effectue le chemin inverse : elle part d'une liste de paires et renvoie la liste décompressée.

Éléments de correction

Code

```haskell
decompress :: (Num b, Eq b) => [(a,b)] -> [a]
decompress [] = []
decompress ((a,n):l) = go a n (decompress l)
  where
    go _ 0 l = l
    go a n l = a : go a (n-1) l
```