Programmation fonctionnelle (L3)
TP-02
Fonctions récursives sur les listes

Stéphane Vialette stephane.vialette@univ-eiffel.fr

16 novembre 2023

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

Dans cet exercice, vous pouvez supposer que les listes en paramètre ne sont pas vide. **Pour chaque fonction, commencer par écrire son type** (et vérifier ensuite en laissant haskell l'inférer).

(a) Écrire la fonction `head'` qui renvoie le premier élément d'une liste.

```
λ: head' [0]
0
λ: head' [1,2,3,4]
1
```

(b) Écrire la fonction `tail'` qui renvoie une liste privée de son premier élément.

```
λ: tail' [0]
[]
λ: tail' [1,2,3,4]
[2,3,4]
```

(c) Écrire la fonction `last'` qui renvoie le dernier élément d'une liste.

```
λ: last' [1,2,3,4]
4
```

(d) Que se passe-t-il si l'on utilise une des fonctions précédentes sur une liste vide ?

(e) Quel est le type d'une fonction `interval` qui prend un entier k en paramètre et renvoie la liste contenant les k premiers entiers strictement positifs ?

(f) Écrire la fonction `interval`.

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

(g) À votre avis quel aurait été son type si vous aviez laissé haskell l'inférer ?

(h) Quel est le type d'une fonction `interval'` qui prend deux entiers s et k en paramètre et renvoie la liste contenant les entiers $s, s+1, \ldots, k$?

(i) Écrire la fonction `interval'`

```
λ: interval' 2 5
[2,3,4,5]
λ: interval' 5 2
[]
```

Exercice 2 : Boite à outils

(a) Écrire la fonction suivante qui renvoie vrai si et seulement si une liste est un *palindrome*.

```
isPalindrome :: Eq a => [a] -> Bool
```

Indication : vous pouvez l'écrire récursivement (en utilisant des fonctions comme `head`, `last`, `tail` ou `init`,... par exemple) mais vous obtiendrez probablement une complexité en $O(n^2)$ alors qu'il est possible de le faire en $O(n)$...

```
λ: isPalindrome []
True
λ: isPalindrome [1]
True
λ: isPalindrome [1,2,1]
True
λ: isPalindrome [1,1,2]
False
```

(b) Écrire la fonction `pairs` qui groupe les éléments d'une liste par paires chevauchantes. Quel est son type ?

```
λ: pairs []
[]
λ: pairs [1]
[]
λ: pairs [1,2,3,4,5]
[(1,2),(2,3),(3,4),(4,5)]
```

(c) Écrire la fonction récursive `prefixes` qui retourne tous les préfixes (suite d'éléments consécutifs en partant du premier) non vides d'une liste. Quel est son type ?

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

(d) Écrire la fonction

`factors :: (Eq a) => [a] -> [[a]]` qui renvoie la liste de tous les facteurs (éléments consécutifs) distincts d'une liste, (i) de façon récursive, (ii) par compréhension de liste.

Indication : la fonction précédente peut être utile.

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

(e) Écrire la fonction

`subseqs :: (Eq a) => [a] -> [[a]]`

qui renvoie la liste de toutes les sous-listes d'une liste.

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

Exercice 3 : Suites arithmétiques

Une *suite arithmétique* est une suite dans laquelle chaque terme permet de déduire le suivant en lui ajoutant une constante appelée *raison*.

Cette définition peut s'écrire sous la forme d'une relation de récurrence, pour chaque indice n :

$$u_{n+1} = u_n + r$$

(a) Écrire, sous forme d'une fonction récursive, la fonction

`isArithSerie :: (Eq a, Num a) => [a] -> Bool`

qui renvoie vrai si la liste est une suite arithmétique.

```
λ: isArithSerie [2,5,8,11]
True
λ: isArithSerie [2,5,7,11]
False
```

(b) Écrire la fonction

`isConstant :: Eq a => [a] -> Bool`

qui renvoie vrai si tous les éléments d'une liste sont égaux. En déduire une nouvelle version de la fonction `isArithSerie`.

(c) Écrire la fonction

```
mkArithSerie :: (Eq a, Num a) => a -> a -> a -> [a]
```

qui calcule les n premier termes d'une suite arithmétique de premier terme u_0 et de raison r.

```
λ: let u0 = 10; r = 5; n = 4 in mkArithSerie u0 r n
[10,15,20,25]
```

Écrire ensuite une nouvelle version de la fonction `mkArithSerie` en utilisant la fonction `Data.List.iterate`.

Exercice 4 : Parité

(a) En utilisant uniquement la connaissance que (i) 0 est pair et (ii) 1 est impair, écrire les fonctions récursives

```
isEven :: (Eq a, Num a) => a -> Bool
```

et

```
isOdd :: (Eq a, Num a) => a -> Bool.
```

La fonction `isEven n` (resp `isOdd n`) renvoie vrai si et seulement si l'entier `n` est pair (resp. impair). Nous supposerons $n \geqslant 0$.

(b) Une liste est *parité alternante* si les termes consécutifs n'ont pas la même parité. La liste vide est parité alternante. Écrire les fonctions récursives

```
isEvenOddAlternating :: Integral a => [a] -> Bool
```

et

```
isOddEvenAlternating :: Integral a => [a] -> Bool.
```

La fonction `isEvenOddAlternating xs` (resp `isOddEvenAlternating xs`) renvoie vrai si et seulement si la liste `xs` est parité alternante et débutte par un entier pair (resp. impair).

Écrire maintenant fonction

```
isAlternating :: Integral a => [a] -> Bool
```

qui renvoie vrai si et seulement si la liste est parité alternante.

Exercice 5 : Tris

(a) Écrire la fonction `msplit` qui sépare une liste en deux, avec les élement d'indices pairs dans la première et les autres dans le seconde. Quel est son type ?

```
λ: msplit []
([],[])
λ: msplit [3,1,2,3,4,5,6]
([3,2,4,6],[1,3,5])
```

(b) Écrire la fonction `merge`. Quel est son type ?

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

(c) Écrire maintenant fonction `mergeSort` qui implémente un tri fusion (*merge sort*).

(d) Écrire la fonction `quickSort` qui implémente un *quicksort*. Pour simplifier, le pivot sera toujours choisi comme le premier élément de la liste. Conseil : la fusion est triviale, l'important c'est de savoir séparer les élements d'une liste en fonction de la valeur du pivot.

Exercice 6 : Bonus d'entraînement `Data.List`

Cet exercice consiste à ré-écrire certaines des fonctions existantes en haskell de façon récursive.

(a) Écrire la récursive fonction `sum` qui renvoie la somme des éléments d'une liste.

```
λ: sum' [1,2,3,4,5]
15
```

(b) Écrire la fonction récursive

```
maximum' :: Ord a => [a] -> a
```

qui renvoie un élément maximum dans une liste. Nous supposerons que la liste est non vide.

(c) Écrire la fonction `length'` qui renvoie la longueur d'une liste.

(d) Écrire la fonction

```
zip' :: [a] -> [b] -> [(a, b)]
```

qui associe les éléments correspondants de deux listes en des paire.

```
λ: zip' [1,2,3,4,5] ['a','b','c','d','e']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]
λ: zip' [1,2,3] ['a','b','c','d','e']
[(1,'a'),(2,'b'),(3,'c')]
λ: zip' [1,2,3,4,5] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
```

(e) Écrire la fonction

```
take' :: (Eq b, Num b) => b -> [a] -> [a]
```

`take' k xs` renvoie la liste contenant les `k` premiers éléments de la liste `xs`. Si `xs` contient moins de `k` éléments, `xs` est renvoyée.

```
λ: take' 3 [1,2,3,4,5]
[1,2,3]
λ: take' 3 [1,2]
[1,2]
λ: take' 0 [1,2,3,4,5]
[]
```

(f) Écrire la fonction

```
drop' :: (Eq b, Num b) => b -> [a] -> [a]
```

`drop' k xs` renvoie la liste `xs` privée de ses `k` premiers éléments. Si `xs` contient moins de `k` éléments, `[]` est renvoyée.

```
λ: drop' 3 [1,2,3,4,5]
[4,5]
λ: drop' 3 [1,2]
[]
λ: drop' 0 [1,2,3,4,5]
[1,2,3,4,5]
```

(g) Écrire la fonction

```
elem' :: Eq a => a -> [a] -> Bool
```

`elem' x xs` renvoie vrai si et seulement si l'élement `x` appraît dans `xs`.

```
λ: 3 `elem'` [1,5,2,4,3,6]
True
λ: 7 `elem'` [1,5,2,4,3,6]
False
```

(h) Écrire la fonction

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

qui renverse une liste.

```
λ: reverse' [1,2,3,4,5]
[5,4,3,2,1]
```