Functional and Logic Programming Fall 2017

S06 : Haskell (Personalised types)

Professor : Le Peutrec Stephane
Assistant : Lauper Jonathan

Exercise 1

Product

```
-- Ex1.a
product' :: Num a => [a] -> a
product' [] = 1
product' (x:xs) = foldr (*) x xs
```

Flatten

```
-- Ex1.b
flatten' :: [[a]] -> [a]
flatten' [] = []
flatten' xss = foldr (++) [] xss
```

AllDifferent

```
-- Ex1.d
allDifferent,allDifferent',allDifferent'',allDifferent''' :: Eq a => [a] -> Bool

allDifferent [] = True
allDifferent (x:xs) = if elem x xs then False else allDifferent xs

allDifferent' [] = True
allDifferent' (x:xs) = inner [x] xs where
  inner :: Eq a => [a] -> [a] -> Bool
  inner _ [] = True
  inner presentValues (x:xs)
    | elem x presentValues = False
    | otherwise = inner (x:presentValues) xs


allDifferent'' xs = inner xs (\x -> x) where
  inner :: Eq a => [a] -> (Bool -> Bool) -> Bool
  inner [] cont = cont True
  inner (x:xs) cont = inner xs (\n -> cont(n && (not (elem x xs))))

allDifferent''' xs = fst (foldr (
                             \v ->
                              \(b,ys) -> if b
                              then if elem v ys
                                then (False,[])
                                else (True,v:ys)
                              else (False,[])
                             )(True,[]) xs)
```

DeleteAll

```
-- Ex1.c
deleteAll :: Eq a => a -> [a] -> [a]
deleteAll elt xs = foldr (\v -> \acc -> if v == elt then acc else v:acc) [] xs
```

Exercise 2

Type declaration

```
-- type declaration
data BinaryTree a = EmptyTree | Node a (BinaryTree a) (BinaryTree a) deriving (Show,Eq,Ord)
```

InsertInTree

```
-- Ex1.a
insertInTree :: (Eq a, Ord a) => a -> BinaryTree a -> BinaryTree a
insertInTree elt EmptyTree = Node elt EmptyTree EmptyTree
insertInTree elt (Node e left right)
  | elt >= e = Node e (insertInTree elt left) right
  | otherwise = Node e left (insertInTree elt right)
```

SearchInTree

```
-- Ex1.b
searchInTree :: (Eq a , Ord a) => a -> BinaryTree a -> Bool
searchInTree _ EmptyTree = False
searchInTree elt (Node e left right)
  | elt == e = True
  | elt > e = searchInTree elt left
  | otherwise = searchInTree elt right
```

SortTree

```
-- Ex1.c
sortTree :: BinaryTree a -> [a]
sortTree EmptyTree = []
sortTree (Node e left right) = (sortTree right) ++ [e] ++ (sortTree left)
```

Exercise 3

Type declaration

```
-- type declaration
type State = Int
type Code = String
type Transition = (State,Char -> Bool,State)
data StateMachine = StateMachine State [State] [Transition] deriving (Show)
data Token = Token StateMachine Code deriving (Show)
```

Token definition

```
-- Tokens definition
t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16 :: Token
t1 = Token (StateMachine 0 [1] [(0,\c -> c == '{',1)]) "begin_block"
t2 = Token (StateMachine 0 [1] [(0,\c -> c == '}',1)]) "end_block"
t3 = Token (StateMachine 0 [1] [(0,\c -> c == '(',1)]) "begin_par"
t4 = Token (StateMachine 0 [1] [(0,\c -> c == ')',1)]) "end_par"
t5 = Token (StateMachine 0 [1] [(0,\c -> c == ';',1)]) "semicolon"
t6 = Token (StateMachine 0 [2] [
               (0,\c -> c == '=',1),
               (1,\c -> c == '=',2)])
     "op_eg"
t7 = Token  (StateMachine 0 [1] [(0,\c -> c == '=',1)]) "op_affect"
t8 = Token  (StateMachine 0 [1] [(0,\c -> c == '+',1)]) "op_add"
t9 = Token  (StateMachine 0 [1] [(0,\c -> c == '-',1)]) "op_minus"
t10 = Token (StateMachine 0 [1] [(0,\c -> c == '*',1)]) "op_mult"
t11 = Token (StateMachine 0 [1] [(0,\c -> c == '/',1)]) "op_div"
t12 = Token ( StateMachine 0 [3] [
     (0,\c -> c == 'i',1),
     (1,\c -> c == 'n',2),
     (2,\c -> c == 't',3)])
      "type_int"
t13 = Token ( StateMachine 0 [2] [
     (0,\c -> c == 'i',1),
     (1,\c -> c == 'f',2)])
      "cond"
t14 = Token ( StateMachine 0 [5] [
     (0,\c -> c == 'w',1),
     (1,\c -> c == 'h',2),
     (2,\c -> c == 'i',3),
     (3,\c -> c == 'l',4),
     (4,\c -> c == 'e',5)])
      "loop"
t15 = Token ( StateMachine 0 [1] [
                (0,isDigit,1),
                (1,isDigit,1)])
      "value_int"
t16 = Token ( StateMachine 0 [1] [
                (0,\c -> isIdentHead c, 1),
                (1,\c -> isIdentBody c,1)])
             "ident" where
  isIdentHead c = (elem c az) || (elem c (map toUpper az)) || c == '_'
  isIdentBody c = isIdentHead c || (isDigit c)
  az = "abcdefghijklmnopqrstuvwxyz"
tokens = [t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16]
```

RecognizedFromState

```
-- Ex3.2
recognizedFromState :: String -> Token -> (Code,String,String)
recognizedFromState str tk = recognizedFromState' 0 str "" tk

recognizedFromState' :: State -> String -> String -> Token -> (Code,String,String)
recognizedFromState' _ [] acc (Token _ code) = (code,acc,"")
recognizedFromState' crtState crtString@(c:cs) acc tk@(Token stateMachine@(StateMachine _ _
↪ transitions) code)
  | ns == -1 = if isFinalState crtState stateMachine
    then (code,acc,crtString)
    else ("","",crtString)
  | otherwise = recognizedFromState' ns cs (acc ++ [c]) tk where
      ns = nextState crtState c stateMachine
```

GetNextRecognizedToken

```haskell
-- Ex3.3
getNextRecognizedToken :: String -> [Token] -> (Code,String,String)
getNextRecognizedToken text [] = ("","",text)
getNextRecognizedToken text tokens = inner text tokens ("","",text) where
  inner :: String -> [Token] -> (Code,String,String) -> (Code,String,String)
  inner _ [] acc = acc
  inner text (tk:tks) acc@(_,crtRecognizedString,_) = case (recognizedFromState text tk) of
    n@(_, txt, _) ->
      if (length txt) > (length crtRecognizedString)
      then inner text tks n
      else inner text tks acc
```

LexAnalyse

```haskell
lexAnalyse :: String -> [Token] -> [Code]
lexAnalyse str tokens = inner (trim str) [] where
  inner :: String -> [Code] -> [Code]
  inner "" acc = acc
  inner crtStr acc = case (getNextRecognizedToken crtStr tokens) of
    (_,"",_) -> acc
    (code,_,rest) -> inner (trim rest) (acc ++ [code])
```

Function for series 05

```haskell
-- remove the first spaces of a String
trim :: String -> String
trim "" = ""
trim str@(c:cs)
  | isSpace c = trim cs
  | otherwise = str

isFinalState :: Int -> StateMachine -> Bool
isFinalState crtState (StateMachine _ finalStates _) = elem crtState finalStates

nextState :: State -> Char -> StateMachine -> State
nextState crtState c (StateMachine _ _ transitions) = applyTransitions transitions where
  applyTransitions :: [Transition] -> State
  applyTransitions [] = -1
  applyTransitions ((start,predicat,end):xs)
    | start == crtState && predicat c = end
    | otherwise = applyTransitions  xs
```