COMP2209 – PROGRAMMING III

Coursework 1

Name: Beh Shu Ao

Student ID: 33354723

Assessment A1

VSCode:

```
import Data.Char (toUpper, isAlpha)
-- CW1 A1
vigenere :: String -> (String -> String, String -> String)
vigenere key = (vigEncrypt key, vigDecrypt key)

vigEncrypt :: String -> String -> String
vigEncrypt key message = zipWith find (cycle key) (convert message)

vigDecrypt :: String -> String -> String
vigDecrypt key = zipWith find (cycle (map negateKeyword key))
    where
        negateKeyword c = toEnum ((26 - (fromEnum c - fromEnum 'A'))`mod` 26 +
fromEnum 'A')

-- shiftCharPosition
-- create the list for the character and position eg: A-0, B-1, Z-25
-- find the postion of the charcter
find :: Char -> Char ->Char
find char key= toEnum (fromEnum 'A' + finalPosition)
    where
        charPosition = fromEnum char - fromEnum 'A'
        keyPostion = fromEnum key - fromEnum 'A'
        finalPosition = (charPosition + keyPostion) `mod` 26

-- remove all non-alphabetic character and whitespace
-- change all character to Uppercase then filter
convert :: String -> String
convert = map toUpper . filter isAlpha

main :: IO()
main = do
    -- input case by tester
    putStr "Enter the keyword you want to use: "
    keyword <- getLine

    let (encryptMsg,decryptMsg)= vigenere (convert keyword)

    putStr "Enter the message you want to encrypt: "
    message <- getLine
```

```
    let encryptedMessage = encryptMsg (convert message)
    putStrLn $ "Encrypted message: "++ encryptedMessage
    putStrLn $ "Decrypted message: "++  decryptMsg encryptedMessage

    -- Test Case 1
    let keyword1 = "lemon"
    let message1 = "attackdawn"
    let (encryptMsg, decryptMsg) =vigenere (convert keyword1)
    putStrLn $ "\nEnter the keyword you want to use: " ++ keyword1 --lemon
    putStrLn $ "Enter the message you want to encrypt: " ++ message1 --
attackdawn

    let encryptedMessage = encryptMsg (convert message1)
    putStrLn $ "Encrypted message: "++ encryptedMessage --Output: LXFOPVHMKA
    putStrLn $ "Decrypted message: "++  decryptMsg encryptedMessage --Output:
ATTACKDAWN

    -- Test Case 2
    let keyword2 = "keyword"
    let message2 = "he llo"
    let (encryptMsg, decryptMsg) =vigenere (convert keyword2)
    putStrLn $ "\nEnter the keyword you want to use: " ++ keyword2 --keyword
    putStrLn $ "Enter the message you want to encrypt: " ++ message2 --he llo

    let encryptedMessage = encryptMsg (convert message2)
    putStrLn $ "Encrypted message: "++ encryptedMessage --Output: RIJHC
    putStrLn $ "Decrypted message: "++  decryptMsg encryptedMessage --Output:
HELLO
```

Output:

```
ghci> :l CW1_A1.hs
[1 of 2] Compiling Main             ( CW1_A1.hs, interpreted )
Ok, one module loaded.
ghci> main
Enter the keyword you want to use: KEYWORD
Enter the message you want to encrypt: HELLOWORLD
Encrypted message: RIJHCNRBPB
Decrypted message: HELLOWORLD

Enter the keyword you want to use: lemon
Enter the message you want to encrypt: attackdawn
Encrypted message: LXFOPVHMKA
Decrypted message: ATTACKDAWN

Enter the keyword you want to use: keyword
Enter the message you want to encrypt: he llo
Encrypted message: RIJHC
Decrypted message: HELLO
```

Assessment A2

```haskell
import Data.List
import Data.Foldable
import Data.Char
import Data.Function (on)
import System.Posix.Internals (lstat)
--CW1 A2

-- remove all non-alphabetic character and whitespace
-- change all character to Uppercase then filter
convert :: String -> String
convert = map toUpper . filter isAlpha

-- sort the element by the second element with snd function
-- orderList = sortBy (compare `on` snd)
-- descList = quicksortDesc  orderList
frequency :: Int -> String -> [[(Char,Int)]]
frequency n ct = map (list . sort)  groupedCharacters where groupedCharacters
= groupByPosition n ct

-- count the occurences of the character in the message
countCharacter:: Char -> String -> Int
countCharacter c = length . filter (==c)

-- combine the character appears and its occurences into a list of tuples
-- and only remain the first element of the list of the same character
-- list = [(c,count) | c<- ['A'..'Z'], let count=countCharacter c ct ,count>0
| c <- nub ct]
list :: [Char] ->[(Char,Int)]
list str = [(c,countCharacter c str) | c <- nub str]

-- arrange them in descending order based on their frequency
quicksortDesc :: [(Char, Int)] -> [(Char, Int)]
quicksortDesc [] = []
quicksortDesc ((z,y):zs) = quicksortDesc rs ++ [(z,y)] ++ quicksortDesc ls
    where
        rs =[(a,b) | (a,b)<-zs, b>y]
        ls =[(a,b) | (a,b)<-zs, b<=y]

-- Split a list into n parts
-- split :: Int -> [(Char,Int)] -> [[(Char,Int)]]
-- split _ [] = [[],[]]
-- split n (x:xs)= let [(odds), (evens)] = split xs in [(x:evens), (odds)]
groupByPosition :: Int -> String -> [[Char]]
groupByPosition n ct = [ [ct !! i | i <- [pos, pos + n..length ct - 1]] | pos
<- [0..n-1]]

main ::IO()
```

```haskell
main = do
    -- user input case
    putStrLn "Enter the message you want to encrypt: "
    message <- getLine
    let keyword = "KEY"
    print (frequency (length keyword) (convert message))

    -- test case 1
    let message1 = "Hello World"
    putStrLn $ "Message to encrypt: " ++ message1
    print (frequency (length keyword) (convert message1))
    --[[('D',1),('H',1),('L',1),('O',1)],
    -- [('E',1),('O',1),('R',1)],
    -- [('L',2),('W',1)]]

    -- test case 2
    let message2 = "What do you want?"
    putStrLn $ "Message to encrypt: " ++ message2
    print (frequency (length keyword) (convert message2))
    -- [[('T',2),('W',2),('Y',1)],
    --  [('A',1),('D',1),('H',1),('O',1)],
    --  [('A',1),('N',1),('O',1),('U',1)]]
```

Output:

```
ghci> :l CW1_A2.hs
[1 of 2] Compiling Main             ( CW1_A2.hs, interpreted )
Ok, one module loaded.
ghci> main
Enter the message you want to encrypt:
TOMMORROW
[[('M',1),('R',1),('T',1)],[('O',3)],[('M',1),('R',1),('W',1)]]
Message to encrypt: Hello World
[[('D',1),('H',1),('L',1),('O',1)],[('E',1),('O',1),('R',1)],[('L',2),('W',1)]]
Message to encrypt: What do you want?
[[('T',2),('W',2),('Y',1)],[('A',1),('D',1),('H',1),('O',1)],[('A',1),('N',1),('O',1),('U',1)]]
```

Assessment A3

```haskell
import Text.Read (Lexeme(String))
-- CW1 A3
renderMaze :: [((Integer,Integer),(Integer,Integer))] -> [String]
renderMaze [] = []
renderMaze maze = map setRow [0..maxY]
    where
        -- [((x1,y1),(x2,y2))]
        -- find the maximum x and y
        maxX = findMaxX (listMaxX maze)
        maxY = findMaxY (listMaxY maze)

        -- list out all the paths from the maze and return it as a list
        isPath :: ((Integer,Integer),(Integer,Integer)) -> [(Integer,Integer)]
        isPath ((x1,y1),(x2,y2)) | x1==x2 = [(x1,y) | y <- [min y1 y2..max y1
y2]]
                                 | y1==y2 = [(x,y1) | x <- [min x1 x2..max x1
x2]]

        -- store all the paths in this list
        pathCoords = concatMap isPath maze

        -- print the coordinates is path with '#'
        setRow :: Integer -> String
        setRow y = [if (x,y) `elem` pathCoords then '#' else ' '| x <-
[0..maxX]]

listMaxX :: [((Integer,Integer),(Integer,Integer))] -> [Integer]
listMaxX [] = []
listMaxX (((x1,y1),(x2,y2)):xs) = x1:x2 : listMaxX xs

findMaxX :: [Integer] -> Integer
findMaxX [x] = x
findMaxX (x:y:xs) = findMaxX (max x y: xs)

listMaxY :: [((Integer,Integer),(Integer,Integer))] -> [Integer]
listMaxY [] = []
listMaxY (((x1,y1),(x2,y2)):xs) = y1:y2 : listMaxY xs

findMaxY :: [Integer] -> Integer
findMaxY [y] = y
findMaxY (y:z:ys) = findMaxY (max y z: ys)

main :: IO()
main = do
    let maze = [((0,0),(0,3)), ((0,2),(2,2)), ((2,1),(4,1)), ((4,0),(4,2)),
((4,2),(5,2)), ((2,1),(2,5)),((1,5),(4,5))]
    putStrLn $ "Max x is " ++ show (findMaxX (listMaxX maze)) -- 5
```

```haskell
    putStrLn $ "Max y is " ++ show (findMaxY (listMaxY maze)) -- 5
    mapM_ putStrLn (renderMaze maze)
```

Output:

```
ghci> :l CW1_A3.hs
[1 of 2] Compiling Main             ( CW1_A3.hs, interpreted )
Ok, one module loaded.
ghci> main
Max x is 5
Max y is 5
#   #
# ###
### ##
# #
  #
 ####
```

Assessment A4

```
import Data.List

connected :: [((Integer,Integer),(Integer,Integer))] -> Bool
connected ps = length (isPath ps) <= 1

isPath :: [((Integer,Integer),(Integer,Integer))] ->
[[((Integer,Integer),(Integer,Integer))]]
isPath (p:ps) = (p : concat qs) : rs where (qs,rs) = partition (p
`intersectionPath`) (isPath ps)
isPath [] = []

-- check the path with other paths one by one
intersectionPath :: ((Integer,Integer),(Integer,Integer))
->[((Integer,Integer),(Integer,Integer))] -> Bool
intersectionPath p ps = any (intersectPoint p) ps

-- check the intersection point exists or not
intersectPoint :: ((Integer,Integer),(Integer,Integer))->
((Integer,Integer),(Integer,Integer)) -> Bool
intersectPoint (p1,p2) (p3,p4) -- p:point
 = (d1 > 0 && d2 < 0 || d1 < 0 && d2 > 0) && (d3 > 0 && d4 < 0 || d3 < 0 &&
d4 > 0)
  || d1 == 0 && onLine p3 p4 p1 -- p1 on distance (p3,p4)
  || d2 == 0 && onLine p3 p4 p2 -- p2 on distance (p3,p4)
  || d3 == 0 && onLine p1 p2 p3 -- p3 on distance (p1,p2)
  || d4 == 0 && onLine p1 p2 p4 -- p4 on distance (p1,p2)

  where d1 = direction p3 p4 p1
        d2 = direction p3 p4 p2
        d3 = direction p1 p2 p3
        d4 = direction p1 p2 p4

-- Find its direction through vector methods
direction :: (Integer,Integer) -> (Integer,Integer) -> (Integer,Integer) ->
Integer
direction q1 q2 q3 = cross_product (pdiff q3 q1) (pdiff q2 q1)
        where
            pdiff (x,y) (x',y') = (x-x',y-y') -- point difference
            cross_product (x,y) (x',y') = x*y'-x'*y --calculate the cross
product by vector

-- Check whether the point is between these points or not
-- If not then just return False then not need to check this possibilities
onLine :: (Integer,Integer) -> (Integer,Integer) -> (Integer,Integer) -> Bool
onLine (q1x,q1y) (q2x,q2y) (qx,qy)
          =  min q1x q2x <= qx
          && qx <= max q1x q2x
```

```
          && min q1y q2y <= qy
          && qy <= max q1y q2y

main :: IO()
main = do
    let maze1 = [((0,0),(0,3)), ((0,2),(2,2)), ((2,1),(4,1)), ((4,0),(4,2)),
((4,2),(5,2)), ((2,1),(2,5)),((1,5),(4,5))]
    let maze2 = [((0,0),(0,3)), ((0,2),(2,2)), ((2,1),(4,1)),((4,0),(4,2)),
((4,2),(5,2)), ((2,1),(2,5)),((3,5),(5,5))]

    putStrLn $ "Maze 1 is connected: " ++ show(connected maze1) -- True
    putStrLn $ "Maze 2 is connected: " ++ show(connected maze2) -- False
    print (intersectPoint ((0,0),(0,3)) ((0,2),(2,2))) -- True
    print (intersectPoint ((0,0),(0,3)) ((3,5),(5,5))) -- False
```

Output:

```
ghci> :l CW1_A4.hs
[1 of 2] Compiling Main             ( CW1_A4.hs, interpreted )
Ok, one module loaded.
ghci> main
Maze 1 is connected: True
Maze 2 is connected: False
True
False
```

Assessment A5

```haskell
import Data.Char
-- CW1 A5
hextonum :: Char -> Int
hextonum x | x `elem` ['0'..'9'] = fromEnum x - fromEnum '0'
           | x `elem` ['a'..'f'] = 10 + (fromEnum x - fromEnum 'a')
           | x `elem` ['A'..'F'] = 10 + (fromEnum x - fromEnum 'A')

numtohex :: Int -> Char
numtohex x | x `elem` [0..9] = toEnum(fromEnum x - fromEnum '0')
           | x `elem` [10..15] = toEnum (fromEnum 'a' +(x-10))


main :: IO()
main = do
    putStrLn $ "Change a from hex to num:"++ show (hextonum 'a') --10
    putStrLn $ "Change 15 from num to hex:"++ [numtohex 15] --'f'

    let hexChars = "0123456789abcdefABCDEF"
    let decimalNumbers = [0..15]
    let hexToDecimal = map hextonum "0123456789abcdefABCDEF"
    let decimalToHex = map numtohex decimalNumbers

    putStrLn "Hex to Decimal Conversion:"
    putStrLn hexChars
    print hexToDecimal

    putStrLn "\nDecimal to Hex Conversion:"
    print decimalNumbers
    putStrLn decimalToHex

    putStr "The test result: "
    print ( map (hextonum . numtohex) decimalNumbers)
```

Output:

```
ghci> :l CW1_A5.hs
[1 of 2] Compiling Main             ( CW1_A5.hs, interpreted )
Ok, one module loaded.
ghci> main
Change a from hex to num:10
Change 15 from num to hex:f
Hex to Decimal Conversion:
0123456789abcdefABCDEF
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,10,11,12,13,14,15]

Decimal to Hex Conversion:
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```