COM2108: Functional Programming – 2022

Functional Programming Design Case Study

This document sets out the case study for the grading assignment and the tasks that you need to complete for that assignment. It explains the Enigma machine (the machine used extensively by Nazi Germany in World War II to protect sensitive communication) and the Bombe (the machine developed by codebreakers at Bletchly Park to help decode messages that had been encrypted with an Enigma machine). Specifically, the Bombe was designed to discover some of the daily settings of the Enigma machines on the various German military networks: the set of rotors in use and their positions in the machine, the rotor start positions for the message, and the wirings of the plugboard.

The purpose of the grading assignment is to determine how *well* you have developed your skills in functional programming. Remember, you will already have achieved 40% in the module by passing the threshold test; any marks you earn here are in addition to that. The grading assignment is **meant** to be difficult: the average module mark is expected to be about 60%. You would get 60% in the module if you achieve $\frac{1}{3}$ of the marks for the grading assignment – in other words, the average student should expect to receive about $\frac{1}{3}$ of the marks for this assignment. Check the marking scheme carefully, noting that *correctness* of your solution only contributes a small amount to the mark.

Please note that there is **a lot** of background reading in this document. The point being, you are being tested on your ability to take a problem and apply a functional approach to solving it. You are being tested not just on your ability to code a solution, but your ability to analyse a problem, develop a solution and test that your solution satisfies the problem.

1 The Enigma Machine

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

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

Figure 1: An Enigma Machine
Museo della Scienza e della Tecnologia "Leonardo da Vinci", CC BY-SA 4.0 https://creativecommons.org/licenses/by-sa/4.0, via Wikimedia Commons

These rotors were internally wired with these mappings, so an input at position A (0) on rotor I would always give an output of E, and an input at position Z (25) on rotor II would always give an output of E. Any rotor could be **offset** by a given number of positions (0 to 25), which has a slightly different effect to the shift in a Caesar cypher (that we looked at in a programming exercise). The wiring of the rotor was *fixed*, so changing the offset shifted the rotor relative to its input/output positions. For example, figure 2 shows rotor I in its "home" position (left) and in position 1 (1 step counter-clockwise) (right), together with the encoding for input 'A' in both cases. In other words, the shift changes the input letter to the rotor, which forces the signal through a different path *internally* to the rotor, but also, the *output signal* is also shifted (again, look at the right-hand part of the diagram, where the output K from the rotor results in a J).

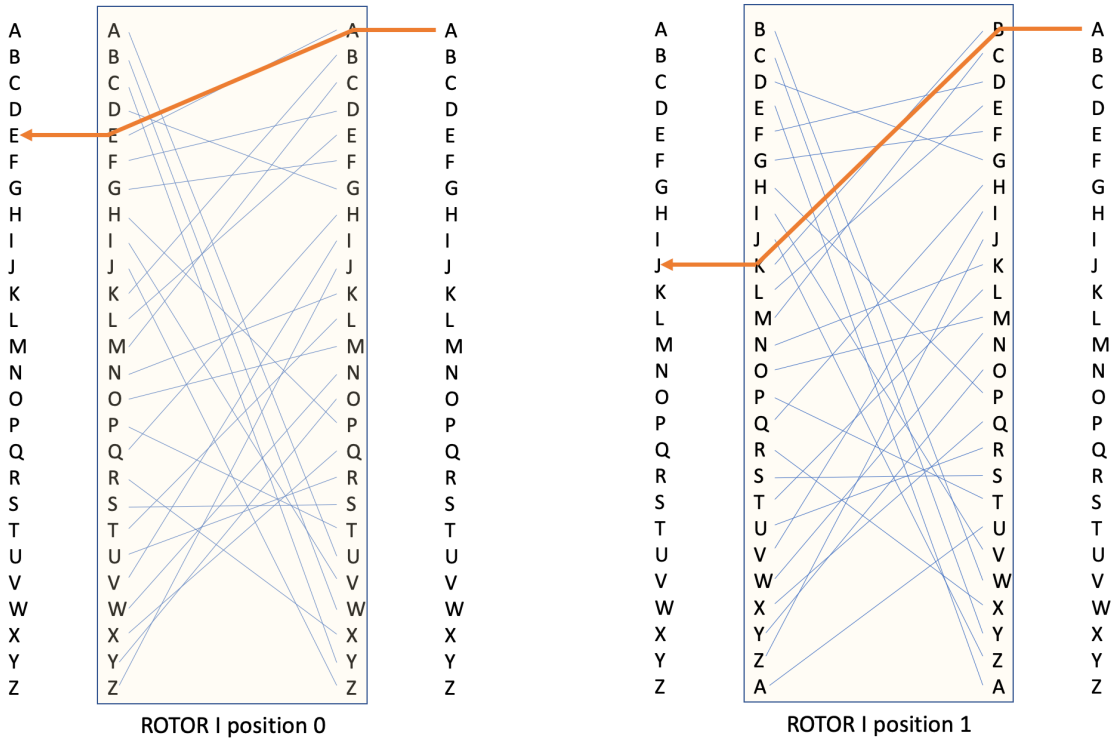

Figure 2: Wiring of rotor I in its "home" position (left), and with offset 1 (right). The red arrows show the output for an input of 'A' in each case.

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 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.

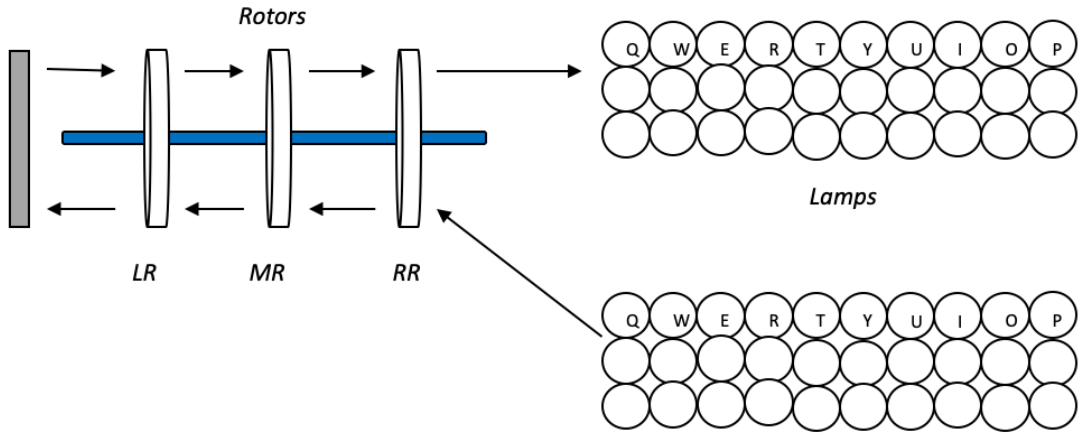

Figure 3: Schematic for a simplified Enigma machine

For example, if RR is RI, MR is RII and LR is RIII, and the offsets were all 0i[1], 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, 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]Actually, the first thing that happens when a key is pressed is that the right rotor advances, so to actually get this output, the initial offsets would have to be (0,0,25). This is explained further in the next section.

1.1 Advancing the rotors

Every rotor had a pin, at a different position for each rotor, which, when passed, would cause the left-adjacent rotor to advance by one position. The *rightmost* rotor advanced with every keypress.

The knock-on positions for each of the rotors was as follows:

Rotor	Knock-on Position	(Numerically)
I	R	17
II	F	5
III	W	22
IV	K	10
V	A	0

Cryptanalysts at Bletchly Park used the mnemonic *Royal Flags Wave Kings Above* to remember these positions.

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 at position **R** before the key was pressed, in which case

1. $OR \rightarrow (OR{+}1) \bmod 26$
2. $OM \rightarrow (OM{+}1) \bmod 26$

So when RR progresses past its knock-on position, OM was advanced. Similarly if OM progresses past its knock-on position, OL was advanced. If either rotor is at the Z position, the next advance takes it to A.

Note that the rotor movement happens *before* the signal passes through the rotors to generate the encrypted letter. So for the example above, the rotors would be in position 0,0,25 (or A, A, Z) initially: if the key A was pressed with the rotors at this position they would advance to position 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.

1.3 Steckering

In wartime use the Enigma was augmented by a **plugboard**, in which pairs of letters were plugged together. If X was 'steckered' to Y then the plugboard 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. Here is an example:

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

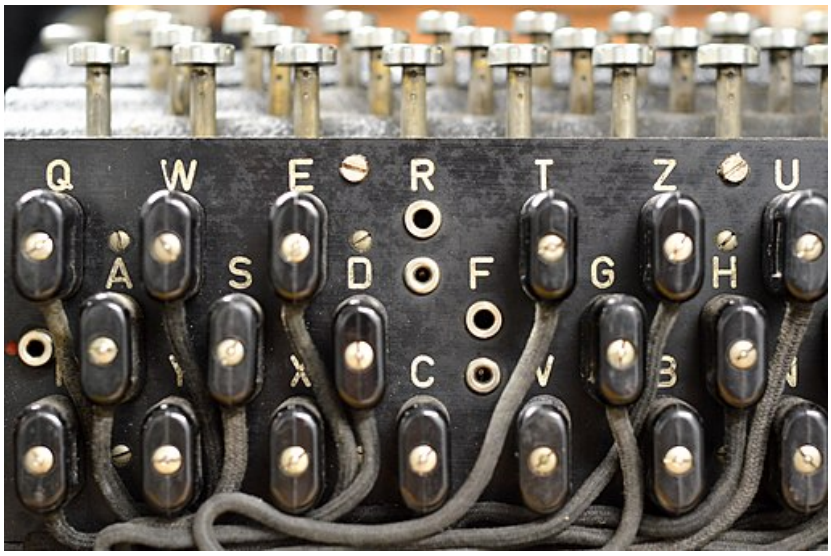

Figure 4: A "Steckerboard" (plugboard) for an Enigma Machine
School of Mathematics - University of Manchester, CC BY 2.0
https://creativecommons.org/licenses/by/2.0, via Wikimedia Commons

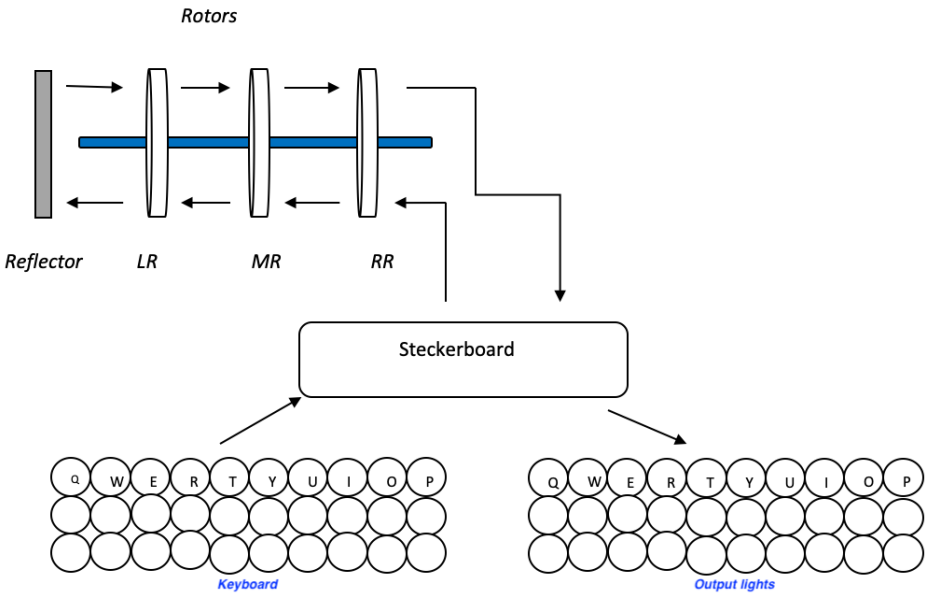

Figure 5: Schematic for a Steckered Enigma Machine

A steckered enigma machine was still symmetrical (that is, you could decode a message from a steckered enigma machine using an identical steckered enigma machine).

2 Breaking the Enigma

2.1 The problem the codebreakers faced.

The Bletchley Park (BP) codebreakers knew how the Enigma machine worked, and they knew that the military version was steckered. They also knew the 5 rotors, with their substitution codes, and the reflector pairs[2].

The codebreaker's daily task was to find the choice of 3 rotors from 5, the initial offsets and the stecker pairs, on the basis of the messages which were being transmitted on that day.

The number of possible solutions was around 10^{23}. Computers hadn't been invented...

2.2 Bombes

The codebreakers didn't have computers but they could build hardware to simulate simple Enigmas, using components adapted from telephone exchanges. These devices were called Bombes (named after an ice cream). A restored Bombe can be seen in Fig. 6, Given a choice of rotors and initial offsets, a bombe could find the encoding for any character at any position in the message.

Each vertical set of 3 dials corresponds to the 3 rotors of an Enigma. So a bombe could run 36 simulations. It took around 20 minutes to go through the 26^3 offsets.

Over 200 Bombes were built, but the vast majority were destroyed at the end of the war, and the remaining few, shortly thereafter.

Figure 6: A Bombe
Ian Petticrew, CC BY-SA 2.0 https://creativecommons.org/licenses/by-sa/2.0, via Wikimedia Commons

2.3 Cribs and Menus

Codebreaking was dependent on having a Crib and a Menu to guide the search through the crib.

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.

[2] Actually, there were different versions of the Enigma machine, and some had up to eight rotors to choose from. Most used just three rotors at any given time, but some had four. There were also different reflectors available. We are simplifying the problem slightly by considering the army/airforce enigma machine that had just five rotors and the fixed reflector B.

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 you 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.

In the Bombe, the implication-chasing process was implemented in hardware. The key component was called a **diagonal board**.

2.4 Codebreaking

Suppose you assume

- a particular arrangement for the Enigma rotors and reflector (say RI, RII, RIII, Reflector B)
- a particular set of initial offsets, say (0,0,25).

You should find out if, for these choices, we can find a set of plugs for the plugboard which is compatible with the crib by using its menu.

- If you can't, change the choice of initial offsets.
- If you run out of choices for initial offsets, change the Enigma configuration (but you should use test cases that do **not** require you to do this - you shouldn't need to code this step).

The crib has implications for the stecker, as illustrated below:

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
input							E															Y	
out							N															E	
cipher	K	A	Z	E	O	C	E	B	U	P	D	H	Z	C	D	T	P	A	Z	T	O	V	I

Suppose the menu is `[21,6,15,3,1,22,17,20,7,16]`

- Start with an assumption about the Steckerboard for the Char at the menu start position. Let's assume it's unsteckered i.e. the initial Steckerboard is `[('Y','Y')]`
- Suppose the enigma encodes Y to E at the menu start position, 21, with the given offsets. Then ('E','V') must be added to the Steckerboard.
- Following the menu, this means that at position 6 the input to the Enigma is E and, if its output is N, so we try to add ('N','E') to the stecker. We can't do this though, because ('E','V') is already in our Steckerboard.

- So the initial steckerboard `[('Y','Y')]` doesn't work out and we try the next, say `[('Y','Z')]`.
- If you reach the end of the menu without finding a contradiction you have a potential solution (i.e. initial offsets and a Steckerboard which is compatible with the menu)
- If all 26 initial Steckerboard pairs have been tried we change the initial offsets and go through this process again.
- If all 26 initial Steckerboard pairs have been tried for all initial offsets, we should return Nothing (no solution can be found).

3 Your Tasks

You are required to write simulators for the Enigma machine and the Bombe (in a simplified form). **Use the stub provided in Enigma.hs.** Also study carefully the way functions are being called in Main.hs: this file shows the way arguments are expected to be passed to the functions you are required to write. (You are expected to write many more functions too, but the three functions `encodeMessage`, `longestMenu` and `breakEnigma` are the three that will be tested.) If your types do not align with that used in the automated testing, your code will not compile.

If you submit code that does not compile, you will receive 0 for the implementation section of your work.

3.1 The Enigma Simulation

At the top level, you must have a function `encodeMessage`. The type definition is provided in the stub file, along with the definition of a type to represent an Enigma machine specification:

```
data Enigma = SimpleEnigma Rotor Rotor Rotor Reflector Offsets
            | SteckeredEnigma Rotor Rotor Rotor Reflector Offsets Stecker
```

The type definitions for the sub-components (Rotor, Reflector, Offsets, Stecker) are *not* correctly provided. You will need to supply these, based upon the sample usage in Main.hs. Look at the intended usages of encodeMessage (as shown in Main.hs), then write appropriate type definitions. However, do not start to write any other code yet!

3.1.1 Document the Design

As discussed in lectures, **before you even think about the code**, you should be thinking about the design. For this assignment, you must create a report in which you clearly explain (diagrams can be helpful) the design of your solution.

3.1.2 Implement your Design

When you have completed your design, you should implement it. A good design should be relatively easy to translate into code; if you are finding the implementation difficult, you should consider going back to your design and putting more detail into it. However **don't**

try to implement it all at once! You should think carefully about the order in which you implement functions and how you test them. (See the next subsection.)

When you implement your code, remember to use good functional style, choose sensible names (for functions, parameters and types), and include comments *where needed*. Note the "*where needed*" – well-written code should be largely self-explanatory. A brief comment explaining the purpose of most functions is a good idea (but you may have some very simple low-level functions that require no explanation), and header documentation (at the top of the file) is essential, but you generally don't need a comment for every line of code (only when it's not clear what they do). You are encouraged to ask for feedback in lab classes about style and comments.

3.1.3 Document your Testing

Add a section to the document you created to document your design. In this section, clearly indicate:

1. The order in which you implemented your functions.
2. For each function:
   - The rationale behind the testing for that function
   - Some illustrative examples, showing the test input and output. This (small) set of examples should clearly demonstrate that your function is working as intended. (A table would work well here!)

Think carefully about your tests. I do not want to see **every** test that you performed, but I want to see evidence that you chose an appropriate range of test data and that your functions will work both for general and edge cases.

3.2 Finding the Longest Menu

As described previously, *cribs* played an important part in cracking the enigma code. (See Section 2.3.) For this part of your assignment, you must:

1. Define data structures for a `Crib` and a `Menu`
2. Write `longestMenu :: Crib -> Menu` which is given a crib and returns the longest menu.

As for the previous stage, you should:

1. Document your design
2. Implement your design
3. Document your testing

3.3 Simulating the Bombe

To simplify things, assume that the choice of rotors and reflector is always the same: the left rotor will be RI, the middle rotor RII and the right rotor RIII. The reflector is the

standard one (which was called reflector B). Once you have working code you can try removing these restrictions.

Note that you are coding a search, one that could go on a long time. You should therefore invent test cases for for which the solution will be found quickly – in particular, think carefully about sensible initial rotor positions. We can easily construct such test cases if we have built the functions to simulate steckered enigmas.

Your challenge is to write a function `breakEnigma :: Crib -> Maybe (Offsets, Stecker)` that will search for a solution, given a crib. Note that the function has a *Maybe* return type, because it might not be able to find a solution!

As for the previous stages, you should:

1. Document your design
2. Implement your design
3. Document your testing

Note that design is **absolutely** critical. (I mean, it always is, but there is no way you can complete this stage without thinking carefully about the design!) If you run into difficulties, try to switch off from the code entirely and focus on the algorithm.

3.4 Critical Reflection

You should now have three main sections in your report:

1. Simulating the Enigma
2. Finding the Longest Menu
3. Simulating the Bombe

Each of these sections will describe the design and the testing for that component (or at least the parts of that component that you have managed to implement).

The final section of your report should be a **Critical Reflection** on your experiences. What is a critical reflection? It is based around the idea that all of our experiences should be part of an ongoing cycle of reflection:

In this final section, you should write a **brief** (no more than half a page) critical reflection that picks up on one or two specific experiences in working with functional programming and discusses how you will use that experience and your reflection upon it to your wider activities in the future.

4 Submission

You must make two separate submissions for this assignment:

1. The report, submitted as a Microsoft Word or text-based PDF file, using the appropriate link (Report) on Blackboard (the name of this document is not important), and

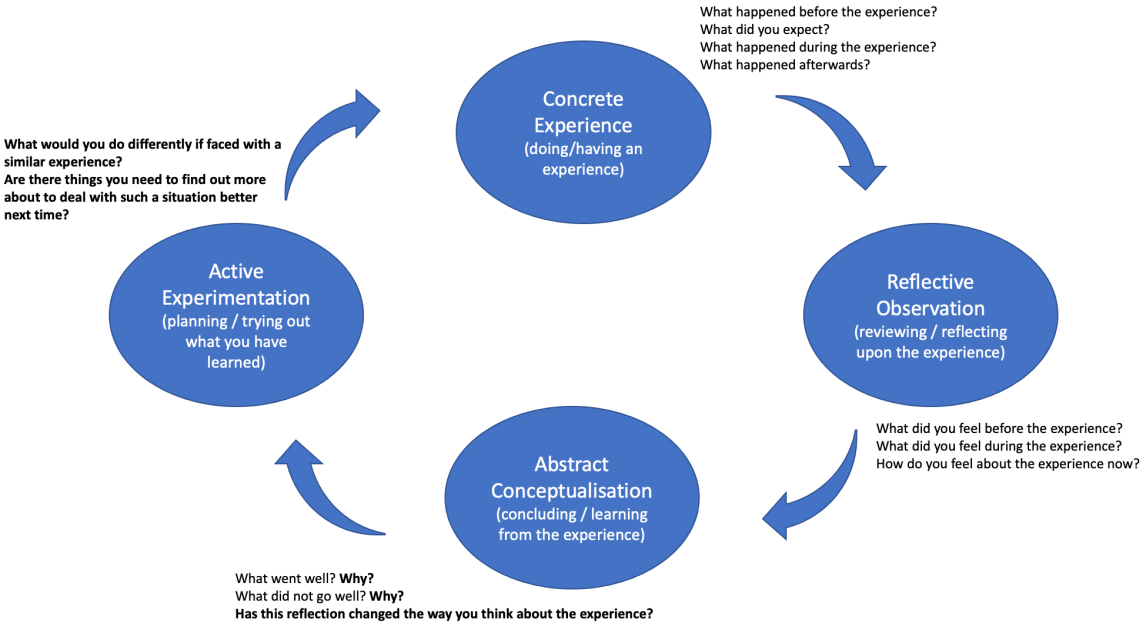

Figure 7: Kolb's experiential learning cycle
Adapted from: Kolb, David A. Experiential learning: Experience as the source of learning and development. FT press, 2014.

2. A single source file, `Enigma.hs`, which implements your solution in a module called Enigma (as per the stub). This source file will be automatically tested, as per the Main.hs example provided on Blackboard (but the inputs for the automated testing will be different to what is in that example). Again, make sure you use the correct (Implementation) link on Blackboard.

5 Assessment

You will be assessed upon both your code and the report that you submit.

For your code:

- It should produce correct results. `encodeMessage` should encode/decode messages correctly.
- It should be well-presented. This includes the organisation of functions, the layout of the lines, the choice of names, whitespace used appropriately for clarity.
- It should have appropriate comments - header comments, function comments, and where appropriate, inline comments.

For your report:

- It should be divided into the four subsections as described.
- For each of the first three sections:

  - The design must be clear, including the functions and the data flow.
  - The description of testing should indicate the *order* in which you have implemented the functions and convince me that you have put serious thought into how to appropriately test each function (and have done that testing!).

- For the final section, you will be marked on how clearly you express your critical reflection and the relevance of the reflection.

Element	Checked for	Proportion of marks
Report		
Design	For each section (encodeMessage, longestMenu, breakEnigma), a clear decomposition from high-level problem down to low-level functions.	30
Testing	For each section, a clear statement of the order of implementation of functions (showing a sensible choice of order) and evidence that you have considered both general and special cases in your test sets.	20
Critical Reflection	You have identified a specific experience and explained how this has changed your practice in the longer term. I am particularly looking for evidence that this is transferrable to practice beyond this particular module.	10
Implementation		
Correctness	encodeMessage	5
	longestMenu	5
	breakEnigma	10
Style	Sensible naming convention, layout, ordering of functions, length of lines, general readability	10*
Comments	Header and (sensible) function comments included, inline comments used where necessary (too many are as bad as too few).	10*

Table 1: Mark breakdown for grading assignment

Notes:

1. If you submit code that does not compile, you will get 0 for the implementation component.
2. The marks for style and comments will be scaled proportionally to the *correctness* of your code. If you get 5/5 for encodeMessage, 1/5 for longestMenu, 2/10 for breakEnigma, 8/10 for style and 5/10 for comments, your mark for the implementation section will be calculated as:

$$(5+1+2)+((8+5)\times\frac{(5+1+2)}{20})=8+5.2=13.2$$

(**not** the 21 that would arise from the straight summation of results.)