Institut Informatique Claude Chappe

Master Informatique
Module 177UD05
Programmation fonctionnelle

Examen du 15 décembre 2020

Aucun document autorisé

Les exercices peuvent traités indépendamment les uns des autres

Exercice n°1 (2 pts)

Soit le type suivant : `data BTree a = Leaf a | Node (BTree a) a (BTree a)`
avec l'exemple : `abtree = Node (Node (Leaf "1") "x" (Leaf "2")) "+" (Leaf "3")`

Ecrivez les fonctions `postFlatten :: BTree a -> [a]` et `infFlatten :: BTree a -> [a]`

Exemples :
```
postFlatten abtree      == ["1","2","x","3","+"]
infFlatten abtree       == ["1","x","2","+","3"]
```

Exercice n°2 (2 pts)

Ecrivez la fonction `zip :: [a] -> [b] -> [(a,b)]`

Exemples :
```
zip [1,2] [3,4,5]       == [(1,3),(2,4)]
zip [1,2,3] [4,5]       == [(1,4),(2,5)]
zip [] [3,4,5]          == []
```

Exercice n°3 (2 pts)

Ecrivez la fonction `unzip :: [(a,b)] -> ([a], [b])`

Exemples :
```
unzip []                == ([], [])
unzip [(1,2),(3,4)]     == ([1,3],[2,4])
```

Exercice n°4 (2 pts)

Ecrivez la fonction `takeWhile :: (a -> Bool) -> [a] -> [a]`

Exemples :
```
takeWhile (<10) [2,4,6,8,10,12]     == [2,4,6,8]
takeWhile (>10) [2,4,6,8,10,12]     == []
```

Exercice n°5 (1 pt)

Dites ce que fait la fonction `sgetLine :: IO String` définie de la manière suivante (cf. page d'après) :

```
getCh :: IO Char
getCh = do hSetEcho stdin False
           x <- getChar
           hSetEcho stdin True
           return x
```

```
sgetLine :: IO String
sgetLine = do c <- getCh
              if c == '\n'
                 then do putChar c
                         return []
                 else do putChar '-'
                         xs <- sgetLine
                         return (c:xs)
```

Exercice n°6 (1 pt)

Dites à quoi sert la classe de type Monad définie de la manière suivante :

```
class Applicative m => Monad m where
      return :: a -> m a
      (>>=) :: m a -> (a -> m b) -> m b
```

Exercice n°7 (10 pts) :

Le but de l'exercice est d'écrire une fonction d'affichage d'une table de vérité d'une expression booléenne donnée. Pour cela, vous allez devoir écrire différentes fonctions intermédiaires.

On utilisera le type `BoolExpr` suivant :

```
data BoolExpr a = Var Char |
                  Not (BoolExpr a) |
                  And (BoolExpr a) (BoolExpr a) |
                  Or (BoolExpr a) (BoolExpr a)
```

avec l'exemple : `aexpr = Or (Var 'D') (And (Var 'B') (Var 'C'))`

On utilisera une table d'associations de chaque variable et sa valeur booléenne définie ainsi :

```
data Assoc = Ass [(Char,Bool)] deriving Show
```

avec l'exemple : `aassoc = Ass [('D', False), ('B', True), ('C', False)]`

a) **(1 pt)** Ecrivez l'instance de la classe `Show` du type `BoolExpr`.

Exemples :
```
show aexpr  == "(D + (B . C))"

exp = Not (And (Or (Var 'B') (Var 'C')) (Var 'D'))
show exp    == "Not (((B + C) . D))"
```

b) **(1 pt)** Ecrivez la fonction `getVal :: Char -> Assoc -> Bool`
On supposera que la variable existe dans la table.

Exemples :
```
getVal 'D' aassoc       == False
getVal 'B' aassoc       == True
```

c) **(2 pts)** Ecrivez une fonction `getVars :: BoolExpr a -> [Char]` qui retourne la liste des variables apparaissant dans une expression booléenne.

Vous utiliserez la fonction `rmdups` suivante :

```
rmdups :: Eq a => [a] -> [a]
rmdups [] = []
rmdups (x:xs) = x : filter (/= x) (rmdups xs)
```

Vous écrirez et utiliserez la fonction `getVar :: BoolExpr a -> [Char].`

Exemples :

```
rmdups "AABBACD"        == "ABCD"

exp = And (Or (Var 'A') (Var 'B')) (And (Var 'A') (Var 'D'))
getVar exp              == "ABAD"

getVars exp             == "ABD"
```

d) **(1 pt)** Ecrivez une fonction d'évaluation d'une expression booléenne :
`eval :: Assoc -> BoolExpr a -> Bool`

Exemple : `eval aassoc aexpr == False`

e) **(2 pts)** Pour générer la table de vérité d'une expression, nous avons besoin de générer toutes les tables d'association possible.

Ecrivez la fonction `genAssocs :: [Char] -> [Assoc]`

Vous utiliserez la fonction `bools` suivante :

```
bools :: Int -> [[Bool]]
bools 0 = [[]]
bools n = map (False :) bss ++ map (True :) bss
      where bss = bools (n-1)
```

Exemples :

```
bools 0 == [[]]

bools 1 == [[False],[True]]

bools 2 == [[False,False],[False,True],[True,False],[True,True]]

genAssocs "A"     == [Ass [('A',False)],Ass [('A',True)]]

genAssocs "AB"    == [Ass [('A',False),('B',False)],
                      Ass [('A',False),('B',True)],
                      Ass [('A',True),('B',False)],
                      Ass [('A',True),('B',True)]]
```

f) **(1 pt)** Ecrivez la fonction `showHeader :: [Char] -> IO ()`. « X » est le nom de la sortie. Vous disposez de la fonction `vars2Str :: [Char] -> String` qui convertit une liste de variables en chaine avec des espaces autour de chaque variable ainsi que de la fonction `replicate :: Int -> a -> [a]` qui réplique n fois sont deuxième argument.

Exemples : `vars2Str "ABC"  == " A  B  C "`

`showHeader "ABC"  == " A  B  C | X`
`                     -------------"`

g) **(1 pt)** Ecrivez la fonction `showRows :: BoolExpr a -> IO ()` qui affiche les lignes de la table de vérité (en dessous de l'entête de la table).

Vous disposez de la fonction `row2Str :: Assoc -> BoolExpr a -> String` qui convertit une table d'association en chaine en fonction de l'expression donnée.

Exemples : `row2Str aassoc aexpr  == " 0  1  0 | 0 \n"`

`showRows aexpr  == " 0  0  0 | 0`
`                   0  0  1 | 0`
`                   0  1  0 | 0`
`                   0  1  1 | 1`
`                   1  0  0 | 1`
`                   1  0  1 | 1`
`                   1  1  0 | 1`
`                   1  1  1 | 1 "`

h) **(1 pt)** Enfin, écrivez la fonction `showTable :: BoolExpr a -> IO ()`
Exemple dans l'image suivante :

```
*Main>
*Main> aexpr
(D + (B . C))
*Main> showTable aexpr
 D  B  C | X
-------------
 0  0  0 | 0
 0  0  1 | 0
 0  1  0 | 0
 0  1  1 | 1
 1  0  0 | 1
 1  0  1 | 1
 1  1  0 | 1
 1  1  1 | 1

*Main>
```