TD 4 : AlgDem

Exercice 1 : Exponentiation rapide récursive terminale

La fonction `expAux` est utilisée pour calculer l'exponentiation rapide de façon récursive terminale.

```
expAux:: Int -> Int -> Int -> Int
expAux a b acc | b == 0          = acc
               | b `mod` 2 == 0  = expAux (a*a) (b `quot` 2) acc
               | otherwise       = expAux (a*a) (b `quot` 2) (acc*a)

expR :: Int -> Int -> Int
expR a b = expAux a b 1
```

Q 1.1 Fonction python non récursive
En suivant la méthode donnée en cours, donnez une implémentation en `python` de `expAux` qui soit non récursive.

facile, avec un acc=0 et un while True!

Q 1.2 Spécification et démonstration Voici la spécification de `expAux a b acc` :

Pré-condition `b >= 0`

Post-condition `expAux a b acc` $= acc \times a^b$.

Variant `b`.

b = 2*(b`quot`2) + b`mod`2.

Nous admettrons pour cette démonstration que `a = b * (a ‘quot‘ b) + a ‘mod‘ b`.
Démontrez que `expAux` vérifie sa spécification.

(HI) : $\forall b' < b,\ b \geq 0 \Rightarrow$ expAux a b' acc $= acc \times a^{b'}$

Cas 1 : $b = 0$

On a : expAux a b acc = acc = acc * 1 = acc * a^b ok.

Cas 2 : $b > 0 \wedge b \equiv 0 \bmod 2$.

expAux a b acc = expAux (a*a) (b`quot`2) acc.
On a : b `quot` 2 ≥ 0 et b `quot` 2 $< b$: on applique (HI)
On pose b' = b `quot` 2 :

expAux (a*a) b' acc = acc * $(a*a)^{b'}$
$= acc * a^{2b'} = acc * a^b$.

Cas 3 : $b > 0 \wedge b \equiv 1 \bmod 2$.

expAux a b acc = expAux (a*a) (b `quot` 2) (acc*a)

=> (HI)

expAux (a*a) (b `quot` 2) (acc*a) = acc*a*(a*a)$^{b'}$

= acc*a^{1}*a$^{2b'}$ = acc * a^{b}

Ok.

Q 1.3 Exponentielle

Déduisez de la question précédente que la fonction `expR a b` vérifie la spécification suivante :

Pré-condition `b >= 0`

Post-condition `expR a b` = a^b.

...

$\forall b' < b, b \geq 0 \Rightarrow$ expR a b = a^b

Cas 0 : b = 0

expR a b = expAux a b 1 = 1 = a^b ok.

Cas 1 : b > 0

expR a b = expAux a b 1 = 1 * a^b

Ok.

Exercice 2 : Insertion triée récursive terminale

Q 2.1 Insertion

La fonction `insererAux` est utilisée pour effectuer l'insertion au cours de l'algorithme de tri par insertion.

```
revAux :: [Int] -> [Int] -> [Int]
revAux [] l = l
revAux (x:l') l = revAux l' (x:l)

insererAux :: Int -> [Int] -> [Int] -> [Int]
insererAux k [] acc = revAux acc [k]
insererAux k (n:l) acc | k <= n    = revAux acc (k:n:l)
                       | otherwise = insererAux k l (n:acc)
```

Cette fonction utilise la fonction `revAux` étudiée en cours et dont nous avons montré que `revAux l l' = reverse l ++ l'`.

Q 2.3 Spécification et démonstration Voici la spécification de `insererAux k l acc` :

Pré-condition `triée(revAux acc l)` et `k` est strictement plus grand que tous les éléments de `acc`.

Post-condition `triée(insererAux k l acc)`, il existe `l1` et `l2` tels que `l = l1 ++ l2` et `insererAux k l acc = revAux acc (l1 ++ [k] ++ l2)`.

Variant `length l`

Nous admettrons les propriétés suivante :

— lorsque `k` est plus grand que tous les éléments de `l1` et plus petit que tous les éléments de `l2` et que `revAux l1 l2` et triée, alors `revAux l1 (k:l2)` est triée,

— lorsque `revAux l1 l2` est triée, `l2` est triée.

Démontrez que `insererAux` vérifie sa spécification.