Haskell (Info 3)
Fonctions et fonctions d'ordre supérieur

Stéphane Vialette (stephane.vialette@univ-eiffel.fr)

22 novembre 2022

Question 1: Échauffement

(a) On souhaire écrire la fonction `double` qui multiplie par deux chaque élément d'une liste. Son type est donné par : `double :: Num a => [a] -> [a]`

Exemple d'utilisation :

```
λ: double []
[]
λ: double [1..5]
[2,4,6,8,10]
λ: double [1.0, 2.0, 3.0, 4.0, 5.0]
[2.0,4.0,6.0,8.0,10.0]
```

Écrire la fonction `double` :

1. en utilisant une compréhension de liste,
2. en utilisant une fonction récursive,
3. en utilisant la fonctionnelle `map` et une lambda, et enfin
4. en utilisant la fonctionnelle `map` sans lambda.

(b) On souhaire écrire la fonction `firstEvenSquares` qui calcule les `k` premiers carrés pairs. Son type est donné par :

```
firstEvenSquares :: Integral a => Int -> [a]
```

Exemple d'utilisation :

```
λ: [firstEvenSquares k | k <- [0.. 5]]
[[],[4],[4,16],[4,16,36],[4,16,36,64],[4,16,36,64,100]]
```

Écrire la fonction `firstEvenSquares` :

1. en utilisant les fonctionnelles `take` et `filter`, et une compréhension de liste,
2. en utilisant les fonctionnelles `map`, `take` et `filter`,
3. en utilisant une fonction récursive non terminale, et enfin
4. en utilisant une fonction récursive terminale.

(c) On souhaire maintenant écrire la fonction `altParPerms` qui calcule les permutations alternantes (sous forme d'une liste de listes d'entiers) d'une longueur donnée. Une permutations est alternante si tous les paires d'éléments successifs n'ont pas la même parité. Son type est donné par :

```
altParPerms :: Int -> [[Int]]
```

Exemple d'utilisation :

```
λ: mapM_ print [altParPerms n | n <- [0..4]]
[[]]
[[1]]
[[1,2],[2,1]]
[[1,2,3],[3,2,1]]
[[1,2,3,4],[2,3,4,1],[3,2,1,4],[3,4,1,2],[2,1,4,3],[4,1,2,3],
 [1,4,3,2],[4,3,2,1]]
```

Nous allons décomposer le problème en deux sous-problèmes. Écrire dans un premier temps la fonction

```
permutations :: [a] -> [[a]]
```

qui retourne toutes les permutations d'une liste données.

```
λ: mapM_ print [permutations [1..n] | n <- [0..3]]
[[]]
[[1]]
[[1,2],[2,1]]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```

Écrire maintenant la fonction `altParPerms` en utilisant la fonction `permutations`.

Petite question bonus (elle est un peu plus compliquée mais pas hors de portée). Nous pouvons être un peu plus astucieux et générer directement les permutations alternantes sans avoir à filtrer l'ensemble des permutations. Proposer une implémentation.

Question 2: Origami

(a) Écrire la fonction `double` qui multiplie par deux chaque élément d'une liste en utilisant `foldl` puis `foldr`.

(b) Écrire, en utilisant les fonctionnelles `foldl` ou `foldr`, la fonction

```
splitPar :: (Foldable t, Integral a) => t a -> ([a], [a])
```

qui, à partir d'une liste d'entiers, calcule une paire de listes, la première contenant tous les entiers pairs et la seconde tous les entiers impairs.

Exemple d'utilisation :

```
λ: splitPar []
([],[])
λ: splitPar [1]
([],[1])
λ: splitPar [2]
([2],[])
λ: splitPar [1..10]
([2,4,6,8,10],[1,3,5,7,9])
```

Écrire la même fonction `splitPar` en utilisant des fonctionnelles d'ordre supérieur différentes de `foldl` et `foldr`.

(c) Écrire, en utilisant les fonctionnelles d'ordre supérieur `foldl` ou `foldr`, la fonction

```
prefixSums :: (Foldable t, Num a) => t a -> [a]
```

qui, à partir d'une liste d'entiers, calcule la liste des sommes cumulées.

Un court exemple d'utilisation :

```
λ: prefixSums []
[]
λ: prefixSums [1]
[1]
```

```
λ: prefixSums [1,2]
[1,3]
λ: prefixSums [1..10]
[1,3,6,10,15,21,28,36,45,55]
```

Écrire maintenant la fonction `invPrefixSums` qui, à partir d'une liste `xs`, calcule la liste `ys` dont les sommes cumulées sont exactement `xs`.

Exemple d'utilisation :

```
λ: invPrefixSums []
[]
λ: invPrefixSums [1]
[1]
λ: invPrefixSums [1..10]
[1,1,1,1,1,1,1,1,1,1]
λ: invPrefixSums (prefixSums [1..10])
[1,2,3,4,5,6,7,8,9,10]
```

(d) Écrire la fonction

```
shift :: [a] -> [a]
```

qui, à partir d'une liste d'entiers `xs`, calcule qui place le premier élément de `xs` à la dernière place. On supposera que `xs` n'est pas la liste vide.

Exemple d'utilisation :

```
λ: shift []
[]
λ: shift [1]
[1]
λ: shift [1,2]
[2,1]
λ: shift [1,2,3,4,5]
[2,3,4,5,1]
```

On considère la fonction

```
rotate :: [a] -> [[a]]
```

qui calcule toutes les rotations d'une liste (aucun ordre n'est imposé).

Exemple d'utilisation :

```
λ: rotate []
[]
λ: rotate [1]
[[1]]
λ: rotate [1,2]
[[2,1],[1,2]]
λ: rotate [1,2,3,4,5]
[[5,1,2,3,4],[4,5,1,2,3],[3,4,5,1,2],[2,3,4,5,1],[1,2,3,4,5]]
```

Écrire la fonction `rotate :: [a] -> [[a]]` en utilisant uniquement la fonction `shift :: [a] -> [a]` et la fonctionnelle `foldl`

(e) Le `fold` sur les entiers peut être définie de la façon suivante :

```
foldi :: (a -> a) -> a -> Int -> a
foldi f q 0 = q
foldi f q i = f (foldi f q (pred i))
```

Écrire les fonctions `add` (*i.e.*, $x + y$ avec x et y entiers), `mult` (*i.e.*, $x \times y$ avec x et y entiers), `expo` (*i.e.*, x^y avec x et y entiers) et `fact` (*i.e.*, $x!$ avec x entier).