Programmation fonctionnelle
TP-Noté
Durée de l'épreuve 01h30

Stéphane Vialette stephane.vialette@univ-eiffel.fr

16 janvier 2024

Rappel : L'ensemble des fonctions du module `Data.List` est accessible via la directive d'importation :

```
import Data.List
```

Exercice 1 : Sous-listes

(a) Sans utiliser la fonction `Data.List.subsequences`, écrire la fonction récursive
sublists : : [a] -> [[a]]
qui calcule toutes les sous-listes (*i.e*, suite d'éléments non nécessairement consécutifs) d'une liste. La liste vide `[]` est, par définition, sous-liste de toute liste. Aucun ordre des sous-listes n'est imposé.

```
λ: sublists []
[[]]
λ: sublists [1]
[[],[1]]
λ: sublists [1,2]
[[],[2],[1],[1,2]]
λ: sublists [1,3,2]
[[],[2],[3],[3,2],[1],[1,2],[1,3],[1,3,2]]
λ: sublists [1,2,2]
[[],[2],[2],[2,2],[1],[1,2],[1,2],[1,2,2]]
λ: [(n, length (sublists [1..n])) | n <- [1..10]]
[(1,2),(2,4),(3,8),(4,16),(5,32),(6,64),(7,128),(8,256),(9,512),(10,1024)]
```

(b) Sans utiliser la fonction `Data.List.subsequences`, écrire la fonction récursive
`fixedSublists :: Int -> [a] -> [[a]]`
qui calcule toutes les sous-listes de taille `k` donnée d'une liste. Si `k < 0` ou si `k` est supérieur à la longueur de la liste sur laquelle la fonction est appliquée, alors la fonction retourne la liste vide. Aucun ordre des sous-listes n'est imposé.

```
λ: fixedSublists (-1) [1,2,3,4]
[]
λ: fixedSublists 1 [1,2,3,4]
[[1],[2],[3],[4]]
λ: fixedSublists 2 [1,2,3,4]
[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
λ: fixedSublists 3 [1,2,3,4]
[[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
λ: fixedSublists 4 [1,2,3,4]
[[1,2,3,4]]
```

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

(c) En utilisant obligatoirement une fonction d'ordre supérieur, écrire le prédicat
`isIncreasing ::(Ord a) => [a] -> Bool`
qui décide si une liste est strictement croissante. La liste vide est une liste strictement croissante.

```
λ: isIncreasing []
True
λ: isIncreasing [1]
True
λ: isIncreasing [1,3,5,8]
True
λ: isIncreasing [3,5,1,8]
False
```

(d) Écrire la fonction
`increasingSublists:: (Ord a) => [a] -> [[a]]`
qui calcule toutes les sous-listes strictement croissantes d'une liste.

```
λ: increasingSublists []
[[]]
λ: increasingSublists [1]
[[],[1]]
λ: increasingSublists [1,2]
[[],[2],[1],[1,2]]
λ: increasingSublists [1,3,2]
[[],[2],[3],[1],[1,2],[1,3]]
λ: increasingSublists [1,3,4,2]
[[],[2],[4],[3],[3,4],[1],[1,2],[1,4],[1,3],[1,3,4]]
λ: increasingSublists [1,2,1,3]
[[],[3],[1],[1,3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
λ: prefixFreeSublists [[1],[1],[]]
[[1]]
```

(e) Rappelons que la fonction
`Data.List.isPrefixOf :: Eq a => [a] -> [a] -> Bool`
retourne vrai si la première liste est préfixe de la seconde. Écrire la fonction
`prefixFreeSublists :: (Ord a) => [[a]] -> [[a]]`
`prefixFreeSublist xss` calcule la plus longue sous-liste de `xss` dont les éléments sont non comparables par la relation de préfixe. La liste vide est un préfixe de toute liste

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

(f) Écrire la fonction
`maximalIncreasingSublists :: (Eq a, Ord a) => [a] -> [[a]]`
qui calcule toutes les sous-listes maximales (*i.e.* de longueur maximale et non comparables par la relation de préfixe) strictement croissantes d'une liste.

```
λ: increasingSublists [1,3,2,4]
[[],[4],[2],[2,4],[3],[3,4],[1],[1,4],[1,2],[1,2,4],[1,3],[1,3,4]]
λ: maximalIncreasingSublists [1,3,2,4]
[[4],[2,4],[3,4],[1,4],[1,2,4],[1,3,4]]
λ: increasingSublists [4,2,1,3]
[[],[3],[1],[1,3],[2],[2,3],[4]]
λ: maximalIncreasingSublists [4,2,1,3]
[[3],[1,3],[2,3],[4]]
```

(g) Écrire la fonction

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

qui calcule toutes les sous-listes maximales strictement croissantes communes entre deux listes.

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

Exercice 2 : Codage et décodages (simples)

Nous considérons dans la suite un système de codage et décodage - très (trop ?) simple - d'un message. Il consiste à itérer à un nombre fixé de fois l'algorithme suivant (appelé algorithme A dans la suite de l'exercice) : partant d'un message `xss` (ce sera une liste dans cet exercice), concaténer les éléments de `xss` qui sont en position impaire (en conservant l'ordre des éléments dans `xss`) aux éléments de `xss` qui sont en position paire (en conservant encore une fois l'ordre des éléments dans `xss`). Le premier élément d'un message est en position 0 (donc en position paire).

Par exemple, pour `xss = "abcdefgh"`, l'algorithme A calcule `"bdfhaceg"`. En effet, les lettres de `xss` qui sont en position impaire forment le mot `"bdfh"` et les lettres de `xss` qui sont en position paire forment le mot `"aceg"`. La concaténation de ces deux mots produit bien le mot `"bdfhaceg"`.

L'algorithme de codage complet est paramétré par un entier `k` qui indique le nombre d'itérations de l'Algorithme A à appliquer. Par exemple, si `k = 4`, l'algorithme de codage pour le message `xss = "abcdefgh"` calcule `"gecahfdb"`. En effet, nous obtenons en appliquant 4 fois l'algorithme A :

`"abcdefgh" -> "bdfhaceg" -> "dhcgbfae" -> "hgfedcba" -> "gecahfdb"`.

(a) Écrire la fonction

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

qui à partir d'une liste `xs` calcule la paire `(os, es)`, où `os` (resp. `es`) est la liste des éléments de `xs` qui sont en position impaire (resp. paire) dans le même ordre que dans `xs`.

```
λ: splitByParityIndex []
([],[])
λ: splitByParityIndex "a"
("","a")
λ: splitByParityIndex "ab"
("b","a")
λ: splitByParityIndex "abcdefg"
("bdf","aceg")
```

(b) En déduire l'écriture de la fonction

```
encode :: (Ord t, Num t) => [a] -> t -> [a]
```

qui code un message en appliquant un nombre fixé de fois l'algorithme A.

```
λ: encode "abcdefg" 0
"abcdefg"
λ: encode "abcdefg" 1
"bdfaceg"
λ: encode "abcdefg" 2
"daebfcg"
```

(c) Écrire la fonction

```
perfectShuffle :: [a] -> [a] -> [a]
```

qui entrelace deux listes. L'entracement des listes `xs` et `ys` est la liste qui débute par le premier élément de `xs`, suivi du premier élément de `ys`, suivi du deuxième élément de `xs`, suivi du deuxième élément de `ys`, ... Si une liste est plus courte qu'une autre, les éléments surnuméraires sont placés à la suite en fin de liste.

```
λ: perfectShuffle "abcd" "MNOP"
"aMbNcOdP"
λ: perfectShuffle "abcde" "MNOP"
"aMbNcOdPe"
λ: perfectShuffle "ab" "MNOP"
"aMbNOP"
λ: perfectShuffle "abcd" "MN"
"aMbNcd"
```

(d) En déduire l'écriture de la fonction

```
decode :: (Ord t, Num t) => [a] -> t -> [a]
```

qui décode un message préallablement codé par un nombre connu d'itérations de l'algorithme A.

```
λ: encode "abcdefg" 1
"bdfaceg"
λ: decode "bdfaceg" 1
"abcdefg"
λ: encode "abcdefg" 2
"daebfcg"
λ: decode "daebfcg" 2
"abcdefg"
λ: let xs = "abcdefgh" in [xs == decode (encode xs k) k | k <- [0..4]]
[True,True,True,True,True]
```

(e) Un crypto-analyste de votre service a laissé la note suivante sur votre bureau hier soir :

```
-- Too easy to crack !
easyCrack :: Eq a => [a] -> [[a]]
easyCrack xs = let xss = iterate (flip encode 1) xs in xs:f xss
  where
    f = takeWhile (/= xs) . tail
```

Que calcule cette fonction ? Qu'en concluez-vous expérimentalement ?