COM2108: Functional Programming – Autumn 2019
Assignment 2: The Enigma Machine, Cribs and Menus

This assignment is 30% of the assessment for COM2108

1 The Enigma Machine

An Enigma Machine

The Enigma machine was based on rotors. A rotor implemented a fixed alphabetic substitution cipher. The ciphers for the original five Enigma rotors RI, RII, RIII, RIV and RV were

```
plain   ABCDEFGHIJKLMNOPQRSTUVWXYZ
RI      EKMFLGDQVZNTOWYHXUSPAIBRCJ
RII     AJDKSIRUXBLHWTMCQGZNPYFVOE
RIII    BDFHJLCPRTXVZNYEIWGAKMUSQO
RIV     ESOVPZJAYQUIRHXLNFTGKDCMWB
RV      VZBRGITYUPSDNHLXAWMJQOFECK
```

Like the ciphers in assignment 1, a rotor could be **offset** by a given number of positions (0 to 25), however this had a slightly different effect to the offset in a the ciphers of that assignment. The wiring of the rotor was *fixed*, so changing the offset shifted the rotor relative to its input/output positions. For example, figure 1 shows rotor I in its "home" position and in position 1 (1 step counter-clockwise), together with the encoding for input 'A' in both cases.

The standard Enigma had 3 rotors, mounted on a shaft. We'll refer to them as the left, middle and right rotors, LR, MR and RR.

A codebook given to all Enigma operators specified a different selection for LR, MR & RR for each day from the available rotors RI,RII..RV. These were mounted on the shaft in the correct order. In addition, initial offsets OL, OM and OR were specified for LR, MR and

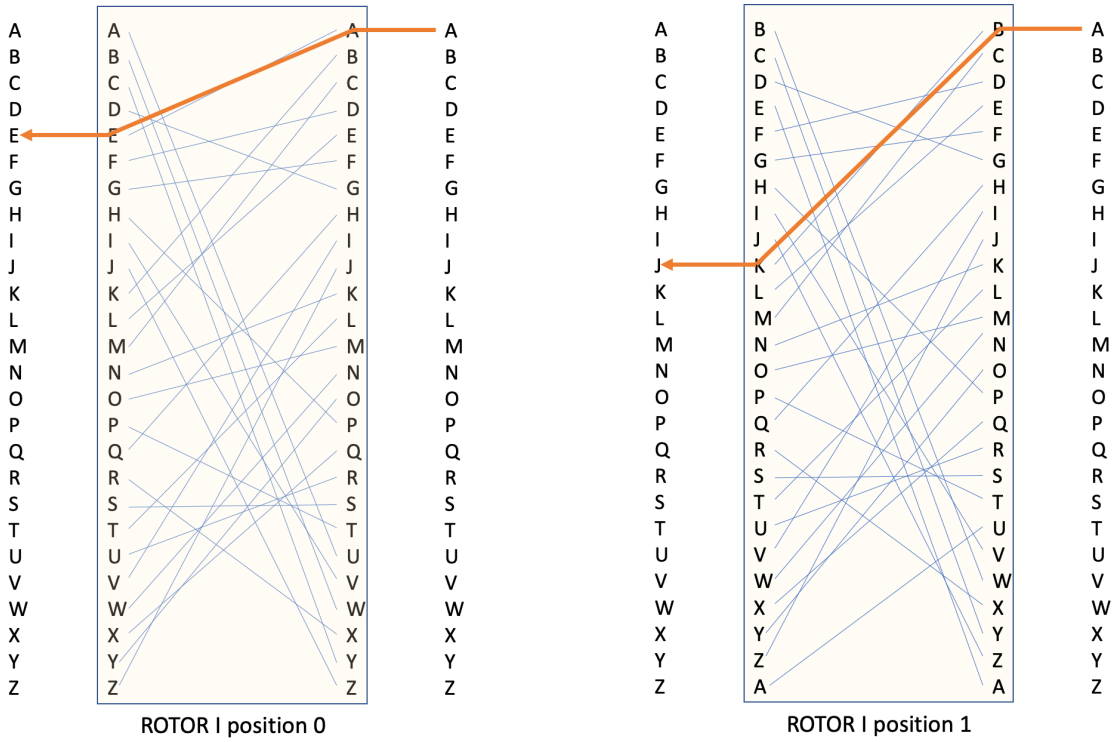

Figure 1: Wiring of rotor I in its "home" position (left), and with offset 1 (right). The red arrows show the output for an input of 'A' in each case.

RR. Thus all the Enigmas in use were set to the same starting positions each day. We'll make the assumption that before sending each message the offsets were reset to OL, OM and OR (in reality it was more complicated).

In the basic Enigma (or 'unsteckered' Enigma – see later for an explanation of steckering), a letter was first transmitted to RR, which encoded it and transmitted the result to MR, which encoded that and transmitted it to LR. The encoded letter from LR was passed to a fixed **reflector** which performed a letter-swap (i.e. the 26 letters were divided into 13 pairs (c1,c2) where c1 was changed to c2, and c2 to c1). The standard reflector pairings (known as reflector B) were

```
(A Y)(B R)(C U)(D H)(E Q)(F S)(G L)(I P)(J X)(K N)(M O)(T Z)(V W)
```

The output from the reflector was then passed back through LR, MR and RR in turn, by reverse-encoding, to the output lamps, where the lamp encoding the original character was lit.

For example, if RR is RI, MR is RII and LR is RIII, and the offsets were all 0, then if A is input, N will light:

Key press	RR	MR	LR	reflector	LR	MR	RR	Lamp
A	E	S	G	L	F	W	N	N

Note that this process is **symmetric and reversible**, i.e. we have just encoded A to N. If,

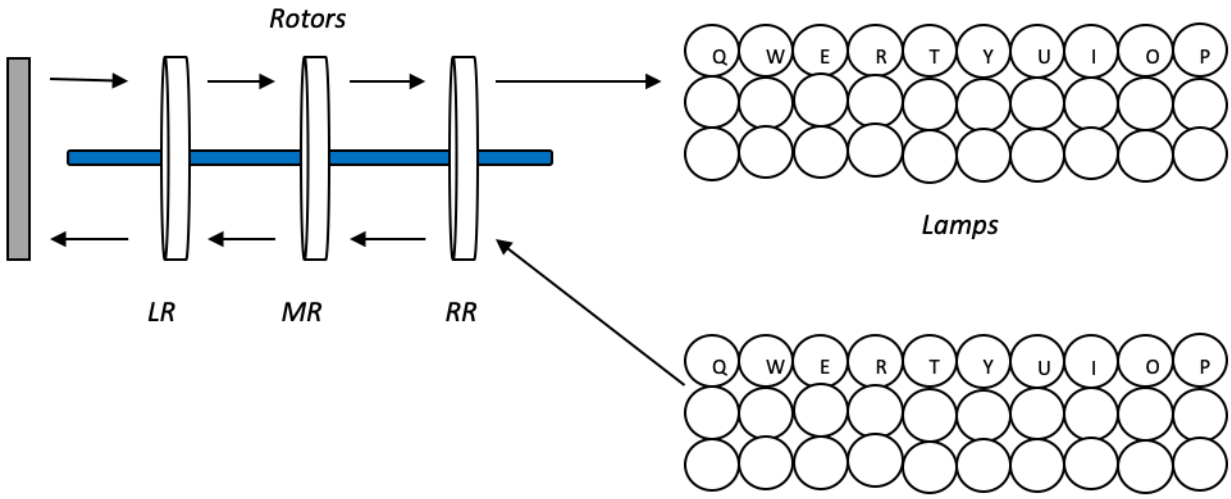

Figure 2: Schematic for a simplified Enigma machine

with the same starting point, we press N we get A:

Key press	RR	MR	LR	reflector	LR	MR	RR	Lamp
N	W	F	L	G	S	E	A	A

1.1 Advancing the rotors

Each key press changed the Enigma encoding:

When a key was pressed, the first action, before encoding the character, was to advance RR by 1 place, i.e. $OR \rightarrow OR + 1$, unless OR was 25, in which case

1. $OR \rightarrow 0$
2. $OM \rightarrow OM{+}1$

So when RR completed a revolution, OM was advanced. Similarly if OM completed a revolution, OL was advanced.

Finally, if the rotors were all on 25, the next key press would reset them all to 0. That's what would happen to generate the example above: if the key A was pressed with the rotors at 25,25,25 they would reset to 0,0,0 and the N lamp would light.

1.2 Decoding

The reversible property means that if the receiver of the message (another Enigma operator) sets her/his Enigma to the same starting point, s/he can decode an incoming message simply by typing it in – the lamps for the original text will appear.

2 Tasks

The machine as described above was called a **Simple Enigma**.

Using your code from assignment 1 (revised/updated as necessary),

1. Define Haskell data structures for a `Rotor`, a `Reflector`, and a set of `Offsets` (which specifies the offset of each rotor as an integer).
2. Using these data structures, add this algebraic type to your code:

   ```
   data Enigma = SimpleEnigma Rotor Rotor Rotor Reflector Offsets
   ```

   (This has *not* been included in the help file, because you need to define the other types before it will work.)
3. Define `enigmaEncode`: Given a character to encode, and an enigma (with appropriate rotors, reflector and offsets), returns the encoded letter.
4. Define `enigmaEncodeMessage`: Given a message to encode, and an enigma, returns the encoded message.

2.1 Steckering

In wartime use the Enigma was augmented by a **'Steckerboard'**, in which pairs of letters were plugged together. If X was 'steckered' to Y then the steckerboard would output Y if given X and vice-versa. Unlike a reflector, the pairings were not complete: there were up to 10 pairs, the remaining letters being transmitted unchanged. The 'steckering' i.e. the letter pairs, was changed every day.

A single steckerboard was placed between the keyboard and the Enigma and between the Enigma and the lamps:

5. Define a data structures for a `Steckerboard`.
6. Extend the algebraic type given above for Enigma so that it is *either* a `SimpleEnigma` or a `SteckeredEnigma`. (The `SteckeredEnigma` should be the same as a `SimpleEnigma`, with the addition of a `Steckerboard` as its final argument.)
7. Extend `enigmaEncode` and `enigmaEncodeMessage` so that they work for a steckered Enigma as well as a simple Enigma.
8. Check that you can still decode a message enciphered by a steckered Enigma with an identical steckered Enigma.

2.2 Cribs and Menus

Alan Turing's method for breaking the Enigma code was based on **'cribs'**. Military messages were highly formulaic and it was possible to make good guesses at phrases they would contain and the points in the message where these phrases might be found, e.g. the sender would be identified early in the message and **WETTERVORHERSAGE** (weather forecast) followed by a place name might appear near the end. A very short message might well be 'nothing to report'. Experts at Bletchley Park were capable of making good guesses at the cribs.

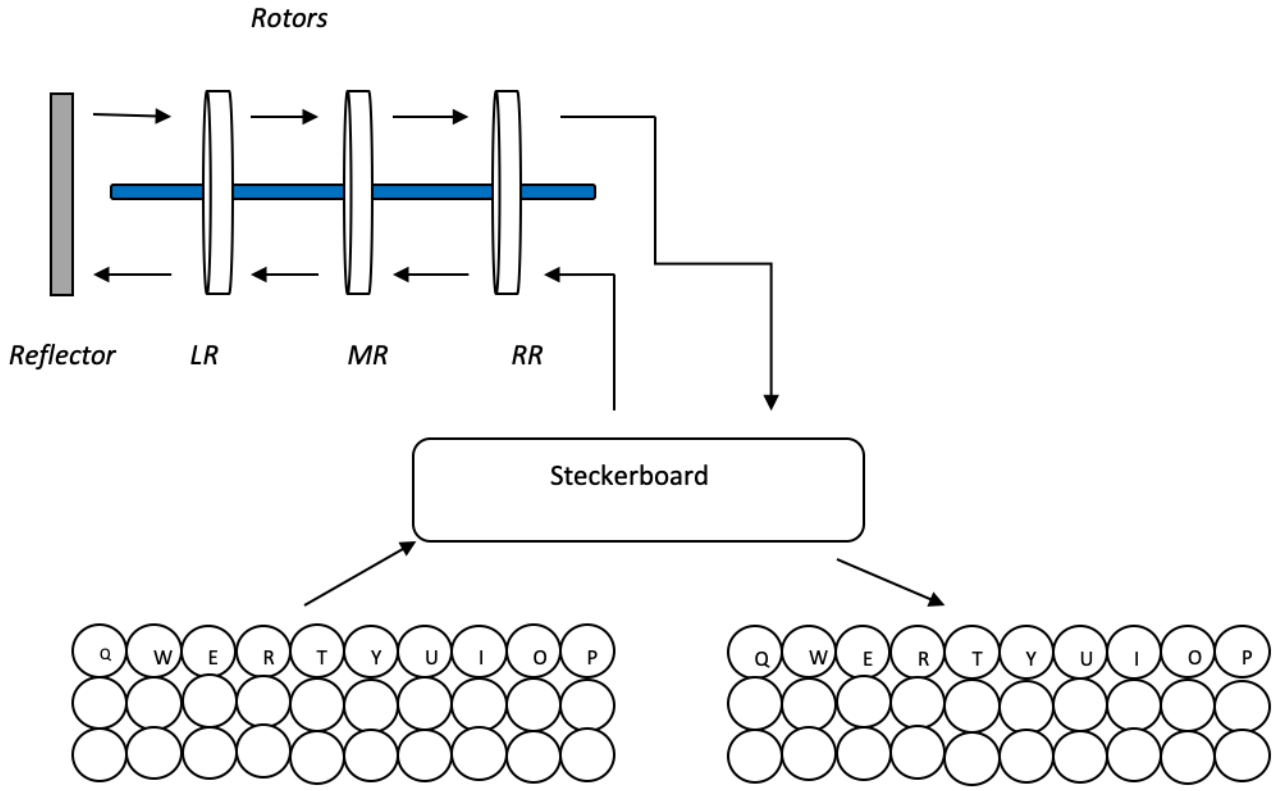

Figure 3: Schematic for a Steckered Enigma Machine

This is a real example of a crib:

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

The code breaking process depended on finding a long chain of letters linking plain and cipher in the crib. A link in this chain between index i and index j means that the cipher character at i is the same as the plain character at j.

e.g. in the above, a chain starting at position 1 is: `[1, 0, 5, 21, 12, 7, 4, ...]`. Such a chain was called a **menu**. At Bletchley Park menus were found by hand but we will write a function to find the longest menu in a crib. In the above example `[13,14,19,22,4,3,6,5,21,12,7,1,0,8,18,16,9]` is one of several menus of length 17.

9. Define data structures for a `Crib` and a `Menu`
10. Write `longestMenu` which is given a crib and returns the longest menu.

3 Marking Scheme

Task	Credit(%)
Data structure definitions: Rotor Reflector SteckerBoard Crib Menu	**10**
Correctness of: enigmaEncode (for SimpleEnigma) enigmaEncodeMessage (for SimpleEnigma) enigmaEncode (for SteckeredEnigma) enigmaEncodeMessage (for SteckeredEnigma) longestMenu **(subtotal)**	 10 5 10 5 20 **50**
Coding style	**20**
Documentation	**20**
Total	**100**

Table 1: Mark breakdown for assignment 2

As for assignment 1, correctness of your code will be assessed using automated testing. This testing **will fail** if you do not write your functions exactly as specified. This means exactly the same names, exactly the same arguments (in exactly the same order) and exactly the same return type for the functions. Furthermore, **you need to ensure that you data structure definitions are correct**, because if they are not, it is also highly likely that the automated testing will fail.

4 Submission

Work should be submitted via the appropriate link on Blackboard. Any work received after the deadline will have standard lateness penalties applied (see `https://sites.google.com/sheffield.ac.uk/comughandbook/general-information/assessment/late-submission`).

You may submit **one** Haskell (`.hs`) file, which should include your name in an initial comment. **You should not submit the help file.** As for assignment 1, it will be provided as-is for testing. It is recommended that you take a copy of your code from assignment 1 and use this as the basis of your work for assignment 2.

In the comments for your code, you should clearly state any known limitations, or be clear about modifications that you made to address any identified limitations – the limitations are things that you should have identified when testing your work (or if you planned your work perfectly, that you identified before you started coding). If know *know* your code has limitations that you have been unable to address, you will achieve a better score if you clearly discuss those limitations in your comments than if you ignore them and your code fails test cases.

5 Individual Work

You are reminded that this is intended to be assessment of *your own, individual work*, in order to assess whether you have achieved with the learning outcomes for this module. If there is any suspicion that that you have utilised unfair means, the department will take appropriate action, which could lead to disciplinary procedures.

You should have had a discussion about this subject with your personal tutor in first year. If you need refreshing on the rules, please see the Computer Science Undergraduate Handbook entry on the topic at

`https://sites.google.com/sheffield.ac.uk/comughandbook/general-information/assessment/unfair-means`

and/or discuss the matter with your personal tutor.