**COM2108 Assignment Report**

* The Enigma Simulation

Design

**Types:**

- Rotor: a tuple of the encryption string and the rotor knock-on position.

- Reflector: a list of tuples of characters containing the character reflections used in the enigma.

- Offsets: a three-tuple containing offsets for the left, middle and right rotors, respectively.

- Stecker: a list of tuples of characters containing a few pairs of letters that are plugs in Steckered Enigmas.

**-encodeMessage :: String -> Enigma -> String**

This function takes a string to encode and an Enigma type and returns the first parameter encoded with the second. This function makes use of the functions capMessage and truncateMessage which serve the purpose of formatting the provided message so the enigma machine can properly encode it.

**-encodeString :: String -> Enigma -> String**

This function takes an all-uppercase String and an Enigma type and returns the first parameter encoded with the second. It makes use of the functions encodeChar and incrementOffsets.

**-encodeChar :: Char -> Enigma -> Char**

This function takes an uppercase Char and an Enigma type and returns the first parameter encoded with the second. It makes use of the functions passByRotors, which encodes a character through the right, middle and left rotors; reflectChar, which reflects a character with the given reflector; and reversePassByRotors which encodes a character through the left, middle and right rotors in reverse.

**-incrementOffsets :: Offsets -> Rotor -> Rotor -> Rotor -> Offsets**

This function takes an Offsets type and three Rotor types and returns the first parameter incremented according to the Rotor types’ knock-on positions. For my implementations I ended up implementing this function as stated in the brief, so the offset in the left is increased once the one in the right progresses past the rotor’s knock-on position.

**-passByRotors :: Char -> Offsets -> Rotor -> Rotor -> Rotor -> Char**

This function takes a Char, Offsets type and three Rotor types and returns the first parameter encoded through the right, middle and left rotors, respectively, and with the given offsets. It makes use of the function passByRotor which encodes the character through a single character.

-**reversePassByRotors :: Char -> Offsets -> Rotor -> Rotor -> Rotor -> Char**

This function takes a Char, Offsets type and three Rotor types and returns the first parameter reverse-encoded through the left, middle and right rotors, respectively, and with the given offsets. It makes use of the function reversePassByRotor which reverse-encodes the character through a single character.

**-reflectChar :: Char -> Reflector -> Char**

This function takes a Char and a Reflector type and returns the first parameter reflected using the second parameter.

**-reversePassByRotor :: Char -> Rotor -> Int -> Char**

This function takes a Char, a Rotor type and an Int and returns the first parameter reverse-encoded through the second parameter with the third parameter being the offset. It makes use of the functions shiftBckwrd and shiftFrwd when there’s an offset, and also makes use of the functions alphaPos and getIndex to actually do the encoding.

-**passByRotor :: Char -> Rotor -> Int -> Char**

This function takes a Char, a Rotor type and an Int and returns the first parameter encoded through the second parameter with the third parameter being the offset. It makes use of the functions shiftBckwrd and shiftFrwd when there’s an offset, and also makes use of the functions alphaPos and getIndex to actually do the encoding.

-**steckerChar :: Char -> Stecker -> Char**

This function takes a Char and a Stecker type and returns the first parameter stickered through the second parameter. This function makes use of the reflectChar function; it checks whether the character given is in the stecker or not and if so, it uses the reflectChar function to reflect it using the pairs in the stecker provided. I was able to do this because both the Reflector and Stecker types are lists of Char tuples.

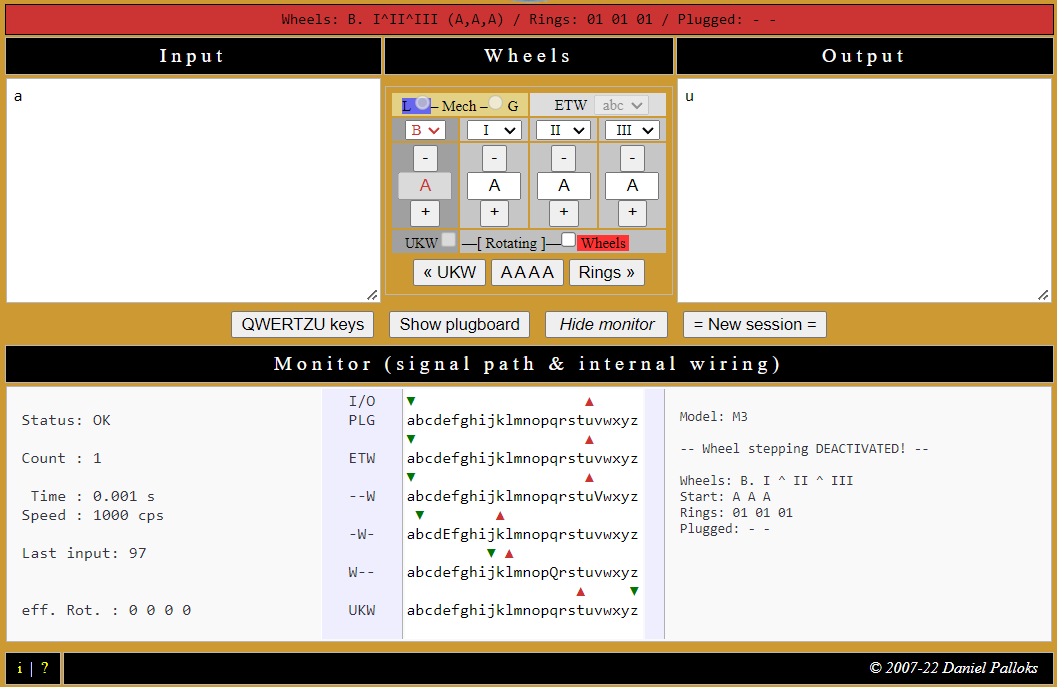
-**isInList :: Char -> [Char] -> Bool**

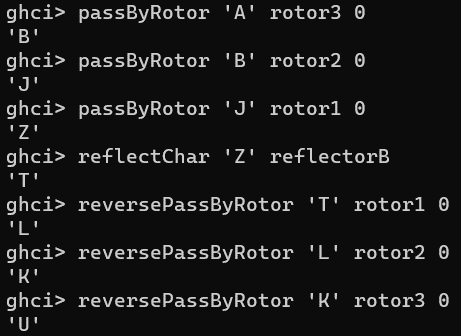
This function takes a Char and a list of Char and returns a Bool that is true if the first parameter is in the list given as the second parameter, false otherwise.

Testing

I implemented the functions in the following order: passByRotor, passByRotors, reflectChar, reversePassByRotor, reversePassByRotors, incrementOffsets, isInList, steckerChar, encodeChar, encodeString and encodeMessage.

For testing of the all the passByRotor functions, I used the Enigma Machine simulation at (<https://people.physik.hu-berlin.de/~palloks/js/enigma/enigma-u_v26_en.html>) which has a monitor that show the progress through the rotors as you encode. I essentially encoded characters with different offsets and then tested my functions for each rotor to check if the output was the same as in the simulation.





* Finding the Longest Menu

Design

**Types:**

- Menu: a list of Int, each representing a position in the crib.

- Crib: a list of Char tuples where each tuple contains a plain character and its encryption.

**-longestMenu :: Crib -> Menu**

This function takes a Crib type as a parameter and returns one of the longest menus you can get from the

Crib provided as a parameter

**-nextInMenu :: Int -> Crib -> [Int]**

This function takes an Int and a Crib type as parameters and returns a list of Int with all the next possible positions in the menu. For this I used the findIndices method from Data.List.

**-arrangeMenus :: [Int] -> [Int] -> [[Int]]**

This function takes two lists of Int as parameters and returns a list of list of Ints where each list is the first parameter concatenated with each of the elements in the second parameter.

**-completeMenu :: [[Int]] -> Crib -> [[Int]]**

This function takes a list of list of Ints and a Crib type as parameters and returns a list of list of Ints which is the complete set of menus that you can make from the Crib provided as the second parameter.

This function uses the functions arrangeMenus and nextInMenu to go through the crib and work out the menus; and the function removeFromCrib to update the crib and make it so the same position can’t appear twice in the same menu.

Testing

­­­­I implemented the functions in the following order: nextInMenu, arrangeMenus, completeMenu, longestMenu.

* **Simulating the Bombe**

Design

breakEnigma :: Crib -> Maybe (Offsets, Stecker)

tryAllOffsets :: Crib -> Menu -> Offsets -> Int -> (Offsets, Stecker)

tryAllAssumptions :: [(Char,Char)] -> Crib -> Menu -> Offsets -> Stecker

createAssumptions :: Crib -> Menu -> Int -> [(Char,Char)]

tryAssumption :: Crib -> Menu -> Stecker -> Offsets -> Stecker

incrementOffsetsBy :: Int -> Offsets -> Offsets

isSteckerValid :: Stecker -> Bool

formatStecker :: Stecker -> Stecker

intersectStecker :: [Char] -> Stecker -> [Char]

removeMirroredSteckers :: Stecker -> Stecker

removeRedundantSteckers :: Stecker -> Stecker

elmRepeats :: [Char] -> [Char] -> Bool