Functional and Logic Programming Fall 2017

S03 : Haskell (Lists and lexical analysis)

Professor : Le Peutrec Stephane
Assistant : Lauper Jonathan

Exercise 1

Fibonnacci

```haskell
-- Ex1.a
fibonacci,fibonacci'  :: Int -> Int
fibonacci n = inner n 0 1 where
  inner :: Int -> Int -> Int -> Int
  inner 0 acc1 _ = acc1
  inner n acc1 acc2 = inner (n-1) acc2 $! (acc1 + acc2)

fibonacci' 0 = 0
fibonacci' n = inner n (\x -> x) where
  inner :: Int -> (Int -> Int) -> Int
  inner 0 cont = cont 0
  inner 1 cont = cont 1
  inner n cont = inner (n-1) (\x -> inner (n-2) (\y -> cont x + y))
```

Product

```haskell
-- Ex1.b
-- PRE : (null []) == false
product',product'' :: Num a => [a] -> a
product' xs = inner xs 1 where
  inner [] acc = acc
  inner (x:xs) acc = inner xs $! (acc * x)

product'' xs = inner xs (\x -> x) where
  inner :: Num a => [a] -> (a -> a) -> a
  inner [] cont = cont 1
  inner (x:xs) cont = inner xs (\n -> cont (n * x))
```

Flatten

```haskell
-- Ex1.c
flatten',flatten'' :: [[a]] -> [a]
flatten' xss = inner xss [] where
  inner [] acc = acc
  inner ((ys):xs) acc = inner xs $! (acc ++ ys)

flatten'' xss = inner xss (\x -> x) where
  inner :: [[a]] -> ([a] -> [a]) -> [a]
  inner [] cont = cont []
  inner (x:xs) cont = inner xs (\n -> cont (x ++ n))
```

DeleteAll

```haskell
-- Ex1.d
deleteAll,deleteAll' :: Eq a => a -> [a] -> [a]
deleteAll elt xs = inner (reverse' xs) [] where
  inner [] acc = acc
  inner (x:xs) acc
    | x == elt = inner xs acc
    | otherwise = inner xs $! (x:acc)

deleteAll' elt xs = inner xs (\x -> x) where
  inner [] cont = cont []
  inner (x:xs) cont
    | x == elt = inner xs (\ns -> cont ns)
    | otherwise = inner xs (\ns -> cont (x : ns))
```

Insert

```haskell
-- Ex1.e
insert',insert'' :: Ord a => a -> [a] -> [a]
insert' elt xs = inner xs [] where
  inner [] acc = reverse' (elt:acc)
  inner l@(x:xs) acc
    | elt <= x = (reverse' acc) ++ (elt:l)
    | otherwise = inner xs $! (x:acc)

insert'' elt xs = inner xs (\x -> x) where
  inner [] cont = cont (elt : [])
  inner (x:xs) cont
    | elt <= x = cont (elt : x : xs)
    | otherwise = inner xs (\ns -> cont (x : ns))
```

Reverse

```haskell
-- when we don't want the inversion of the list during the algorithm
reverse' :: [a] -> [a]
reverse' xs = inner xs [] where
  inner [] acc = acc
  inner (x:xs) acc = inner xs $! (x:acc)
```

Exercise 2

Type declaration

```haskell
-- type definition
type State = Int
type Transition = (State,Char -> Bool,State)
type StateMachine = (State,[State],[Transition])
type Code = String
type Token = (StateMachine,Code)
```

Tokens definition

```
t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16 :: Token
t1 = ((0,[1],[(0,\c -> c == '{',1)]),"begin_block")
t2 = ((0,[1],[(0,\c -> c == '}',1)]),"end_block")
t3 = ((0,[1],[(0,\c -> c == '(',1)]),"begin_par")
t4 = ((0,[1],[(0,\c -> c == ')',1)]),"end_par")
t5 = ((0,[1],[(0,\c -> c == ';',1)]),"semicolon")
t6 = (
  (
    0,
    [2],
    [
      (0,\c -> c == '=',1),
      (1,\c -> c == '=',2)
    ]),
  "op_eg")
t7 = ((0,[1],[(0,\c -> c == '=',1)]),"op_affect")
t8 = ((0,[1],[(0,\c -> c == '+',1)]),"op_add")
t9 = ((0,[1],[(0,\c -> c == '-',1)]),"op_minus")
t10 = ((0,[1],[(0,\c -> c == '*',1)]),"op_mult")
t11 = ((0,[1],[(0,\c -> c == '/',1)]),"op_div")
t12 = (
  (0,
   [3],
   [
     (0,\c -> c == 'i',1),
     (1,\c -> c == 'n',2),
     (2,\c -> c == 't',3)
   ]),
    "type_int")
t13 = (
  (0,
   [2],
   [
     (0,\c -> c == 'i',1),
     (1,\c -> c == 'f',2)
   ]),
    "cond")
t14 = (
  (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 = (
  (0,
   [0],
   [
     (0,isDigit,0)
   ]
  ),"value_int")
t16 = (
  (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]
```

Function from series 03

```haskell
isToken :: String -> StateMachine -> Bool
isToken str automata@(initState,_,_) = reconizedFromState initState str automata

reconizedFromState :: State -> String -> StateMachine -> Bool
reconizedFromState crtState "" automata = isFinalState crtState automata
reconizedFromState crtState (c:str) automata = if nextState' == -1
  then False
  else reconizedFromState nextState' str automata where
  nextState' = nextState crtState c automata

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

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

GetToken

```haskell
getToken :: String -> [Token] -> Code
getToken str [] = error ("no recognize token : " ++ str)
getToken str ((automata,code):tks) = if isToken str automata
  then code
```

LexAnalyse

```haskell
lexAnalyse :: String -> [Code]
lexAnalyse str = inner (words str) where
  inner :: [String] -> [Code]
  inner [] = []
```