Programmation fonctionnelle (L3)
TP-04
TP noté (1h30)

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

15 décembre 2023

Les 4 exercices sont indépendants. Le barème est donné uniquement à titre indicatif. Seuls les documents du cours et une page recto-verso manuscrite sont autorisés.

Votre TP doit impérativement se trouver dans le répertoire `EXAM` sous la forme d'un unique fichier nommé `TPNote.hs`.

Exercice 1 : Rejet

(a) (1 point) Écrire le prédicat

```
isUpperChar :: Char -> Bool
```

qui décide si un caractère est une lettre majuscule.

```
λ: isUpperChar 'K'
True
λ: isUpperChar 'k'
False
λ: and (map isUpperChar ['A'..'Z'])
True
λ: or (map isUpperChar "[\\]^_`./?=%*$-+")
False
```

(b) (4 points) Nous souhaitons écrire une fonction

```
rejectNullOrUpperFirstChar :: [String] -> [String]
```

qui supprime dans une liste toute les chaînes de caractères nulles ainsi que celles qui commencent par une lettre majuscule.

```
λ: rejectNullOrUpperFirstChar []
[]
λ: rejectNullOrUpperFirstChar ["aB", "", "Cd", "eFg", "HIH", "", "Klmn", "opq", "123"]
["aB","eFg","opq","123"]
```

(a) (1 point) Écrire une version `rejectNullOrUpperFirstChar1` de la fonction en utilisant obligatoirement une fonction récursive avec des gardes.

(b) (1 point) Écrire une version `rejectNullOrUpperFirstChar2` de la fonction en utilisant obligatoirement une compréhension de liste.

(c) (1 point) Écrire une version `rejectNullOrUpperFirstChar3` de la fonction en utilisant obligatoirement la fonction `filter`.

(d) (1 point) Écrire une version `rejectNullOrUpperFirstChar4` de la fonction en utilisant obligatoirement un `fold`.

Exercice 2 : Boîte à outils

(a) (1 point) Écrire la fonction

```
chunk3 :: [a] -> [[a]]
```

qui structure une liste donnée en sous-listes de longueur trois, sauf la dernière sous-liste qui peut éventuellement comporter moins de trois éléments (c'est la cas si la longueur de la liste considérée n'est pas un multiple de trois).

```
λ: chunk3 [1..9]
[[1,2,3],[4,5,6],[7,8,9]]
λ: chunk3 [1..10]
[[1,2,3],[4,5,6],[7,8,9],[10]]
λ: chunk3 [1..11]
[[1,2,3],[4,5,6],[7,8,9],[10,11]]
λ: chunk3 [1..12]
[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
λ: chunk3 [1..13]
[[1,2,3],[4,5,6],[7,8,9],[10,11,12],[13]]
```

(b) (1 point) Écrire la fonction

```
pairsOfConsecutiveEvens :: Integral a => [a] -> [(a, a)]
```

qui retourne la liste des paires non chevauchantes d'entiers pairs consécutifs (en position). Autrement dit, les entiers impairs sont écartés et les entiers pairs consécutifs sont groupés deux par deux (si le nombre d'entiers pairs consécutifs est impair, le dernier est écarté).

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

(c) (1 point) Écrire la fonction

```
pairAndSwapConsecutives :: [a] -> [(a, a)]
```

qui groupe les éléments consécutifs par paires inversées chevauchantes.

```
λ: pairAndSwapConsecutives1 []
[]
λ: pairAndSwapConsecutives1 [1]
[]
λ: pairAndSwapConsecutives1 [1,2]
[(2,1)]
λ: pairAndSwapConsecutives1 [1,2,3,4,5,6,7,8,9]
[(2,1),(3,2),(4,3),(5,4),(6,5),(7,6),(8,7),(9,8)]
```

(d) (2 points) Écrire la fonction

```
runs :: Ord a => [a] -> [[a]]
```

qui structure une liste en sous-listes d'éléments consécutifs strictement croissants.

```
λ: runs [1,2,3,4,5,6,7,8,9]
[[1,2,3,4,5,6,7,8,9]]
λ: runs [1,2,1,4,5,3,3,5,6,5,8,9]
[[1,2],[1,4,5],[3],[3,5,6],[5,8,9]]
λ: runs [9,8,7,6,5,4,3,2,1]
[[9],[8],[7],[6],[5],[4],[3],[2],[1]]
```

Exercice 3 : Sommes cumulées

Nous souhaitons écrire la fonction

```
sumCumFun :: (Int -> Int) -> Int -> [Int]
```

L'appel `sumCumFun f n` calcule les sommes cumulées de la fonction `f :: Int -> Int` pour tous les entiers allant de `1` à `n`. Autrement dit, pour une fonction `f` et un entier `n`, le `k`-ème élément ($1 \leqslant k \leqslant n$) de la liste calculée par la fonction `sumCumFun f n` est :

$$\sum_{i=1}^{k} \texttt{f i}.$$

```
λ: sumCumFun (\x -> x) 5           -- ou plus simplement sumCumFun id 5
[1,3,6,10,15]
λ: [1,1+2,1+2+3,1+2+3+4,1+2+3+4+5] -- pour vérifier
[1,3,6,10,15]
λ: sumCumFun (\x -> x*x) 5
[1,5,14,30,55]
λ: sumCumFun (\x -> 2*x) 5         -- ou plus simplement sumCumFun (2*) 5
[2,6,12,20,30]
λ: sumCumFun (+ 1) 5
[2,5,9,14,20]
```

(a) (1.5 points) Écrire une version `sumCumFun1` de la fonction à l'aide de compréhensions de listes.

(b) (2 points) Écrire une version `sumCumFun2` de la fonction à l'aide d'une fonction récursive.

(c) (2 points) Écrire une version `sumCumFun3` de la fonction à l'aide d'une ou plusieurs fonctions d'ordre supérieur.

Exercice 4 : Tri

Considérons un algorithme de tri qui consiste à comparer répétitivement les éléments consécutifs dans une liste, et à les permuter lorsqu'ils sont mal triés. Ce tri (fonction `examSort`) sera mis en œuvre à l'aide de la fonction utilitaire `examSortStep`.

(a) (1 point) Écrire la fonction

```
examSortStep :: Ord a => a -> [a] -> [a]
```

La fonction `examSortStep x xs` parcourt la liste `x:xs` et compare les éléments consécutifs. Lorsque deux éléments consécutifs ne sont pas dans l'ordre, ils sont permutés. On remarquera que, par définition, si `xs` est une liste triée alors `examSortStep x xs` produit une liste entièrement triée (c'est le schéma récursif de notre algorithme de tri).

```
λ: examSortStep 5 [6,7,8,9,1,2,3,4]
[5,6,7,8,1,2,3,4,9]
λ: examSortStep 1 [4,9,6,3,7,8,2,5]
[1,4,6,3,7,8,2,5,9]
λ: examSortStep 9 [4,1,6,3,7,8,2,5]
[4,1,6,3,7,8,2,5,9]
λ: examSortStep 5 [1,2,3,4,6,7,8,9]
[1,2,3,4,5,6,7,8,9]
λ: examSortStep 3 []
[3]
```

(b) (1.5 points) Écrire la fonction

```
examSort :: Ord a => [a] -> [a]
```

qui trie une liste d'éléments à l'aide de notre algorithme du tri.

```
λ: examSort [5,4,6,3,7,9,1,2,8]
[1,2,3,4,5,6,7,8,9]
```

(c) (2 points) En déduire l'écriture des fonctions

```
examSortFunStep :: Ord a => (a -> a -> Bool) -> a -> [a] -> [a]
```

et

```
examSortFun :: Ord a => (a -> a -> Bool) -> [a] -> [a]
```

qui trie une liste d'éléments à l'aide de notre algorithme de tri paramétré par une fonction qui, pour deux éléments `x` et `y`, retourne vrai si et seulement si `x < y` pour l'ordre < considéré. Par exemple, en considérant les trois fonctions suivantes :

```
descendingOrder :: Ord a => a -> a -> Bool
descendingOrder x y = x > y

oddEvenOrderThenAscendingOrder :: Integral a => a -> a -> Bool
oddEvenOrderThenAscendingOrder x y
  | odd  x && even y = True
  | even x && odd  y = False
  | otherwise        = x < y

mod3OrderThenAscendingOrder :: Integral a => a -> a -> Bool
mod3OrderThenAscendingOrder x y
  | x `mod` 3  < y `mod` 3 = True
  | x `mod` 3 == y `mod` 3 = x < y
  | otherwise              = False
```

nous obtenons :

```
λ: examSortFun descendingOrder                [5,4,6,3,7,9,1,2,8]
[9,8,7,6,5,4,3,2,1]
λ: examSortFun oddEvenOrderThenAscendingOrder [5,4,6,3,7,9,1,2,8]
[1,3,5,7,9,2,4,6,8]
λ: examSortFun mod3OrderThenAscendingOrder    [5,4,6,3,7,9,1,2,8]
[3,6,9,1,4,7,2,5,8]
```

Bonus. Implémentez la fonction `examSort` à l'aide de la fonction `examSortFun`.