Programmation fonctionnelle
TP-03
Fonctions d'ordre supérieur

Stéphane Vialette stephane.vialette@univ-eiffel.fr

23 novembre 2023

Avertissement : dans tout le TP, on fait fréquemment référence à des fonctions du module `Data.List`, qu'on supposera déjà importé :

```
import Data.List
ma_fonction xs = head (tail xs)
```

Toutes les fonctions à programmer dans ce TP portent un nom différent des fonctions prédéfinies de Haskell. Quand ce n'est pas le cas, on peut utiliser le mécanisme des imports dits « qualifiés » pour éviter toute confusion :

```
import qualified Data.List as L
ma_fonction xs = L.head (L.tail xs)
```

Exercice 1 : Échauffement (à faire avant le début du TP)

(a) En utilisant un `fold`, écrire le prédicat `elem'` qui décide si un élément apparaît dans une liste.

(b) Autour de la fonction map.

(a) Écrire trois versions `map1`, `map2` et `map3` de la fonction `map` (qui construit une nouvelle liste en appliquant une fonction à chaque élément de la liste reçue), en utilisant respectivement (i) une fonction récursive, (ii) une compréhension de liste, (iii) la fonction `foldr`.

```
λ: map (\x -> [x,x]) [1..4]
[[1,1],[2,2],[3,3],[4,4]]
```

(b) Voici deux versions supplémentaires de la fonction `map` utilisant la fonction `foldl` plutôt que `foldr`. Sont-elles correctes ? Si non, comment faudrait-il les corriger ? Que peut-on dire de leur complexité ?

```
map4 f = foldl (\acc x -> f x:acc) []
map5 f = foldl (\acc x -> acc ++ [f x]) []
```

(c) La fonction `map` de Haskell accepte aussi des listes infinies :

```
λ: take 5 $ map (^2) [1..]
[1, 4, 9, 16, 25]
```

Essayer ce test avec les versions `map1` à `map5` ci-dessus et interpréter les résultats.

(d) Écrire trois versions `filter1`, `filter2` et `filter3` de la fonction `filter` qui filtre les éléments d'une liste à l'aide d'un prédicat en utilisant respectivement (i) une fonction récursive, (ii) une compréhension de liste, (iii) la fonction `foldr` ou `foldl`.

```
λ: filter even [1..10]
[2,4,6,8,10]
```

(e) Écrire deux versions `takeWhile1` et `takeWhile2` de la fonction

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

qui conserve les éléments du début de la liste tant que le prédicat en argument appliqué à l'élément courant vaut `True`, en utilisant respectivement (i) une fonction récursive puis (ii) la fonction `foldr` ★.

```
λ: takeWhile (<5) [1..10]
[1,2,3,4]
λ: takeWhile odd [1..10]
[1]
```

(f) Écrire deux versions `dropWhile1` et `dropWhile2` de la fonction

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

qui rejette les éléments du début de la liste vérifiant le prédicat en argument, en utilisant respectivement (i) une fonction récursive puis (ii) la fonction `foldl` ★.

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

Que pensez-vous d'une implémentation par compréhension de liste ?

(g) Écrire deux versions `zipWith1` et `zipWith2` de la fonction

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

en utilisant respectivement (i) une fonction récursive puis (ii) d'autres fonctions d'ordre supérieur.

```
λ: zipWith (*) [1,2,3,4] [5,6,7,8]
[5,12,21,32]
λ: zipWith replicate [1,2,3,4] [5,6,7,8]
[[5],[6,6],[7,7,7],[8,8,8,8]]
λ: zipWith (,) [1..] ['a'..'g']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e'),(6,'f'),(7,'g')]
```

Que pensez-vous d'une implémentation par compréhension de liste ?

(h) Écrire trois versions `reverse1`, `reverse2` et `reverse3` de la fonction

```
reverse :: [a] -> [a]
```

qui renverse une liste en utilisant respectivement (i) une fonction récursive, (ii) la fonction `foldl` puis (iii) la fonction `foldr`. Ont-elles toutes la même complexité ? Que pouvez-vous en conclure ?

Exercice 2 : Suite infinie de Fibonacci

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Notée (F_n), elle est définie par $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geqslant 2$. Les premiers termes de (F_n) sont 1, 1, 2, 3, 5, 8, 13 et 21.

(a) On peut définir cette suite infinie à l'aide la fonction récursive suivante :

```
fibonacciSeq1 :: [Integer]
fibonacciSeq1 = [f n | n <- [0..]]
  where
    f 0 = 0
    f 1 = 1
    f n = f (n-1) + f (n-2)
```

Quelle est sa complexité ? Peut-on faire mieux ?

(b) Écrire une nouvelle version récursive `fibonacciSeq2` plus efficace, qui ne recalcule pas plusieurs fois le même terme (*indication : calculer* `[x+y, y +(x+y), (x+y)+(y+(x+y)), ...]`)★ .

(c) Écrire une nouvelle version aussi efficace `fibonacciSeq3` qui utilise la fonction `iterate` vue en cours (*indication : on peut calculer des paires de termes consécutifs).*

(d) Analysez et commentez la fonction suivante :

```
fibonacciSeq4 :: [Integer]
fibonacciSeq4 = 0:zipWith (+) (1:fibonacciSeq4) fibonacciSeq4
```

Exercice 3 : Autour des permutations ★

La fonction `permutations` du module `Data.List` permet de générer la liste de toutes les permutations d'une liste.

```
λ: :type permutations
permutations :: [a] -> [[a]]
λ: permutations []
[[]]
λ: permutations [1,2,3]
[[1,2,3],[2,1,3],[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
```

(a) **Permutations par insertions**.

(a) Écrire la fonction `distrib :: a -> [a] -> [[a]]`

La fonction `distrib x xs` renvoie la liste de toutes les listes que l'on peut obtenir en insérant `x` dans `xs`. Par exemple :

```
λ: distrib 1 [2,3,4]
[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]
```

(b) En déduire une version `permutations1` de la fonction `permutations` qui utilise `distrib`.

Indice : vous allez probablement avoir besoin d'aplatir une liste...

(c) **Permutations par mélange**.

(a) Écrire la fonction

`shuffles :: [a] -> [a] -> [[a]]`

La fonction `shuffles ys xs` renvoie la liste de toutes les listes que l'on peut obtenir en intercalant les éléments des listes `xs` et `ys`, comme on rassemble deux paquets de cartes en les mélangeant. L'ordre relatif des éléments de `xs` et `ys` est inchangé dans les listes résultats.

```
λ: shuffles [1,2] []
[[1,2]]
λ: shuffles [1,2] [3,4]
[[1,2,3,4],[1,3,2,4],[1,3,4,2],[3,1,2,4],[3,1,4,2],[3,4,1,2]]
```

Indice : observez les premières valeurs des listes renvoyées par `shuffles`...

(b) La fonction `splitAt` du module `Data.List` permet de séparer une liste en deux à une position donnée. Par exemple :

```
λ: :type splitAt
splitAt :: Int -> [a] -> ([a], [a])
λ: [splitAt i [1,2,3,4] | i <- [0,1,2,3,4]]
[([],[1,2,3,4]),([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4]),([1,2,3,4],[])]
```

En déduire une version `permutations2` de la fonction `permutations` qui utilise `shuffles` et `splitAt`.

(c) **Permutations par ajout en tête**. L'ensemble des permutations des éléments de $X = \{x_1, x_2, \ldots\}$ peut se partitionner en deux sous ensembles : d'une part, les permutations de X qui débutent par x_1 et d'autre part, celles qui ne débutent pas par x_1. Ces deux sous-ensembles sont, par définition, disjoints. En déduire une version `permutations3` de la fonction `permutations`.

Exercice 4 : Applications

(a) Considérons la fonction d'ordre supérieur `unfold` définie de la façon suivante :

```
unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
unfold p h t x
  | p x       = []
  | otherwise = h x:unfold p h t (t x)
```

La fonction `unfold` permet d'encapsuler un motif simple de récursion pour calculer une liste. Par exemple, la fonction `intToBins` définie par :

```
intToBins :: Integer -> [Integer]
intToBins = unfold (== 0) (`mod` 2) (`div` 2)
```

permet de convertir un entier en binaire :

```
λ: intToBins 0
[]
λ: intToBins 5
[1,0,1]
λ: intToBins 42
[0,1,0,1,0,1]
```

À l'aide de la fonction d'ordre supérieur `unfold` ci-dessus :

(a) Calculer la liste `[0,1,2,3,4,5,6,7,8,9]`.

(b) Écrire une version `map'` de la fonction `map :: (a -> b) -> [a] -> [b]`.

(c) Écrire une version `iterate'` de la fonction `iterate :: (a -> a) -> a -> [a]`.

(d) Écrire la fonction

```
altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
```

qui applique alternativement ses deux fonctions passées en arguments. Par exemple :

```
λ: altMap (+10) (*100) [1,2,3,4,5]
[11,200,13,400,15]
λ: altMap (+10) id [1,2,3,4,5]
[11,2,13,4,15]
λ: altMap id (*100) [1,2,3,4,5]
[1,200,3,400,5]
λ: altMap id id [1,2,3,4,5]
[1,2,3,4,5]
```

(e) Écrire la fonction

```
digits :: (Integral a) => a -> [a]
```

permettant de convertir un entier en la liste des chiffres le composant. Par exemple :

```
λ: digits1 0
[0]
λ: digits1 1
[1]
λ: digits1 12345
[1,2,3,4,5]
```

(f) L'*algorithme de Luhn* détermine si un numéro de carte de crédit Visa est valide ou non. Pour un numéro de carte de crédit donné :

1. Doublez la valeur d'un chiffre sur deux de la droite vers la gauche, en commençant à partir de l'avant-dernier chiffre.
2. Ajoutez les chiffres du résultat de l'étape 1 aux chiffres restants du numéro de la carte de crédit.
3. Si le résultat mod 10 est égal à 0, le numéro est valide. Si le résultat en base 10 est différent de 0, la validation échoue.

(source IBM Sterling Order Management Software).

Par exemple, considérons un numéro de carte de crédit portant le numéro 4624 7482 3324 9080.

1. Doublez la valeur d'un chiffre sur deux de la droite vers la gauche, en commençant à partir de l'antépénultième chiffre.

$$8 \times 2 = 16$$
$$9 \times 2 = 18$$
$$2 \times 2 = 4$$
$$3 \times 2 = 6$$
$$8 \times 2 = 16$$
$$7 \times 2 = 14$$
$$2 \times 2 = 4$$
$$4 \times 2 = 8$$

2. Ajoutez les chiffres du résultat de l'étape 1 aux chiffres restants du numéro de la carte de crédit.

$$1+6+1+8+4+6+1+6+1+4+4+8=50$$
$$6+4+4+2+3+4+0+0=23$$
$$50+23=73$$

3. Si le résultat mod 10 est égal à 0, le numéro est valide. Si le résultat en base 10 est différent de 0, la validation échoue.

$$73 \equiv 3 \pmod{10}$$

Donc le numéro 4624 7482 3324 9080 n'est pas un numéro de carte de crédit valide. Par contre, 4624 7482 3324 9780 est un numéro de carte de crédit valide.

Écrire le prédicat

```
luhn :: [a] -> Bool
```

qui décide si un numéro de carte de crédit est valide. Par exemple :

```
λ: luhn [4,6,2,4, 7,4,8,2, 3,3,2,4, 9,0,8,0]
False
λ: luhn [4,6,2,4, 7,4,8,2, 3,3,2,4, 9,7,8,0]
True
```

Exercice 5 : Bonus d'entraînement : Carré magique par compréhensions de listes

Un *carré magique* d'ordre n est composé de n^2 entiers strictement positifs, écrits sous la forme d'un tableau carré. Ces nombres sont disposés de sorte que leurs sommes sur chaque ligne, sur chaque colonne et sur chaque diagonale principale soient égales. On nomme alors *constante magique* la valeur de ces sommes. Un *carré magique normal* est un cas particulier de carré magique, constitué de tous les nombres entiers de 1 à n^2, où n est l'ordre du carré. Nous allons nous intéresser aux carrés magiques d'ordre 3.

(a) À l'aide d'une compréhension de liste, écrire la fonction

```
gen3 :: Int -> [(Int, Int, Int)]
```

`gen3 lb` calcule tous les triplets `(x1,x2,x3)` d'éléments distincts compris entre `lb` et `lb+8`. Attention, `(1,2,3)` et `(2,1,3)` sont des triplets différents. Par exemple :

```
λ: length (gen3 1)
504
λ: take 5 (gen3 1)
[(1,2,3),(1,2,4),(1,2,5),(1,2,6),(1,2,7)]
λ: length (gen3 101)
504
λ: take 5 (gen3 101)
[(101,102,103),(101,102,104),(101,102,105),(101,102,106),(101,102,107)]
```

(b) À l'aide d'une compréhension de liste, écrire la fonction

```
gen3' :: Int -> [Int] -> [(Int, Int, Int)]
```

`gen3' lb xs` calcule tous les triplets `(x1,x2,x3)` d'éléments distincts compris entre `lb` et `lb+8` qui ne sont pas dans `xs`. Par exemple :

```
λ: length (gen3' 1 [1,2,3])
120
λ: take 5 (gen3' 1 [1,2,3])
[(4,5,6),(4,5,7),(4,5,8),(4,5,9),(4,6,5)]
```

(c) À l'aide d'une compréhension de liste, écrire maintenant la fonction

```
magicSquare3 :: Int -> [((Int, Int, Int), (Int, Int, Int), (Int, Int, Int))]
```

qui calcule tous les carrés magiques d'ordre 3 constitués de tous les nombres entiers de `lb` à `lb+8` (où `lb` est le paramètre de la fonction).

```
λ: length (magicSquare3 1)
8
λ: take 2 (magicSquare3 1)
[((2,7,6),(9,5,1),(4,3,8)),((2,9,4),(7,5,3),(6,1,8))]
λ: take 2 (magicSquare3 101)
[((102,107,106),(109,105,101),(104,103,108)), ((102,109,104),(107,105,103),(106,101,108))]
```

(d) Pour terminer, écrire la fonction

```
normalMagicSquare3 :: [((Int, Int, Int), (Int, Int, Int), (Int, Int, Int))]
```

qui calcule tous les carrés magiques normaux d'ordre 3.

```
λ: length normalMagicSquare3
8
λ: take 2 normalMagicSquare3
[((2,7,6),(9,5,1),(4,3,8)),((2,9,4),(7,5,3),(6,1,8))]
```