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.

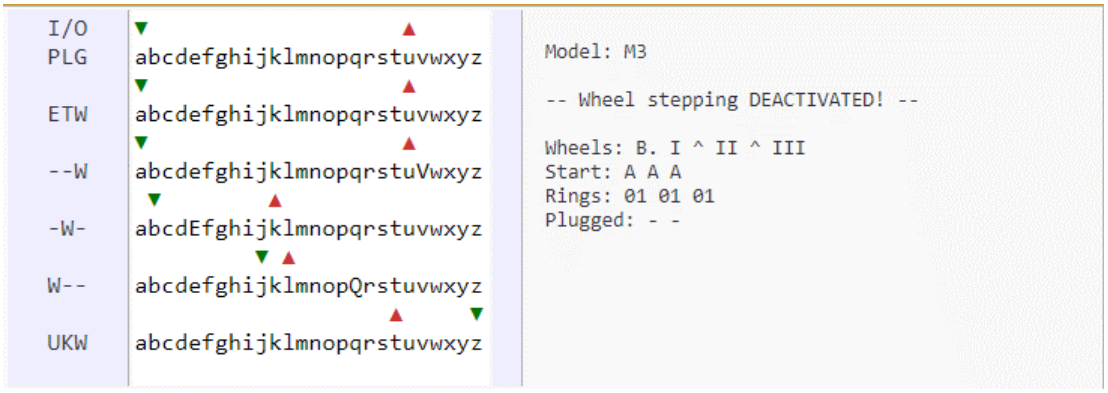
-isInList :: Char -> [Char] -> Bool
This function takes a Char and a list of Char and returns a Bool that is true if the first parameter is in the list given as the second parameter, false otherwise.

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 and encoding 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.

Example:




```
ghci> passByRotor 'A' rotor3 0
'B'
ghci> passByRotor 'B' rotor2 0
'J'
ghci> passByRotor 'J' rotor1 0
'Z'
ghci> reflectChar 'Z' reflectorB
'T'
ghci> reversePassByRotor 'T' rotor1 0
'L'
ghci> reversePassByRotor 'L' rotor2 0
'K'
ghci> reversePassByRotor 'K' rotor3 0
'U'
ghci> encodeChar 'A' (SimpleEnigma rotor1 rotor2 rotor3 reflectorB (0,0,0))
'U'
```

The simulation also has an option to add your own steckerboard so I added my own steckerboard and compared with the results I got for encodeMessage. The results stop matching after a few characters which worried me at first but I realised it was because the implementation of this simulation uses double stepping and increases the offsets when they reach knock on position whereas I implemented it where it increases the offsets when they progress past knock-on position. Below are some additional tests for low-level functions:

```
ghci> incrementOffsets (0,0,0) rotor1 rotor2 rotor3
(0,0,1)
ghci> incrementOffsets (0,0,22) rotor1 rotor2 rotor3
(0,1,23)
ghci> incrementOffsets (0,0,21) rotor1 rotor2 rotor3
(0,0,22)
ghci> incrementOffsets (0,5,22) rotor1 rotor2 rotor3
(1,6,23)
ghci> incrementOffsets (0,5,24) rotor1 rotor2 rotor3
(0,5,25)
ghci> reflectorB
[('A','Y'),('B','R'),('C','U'),('D','H'),('E','Q'),('F','S'),('G','L'),('I','P'),('J','X'),
ghci> reflectChar 'A' reflectorB
'Y'
ghci> reflectChar 'Y' reflectorB
'A'
ghci> reflectChar 'E' reflectorB
'Q'
ghci> reflectChar 'P' reflectorB
'I'
```

```
ghci> stecker = [('F','S'),('G','L'),('I','P'),('J','X'),('K','N')]
ghci> steckerChar 'A' stecker
'A'
ghci> steckerChar 'F' stecker
'S'
ghci> steckerChar 'L' stecker
'G'
ghci> steckerChar 'B' stecker
'B'
```

- 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. This function makes use of the function arrangeMenus to put all the initial indices in a list of lists and work out the menus from there. It also uses the function completeMenu with the arranged menus given as a parameter and gets the maximum, comparing length of all the menus returned by completeMenu.

-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.

To test nextInMenu I used the crib and message from the first Main.hs. I wrote down the letters in my notebook like in the assignment brief, with the index above them and I manually checked if the next indices the function returned were the correct ones.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
W	E	T	T	E	R	V	O	R	H	E	R	S	A	G	E	B	I	S	K	A	Y	A
R	W	I	V	T	Y	R	E	S	X	B	F	O	G	K	U	H	Q	B	A	I	S	E

```
ghci> nextInMenu 22 crib
[1,4,10,15]
ghci> nextInMenu 6 crib
[5,8,11]
ghci> nextInMenu 10 crib
[16]
ghci> nextInMenu 17 crib
[]
```

To test arrangeMenus I simply gave the functions a few different outputs and saw if it behaved like I wanted it to. The function is supposed to return a list of all the lists formed from appending each element in the second list given as a parameter to the first one. So, if I input arrangeMenus [1,2,3] [4,5,6] I expect the result [[1,2,3,4], [1,2,3,5], [1,2,3,6]]; and if I input arrrangeMenus [4,5,6] [1,2,3] I expect the result [[4,5,6,1], [4,5,6,2], [4,5,6,3]].

```
ghci> arrangeMenus [1,2,3] [4,5,6]
[[1,2,3,4],[1,2,3,5],[1,2,3,6]]
ghci> arrangeMenus [4,5,6] [1,2,3]
[[4,5,6,1],[4,5,6,2],[4,5,6,3]]
```

To test completeMenu I manually picked a few of the longest menus that were returned and run followed the menu by hand to check if it was correct, I used three different cribs and checked 5 of the longest menus in each of the list of menus for each crib. When running completeMenu with the first crib in the first Main.hs file I get 755 menus. I can get all the longest menus with the list comprehension: [x | x<-completeList, length x == length longest] In this example all of the longest menus are valid.

```
ghci> menu = longestMenu (zip "WETTERVORHERSAGEBISKAYA" "RWIVTYRESXBFOGKUHQBAISE")
ghci> completeList = completeMenu (arrangeMenus [] [0..(length crib - 1)]) crib
ghci> [x | x<-completeList, length x == length menu]
[[13,14,19,22,1,0,5,21,12,7,4,3,6,8,18,16,9],[13,14,19,22,1,0,8,12,7,4,3,6,5,21,18,16,9],[13,14,19,22,4,3,6
,5,21,12,7,1,0,8,18,16,9],[13,14,19,22,4,3,6,8,12,7,1,0,5,21,18,16,9]]
```

- **Simulating the Bombe**

Design

-breakEnigma :: Crib -> Maybe (Offsets, Stecker)
This function takes a Crib type as a parameter and returns a Maybe type that is either Just (Offsets, Stecker) or Nothing. The tuple of Offsets and Stecker types is the possible solution found for the crib given as a parameter. This function makes use of the functions longestMenu, to work out the menu for breaking the enigma; and tryAllOffsets to run the Bombe Simulation.

-tryAllOffsets :: Crib -> Menu -> Offsets -> Int -> (Offsets, Stecker)
This function takes a Crib type, a Menu type, an Offsets type and an Int as parameter and returns a tuple of Offests and Stecker types which are the possible solutions worked out from the crib using the menu and initial offsets. The Int given as the fourth parameter serves the purpose of stopping the function when, after the function is recursively called 17576-n times, it stops. 17576 = 26*26*26: all possible initial offsets. This function makes use of the functions formatStecker, to make sure the Stecker doesn't have any redundant plugs; createAssumptions to create a list of all the possible initial assumptions; tryAllAssumptions to run the simulation with all the assumptions calculated before; and incrementOffsets to step through the offsets.

-tryAllAssumptions :: [(Char,Char)] -> Crib -> Menu -> Offsets -> Stecker
This function takes a list of Char tuples, and Crib, Menu and Offsets types as parameters and returns a Stecker as a possible solution for the enigma. This function makes use of the function encodeString to verify the possible solution before returning it; tryAssumption to run the simulation with a single initial assumption; and fmtStecker to provide a proper stecker to the enigma given to encodeString.

-createAssumptions :: Crib -> Menu -> Int -> [(Char,Char)]
This function takes a Crib and Menu types and an Int and returns a list of Char tuples with all the initial assumptions for the given crib and menu. This function makes use of the functions alphaPos and charFromAlphaPos to identify the initial character and get the next ones, iteratively.

-tryAssumption :: Crib -> Menu -> Stecker -> Offsets -> Stecker
This function takes a Crib, Menu, Stecker and Offsets types and returns a possible stecker, if it can be found, running the bombe simulation with the given parameters. This function makes use of the functions isSteckerValid to continually check if the Stecker doesn't have a contradiction as the simulation makes its way down the menu; incrementOffsetsBy to increment the offsets to the position of the menu before encoding the character to add to the Stecker; formatStecker to format the stecker provided to the enigma used for encoding; and encodeChar to encode the characters and work out the stecker.

-incrementOffsetsBy :: Int -> Offsets -> Offsets
This function takes an Int and an Offsets type and returns the Offsets type provided incremented according to the first parameter. This function makes use of the function incrementOffsets to increment the offsets iteratively.

-isSteckerValid :: Stecker -> Bool
This function takes a Stecker type and returns a Bool that is true is the Stecker doesn't have a contradiction and false if it does. It makes use of the functions intersect and elmRepeats to find contradictions; and formatStecker to format the stecker before checking if it is valid.

-formatStecker :: Stecker -> Stecker
This function takes a Stecker type as a parameter and returns the given stecker formatted i.e., without redundant plugs such as ('B','B') or removed one of two mirrored plugs such as ('A','V') and ('V','A'). it makes use of the functions removeRedundantSteckers to handle the former and removeMirroredSteckers to handle the latter. I also use nub from Data.List to remove duplicates.

-removeMirroredSteckers :: Stecker -> Stecker
This function takes a Stecker type and returns the Stecker given keeping only one of all mirrored plugs such as ('A','V') and ('V','A').

-removeRedundantSteckers :: Stecker -> Stecker
This function takes a Stecker type and returns the Stecker given without all the plugs that plug a character to itself such as ('B','B').

-elmRepeats :: [Char] -> [Char] -> Bool
This function takes two lists of Char, the first of which is supposed to be empty, and returns a Bool that is true if the second list given as parameters has any duplicate element. This function makes use of the function isInList which I defined to check if a given element is in a given list.

Testing

I implemented the functions in the following order: elmrepeats, isSteckerValid, incrementOffsetsBy, tryAssumption, createAssumptions, tryAllAssumptions, tryAllOffsets, removeMirroredSteckers, removeRedundantSteckers, formatStecker and breakEnigma.

To test elmRepeats I simply provided it with several lists where elements repeated and some where no elements repeated and checked if it returned the expected Boolean.

```
ghci> elmRepeats [] ['A','B','C','D','E','F','G']
False
ghci> elmRepeats [] ['A','B','C','D','E','F','G','A']
True
ghci> elmRepeats [] ['A','B','C','D','E','F','G','B']
True
ghci> elmRepeats [] ['A','B','C','D','E','F','G','E']
True
ghci> elmRepeats [] ['A','B','C','D','E','F','G','Z']
False
```

To test isSteckerValid I Gave the function a few Steckers and checked if the function correctly classified them.

```
ghci> isSteckerValid [('A','T'),('F','G'),('N','M'),('Q','O')]
True
ghci> isSteckerValid [('A','T'),('F','G'),('N','O'),('Q','O')]
False
ghci> isSteckerValid [('A','T'),('F','G'),('O','M'),('Q','O')]
False
ghci> isSteckerValid [('A','T'),('F','G'),('O','Q'),('Q','O')]
True
ghci> isSteckerValid [('A','T'),('F','G'),('O','Q'),('Q','O'),('B','B')]
True
```

To test incrementOffsetsBy first worked out the result by hand with the given parameters and then checked if my function was returning the offsets correctly incremented.

Test	Expected Result	Result I got
incrementOffsetsBy 23 (0,0,0)	(0,1,23)	(0,1,23)
incrementOffsetsBy 23 (0,0,0)	(0,2,18)	(0,2,18)
incrementOffsetsBy 23 (0,0,0)	(0,3,17)	(0,3,17)
incrementOffsetsBy 23 (0,0,0)	(1,14,3)	(1,14,3)

To test tryAssumption I encoded the crib WETTERVORHERSAGEBISKAYA with different offsets and steckers and tried checked if I could reach a solution with the correct assumption and offset and if I couldn't if I provided the wrong offsets and initial assumption. Unfortunately, in the test example below the menu wasn't long enough to find all the plugs in the Stecker but it could find one.

```
ghci> string = "WETTERVORHERSAGEBISKAYA"
ghci> crypt = encodeString string (SteckeredEnigma rotor1 rotor2 rotor3 reflectorB (0,0,25) [('F','T'),('E','R')])
ghci> crib = zip string crypt
ghci> longestMenu crib
[0,17,21,5,15,14]
ghci> tryAssumption crib (longestMenu crib) [('W','X')] (0,0,25)
[]
ghci> tryAssumption crib (longestMenu crib) [('W','D')] (0,0,25)
[]
ghci> tryAssumption crib (longestMenu crib) [('W','E')] (0,0,25)
[]
ghci> tryAssumption crib (longestMenu crib) [('W','W')] (0,0,25)
[('E','R')]
```

To test createAssumptions I got random cribs and its menus and checked if it would correctly create the 26 initial assumptions for the menu at start position.

```
ghci> createAssumptions crib (longestMenu crib) 26
[('W','W'),('W','X'),('W','Y'),('W','Z'),('W','A'),('W','B'),('W','C'),('W','D'),('W','E'),('W','F'),('W','G'),('W','H'
,('W','I'),('W','J'),('W','K'),('W','L'),('W','M'),('W','N'),('W','O'),('W','P'),('W','Q'),('W','R'),('W','S'),('W','T'
,('W','U'),('W','V')]
```

To test tryAllAssumptions and tryAllOffsets I used Debug.Trace to trace the result as it tried to work it out and checked if the functions were doing what I expected them to. I can't illustrate this as I have already removed the import and al the traces from my program and I forgot to take screenshots of the testing as I was doing it.

To test breakEnigma I used the three Main files that were provided.

Below are some tests of low-level functions from this part:

```
ghci> removeRedundantSteckers [('R','R'),('E','E'),('W','W'),('I','I'),('Y','Y'),('E','E'),('S','S'),('A','A'),('F','T')]
[('F','T')]
ghci> removeRedundantSteckers [('R','R'),('E','E'),('W','W'),('I','I'),('Y','Y'),('E','E'),('S','S'),('F','T'),('F','T')
,('K','K')]
[('F','T'),('F','T')]
ghci> removeMirroredSteckers [('R','R'),('E','E'),('W','W'),('I','I'),('Y','Y'),('E','E'),('S','S'),('T','F'),('F','T'),
('K','K')]
[('R','R'),('W','W'),('I','I'),('Y','Y'),('E','E'),('S','S'),('F','T'),('K','K')]
ghci> formatStecker [('R','R')('E','E'),('W','W'),('I','I'),('Y','Y'),('E','E'),('S','S'),('A','A'),('F','T'),('K','K')]
ghci>formatStecker [('R','R'),('E','E'),('W','W'),('I','I'),('Y','Y'),('E','E'),('S','S'),('T','F'),('F','T'),('K','K')]
[('F','T')]
```

• Critical Reflection

Working with functional programming has been one of the most fun, yet annoying tasks I've had in university so far. Everything seems counter-intuitive at first but once it all clicks, it couldn't be simpler. One of the aspects I liked the most about functional programming is how you can almost read it as a text; meaning that if I can clearly express what I want to implement, in writing, translating that to code should be pretty straight forward. I feel like this time with Haskell enabled me to work better in other programming languages and has helped me to take a step back from my code and thinking about it in a way that makes it easy to implement. I found myself constantly having to go back to the design to think better about what arguments to provide to functions and how to make sure that every function was simple to code. Thinking about my design and implementing the code from the bottom up is something I'm definitely going to be doing for every coding assignment I have. It not only ensures that I have everything organised and I have a clear path to go through, but also makes it so I can gauge my progress as I go since I have to thoroughly test every function before, I use it in other functions. All in all, functional programming was incredibly useful to learn and will certainly help me in my future coding in any language.