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

Stéphane Vialette stephane.vialette@univ-eiffel.fr

17 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)]
```

Solution:
Trouvons le schéma récursif.

- Cas d'arrêt : La liste vide contient la liste vide comme unique sous-liste. Attention il s'agit de la liste qui contient la liste vide (c-à-d `[[]]`) et non de la liste vide (c-à-d `[]`).
- Récurrence : Considérons une liste `xs` dont le premier élément est `x` suivi de la liste `xs'`. Une sous-liste de `xs` contient ou non l'élément `x`. Notons `xss` toutes les sous-listes de `xs'`. Les sous-listes de `xs` sont les sous-listes de `xs'` (c-à-d `xss`) plus les sous-listes de `xs'` auxquelles on ajoute `x` (c-à-d `map (x:) xss`).

Nous obtenons :

```
sublists :: [a] -> [[a]]
sublists []     = [[]]
```

```haskell
sublists (x:xs) = xss ++ map (x:) xss
  where
    xss = sublists xs
```

Une écriture équivalente :

```haskell
sublists :: [a] -> [[a]]
sublists []     = [[]]
sublists (x:xs) = xss ++ [x:xs | xs <- xss]
  where
    xss = sublists xs
```

Pourquoi est-il important que le cas d'arrêt soit `[[]]` et non `[]` ? C'est simple, `map (x:) [[]]` produit la liste `x:[]` (c-à-d `[x]`) alors que `map (x:) []` produit la liste vide (la fonction `map` n'applique la fonction `(x:)` à aucun élément ... puisque la liste est vide).
Pour les plus curieux d'entre-vous (vous pouvez consulter `https://stackoverflow.com/questions/7402528/whats-the-point-of-const-in-the-haskell-prelude` et `https://blog.ssanj.net/posts/2018-04-10-how-does-filterm-work-in-haskell.html` pour commencer) :

```haskell
import Control.Monad

sublists :: [a] -> [[a]]
sublists = filterM $ const [False, True]
```

(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]]
```

Solution:
Deux approches sont possibles. La première consiste à filtrer le calcul de la fonction `sublists`. La seconde est récursive (c-à-d que l'on ne calcule que les sous-listes de la bonne taille).
Pour la première solution, il vient :

```haskell
fixedSublists :: Int -> [a] -> [[a]]
fixedSublists k = filter ((==) k . length) . sublists
```

Pour la seconde solution, il faut conserver le nombre d'éléments qu'il reste à ajouter dans les sous-listes. Nous obtenons :

```haskell
fixedSublists :: Int -> [a] -> [[a]]
fixedSublists 0 []     = [[]]
fixedSublists _ []     = []
fixedSublists k (x:xs)
  | k < 0     = []
  | otherwise = map (x:) (fixedSublists (k-1) xs) ++ fixedSublists k xs
```

L'avantage de la seconde solution est que nous évitons la génération de nombreuses listes.

(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
```

Solution:
Pour vérifier qu'une liste est strictement croissante, il suffit de vérifier que les éléments consécutifs pris deux à deux sont strictement croissant. Une version récursive est immédiate :

```haskell
isIncreasing :: Ord a => [a] -> Bool
isIncreasing []        = True -- la liste vide est strict. croissante
isIncreasing [_]       = True -- la liste à un élément est strict. croissante
isIncreasing (x:x':xs) = x < x' && isIncreasing (x':xs)
```

Il est demandé ici d'utiliser une fonction d'ordre supérieur. Nous savons que nous devons considérer les éléments par paires, cela suggère l'utilisation de la fonction `zip`. Puisque nous devons comparer les éléments de chaque paire, la fonction `zipWith` combinée au prédictat `(<)` nous permettra de verifier toutes les paires. Nous obtenons :

```haskell
isIncreasing :: Ord a => [a] -> Bool
isIncreasing [] = True  -- la liste vide est strict. croissante
isIncreasing xs =  and $ zipWith (<) xs (tail xs)
```

Remarquez que pour utiliser sereinement la fonction `tail`, nous vérifions dans un premier temps que la liste n'est pas vide. Notez également l'utilisation de la fonction `and` :
`and :: Foldable t => t Bool -> Bool`
qui dans notre cas peut se comprendre comme :
`and :: [Bool] -> Bool`
Nous pouvons également préférer l'utilisation de `all` plutôt que celle de `and` (c'est, il est vrai, plus élégant ici) :
`Data.Foldable.all :: Foldable t => (a -> Bool) -> t a -> Bool`
qui dans notre cas peut se comprendre comme :
`Data.Foldable.all :: (a -> Bool) -> [a] -> Bool`
Nous obtenons alors (notez l'utilisation de la lambda) :

```haskell
isIncreasing :: Ord a => [a] -> Bool
isIncreasing [] = True  -- la liste vide est strict. croissante
isIncreasing xs = all (\(x, x') -> x < x') $ zip xs (tail xs)]
```

Pour les amoureux inconditionnels du `fold`, c'est évidemment possible :

```
isIncreasing'' :: Ord a => [a] -> Bool
isIncreasing'' []     = True
isIncreasing'' (x:xs) = case foldl s (Just x) xs of
  Nothing -> False
  Just _  -> True
  where
    s Nothing  x  = Nothing
    s (Just x) x' = if x < x' then Just x' else Nothing
```

Bof . . . ce n'est ni forcément très lisible ni particulièrement élégant ! Nous pouvons améliorer la lisibilité de la fonction en utilisant `Data.Maybe.isJust :: Maybe a -> Bool` :

```
import Data.Maybe

isIncreasing'' :: Ord a => [a] -> Bool
isIncreasing'' []     = True
isIncreasing'' (x:xs) = isJust $ foldl s (Just x) xs
  where
    s Nothing  x  = Nothing
    s (Just x) x' = if x < x' then Just x' else Nothing
```

Toujours pas vraiment satisfaisant !

(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]]
```

Solution:
D'une part nous savons générer toutes les sous-listes et d'autre part nous savons vérifier si une liste est stictement croissante. Il suffit donc d'appliquer un filtre sur toutes les sous-listes.
Il vient :

```
increasingSublists:: (Ord a) => [a] -> [[a]]
increasingSublists = filter isIncreasing . sublists
```

(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]]
```

Solution:

C'est un peu plus délicat que la fonction précédente. En fait il y a plusieurs réponses possibles valides à la question, choisissons de conserver les sous-listes les plus longues. Examinons une approche récursive.

- Cas d'arrêt : Le cas d'arrêt est clairement la liste vide qui produit la liste vide.
- Récurrence : Soit `xs` la première sous-liste et `xss` les sous-listes suivantes. S'il existe une sous-liste de `xss` dont `xs` est préfixe, il ne faut bien sûr pas conserver `xs` dans la solution. Sinon, il faut conserver `xs` et supprimer de `xss` toutes les sous-listes dont `xs` est préfixe.

Il vient :

```
prefixFreeSublists :: (Ord a) => [[a]] -> [[a]]
prefixFreeSublists [] = []
prefixFreeSublists (xs:xss)
  | any (isPrefixOf xs) xss = prefixFreeSublists xss
  | otherwise               = xs:prefixFreeSublists xss'
  where
    xss' = filter (\ ys -> not (ys `isPrefixOf` xs)) xss
```

(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]]
```

Solution:

Il suffit de combiner la génération des sous-listes strictement croissantes (c'est le rôle de la fonction `increasingSublists`) avec la fonction `increasinprefixFreeSublistsgSublists`. Nous obtenons :

```
maximalIncreasingSublists :: (Eq a, Ord a) => [a] -> [[a]]
maximalIncreasingSublists = prefixFreeSublists . increasingSublists
```

(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]]
```

Solution:

Nous savons calculer les sous-listes strictement croissantes. Il faut donc trouver les sous-listes communes puis calculer les éléments maximaux (ce que sait très bien faire la fonction `prefixFreeSublists`). Pour trouver les éléments communs, le plus simple est d'écrire une compréhension de liste : le générateur produit toutes les sous-listes strictement croissantes d'une première liste et nous utilisons un prédicat pour vérifier qu'elle est présente parmi les sous-listes strictement croissantes de la seconde liste.

Il vient :

```haskell
commonMaximalIncreasingSublists :: Ord a => [a] -> [a] -> [[a]]
commonMaximalIncreasingSublists xs ys = prefixFreeSublists [mixs | mixs <- mixss,
                                                                   mixs `elem` miyss]
  where
    mixss = increasingSublists xs
    miyss = increasingSublists ys
```

Nous sommes probablement d'accord, ce n'est pas l'algorithme le plus efficace pour calculer les sous-listes strictement croissantes communes ! Les plus curieux d'entre-vous peuvent jeter un œil dans `rosettacode` (`https://rosettacode.org/wiki/Longest_common_subsequence#Haskell`) pour y réflechir et découvrir l'utilisation des tableaux en programmation fonctionnelle :

```haskell
import Data.Array

lcs xs ys = a!(0,0) where
  n = length xs
  m = length ys
  a = array ((0,0),(n,m)) $ l1 ++ l2 ++ l3
  l1 = [((i,m),[]) | i <- [0..n]]
  l2 = [((n,j),[]) | j <- [0..m]]
  l3 = [((i,j), f x y i j) | (x,i) <- zip xs [0..], (y,j) <- zip ys [0..]]
  f x y i j
    | x == y    = x : a!(i+1,j+1)
    | otherwise = longest (a!(i,j+1)) (a!(i+1,j))
```

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")
```

Solution:
Nous pouvons utiliser deux fonctions internes mutuellement récursives : `goE` si la position courante est une position paire (ce sera donc la première fonction appelée car `0` est un chiffre pair) et `goO` si la position courante est une position impaire. Nous obtenons (en n'oubliant pas de renverser les deux listes au final puisque nous construisons les deux listes en ordre inverse) :

```
splitByParityIndex :: [a] -> ([a], [a])
splitByParityIndex = (\ (os, es) -> (reverse os, reverse es)) . goE ([], [])
  where
    goE (os, es) []     = (os, es)
    goE (os, es) (x:xs) = goO (os, x:es) xs

    goO  (os, es) []     = (os, es)
    goO  (os, es) (x:xs) = goE (x:os, es) xs
```

Les fonctions sont mutuellement récursives car une position impaire succède à un position paire . . . et inversement.
Bon . . . ce n'est tout de même pas très élégant ! Il est beaucoup plus judicieux d'indexer les éléments par leur position (c'est facile, il suffit d'appliquer `zip [0..]` à notre liste) et de filtrer séparément les éléments en position impaire et les éléments en position paire. Bien sûr, il ne faut pas oublier de supprimer tous les indexes avant de retourner notre calcul (c-à-d `map snd` sur chaque liste). Nous obtenons alors :

```
splitByParityIndex :: [a] -> ([a], [a])
splitByParityIndex xs = (map snd ios, map snd ies)
  where
      ixs = zip [0..] xs
      ios = filter (odd  . fst) ixs
      ies = filter (even . fst) ixs
```

Encore une fois, pour les plus curieux d'entre-vous :

```
import Control.Arrow

splitByParityIndex :: [a] -> ([a], [a])
splitByParityIndex = (proj *** proj) . (f odd &&& f even) . zip [0..]
  where
    f p  = filter (p . fst)
    proj = map snd
```

(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"
```

Solution:

Première solution, on utilise une simple fonction récursive pour appliquer l'algorithme A un nombre de fois fixé :

```
encode :: (Ord t, Num t) => [a] -> t -> [a]
encode xs k
  | k <= 0    = xs
  | otherwise = encode (os ++ es) (k-1)
  where
    (os, es) = splitByParityIndex xs
```

Aucune difficulté.

Une seconde approche, mais qui n'est pas totalement équivalente (en particulier, cette approche nécessite de modifier légèrement le type de la fonction). Nous itérons l'algorithme A et récupérons la solution. Nous obtenons (en mettant explicitement en avant l'algorithme A dans le code) :

```
encode :: [a] -> Int -> [a]
encode xs k = head . drop k $ iterate algoA xs
  where
    algoA xs = os ++ es
      where
          (os, es) = splitByParityIndex xs
```

ou de façon équivalente

```
encode :: [a] -> Int -> [a]
encode xs k = (!! max 0 k) $ iterate algoA xs
  where
    algoA xs = os ++ es
      where
          (os, es) = splitByParityIndex xs
```

Dans les deux cas, il faut un `Int` car le type de la fonction `take` est

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

et celui de la fonction `(!!)` est

```
take :: Int -> [a] -> [a].
```

(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"
```

Solution:
Inutile de chercher à faire très compliqué ici. Nous considérons les éléments deux par deux. Il vient :

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

(d) En déduire l'écriture de la fonction

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

qui décode un message préallablement codé par un nombre connu d'itérations de l'algorithme $\mathcal{A}$.

```
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]
```

Solution:
Nous savons que la première partie du mot est constituée des éléments en position impaire et que la seconde partie du mot est constituée des éléments en position paire. Il suffit ensuite de mélanger ces deux listes à l'aide de la fonction `perfectShuffle`. Il faut bien sûr appliquer cet algorithme un certain nombre de fois, une fonction récursive nous permettra de prendre cet élément en compte.
Nopus obtenons :

```haskell
decode :: (Ord t, Num t) => [a] -> t -> [a]
decode xs k
  | k <= 0    = xs
  | otherwise = decode (perfectShuffle es os) (k-1)
  where
    n        = length xs
    (os, es) = splitAt (length xs `div` 2) xs
```

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

```haskell
-- 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 ?

Solution:
La fonction itère l'algorithme d'encodage jusqu'à - éventuellement - rencontrer le mot initial.

```
λ: mapM_ print $ easyCrack "ABCDEFGHIJ"
"ABCDEFGHIJ"
"BDFHJACEGI"
"DHAEIBFJCG"
"HEBJGDAIFC"
"EJDICHBGAF"
"JIHGFEDCBA"
"IGECAJHFDB"
"GCJFBIEAHD"
"CFIADGJBEH"
"FAGBHCIDJE"
```

L'expérimentation suggère que l'algorithme de codage, itéré un nombre suffisant de fois, va produire le mot initial. Voyons expérimentalement le délai :

```
λ: mapM_ print [(length xs, length (easyCrack xs)) | xs <- inits ['A'..'Z']]
(0,1)
(1,1)
(2,2)
(3,2)
(4,4)
(5,4)
(6,3)
(7,3)
(8,6)
(9,6)
(10,10)
(11,10)
(12,12)
(13,12)
(14,4)
(15,4)
(16,8)
(17,8)
(18,18)
(19,18)
(20,6)
(21,6)
(22,11)
(23,11)
(24,20)
(25,20)
(26,18)
```

Intéressant, le nombre d'itérations pour retomber sur le mot initial n'est pas une fonction croissante de la taille du mot.