Programmation fonctionnelle (L3)
TP noté (2h00)

Antoine Meyer	antoine.meyer@univ-eiffel.fr
Carine Pivoteau	carine.pivoteau@univ-eiffel.fr
Fabian Reiter	fabian.reiter@univ-eiffel.fr
Stéphane Vialette	stephane.vialette@univ-eiffel.fr

Les 4 exercices sont indépendants. Le seul document autorisé est une feuille A4 recto-verso manuscrite. Dans la mesure du possible, essayez de ne pas écrire du code grossièrement inefficace (par exemple, pas de `[x] ++ xs` lorsque `x : xs` fait sens !).

Votre TP doit impérativement se trouver dans le répertoire `EXAM` sous la forme d'un unique fichier nommé `TPNote.hs`. Un squelette de même nom vous est fourni.

Exercice 1 : Compréhensions de listes

Vous devez **obligatoirement** utiliser une **compréhension de liste** pour toutes les fonctions de cet exercice (vous pouvez bien sûr définir et utiliser si besoin des fonctions internes qui sont appelées depuis votre compréhension de liste).

(a) Écrire la fonction :

```
evenInside :: [[Int]] -> [[Int]]
```

La fonction `evenInside` prend en argument une liste de listes d'entiers. Pour chaque sous-liste, elle conserve uniquement les entiers pairs. La structure des listes doit être préservée.

```
λ> evenInside [[1..5],[2..7],[3..9]]
[[2,4],[2,4,6],[4,6,8]]
λ> evenInside [[1,2],[3,5]]
[[2],[]]
```

(b) Écrire la fonction :

```
evenInsideFlat :: [[Int]] -> [Int]
```

La fonction `evenInsideFlat` prend en argument une liste de listes d'entiers et renvoie une liste plate contenant uniquement les entiers pairs extraits de toutes les sous-listes.

```
λ> evenInsideFlat [[1..5],[2..7],[3..9]]
[2,4,2,4,6,4,6,8]
```

(c) Dans une liste d'entiers, un *point fixe* est un élément qui est égal à sa propre position dans la liste (en comptant les positions à partir de zéro). Écrire la fonction :

```
countFixedPoints :: [Int] -> Int
```

La fonction `countFixedPoints xs` retourne le nombre de points fixes dans la liste `xs`.

```
λ> countFixedPoints [0,1,2,3]
4
λ> countFixedPoints [1,2,3,4]
0
λ> countFixedPoints [0,2,2,3]
3
```

(d) Écrire la fonction :

```
cleanString :: String -> String
```

La fonction `cleanString s` prend une chaîne de caractères `s` et renvoie une nouvelle chaîne de caractères contenant uniquement ses lettres minuscules (de `'a'` à `'z'`) et telle que tous les espaces ont été transformés en underscores `'_'`. Tous les autres caractères (majuscules, chiffres, ponctuation, ...) doivent être supprimés.

```
λ > cleanString "Hello World!"
"ello_orld"
λ > cleanString "Haskell 9.4.8 est top !"
"askell__est_top_"
```

(e) Dans cet exercice, on représente des points par des couples d'entiers (x, y). Écrire la fonction :

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

`inDisk r` retourne la liste des points à coordonnées entières qui se situent à l'intérieur ou sur le bord du disque de rayon `r` centré en $(0, 0)$. On rappelle que dans un triangle rectangle, le carré de la longueur de l'hypoténuse (ou côté opposé à l'angle droit) est égal à la somme des carrés des longueurs des deux autres côtés (ça peut toujours servir !).

```
λ > inDisk 0
[(0,0)]
λ > inDisk 1
[(-1,0),(0,-1),(0,0),(0,1),(1,0)]
λ > inDisk 2
[(-2,0),(-1,-1),(-1,0),(-1,1),(0,-2),(0,-1),(0,0),(0,1),(0,2),(1,-1),(1,0),(1,1),(2,0)]
```

Exercice 2 : Fonctions récursives et fonctions d'ordre supérieur

(a) En utilisant obligatoirement des fonctions d'ordre supérieur écrire une nouvelle implémentation de la fonction `evenInside` de l'exercice 1. Pour éviter les collisions de noms, vous devez nommer votre fonction `evenInside'`.

(b) En utilisant obligatoirement des fonctions d'ordre supérieur écrire une nouvelle implémentation de la fonction `countFixedPoints` de l'exercice 1. Pour éviter les collisions de noms, vous devez nommer votre fonction `countFixedPoints'`.

(c) Dans une liste d'entiers, on appelle *pic* un élément strictement supérieur à ses deux voisins immédiats. Le premier et le dernier élément ne peuvent donc pas être des pics.

En utilisant obligatoirement une fonction récursive, écrire la fonction

```
countPeaks :: [Int] -> Int
```

qui calcule le nombre de pics dans une liste d'entiers.

```
λ > countPeaks []
0
λ > countPeaks [1]
0
λ > countPeaks [1,2]
0
λ > countPeaks [1,2,3]
0
λ > countPeaks [1,2,1]
1
λ > countPeaks [1,2,3,2,1,2,1,4,6,8,5]
3
```

(d) Écrire la fonction :

```
unique :: Eq a => [a] -> [a]
```

`unique xs` retourne la liste des éléments de `xs` sans doublons, en conservant l'ordre d'apparition des éléments dans la liste initiale. Vous ne pouvez pas utiliser la fonction `Data.List.nub`.

```
λ > unique []
[]
```

```
λ > unique [1,2,3]
[1,2,3]
λ > unique [1,1,2,3,3,1,1,1,1,2,3]
[1,2,3]
```

(e) Pour cette question, vous devez obligatoirement utiliser la fonction `foldr`.

Donnez une nouvelle implémentation de la fonction `filter :: (a -> Bool) -> [a] -> [a]`. Pour éviter les collisions de noms, vous devez nommer votre fonction `filter'`.

(f) Pour cette question, vous devez obligatoirement utliser la fonction `foldl`.

On imagine un petit robot posé sur une ligne graduée horizontale. Sa position de départ est 0. On lui envoie une série de commandes sous forme d'une chaîne de caractères, et on veut savoir où il se trouve à la fin de son voyage.

Les commandes possibles sont les suivantes :
— `L` : le robot se déplace d'une unité vers la gauche (i.e. sa position diminue de 1),
— `R` : le robot se déplace d'une unité vers la droite (i.e. sa position augmente de 1),
— tout autre caractère est ignoré.
Écrire la fonction :

```
driveRover :: String -> Int
```

`driveRover s` calcule la position finale du robot après avoir exécuté les commandes de la chaîne `s`.

```
λ > driveRover ""
0
λ > driveRover "LLRR"
0
λ > driveRover "RRLL"
0
λ > driveRover "RRLLXRRYLL"
0
λ > driveRover "LLLLLR"
-4
λ > driveRover "RRRLRRRRLL"
4
```

Exercice 3 : Mesures de capteurs et filtres numériques

Dans cet exercice, on considère un capteur de mesures (par exemple un capteur de température, un sismographe ou un capteur audio) qui produit une série de valeurs brutes représentées par une liste de `Float`.

Ces données étant généralement bruitées, elles doivent être traitées avant analyse.

Dans la suite, on appelle filtre numérique toute fonction de type `[Float] -> [Float]` qui transforme une liste de mesures d'entrée en une nouvelle liste de mesures de sortie.

(a) Un *amplificateur* est un filtre numérique qui multiplie chaque mesure par une constante donnée. Écrire la fonction :

```
amplify :: Float -> [Float] -> [Float]
```

`amplify k xs` retourne la liste obtenue en multipliant chaque élément de `xs` par la constante `k`.

```
λ > amplify 2 [1.0, 2.0, 3.0]
[2.0, 4.0, 6.0]
λ > amplify 0.5 [1.0, 2.0, 3.0]
[0.5, 1.0, 1.5]
```

(b) Un *noise gate* est un filtre numérique qui supprime les mesures dont la valeur absolue est inférieure à un seuil donné.

Écrire la fonction : `noiseGate :: Float -> [Float] -> [Float]`

`noiseGate t xs` retourne la liste obtenue en remplaçant par 0 chaque élément de `xs` dont la valeur absolue est inférieure à `t`, et en laissant les autres éléments inchangés.

```
λ > noiseGate 0.5 [0.0, 0.3, 0.5, 0.7, -0.2, -0.6]
[0.0, 0.0, 0.5, 0.7, 0.0, -0.6]
```

(c) Un *filtre de seuil* est un filtre numérique paramétré par deux seuils `tMin` et `tMax` et qui traite chaque mesure `x` de la façon suivante :
— si `x` est inférieure à `tMin`, alors la mesure de sortie est `tMin`,
— si `x` est supérieure à `tMax`, alors la mesure de sortie est `tMax`,
— sinon, la mesure de sortie est inchangée (i.e. égale à `x`).
On suppose que `tMin` est strictement inférieur à `tMax`.

Écrire la fonction :

```
clipping :: Float -> Float -> [Float] -> [Float]
```

`clipping min max xs` retourne la liste obtenue en appliquant le filtre de seuil avec les seuils `min` et `max` à chaque élément de `xs`.

```
λ > clipping 2.5 3.5 [1.0, 2.0, 3.0, 4.0]
[2.5, 2.5, 3.0, 3.5]
λ > clipping 3.5 4.5 [1.0, 2.0, 3.0, 4.0]
[3.5, 3.5, 3.5, 4.0]
```

(d) Un *réducteur de bruit* est un filtre numérique qui remplace chaque mesure par la moyenne de cette mesure et de son ou ses voisins immédiats. Par exemple, si la liste de mesures est `[1.0, 2.0, 3.0, 4.0]`, alors la liste de sortie est `[1.5, 2.0, 3.0, 3.5]`. Écrire la fonction :

```
smooth :: [Float] -> [Float]
```

`smooth xs` retourne la liste obtenue en appliquant le réducteur de bruit à chaque élément de `xs`.

```
λ > smooth [1.0, 2.0, 3.0, 7.0]
[1.5,2.0,4.0,5.0]
λ > smooth [1.0, 2.0, 6.0]
[1.5,3.0,4.0]
λ > smooth [1.0, 2.0]
[1.5,1.5]
```

(e) Un *filtre de lissage exponentiel* est paramétré par un facteur $\alpha \in [0, 1]$. La première mesure de sortie y_0 est égale à la première mesure d'entrée x_0. Pour les mesures suivantes, on calcule :

$$y_n = \alpha \cdot x_n + (1 - \alpha) \cdot y_{n-1}$$

C'est un filtre très utilisé en audio et en électronique pour supprimer les hautes fréquences de manière plus douce que le smooth.

Écrire la fonction :

```
exponentialSmooth :: Float -> [Float] -> [Float]
```

`exponentialSmooth alpha xs` retourne la liste obtenue en appliquant le filtre de lissage exponentiel avec le facteur α à la liste de mesures `xs`.

```
λ > exponentialSmooth 0.5 [1.0, 2.0, 3.0, 4.0]
[1.0, 1.5, 2.25, 3.125]
λ > exponentialSmooth 0.8 [1.0, 2.0, 3.0, 4.0]
[1.0,1.8,2.76,3.752]
```

(f) Un *filtre par convolution* est un processus où chaque mesure de sortie est calculée comme une somme pondérée d'une fenêtre de mesures d'entrée. Ce filtre est défini par une liste de poids $[w_1, w_2, \ldots, w_k]$ appelée *noyau*. Pour chaque position possible dans la liste de mesures, on aligne le noyau avec k mesures consécutives partir de cette position, on multiplie chaque mesure par le poids correspondant, et on fait la somme.

Écrire la fonction :

```
convolve :: [Float] -> [Float] -> [Float]
```

`convolve weights xs` retourne la liste des sommes pondérées. La liste résultante aura pour longueur $L - k + 1$ (où L est la longueur de `xs` et k celle de `weights`). Si $L < k$, la fonction retourne une liste vide.

```
λ> convolve [0.5, 0.5] [10.0, 20.0, 30.0, 40.0]  -- moyenne glissante sur 2 éléments
[15.0, 25.0, 35.0]
λ> convolve [-1.0, 1.0] [10.0, 10.0, 25.0, 25.0] -- filtre de détection de saut
[0.0,15.0,0.0]
```

(g) Un *traitement* est une fonction qui prend une liste de filtres numériques et une liste de mesures, et qui retourne la liste de mesures obtenue en appliquant successivement tous les filtres de la liste à chaque élement de la liste de mesures. Les filtres numériques doivent être appliqués dans l'ordre d'apparition dans la liste de filtres. Par exemple, si la liste de filtres est `[amplify 2, smooth]` et la liste de mesures est `[1.0, 2.0, 3.0]`, alors la liste de sortie est `[2.5, 4.0, 6.5]` (on applique d'abord le filtre `amplify 2` pour obtenir `[2.0, 4.0, 6.0]`, puis on applique le filtre `smooth` pour obtenir `[2.5, 4.0, 6.5]`).

Écrire la fonction :

```
applyFilters :: [[Float] -> [Float]] -> [Float] -> [Float]
```

`applyFilters fs xs` retourne la liste obtenue en appliquant successivement tous les filtres de la liste `fs` à la liste de mesures `xs`.

```
λ> applyFilters [amplify 2, smooth] [1.0, 2.0, 3.0]
[3.0, 4.0, 5.0]
λ> applyFilters [clipping 2.5 3.5, amplify 2] [1.0, 2.0, 3.0, 4.0]
[5.0, 5.0, 6.0, 7.0]
λ> applyFilters [amplifier 1.25, convolve [0.5, 0.5, 0.5]] [i + 0.5 | i <- [1..10]]
[4.6875,6.5625,8.4375,10.3125,12.1875,14.0625,15.9375,17.8125]
```

Exercice 4 : Les pancakes déterministes

Le *problème des pancakes* a été introduit par Donald Knuth dans les années 70 comme un problème d'algorithmique amusant et pédagogique. Dans sa version classique, on dispose d'une pile de pancakes de tailles différentes, et on cherche à les trier du plus grand en bas au plus petit en haut, en utilisant uniquement une opération appelée *flip* : on retourne les k premiers pancakes (ceux du haut) de la pile à l'aide d'une spatule. Par exemple, si la pile initiale est

$$5, 3, 8, 2, 9, 1, 7, 4, 6$$

(le pancake du haut porte le numéro 5, celui en dessous porte le numéro 3, etc.), et que l'on effectue un flip avec $k = 4$, on obtient la pile

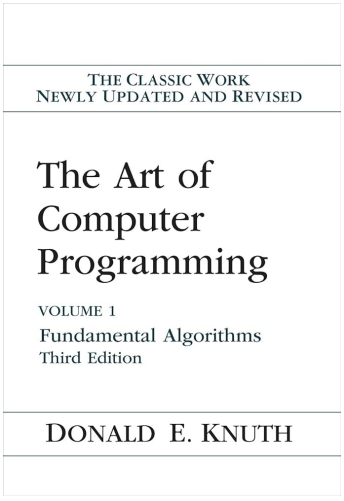
$$2, 8, 3, 5, 9, 1, 7, 4, 6$$

(les 4 premiers pancakes ont été retournés).

Dans la version *déterministe simplifiée* (la version qui nous concerne aujourd'hui), on s'intéresse uniquement à placer le pancake numéro 1 en tête de la pile. De plus, le processus est complètement déterministe :

- On commence avec une liste d'entiers représentant les pancakes empilés les uns sur les autres. Par exemple, la liste `[5,3,8,2,9,1,7,4,6]` représente une pile de pancakes où le pancake du haut porte le numéro 5, le pancake en dessous porte le numéro 3, etc.
- On effectue une série de flips selon la procédure suivante :
  - On considère uniquement le pancake en haut de la pile, et on note son numéro k.
  - On retourne les k premiers pancakes de la pile.
  - On répète le processus et on s'arrête lorsque le pancake numéro 1 est en tête de la pile [1].

1. Petite remarque de culture générale : même si la terminaison de ce processus est garantie (*i.e.*, on peut prouver que le pancake numéro 1 finit toujours par arriver en première position), on ne sait pas si le nombre d'étapes nécessaires est toujours borné par un polynôme en la taille n de la pile. Autrement dit, on ignore si, dans le pire des cas, cet algorithme pourrait nécessiter un nombre d'opérations qui croît très rapidement (par exemple de manière exponentielle) par rapport à la taille de la permutation.



Par exemple, en partant de la pile initiale

$$5, 3, 8, 2, 9, 1, 7, 4, 6$$

on effectue les flips suivants :

$$9, 2, 8, 3, 5, 1, 7, 4, 6$$
$$6, 4, 7, 1, 5, 3, 8, 2, 9$$
$$3, 5, 1, 7, 4, 6, 8, 2, 9$$
$$1, 5, 3, 7, 4, 6, 8, 2, 9$$

et on s'arrête car le pancake numéro 1 est en haut de la pile.

(a) Écrire le prédicat

```
isPancakeStack :: [Int] -> Bool
```

qui retourne vrai si et seulement la liste passée en paramètre est une pile de pancakes, c'est à dire une permutation de 1 à n où n est la longueur de la liste (chaque élément doit apparaître exactement une fois).

```
λ > isPancakeStack []
True
λ > isPancakeStack [2]
False
λ > isPancakeStack [1,2]
True
λ > isPancakeStack [1,3]
False
λ > isPancakeStack [5,3,8,2,9,1,7,4,6]
True
```

(b) Écrire la fonction

```
reversePrefix :: Int -> [a] -> [a]
```

qui retourne une nouvelle liste où les k premiers éléments d'une liste donnée sont renversés. Si l'entier k est plus grand que la longueur de la liste, on renverse toute la liste. Si l'entier k est négatif ou nul, on retourne la liste inchangée.

```
λ > reversePrefix 0 [5,3,8,2,9,1,7,4,6]      -- liste inchangee
[5,3,8,2,9,1,7,4,6]
λ > reversePrefix 2 [5,3,8,2,9,1,7,4,6]
[3,5,8,2,9,1,7,4,6]
```

```
λ > reversePrefix 4 [5,3,8,2,9,1,7,4,6]
[2,8,3,5,9,1,7,4,6]
λ > reversePrefix 9 [5,3,8,2,9,1,7,4,6]
[6,4,7,1,9,2,8,3,5]
λ > reversePrefix 10 [5,3,8,2,9,1,7,4,6]
[6,4,7,1,9,2,8,3,5]
```

(c) Écrire la fonction

```
dPancakeFlipping :: [Int] -> [[Int]]
```

qui retourne la liste des étapes successives du processus de flips déterministes, en commençant par la pile initiale. La liste retournée doit contenir la pile initiale, puis chaque pile obtenue après chaque flip, et se terminer par la pile finale où le pancake numéro 1 est en tête de la pile. Cet ordre est imposé (il doit donc être respecté).

```
λ > dPancakeFlipping [5,3,8,2,9,1,7,4,6]
[[5,3,8,2,9,1,7,4,6],
 [9,2,8,3,5,1,7,4,6],
 [6,4,7,1,5,3,8,2,9],
 [3,5,1,7,4,6,8,2,9],
 [1,5,3,7,4,6,8,2,9]]
λ > dPancakeFlipping [1,4,3,2,8,6,5,7,9]
[[1,4,3,2,8,6,5,7,9]]
```