TD1 : PF

Exercice 1 : Implémentations de l'exponentielle

Nous allons implémenter quatre versions différentes de l'exponentielle : la fonction `monExp` de type `Int -> Int -> Int` telle que `monExp a n` renvoie a^n.

Q 1.1 Version simple

En utilisant la relation entre a^n et a^{n-1} donner une implémentation récursive de `monExp`.

```
monExp :: Int -> Int -> Int.
monExp a 0 = 1
monExp a n = a * monExp a n-1
```

Q 1.2 Version récursive terminale

En utilisant une fonction intermédiaire (par exemple dans une clause `where ...`) qui manipule un accumulateur, donner une définition récursive terminale de `monExp`.

```
monExp a n = monExpAux 1 n .
  where
    monExpAux acc 0 = acc
    monExpAux acc p = monExp (acc*a) (p-1).
```

Q 1.3 Version rapide

Voici deux identités :

$$\begin{aligned} a^n &= a^{n/2} \times a^{n/2} && \text{si } n \text{ pair} \\ a^n &= a \times a^{n/2} \times a^{n/2} && \text{si } n \text{ impair} \end{aligned}$$

Utilisez-les pour écrire une dernière version de `monExp`. Pensez à utiliser des variables pour éviter les calculs redondants.

Vous pouvez utiliser la fonction `quotRem` de type `Int -> Int -> (Int,Int)` : `quotRem a b` calcule la paire `(q,r)` où `q` et `r` sont respectivement le quotient et le reste de la division euclidienne de `a` par `b`.

```
monExp a n = case R of
               0 -> R*K
               otherwise -> a*K*K
  where
    (p,R) = n `quotRem` 2
    K = monExp a p
```

Q 1.4 Avec des fonctions du `Prelude`

En utilisant les fonctions suivantes :

- `product` de type `[Int] -> Int` : `product l` renvoie le produit des éléments de `l`
- `replicate` de type `Int -> Int -> [Int]` : `replicate n a` renvoie une liste `[a, ..., a]` qui contient `n` occurrences de `a`,

donnez une dernière version de la fonction `monExp`.

```
monExp a n = product Replicate n a
  where product :: [Int] -> Int.
        product [] = 1
        product a:l = a * product l

        Replicate :: Int -> Int -> [Int]
        Replicate 0 a = []
        Replicate n a = a : replicate (n-1) a
```

Exercice 2 : Approche naïve

Nous allons commencer par implémenter un algorithme simple :
— nous allons parcourir la liste,
— pour chaque élément `a` de la liste :
  — nous allons compter le nombre `n` d'occurrences de `a` dans la suite de la liste,
  — nous allons ensuite éliminer cet élément de la suite de la liste,
  — *compter* récursivement les éléments de la liste ainsi obtenue,

---

  — pour finalement ajouter en tête de la liste associative obtenue la paire `(a,n)`.

Cette description algorithmique fait apparaître la nécessité d'implémenter deux fonctions intermédiaires :
— `compterOcc` de type `Int -> [Int] -> Int` telle que `compterOcc a l` renvoie le nombre d'occurrences de `a` dans `l`.
— `eliminer a l` qui renvoie la liste obtenue à partir de `l` en éliminant les occurrences de `a`.

Q 2.1 Implémentation
— Écrire la fonction `compterOcc`.
— Écrire la fonction `eliminer`.
— Écrire la fonction `compter`.

```
comptarOcc :: Int -> [Int] -> Int

comptarOcc _ [] = 0

comptarOcc a (x:l)
    | a == x = 1 + comptarOcc a l
    | otherwise = comptarOcc a l
```