# Functional and Logic Programming Fall 2017

## S03 : Haskell (Lists and lexical analysis)

Professor : Le Peutrec Stephane
Assistant : Lauper Jonathan

## Exercise 1

### flatten

```
-- Ex1.a
flatten :: [[a]] -> [a]
flatten [] = []
flatten (x:xs) = x ++ (flatten xs)
```

### partitions

```
-- Ex1.b
partitions :: [a] -> [[a]]
partitions [] = [[]]
partitions (x:xs) = [x:parts | parts <- partitions xs] ++ -- first case, x belongs to the set
                    [parts | parts <- partitions xs]      -- second case, x does not belongs to the set
```

### permutations

```
-- Ex1.c
permutations :: Eq a => [a] -> [[a]]
permutations [] = [[]]
-- the permutations of xs are all the of x with all the permutations of xs \ x
permutations xs = [x:ys | x <- xs, ys <- permutation (delete' x xs)] where
  delete' :: Eq a => a -> [a] -> [a]
  delete' _ [] = []
  delete' (e) (x:xs) = if e == x then xs else x:(delete' e xs)
```

## Exercise 2

### length'

```
-- Ex2.a
-- why using a comprehension ?
length' :: [a] -> Int
length' l = sum [1 | _ <- l]
```

### deleteAll'

```
-- Ex2.b
deleteAll' :: Eq a => a -> [a] -> [a]
deleteAll' e l = [x | x <- l, x /= e]
```

### toUpperString

```haskell
-- Ex2.c
toUpperString :: String -> String
toUpperString s = [toUpper c | c <- s]
```

## Exercise 3.a

### isFinalState

```haskell
isFinalState :: Int -> Bool
isFinalState n
  | n == 4 = True
  | otherwise = False
```

### firstState

```haskell
firstState :: Int
firstState = 0
```

### transition

```haskell
transition :: Int -> Char -> Int
transition 0 'a' = 1
transition 1 'a' = 3
transition 1 'b' = 2
transition 2 'a' = 4
transition 2 'b' = 2
transition 3 'b' = 4
transition 4 'a' = 1
transition _ _ = -1
```

### isToken

```haskell
isToken :: String -> Bool
isToken str = reconizedFromState firstState str

{- Or by hand...
isToken "" = False
isToken str = iter firstState str where
  iter crtState [] = isFinalState crtState
  iter crtState (c:str) = if newState == -1
    then False
    else iter newState str where
    newState = transition crtState c
-}
```

### reconizedFromState

```haskell
reconizedFromState :: Int -> String -> Bool
reconizedFromState (-1) _ = False
reconizedFromState crtState "" = isFinalState crtState
reconizedFromState crtState (c:str) = reconizedFromState (transition crtState c) str
```

## Exercise 3.b

### Type declaration

```
type State = Int
type Transition = (State,Char,State)
type Automata = (State,[State],[Transition])
```

### isToken

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

### reconizedFromState

```
reconizedFromState :: State -> String -> Automata -> 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

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

### nextState

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

## Exercise 3.c

The first technic (using function) is clearer to write and to read. The transitions are clearly defined and modifying them is easy because we just have to add pattern matching case (and that is the same for the *isFinalState* and *firstState* functions). The two remaining functions, *isToken* and *reconizedFromState* are very straightforward to write and read to.
The second technic (using type aliases and record-like structures) is a little bit longer to write and read. The implementation (mostly *reconizedFromState* and *nextState*)is longer than the previous one.
The big advantage of the second technic is its scaleability. If we have to produce the automata from, for example, a regular expression we have to parse, it's easier to generates the data structures than generating function likes the one in part *a*.
The advantage of the first technic is that we can use function composition and higher-order function to manipulate our automata.

For me, it's easier to manipulate structured data instead of function in this case. For example, if we want, during the computation of an automaton, to know something about a specific transition, it would be easier to implement this if we have a list of transitions instead of having a function to analyse.