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.

Solution:

Avec une compréhension de liste :

```
double :: Num a => [a] -> [a]
double xs = [2*x | x <- xs]
```

Avec une fonction récursive :

```
double :: Num a => [a] -> [a]
double [] = []
double (x : xs) = (2 * x) : double xs
```

Avec la fonctionnelle `map` et une lambda :

```
double :: Num a => [a] -> [a]
double xs = map (\ x -> 2*x) xs
```

Mais la fonction `\ x -> 2*x` n'est autre que la fonction `(*) 2`. Nous pouvons donc donc simplifier l'écriture en :

```haskell
double :: Num a => [a] -> [a]
double xs = map ((*) 2) xs
```

En utlisant une *section* et une η-*réduction*, nous obtenons finalement une écriture assez compacte tout en restant très lisible.

```haskell
double :: Num a => [a] -> [a]
double = map (2 *)
```

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

```haskell
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.

Solution:

Nous allons utiliser une approche fonctionnelle en *découpant* le problème en sous-problèmes :

1. générer les carrés,
2. ne retenir que les carrés pairs, et
3. sélectionner le bon nombre de carrés pairs.

Puisqu'haskell est *paresseux*, nous n'allons pas alourdir la fonction de génération des carrés en comptant ; nous générons une liste potentiellement infinie et nous sélectionnerons ceux qui nous interesse dans un second temps.

En utilisant les fonctionnelles `take` et `filter`, et une compréhension de liste :

```haskell
firstEvenSquares :: Integral a => Int -> [a]
firstEvenSquares k =  take k $ filter even [x*x | x <- [1..]]
```

En utilisant les fonctionnelles `map`, `take` et `filter` :

```haskell
firstEvenSquares :: Integral a => Int -> [a]
firstEvenSquares k = take k . filter even $ map (\x -> x*x) [1..]
```

Si nous préférons une fonction plus simple, nous pouvons remplacer la lambda par une fonction explicitement définie :

```haskell
firstEvenSquares :: Integral a => Int -> [a]
firstEvenSquares k = take k . filter even $ map f [1..]
  where
    f x = x*x
```

De manière générale, préférez une fonction explicitement définie si la lambda est un peu compliquée ou trop longue, cela facilite grandement la lecture. Je ne suis pas persuadé que ce soit le cas dans cet exemple . . . mais c'est une affaire de goût !

En utilisant une fonction récursive non terminale :

```haskell
firstEvenSquares :: (Integral a) => Int -> [a]
firstEvenSquares k = go 1 0
  where
    go n c
      | c == k      = []
      | even (n*n)  = (n*n) : go (n+1) (c+1)
      | otherwise   = go (n+1) c
```

Pour obtenir une fonction récursive terminale, nous utilisons un accumulateur dans lequel nous allons conduire le calcul :

```haskell
firstEvenSquares :: (Integral a) => Int -> [a]
firstEvenSquares k = reverse $ go [] 1 0
  where
    go acc n c
      | c == k      = acc
      | even (n*n)  = go ((n*n) : acc) (n+1) (c+1)
      | otherwise   = go acc           (n+1) c
```

Cette solution est totalement *idiomatique* en haskell. Attention à l'initialisation de l'accumulateur (sa valeur précise est dépendante du problème considéré).

Une fonction récursive non terminale est, en générale, plus simple et plus lisible que sa version récursive terminale. La question que vous devez alors vous poser est la suivante : ai-je un intérêt à ré-écrire ma fonction sous forme récursive terminale ? Des éléments de réponse sont bien souvent donnés par la mise en évidence des fonctions coûteuses dans votre programme (il existe bien sûr des outils pour l'analyse de programme haskell). C'est sur fonctions principalement qu'il convient de porter nos efforts dans un premier temps.

Pour conclure cet exercice, en règle générale, préférez la composition de fonctionnelles d'ordre supérieur à l'écriture explicite de fonctions.

(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 :

```haskell
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`.

Solution:

Bien sûr, il existe la fonction `Data.List.permutations` qui sait très bien faire ça, mais essayons de programmer notre propre version. Comment générer toutes les permutations de la liste $[x_1, x_2, \ldots, x_n]$? Pensons récursif ! Ok, supposons donc dans un premier temps que nous sachions générer toutes les permutations d'une liste plus courte. Pouvons nous alors utiliser de façon simple ce résultat pour générer toutes les permutations de la liste $[x_1, x_2, \ldots, x_n]$? Oui, bien sûr ! Si nous savons générer toutes les permutations de la liste $[x_2, \ldots, x_n]$, il suffit d'insérer x_1 à toutes les positions pour produire toutes les permutations d'une liste $[x_1, x_2, \ldots, x_n]$. Par exemple, à partir de $[[2, 3], [3, 2]]$, nous obtenons notre résultat en :

1. insérant 1 dans $[2, 3]$ partout où cela est possible pour obtenir la liste résultat $[[1, 2, 3], [2, 1, 3], [2, 3, 1]]$,
2. insérant 1 dans $[3, 2]$ partout où cela est possible pour obtenir la liste résultat $[[1, 3, 2], [3, 1, 2], [3, 2, 1]]$.

Il suffit dans un dernier temps de regrouper ces deux listes en une liste unique. Ca y est, nous avons notre schéma récursif !. Il s'agit maintenant d'implémenter ça en haskell.

```
perms :: [a] -> [[a]]
perms []       = [[]]
perms (x : xs) = concatMap (ins x) (perms xs)
  where
    ins y []             = [[y]]
    ins y all@(y' : ys) = (y : all) : map (y' :) (ins y ys)
```

Remarquez que `map (y' :) yss` insère `y'` en première position dans toutes les listes de `ys`.

Pour obtenir la fonction `altParPerms`, nous générons toutes les permutations et filtrons pour ne conserver que les permutations pour lesquelles les éléments alternent en parité. Pas de réelles difficultés ici.

```
altParPerms :: Integral a => a -> [[a]]
altParPerms n = filter altPar $ perms [1..n]
  where
    altPar []  = True
    altPar [x] = True
```

```
    altPar xs  = all test $ zip xs (tail xs)
      where
        test (x, x') = x `mod` 2 /= x' `mod` 2
```

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.

Solution:

Une permutation alternante commence par un entier pair ou par un entier impair. C'est une évidence, mais cela nous permet de distinguer deux sous-problèmes. Nous allons donc faire *deux* paquets et considérer toutes les permutations des entiers pairs puis toutes les permutations des entiers impairs (et ça nous savons faire maintenant). Si nous parvenons à les mélanger parfaitement à l'instar d'un jeu de cartes, c'est terminé !

Pour le mélange parfait, ce n'est pas bien compliqué : nous prenons le premier élément de la première liste, le premier élément de la seconde liste, et ... on recommence.

```
perfectShuffle :: [a] -> [a] -> [a]
perfectShuffle []       []       = []
perfectShuffle xs       []       = xs
perfectShuffle []       ys       = ys
perfectShuffle (x : xs) (y : ys) = x : y : perfectShuffle xs ys
```

Maintenant, il suffit de suivre notre décomposition en deux sous-problèmes et s'en remettre au mélange parfait pour produire les permutations alternantes.

```
altParPerms :: (Num a, Enum a) => a -> [[a]]
altParPerms n = startWithEven ++ startWithOdd
  where
    evenPerms = perms [2,4..n]
    oddPerms  = perms [1,3..n]
    startWithEven = [perfectShuffle xs ys | xs <- evenPerms
                                          , ys <- oddPerms]
    startWithOdd  = [perfectShuffle ys xs | xs <- evenPerms
                                          , ys <- oddPerms]
```

Question 2: Origami

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

Solution:

Regardons dans un premier temps `foldl`. Puisque nous parcourons la liste de gauche à droite, il faut placer l'élément courant en fin de liste :

```
double :: Num a => [a] -> [a]
double = foldl (\acc x -> acc ++ [2*x]) []
```

C'est terriblement inefficace. Il est préférable de l'ajouter en tête de l'accumulateur et de renverser la liste en fin de calcul :

Avec `foldr`, c'est beaucoup plus facile. Nous parcourons la liste de droite à gauche, nous pouvons donc ajouter l'élément courant en tête de l'accululateur sans avoir à le renverser en fin de calcul.

```
double :: Num a => [a] -> [a]
double = foldr (\x acc -> (2*x) : acc) []
```

C'est un point **très important** à retenir : lorsqu'il s'agit de calculer une liste à partir d'une liste, il est préférable d'utiliser `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])
```

Solution:
Puisque nous devons calculer une paire de listes, il est naturel que l'accumulateur soit également une paire de listes. De plus, nous devons construire une liste à partir d'une liste, la fonctionnelle `foldr` est le choix qui s'impose. L'implémentation est alors presque immédiate.

```
splitPar :: (Foldable t, Integral a) => t a -> ([a], [a])
splitPar = foldr step ([], [])
  where
    step x (evens, odds)
      | even x    = (x : evens, odds)
      | otherwise = (evens, x: odds)
```

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

Solution:
Il y a pléthore de solutions. Commençons par une implémentation qui pique un peu les yeux.

```
splitPar :: Integral a => [a] -> ([a], [a])
splitPar xs = (xs \\ [1,3..n], xs \\ [2,4..n])
  where
    n = fromIntegral $ length xs
```

Remarquons que `xs \\ [1,3..]` ne peut pas fonctionner. Pourquoi ? Remarquons également l'emploi de `fromIntegral`. Pourquoi ?

Le plus simple (et le plus élégant) est de filtrer deux fois la liste.

```
splitParity :: Integral a => [a] -> ([a], [a])
splitParity xs = (filter even xs, filter odd xs)
```

(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]
```

Solution:
La question nous *impose* en quelque sorte d'utiliser `foldl`, il ne faudra donc pas oublier de retourner la liste résultat.

```
prefixSums :: (Foldable t, Num a) => t a -> [a]
prefixSums = reverse . foldl step []
  where
    step []           x = [x]
    step acc@(s : _) x = (x+s) : acc
```

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]
```

Solution:

```
invPrefixSums :: (Foldable t, Num a) => t a -> [a]
invPrefixSums = reverse . foldl step []
  where
    step []           x = [x]
    step acc@(s : _) x = (x - sum acc) : acc
```

(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]
```

Solution:
Aucune difficulté ici.

```haskell
shift :: [a] -> [a]
shift [] = []
shift (x : xs) = xs ++ [x]
```

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`

Solution:
De prime abord, le problème paraît un peu plus compliqué. Nous aurons sans doute fait un grand pas si nous comprenons le lien entre `shift` et `rotate`. Voyons … `shift [1,2,3,4,5]` produit `[2,3,4,5,1]`. Ca y est, c'est gagné, nous avons le lien, `[2,3,4,5,1]` est une des rotations de `[1,2,3,4,5]`. En toute généralité, `shift xs` calcule une rotation de `xs` et donc, puisqu'il s'agit d'itérer une fonction, nous pourrions très bien écrire quelque chose comme :

```haskell
rotate' :: [a] -> [[a]]
rotate' xs = take (length xs) $ iterate shift xs
```

Le problème est que l'on utilise ici les fonctions `take`, `length`, `iterate` et `shift`, alors que nous ne devons utiliser que les fonctions `foldl` et `shift`. C'est embêtant ! Prenons un peu de hauteur. En fait, nous avons utilisé les fonctions `take` et `length` pour *compter* (*i.e.*, combien de fois devons nous appeler la fonction `shift` ?). Mais c'est quelque chose que nous pouvons très bien faire avec `foldl` puisque cette

fonction d'ordre supérieur parcourt tous les éléments de la liste. Pour la fonction `iterate`, une utilisation astucieuse de l'accumulateur de `foldl` devrait nous permettre de nous en passer. Ce n'est finalement pas si difficile.

```
rotate :: [a] -> [[a]]
rotate xs = foldl f [] xs
  where
    f []            _ = [xs]
    f acc@(xs' : _) _ = shift xs' : acc
```

(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).

Solution:
Avant toute chose, il convient de bien comprendre le fonctionnement de `foldi`. Commençons par le cas d'arêt. Nous avons `foldi f x 0` = `x`. Pour l'appel récursif, il sera peut-être plus simple de voir `pred x` comme `x-1`. Nous avons

```
foldi f x y = f (foldi f x (y-1))
            = f (f (foldi f x (y-2)))
            = f (f (f (foldi f x (y-3))))
              ⋮
            = f (f (f ... (f (foldi f x 0)) ... ))
            = f (f (f ... (f x) ... ))
```

Le *vrai* calcul se fait donc lors des retours des appels récursifs (il y a `y` appels) avec `x` comme valeur initial.

Pour `add x y`, on remarque que

$$x + y = 1 + (1 + (\ldots(1 + x)\ldots))$$

Ca y est, nous avons une écriture avec une fonction itérée et une valeur initiale (c'est `x`).

```
add :: Int -> Int -> Int
add = foldi (+ 1)
```

Pour `mult x y`, on obtient $xy = x + (x + (\ldots(x + 0)\ldots))$. Ca y est, une nouvelle fois nous avons une écriture avec une fonction itérée et une valeur initiale (c'est `0`).

```
mult :: Num a => a -> Int -> a
mult i = foldi (+ i) 0
```

Pour `expo x y`, nous tombons presque dans la routine. Nous avons immédiatement

$$x^y = x * (x * (\ldots(x \times 1)\ldots))$$

```haskell
expo :: Num a => a -> Int -> a
expo i = foldi (* i) 1
```

La fonction `fact x` semble un peu plus difficile. Commençons alors par quelque de simple ! On sait que $n! = n \times (n-1)!$. Nous devons bien pouvoir tirer quelque chose de ça ? En itérant, nous obtenons

$$\begin{aligned} n! &= n \times (n-1)! \\ &= n \times (n-1) \times (n-2)! \\ &\vdots \\ &= n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \end{aligned}$$

Aïe, ça ne ressemble par beaucoup à nos précédentes décompositions. Ne désespérons pas et revenons à notre première idée, à savoir $n! = n \times (n-1)!$. Mais nous avons vu que nous pouvions implémenter la multiplication par un `foldi`. Ok, trouvé !, c'est jeu à deux bandes, il nous faut combiner `foldi` avec un appel récursif (et nous n'oublierons pas le cas d'arrêt)

```haskell
fact :: Int -> Int
fact 1 = 1
fact x = foldi (+ x) 0 (fact (x-1))
```

Nous pouvons même réécrire la fonction pour gérer $0!$.

```haskell
fact :: Int -> Int
fact 0 = 1
fact 1 = 1
fact x = foldi (+ x) 0 (fact (x-1))
```