Programmation fonctionnelle (L3)
TP-Noté (session 2) – 2h

Antoine Meyer Carine Pivoteau
Fabian Reiter Stéphane Vialette

5 juillet 2023

Les trois exercices sont indépendants. Le sujet comporte 8 pages (il y a beaucoup d'exemples pour vous guider). 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 : Manipulation de listes

(a) Écrire la fonction

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

qui, étant donnée une liste, ne conserve qu'une seule occurrence de chaque suite d'éléments consécutifs identiques. La contrainte de classe `Ord a` ne peut pas être ajoutée.

```
λ: compress [1,2,2,3,3,3,2,2,2,2,1]
[1,2,3,2,1]
λ: compress . concat . take 4 $ repeat [1,2,3]
[1,2,3,1,2,3,1,2,3,1,2,3]
```

(b) Écrire la fonction

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

qui, pour un entier `k` et une liste `xs`, duplique `k` fois chaque élément de `xs`. Si `k` est négatif ou nul, la fonction retourne la liste vide.

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

(c) Écrire la fonction

```
flatten :: [[a]] -> [a]
```

qui transforme une liste de listes d'éléments en une liste contenant ces mêmes élément dans le même ordre.

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

(d) Écrire la fonction

```
minMax :: Ord a => [a] -> Maybe (a, a)
```

qui retourne l'élément minimum et l'élément maximum dans une liste. Si la liste est vide, la fonction retourne `Nothing`.

```
λ: minMax []
Nothing
```

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

(e) Le *FizzBuzz* est à l'origine un jeu pour apprendre aux enfants le principe de la division. Les enfants doivent énoncer les chiffres dans l'ordre et remplacer le nombre par *Fizz* s'il est divisible par 3 ou *Buzz* s'il est divisible par 5. Nous appelons `fizzBuzz` la liste infinie dont l'élément d'indice `i` (`i = 1,2,...`) est :
— la chaîne de caractères `"FizzBuzz"` si `i` est divisible par 3 et par 5,
— la chaîne de caractères `"Fizz"` si `i` est divisible par 3,
— la chaîne de caractères `"Buzz"` si `i` est divisible par 5, et
— la chaîne de caractères `"i"` si `i` n'est divisible ni par 3 ni par 5.
Définir la liste infinie

```
fizzBuzz :: [String]
```

```
λ: take 16 $ fizzBuzz
["1","2","Fizz","4","Buzz","Fizz","7","8","Fizz","Buzz","11","Fizz","13","14","FizzBuzz","16"]
λ: take 10 . drop 100 $ fizzBuzz
["101","Fizz","103","104","FizzBuzz","106","107","Fizz","109","Buzz"]
```

Exercice 2 : Expressions bien parenthésées

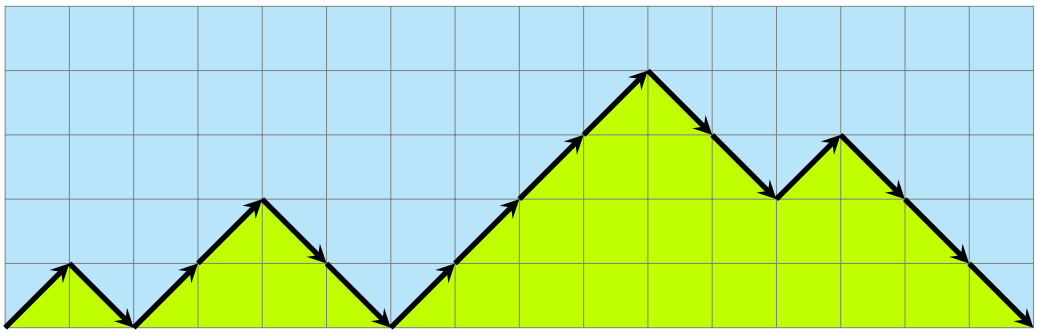
Une expression est *bien parenthésée* si le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes, et si, quelle que soit la position dans l'expression, parmi les symboles qui précédent cette position, le nombre de parenthèses ouvrantes est toujours supérieur ou égal au nombre de parenthèses fermantes. Par exemple, `"()"`, `"()(())(()()())"` et `"()(())((((()))))"` sont des expressions bien parenthésées, mais pas `")("`, `"())("` et `"((()))()"`.

Considérons le type Haskell suivant qui représente la suite de parenthèses dans une expression :

```
data P = L | R deriving (Show)
```

Ici, le constructeur de valeur `L` (left) joue le rôle d'une parenthèse ouvrante et `R` (right) celui d'une parenthèse fermante.

(a) Nous pouvons associer à toute expression parenthésée un chemin dans le plan : partant de l'origine, une parenthèse ouvrante est associé à un déplacement de vecteur $(1, 1)$ et une parenthèse fermante est associé à un déplacement de vecteur $(1, -1)$. Par exemple, le chemin associé à l'expression bien parenthésée `"()(())((((()))))"` est donné ci-après.



La *hauteur* de l'expression parenthésée est l'ordonnée maximale d'un point de la trajectoire associée. Par exemple, `"()(())((((()))))"` est une expression (bien) parenthésée de hauteur 4. Écrire la fonction

```
heightWellFormedExpression :: [P] -> Int
```

qui calcule la hauteur maximale d'une expression bien parenthésée.

```
λ: heightWellFormedExpression [L,L,R,L,R,R]
2
λ: heightWellFormedExpression [L,L,L,R,R,R]
3
λ: heightWellFormedExpression [L,L,L,R,R,R,L,R]
3
λ: heightWellFormedExpression [L,L,R,L,L,L,R,R,R,R]
4
```

(b) Écrire le prédicat

```
isWellFormedExpression :: [P] -> Bool
```

qui décide si une expression est bien parenthésée, en essayant d'obtenir un complexité optimale.

```
λ: isWellFormedExpression [L,L,R,L,R,R]
True
λ: isWellFormedExpression [L,L,L,R,R,R]
True
λ: isWellFormedExpression [L,R,R,L]
False
λ: isWellFormedExpression [L,L,L,R,R,R,L,R]
True
λ: isWellFormedExpression [L,L,L,R,R,R,R,L]
False
```

(c) Écrire la fonction

```
countReturnZeroWellFormedExpression :: [P] -> Int
```

qui calcule le nombre de retours sur l'axe des abscisses du chemin correspondant à une expression bien parenthésée.

```
λ: countReturnZeroWellFormedExpression []
0
λ: countReturnZeroWellFormedExpression [L,R]
1
λ: countReturnZeroWellFormedExpression [L,L,R,L,R,L,R,R]
1
λ: countReturnZeroWellFormedExpression [L,R,L,R,L,R,L,R]
4
λ: countReturnZeroWellFormedExpression [L,R,L,L,R,R,L,L,L,L,R,R,L,R,R,R]
3
```

(d) Écrire la fonction

```
generateWellFormedExpression :: Int -> [[P]]
```

qui calcule toutes les expressions bien parenthésées d'une semi-longueur donnée. On utilise la semi-longueur car le nombre de symboles est forcément pair. On peut remarquer qu'une expression bien parenthésée de taille non nulle (1) commence nécessairement pas une parenthèse ouvrante, (2) suivie d'une expression bien parenthésée, (3) suivie d'une parenthèse fermante, et (4) éventuellement suivie d'une expression bien parenthésée.

```
λ: generateWellFormedExpression 0
[[]]
λ: generateWellFormedExpression 1
[[L,R]]
λ: generateWellFormedExpression 2
[[L,R,L,R],[L,L,R,R]]
λ: generateWellFormedExpression 3
[[L,R,L,R,L,R],[L,R,L,L,R,R],[L,L,R,R,L,R],[L,L,R,L,R,R],[L,L,L,R,R,R]]
```

Exercice 3 : Arbre binaires avec compteurs

Nos considérons dans cet exercice des arbres binaires de recherche munis de compteurs permettant, pour chaque élément présent dans l'arbre binaire de recherche, de connaître son nombre d'occurrences. La valeur du compteur associé à un élément est donc toujours un entier strictement positif.

Considérons les éléments suivants :

```haskell
-- Counter Binary Search Tree
data CBSTree a = Empty | Branch { left  :: CBSTree a
                                , key   :: a
                                , right :: CBSTree a
                                , count :: Int }

branchChar :: Char
branchChar = '-'

splitChar :: Char
splitChar = '+'

branchIndent :: Int
branchIndent = 5

branchNil :: Char
branchNil = 'X'

instance Show a => Show (CBSTree a) where
  show = showBSTree 0
    where
      drawBranch 0 = ""
      drawBranch n = replicate (n - branchIndent) branchChar ++
                     [splitChar]                              ++
                     replicate (branchIndent-1) branchChar

      showBSTree n Empty = drawBranch n ++ branchNil:"\n"
      showBSTree n t     = showBSTree (n + branchIndent) (right t) ++
                           drawBranch n ++ "(" ++ show (key t)     ++
                           "," ++ show (count t)  ++ ")\n"         ++
                           showBSTree (n + branchIndent) (left t)

-- create a CBSTree containing one element
leafCBSTree :: a -> CBSTree a
leafCBSTree x = Branch { left = Empty, key = x, right = Empty, count = 1 }

-- test null CBSTree
nullCBSTree Empty = True
nullCBSTree _     = False
```

Dans la suite, pour alléger la présentation, tout arbre binaire de recherche désignera en fait un arbre binaire de recherche avec compteurs. De plus, la structure des arbres binaires varient suivant l'ordre des insertions des éléments et il est donc possible – et correct sauf mention explicite d'un ordre imposé – que vous n'obteniez pas les mêmes sorties que les exemples donnés.

(a) Écrire la fonction

`insertCBSTree :: (Ord a) => a -> CBSTree a -> CBSTree a`

qui insère une occurrence d'un élément dans un arbre binaire de recherche.

```
λ: insertCBSTree 'a' Empty
+----X
```

```
('a',1)
+----X
λ: insertCBSTree 'b' . insertCBSTree 'a' $ Empty
-----+----X
+----('b',1)
-----+----X
('a',1)
+----X
λ: insertCBSTree 'a' . insertCBSTree 'b' . insertCBSTree 'a' $ Empty
-----+----X
+----('b',1)
-----+----X
('a',2)
+----X
```

(b) Écrire la fonction

`multInsertCBSTree :: (Ord a) => a -> Int -> CBSTree a -> CBSTree a`

`multInsertCBSTree k c t` insère `c` occurrences de l'élément `k` dans l'arbre binaire de recherche `t`.

```
λ: multInsertCBSTree 'a' 2 Empty
+----X
('a',2)
+----X
λ: multInsertCBSTree 'b' 5 . multInsertCBSTree 'a' 2 $ Empty
-----+----X
+----('b',5)
-----+----X
('a',2)
+----X
λ: multInsertCBSTree 'a' 8 . multInsertCBSTree 'b' 5 . multInsertCBSTree 'a' 2 $ Empty
-----+----X
+----('b',5)
-----+----X
('a',10)
+----X
```

(c) Écrire la fonction

`fromListCBSTree :: (Ord a) => [a] -> CBSTree a`

`fromListCBSTree xs` construit un arbre binaire de recherche en insérant les éléments de `xs` par une lecture de gauche à droite (l'ordre est ici imposé).

```
λ: fromListCBSTree []
X
λ: fromListCBSTree "cabcdaba"
-----+----X
+----('d',1)
-----+----X
('c',2)
----------+----X
-----+----('b',2)
----------+----X
+----('a',3)
-----+----X
```

(d) Écrire la fonction

`countCBSTree :: CBSTree a -> Int`

qui retourne le nombre d'éléments dans un arbre binaire de recherche (multiplicités incluses).

```
λ: countCBSTree Empty
0
λ: countCBSTree $ fromListCBSTree "cabcdaba"
8
```

(e) Écrire la fonction

```
preOrderCBSTree :: CBSTree a -> [(a, Int)]
```

qui retourne les paires `(k,c)` (*i.e.* `c` occurrences de l'élément `k`) obtenues par un parcours préfixe dans un arbre binaire de recherche.

```
λ: preOrderCBSTree $ fromListCBSTree "cabcdaba"
[('c',2),('a',3),('b',2),('d',1)]
```

(f) Écrire la fonction

```
mapCBSTree :: (Ord b) => (a -> b) -> CBSTree a -> CBSTree b
```

qui applique une fonction donnée sur tous les éléments d'un arbre binaire de recherche.

```
λ: mapCBSTree succ $ fromListCBSTree "cabcdaba"
+----X
('e',1)
----------+----X
-----+----('d',2)
----------+----X
+----('c',2)
----------+----X
-----+----('b',3)
----------+----X
λ: mapCBSTree ((<) 'b') $ fromListCBSTree "cabcdaba"
+----X
(True,3)
-----+----X
+----(False,5)
-----+----X
```

(g) Écrire la fonction

```
mergeCBSTree2 :: (Ord a) => CBSTree a -> CBSTree a -> CBSTree a
```

qui fusionne deux arbres binaires de recherche en insérant les éléments du second dans le premier.

```
λ: mergeCBSTree2 (fromListCBSTree "abc") (fromListCBSTree "babae")
---------------+----X
----------+----('e',1)
---------------+----X
-----+----('c',1)
----------+----X
+----('b',3)
-----+----X
('a',3)
+----X
λ: mergeCBSTree2 (fromListCBSTree "aa") (fromListCBSTree "aabaa")
-----+----X
+----('b',1)
-----+----X
('a',6)
+----X
```

(h) En déduire l'écriture de la fonction

```
mergeCBSTree :: (Ord a) => [CBSTree a] -> CBSTree a
```

qui fusionne une liste d'arbres binaires de recherche, en utilisant la fonction d'ordre supérieur `foldr`.

```
λ: nullCBSTree $ mergeCBSTree []
True
λ: mergeCBSTree [fromListCBSTree "abcabc"]
----------+----X
-----+----('c',2)
----------+----X
+----('b',2)
-----+----X
```

```
        ('a',2)
        +----X
        λ: mergeCBSTree [fromListCBSTree xs | xs <- ["ab", "bac", "z"]]
        ----------+----X
        -----+----('z',1)
        ---------------+----X
        ----------+----('c',1)
        ---------------+----X
        +----('b',2)
        -----+----X
        ('a',2)
        +----X
```