COMP3258

Functional Programming

Tutorial Session 6: Mid Term Review

Problem 1

Write a Haskell function `wordVowels :: String → [(String, Int)]` that takes a sentence (a string containing words separated by spaces) and returns a list of tuples, where each tuple contains a word and the number of vowels in that word.

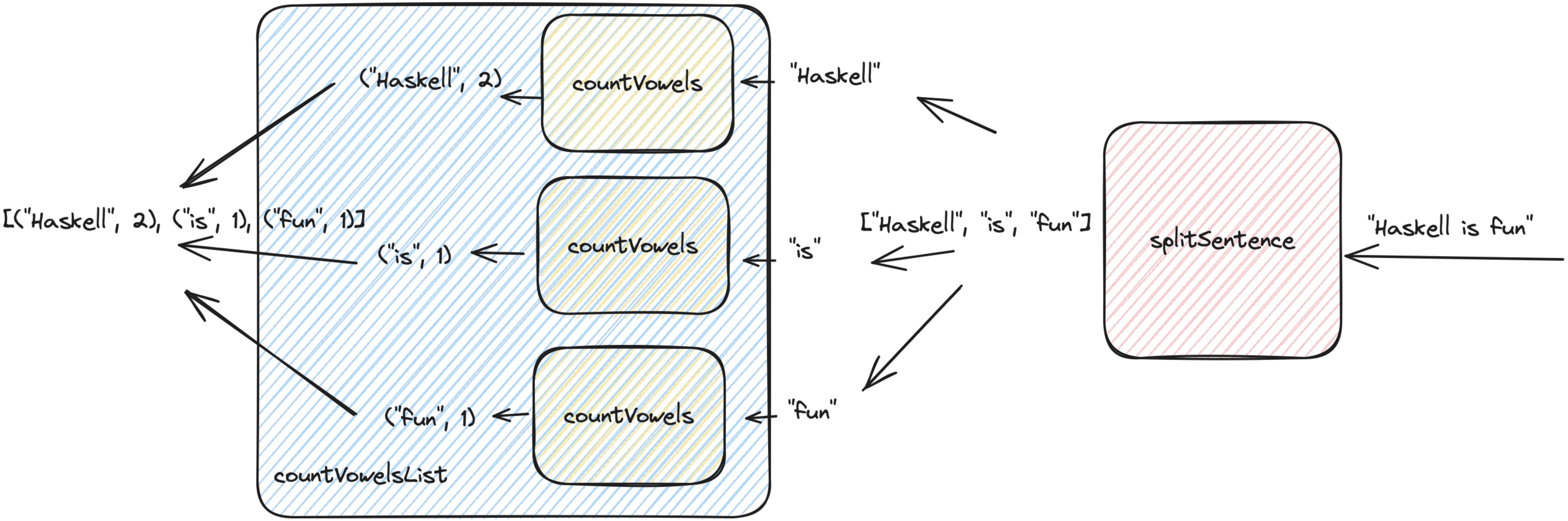
The vowels are `'a'`, `'e'`, `'i'`, `'o'`, and `'u'` (both uppercase and lowercase). For example:

```
wordVowels "Haskell is fun"    = [("Haskell", 2), ("is", 1), ("fun", 1)]
wordVowels “Lists are useful" = [("Lists", 1), ("are", 2), ("useful", 3)]
wordVowels "HELLO WORLD"       = [("HELLO", 2), ("WORLD", 1)]
```

(a) The function should use **list comprehensions** and may use **basic functions** and **library functions**, but not recursion.

(b) Write a second function version of `wordVowels`, this time that must use **recursion** and may use **basic functions**, but you should not use **list comprehensions** and **library functions**.

Problem 1



countVowels
countVowels
countVowels
countVowelsList
splitSentence
"Haskell is fun"
["Haskell", "is", "fun"]
"Haskell"
"is"
"fun"
("Haskell", 2)
("is", 1)
("fun", 1)
[("Haskell", 2), ("is", 1), ("fun", 1)]

Subproblems

[1] Split a sentence into words

```
splitSentence :: String -> [String]

>>> splitSentence "Haskell is fun”
[“Haskell”, “is”, “fun”]

>>> splitSentence “Lists are useful”
[“Lists”, “are”, “useful”]
```

[2] Count vowels in a list of words

```
countVowelsList :: [String] -> [(String, Int)]

>>> countVowelsList [“Haskell”, “is”, “fun”]
[("Haskell", 2), ("is", 1), ("fun", 1)]

>>> countVowelsList [“Lists”, “are”, “useful”]
[("Lists", 1), ("are", 2), ("useful", 3)]
```

```
wordVowels :: String -> [(String, Int)]
wordVowels = countVowelsList . splitSentence
```

Subproblem [1]

[1] Split a sentence into words

```
splitSentence :: String → [String]

>>> splitSentence "Haskell is fun”
[“Haskell”, “is”, “fun”]

>>> splitSentence “Lists are useful”
[“Lists”, “are”, “useful”]
```

```haskell
splitSentenceA :: String → [String]
splitSentenceA = words

splitSentenceB :: String → [String]
splitSentenceB str = foldr processWord [] str
  where
      processWord :: Char → [String] → [String]
      processWord ' '  acc     = [] : acc
      processWord curr []      = [[curr]]
      processWord curr (w:ws)  = (curr : w) : ws

splitSentenceC :: String → [String]
splitSentenceC [] = []
splitSentenceC (x:xs) | x == ' ' = [] : splitSentenceC xs
                      | otherwise = case (splitSentenceC xs) of
                            [] → [[x]]
                            (y:ys) → (x:y):ys
```

Subproblem [2]

[2.1] Count vowels in a word

```
countVowels :: String → (String, Int)

>>> countVowels “Haskell”
(“Haskell”, 2)

>>> countVowels “Lists”
(“Lists”, 2)
```

```
countVowelsListA :: [String] → [(String, Int)]
countVowelsListA xs = [countVowelsA x | x ← xs]

countVowelsA :: String → (String, Int)
countVowelsA s = (s, length $ filter (`elem` "aeiouAEIOU") s)
```

Subproblem [2]

[2.1] Count vowels in a word

```
countVowels :: String → (String, Int)

>>> countVowels “Haskell”
(“Haskell”, 2)

>>> countVowels “Lists”
(“Lists”, 2)
```

```haskell
countVowelsListB :: [String] → [(String, Int)]
countVowelsListB [] = []
countVowelsListB (x:xs) = countVowelsB x : countVowelsListB xs

countVowelsB :: String → (String, Int)
countVowelsB s = (s, countVowelsB' s)
    where countVowelsB' [] = 0
          countVowelsB' (x:xs) = if isVowel x then 1 + countVowelsB' xs else countVowelsB' xs
```

Problem 1

```haskell
wordVowels :: String -> [(String, Int)]
wordVowels s = wordVowels' s "" 0 []

wordVowels' :: String -> String -> Int -> [(String,Int)] -> [(String,Int)]
wordVowels' [] s n acc     = acc ++ [(s,n)]
wordVowels' (x:xs) s n acc
    | isVowel x            = wordVowels' xs (s ++ [x]) (n+1) acc
    | x == ' '             = wordVowels' xs "" 0 (acc ++ [(s,n)])
    | otherwise            = wordVowels' xs (s ++ [x]) n acc
```

Problem 2a

Consider the following data type representing boolean expressions with a single variable:

```
data BExpr = X                        -- single boolean variable
           | And BExpr BExpr          -- logical AND
           | Or BExpr BExpr           -- logical OR
           | Not BExpr                -- logical NOT
           | Impl BExpr BExpr         -- logical implication
           | Equiv BExpr BExpr        -- logical equivalence
```

(a) Write a function `evalB :: BExpr -> Bool -> Bool`, which takes a boolean expression and the value of the single boolean variable `X`, and returns the value of the expression.

```
evalB (And X (Or X (Not X))) True = True
evalB (Impl X X) False            = True
evalB (Equiv X (Not X)) True      = False
```

Problem 2a

```haskell
evalB :: BExpr -> Bool -> Bool
evalB X val                 = val
evalB (And b1 b2) val       = evalB b1 val && evalB b2 val
evalB (Or b1 b2) val        = evalB b1 val || evalB b2 val
evalB (Not b) val           = not (evalB b val)
evalB (Impl b1 b2) val      = not (evalB b1 val) || evalB b2 val
evalB (Equiv b1 b2) val     = evalB b1 val == evalB b2 val
```

Problem 2b

Write a function `toInfixNotation :: BExpr → [String]` that converts a boolean expression to its equivalent in infix notation. In infix notation, the operator is placed between its operands, like:

```
Not X                      in infix notation is    NOT X
And X Y                    in infix notation is    X AND Y
Or X (Not Y)               in infix notation is    X OR (NOT Y)
Impl X (And Y Z)           in infix notation is    X IMPL (Y AND Z)
Equiv X (Or Y (Not Z)) in infix notation is    X EQUIV (Y OR (NOT Z))
```

The function should return the infix notation as a list of strings. For example:

```
toInfixNotation (And X (Or X (Not X))) = ["X","AND","X","OR","NOT","X"]
```

Problem 2b

```
toInfixNotation :: BExpr → [String]
toInfixNotation X               = ["X"]
toInfixNotation (And e1 e2)     = toInfixNotation e1 ++ ["AND"] ++ toInfixNotation e2
toInfixNotation (Or e1 e2)      = toInfixNotation e1 ++ ["OR"] ++ toInfixNotation e2
toInfixNotation (Not e)         = ["NOT"] ++ toInfixNotation e
toInfixNotation (Impl e1 e2)    = toInfixNotation e1 ++ ["IMPL"] ++ toInfixNotation e2
toInfixNotation (Equiv e1 e2) = toInfixNotation e1 ++ ["EQUIV"] ++ toInfixNotation e2
```

Problem 3

In Haskell, how does the order of generators in list comprehensions affect the resulting list, and what is the significance of guards in list comprehensions? Illustrate your answer with a concrete example of a list comprehension with two generators and a guard.

- the order of generators in list comprehensions affects the resulting list

- Haskell list comprehensions can also include guards, which are boolean expressions that act as filters, allowing elements to be included in the output list only if the guard condition is True.

```
list1 = [1, 2, 3]
list2 = ['A', 'B', 'C']

comprehension1 = [(x, y) | x ← list1, y ← list2, x /= 2] -- [(1,'A'),(1,'B'),(1,'C'),(3,'A'),(3,'B'),(3,'C')]
comprehension2 = [(x, y) | y ← list2, x ← list1, x /= 2] -- [(1,'A'),(3,'A'),(1,'B'),(3,'B'),(1,'C'),(3,'C')]
```

Collect Your Paper Now.

There're 6 stacks, classified by ID
30355, 30356, 30357, 30358, 30359, 3036

Double check your scores with Moodle

If some grading issues are identified, you can talk with me,
but need to confirm with professor later