COM2108 Functional Programming Assignment: The Enigma Problem Report

Priscilla Emasoga

December 8, 2022

1 The Enigma Simulation

1.1 Design

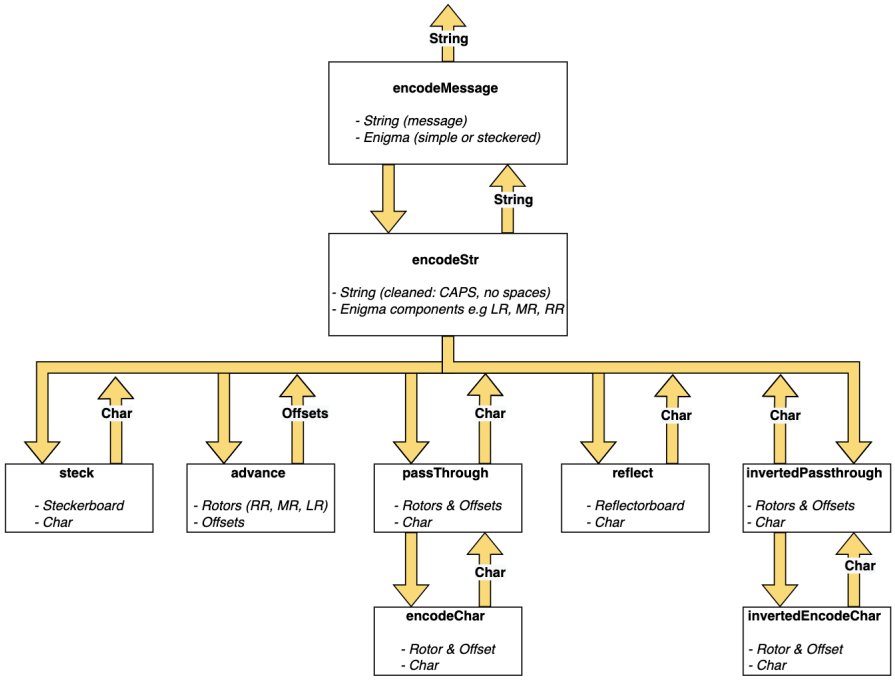

Figure 1: The Enigma Design

Figure 1 demonstrates the design of the enigma simulation. Following the top-down design approach, the **encodeMessage** function is the top-level function that takes two kinds of enigma data types (simple and steckered enigmas) and an unencrypted message and returns the encrypted version of the message or vice versa. To do this, It utilises the various low-level functions defined under it, starting from the **encodeStr** function.

The **encodeStr** function plays an integral role in the encryption or decryption of a message. It takes all the components of the specified Enigma (Steckerboard, Reflectorboard and a choice of three rotors) and a capitalised version of the input message with all characters and spaces removed. Using the low-level functions defined under it, it encrypts each character in the message and returns the encrypted message to the top-level **encodeMessage** function. The steck and reflect functions are independent functions that perform similar tasks. Given an input character and a Steckerboard (a list of character pairs), the **steck** function simply performs a letter swap if it finds a matching pair on the board. Similarly, the **reflect** function goes through the same procedure but with a **Reflectorboard**. The **advance** function is another independent function that advances each rotor based on their specified knock-on positions.

The **passThrough** and **invertedPassThrough** functions also play a critical role in encryption. They both depend on the **encodeChar** and **invertedEncodeChar** functions to carry out their functionality, respectively. With the aid of the **encodeChar** function, the **passThrough** function encodes a letter three times, passing it through the right, middle and left rotors. The reverse can be observed for the **invertedPassThrough** as encryption goes from the left to the right rotor.

1.2 Implementation and Testing

1.2.1 encodeChar and invertedEncodeChar

Following a bottom-up approach, implementation started with the **encodeChar** **intervertedEncode** functions, as they don't depend on any other functions in the design. I also defined two additional functions (**innerShift** and **outterShift**), which help perform the letter shifting before and after a rotor encrypts them. These two functions are used exclusively in the **encodeChar** and **invertedEncodeChar** functions for encryption.

Table 1 shows some test cases performed on the encode functions using all the rotors Enigma provides. To make it easier to decode by hand, I have kept the test cases simple for each rotor. The crucial thing to note

Rotor	Offset	Char	encodeChar	invertedEncode
I	0	A	E	U
I	1	A	J	V
II	0	A	A	A
III	0	A	B	T
IV	0	A	E	H
V	0	A	V	Q

Table 1: Testing the encode functions

in the table is the row highlighted in grey, which demonstrates the rotor's shift mechanism in relation to the specified offset.

1.2.2 steck, reflect and advance

I proceeded to implement the **steck** and **reflect** function, as they are also independent functions and perform similar tasks. I was able to achieve the desired functionality of these functions using a simple recursive definition with guards. The implementation of the independent advance function came shortly after that and I settled on advancing my middle rotor once the right rotor hit its knock-on position and the left rotor once the middle rotor hit its knock-on position. Table 2 below highlights tests for the rotor advancing mechanism for both general and edge cases.

Rotors (LR, MR, RR)	Initial Offsets	Output
(III, II, I)	(0,0,0)	(0,0,1)
(III, II, I)	(0,0,16)	(0,1,17)
(III, II, I)	(0,4,16)	(1,5,17)
(III, II, I)	(25,4,16)	(0,5,17)

Table 2: Testing the advance function

1.2.3 passThrough and invertedPassThrough

After implementing all the independent low-level functions, it became straightforward to implement the dependent ones with simple function calls. The **passThrough** uses a super compact function composition to pass a character from the right rotor to the left rotor. The **encodeChar** function is called three times with the different rotors, which encrypts the letter the

same number of times. A similar approach was implemented for the **invertedPassThrough** function but inversely.

1.2.4 encodeStr and encodeMessage

Finally, I implemented the **encodeStr** function, which brings it all together. Since it has access to all the enigma components, it calls each independent function. Using a where clause, first the letter is swapped (if simple enigma, the same letter is returned), then passed unto the **passThrough** function and to the reflector. The encryption process ends after the letter is returned from the **invertedPassthrough** function. This is done recursively for every letter in the string until the entire string is completely encoded. The encoded string is then returned to the **encodeMessage** function, implemented using a simple pattern matching for the different Enigmas.

Rotors (LR, MR, RR)	Offsets	Message	Output
(III, II, I)	(0,0,25)	ILOVEHASKELL	RILCDKIAEGGC
(III, II, I)	(0,0,25)	ilovehaskell	RILCDKIAEGGC
(III, II, I)	(0,0,25)	iloveh*aske;ll	RILCDKIAEGGC
(III, II, I)	(0,0,25)	RILCDKIAEGGC	ILOVEHASKELL

Table 3: Testing the Enigma

Table 3 highlights some test cases I ran on the finished enigma simulator. First, I needed to ensure the system ignored the unwanted characters in the input message, hence the first three test cases in rows one to three. The last test case demonstrates the system is able to decode an encrypted message given the same configuration. To test longer messages, I encrypted them using other enigma simulators online and decrypted them with my system.

2 The Longest Menu

2.1 Design

Figure 2 depicts the design of the Longest Menu simulation. The top-level function takes a crib and returns the longest menu using the **genMenus** function, which generates a list of all the menus in the crib. The **findPaths** function plays a vital role in generating the longest menus. Given a pair in the crib, it generates a list of all the chain of pairs that can be reached from that position in the crib. These are passed to the **genMenus** function, which selects the longest chain and extracts its menu.

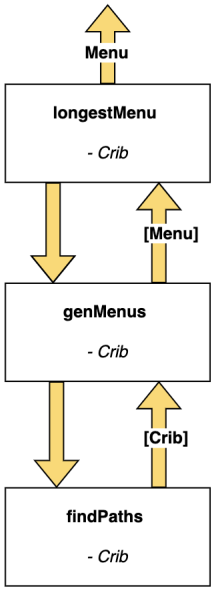

Figure 2: The LongestMenu Design

2.2 Implementation and Testing

2.2.1 findPaths

I started with the implementation of the design with **findPaths** function following an approach similar to that of depth-first search. As highlighted in the design, this function generates a list of the chains in a crib. The crib it takes has been modified, and index positions are attached to each character pair in the crib. This is to make it easy to extract the menu once the longest path has been found.

To simplify the test cases in this report, I have designed a simple crib, and I will be using it for the various functions in the design.

pos	0	1	2	3	4
plain	M	N	G	G	B
cipher	K	G	F	B	L

Table 4: An Example Crib

Table 5 shows the tests for the **findPaths** function. The critical test

Start Pos	Output List
(0, M, K)	[(0, M, K)]
(3, G, B)	[(3, G, B),(4, B, L)]
(1, N, G)	[(1, N, G), (2, G, F)], [(1, N, G), (3, G, B),(4, B, L)]

Table 5: Testing the findPaths function

is in the last row, as it shows the ability of the search to backtrack and continue exploring when it encounters a duplicate link in the crib. In the end, it returns all the paths it's explored.

2.2.2 genMenus

The **genMenus** functions recursively loops through all the positions in the modified crib (i.e a crib with index position attached). For each pair it:

- Calls the **findPaths** function to generate all chains.
- Picks the longest chain in the list and extracts it index positions to make a menu.

2.2.3 longestMenu

Implementing the top-level **longestMenu** function came very easy as all the low-level functions has been implemented and properly tested. **[1 3 4]** is the longest menu generated for the example crib in Table 4. On the event of multiple longest menus, it returns the first one in the list.

3 Break Enigma

3.1 Design

The responsibility of the **Bombe** machine is to find a result which consists of a **Steckerboard** and initial Offsets. The design started from the top-level **breakEnigma** function. This function takes a crib, and generates its longest menu. The Crib, menu and an initially assumed **Steckerboard** are then passed to the **tryAllOffsets** function to begin the search. **tryAllOffsets** recursively tries to find a set of offsets and **Steckerboard** using all possible combinations of offsets starting from (0,0,0) to (25,25,25). If it finds a result, it returns the **Steckerboard** and the pair of offsets back to the **breakEnigma** function or Nothing otherwise.

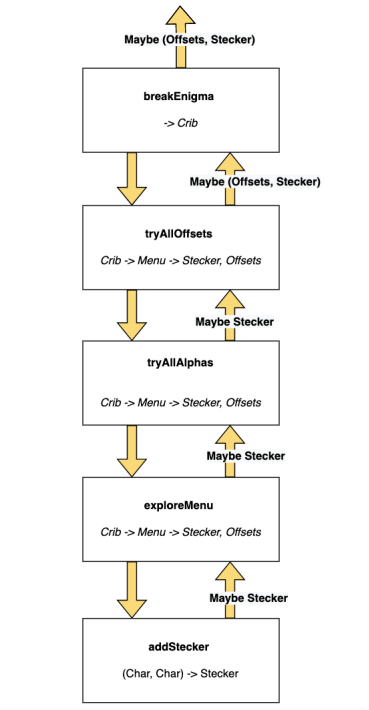

Figure 3: The Bombe Design

The **tryAllAlphas** function searches for a compatible **Steckerboard** for the crib using the set of offsets provided by its top-level function. If a contradiction is found during the search, it makes the next assumption until all characters in the alphabet are exhausted. In that case, it returns **Nothing** and waits for another pair of offsets.

The exploreMenu recursively tries to build up the **Steckerboard** following the longest menu from start to end. It does this using the low-level **addStecker** function, whose main responsibility is to add a new pair of characters to the Steckboard. If a contradiction is found, it returns Nothing.

3.2 Implementation and Testing

3.2.1 addStecker

Similar to the first two problems, i utilized a bottom-up approach for the implementation of the Bombe. I started with the independent **addStecker**

function, and made my way up to the top-level **breakEnigma** function. Table 6 shows some of the tests I performed on this function. The first two rows are edge cases that captures the functions response to duplicate pairs.

Pair	Initial Stecker	Output Stecker
(A, B	[(A, B)]	[(A, B)]
(B, A)	[(A, B)]	[(A, B)]
(C,D)	[(A, B)]	[(A, B) (C, D)]
A, D)	[(A, B)]	Nothing

Table 6: Testing the **addStecker** function

4 Critical Reflection

This module has significantly influenced the way I approach programming problems. For example, the top-design strategy helped me decompose the bigger problems into smaller parts that can be represented by single functions. This helped me better understand the problem and control the order of implementation. Using a bottom-up implementation strategy also helped me a lot in testing. I was able to identify and fix errors early in the program by writing various tests. All low-level functions were tested extensively before moving to higher ones. If I encounter a problem with a higher-level function, I am confident the issue is not originating from the low-level functions.

Finally, this module has also improved my understanding of recursion. This is an aspect of programming I particularly struggled with before learning functional programming.