COM2108 Functional Programming Project Report

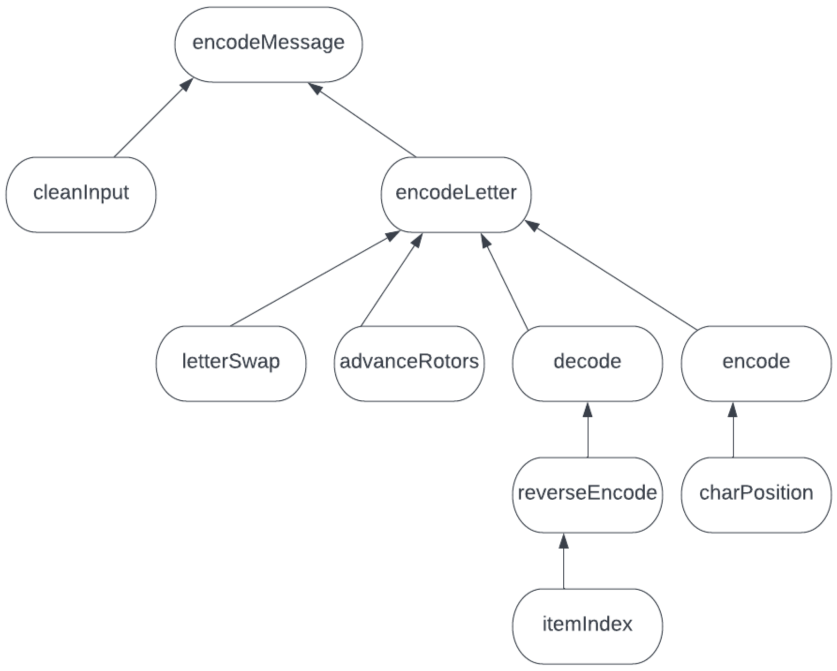
The aim of this project was to design, implement and test a Haskell program that represents the work of the Enigma and Bombe machines.

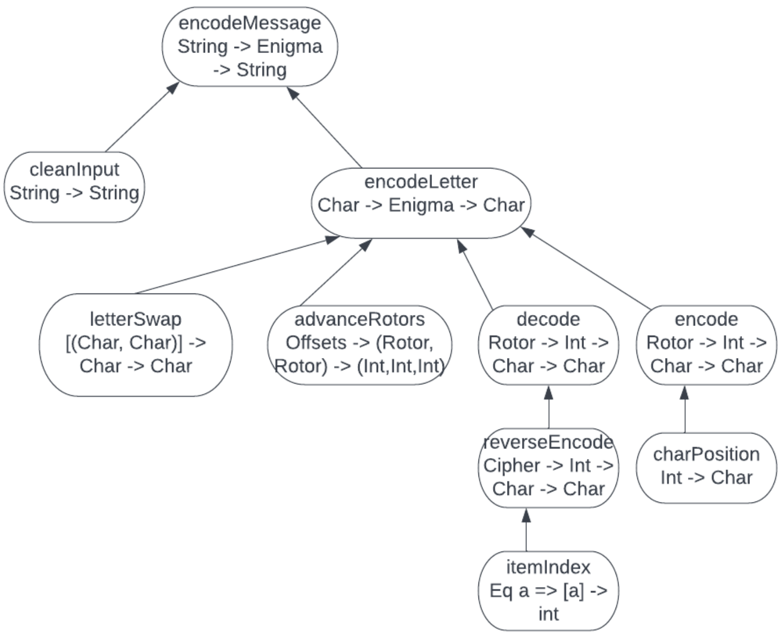
Simulating the Enigma (encodeMessage)

1. Design

I have taken the top-down approach to design my solution for encoding a message using Enigma. To do that I designed a cleanInput function to get the String as upper case letters, then I designed the encodeLetter function that uses all the lower-level functions to encode a single letter. The implementation order is showed in the table in the testing section.

To encode a letter I had to create a function representing the work of a fixed reflector (letterSwap), a function to advance the rotors, including their knock-on positions, (advanceRotors) and functions encode, decode and reverseEncode to find the right letter after all the calculations. The function charPosition finds the letter at n position and the function itemIndex finds the index of an item in a list.




encodeMessage
String -> Enigma -> String
cleanInput
String -> String
encodeLetter
Char -> Enigma -> Char
letterSwap
[(Char, Char)] -> Char -> Char
advanceRotors
Offsets -> (Rotor, Rotor) -> (Int,Int,Int)
decode
Rotor -> Int -> Char -> Char
encode
Rotor -> Int -> Char -> Char
reverseEncode
Cipher -> Int -> Char -> Char
charPosition
Int -> Char
itemIndex
Eq a => [a] -> int

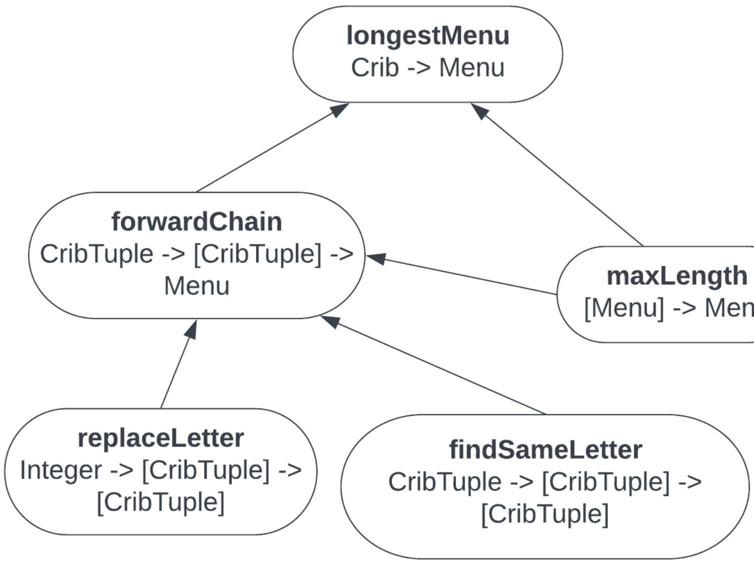
2. Testing

Functions (in order of implementation)	Test Inputs	Test Outputs
itemIndex	itemIndex 'x' ['a','b','p','x','w'] the edge case of an empty list is handled by encodeMessage	3
reverseEncode	reverseEncode "abcdefg" 1 'b' reverseEncode "a" 26 'a'	‘A’ ‘A’
decode	decode rotor5 13 'f'	‘U’
charPosition	charPosition 0	‘A’
encode	encode rotor4 10 'c'	‘J’
advanceRotors	advanceRotors (0,10,10) (rotor1,rotor2) advanceRotors (0,0,25) (rotor1,rotor2)	(0,10,11) (0,0,0)
letterSwap	letterSwap [('a','b'),('b','c'),('c','d')] 'a'	‘b’
encodeLetter	encodeLetter 'a' enigma1	‘p’
cleanInput	cleanInput "2 !aAbB?"	"AABB"
encodeMessage	encodeMessage "ZFCCQBWHYKABMOWUDEODZI" enigma1	"HEREISATESTINPUTSTRING"

Finding the longest menu (longestMenu)

1. Design

To find the longest menu I had to design functions to find connected tuples (findSameLetter), change the ones I have visited to 0 to not repeat them (replaceLetter), recursively find all of the menus and return the longest one.


longestMenu
Crib -> Menu
forwardChain
CribTuple -> [CribTuple] -> Menu
maxLength
[Menu] -> Menu
replaceLetter
Integer -> [CribTuple] -> [CribTuple]
findSameLetter
CribTuple -> [CribTuple] -> [CribTuple]

2. Testing

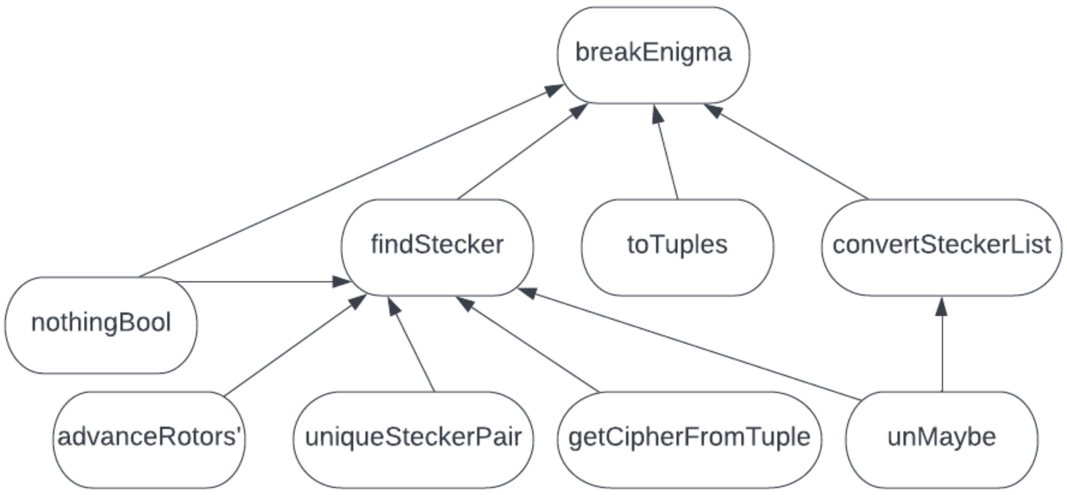
Functions (in order of implementation)	Test Inputs	Test Outputs
replaceLetter	replaceLetter 1 [('a','b',0),('b','c',1),('c','d',2)]	[('a','b',0),('0','0',1),('c','d',2)]
findSameLetter	findSameLetter ('a','b',0) [('b','c',1),('n','m',2),('b','t',3)]	[('b','c',1),('b','t',3)]
maxLength	maxLength [[1,2], [1,2,3],[1,2,3,4,5,6,7]]	[1,2,3,4,5,6,7]
forwardChain	forwardChain ('a','b',0) [('b','c',1),('c','d',2),('d','e',3)]	[0,1,2,3]
longestMenu	longestMenu [('a','b'),('d','e'),('b','c'),('c','d'),('a','c'),('c','b')]	[4,5,2,3,1]

Simulating the Bombe (breakEnigma)

1. Design

To break the Enigma we need to start from the first index from the longest menu, assume the initial steckerboard and offsets, and follow the menu to find if there are any contradictions. If there are, we change the initial steckerboard until we ran out of letters, then we change the offsets and try all the letters again. If we find the steckerboard without any contradictions that gives us the answer.

In order to do that, I designed the functions showed below. I also created a new type SteckerPair that is a tuple of chars to link the output and cipher. unMaybe, getCipherFromTuple, convertSteckerList, toTuples and nothingBool are help functions that simply change the input type when needed. The function uniqueSteckerPair takes a SteckerPair and a Stecker and adds the pair to the stecker if both plain and cipher letters has not been added yet. If there is a contradiction, it returns Nothing. The function advanceRotors' works the same as I the first part of the assignment but does that recursively to go through all the possible offsets. findStecker and breakEnigma are two highest functions. findStecker finds and returns Maybe Stecker



2. Testing

Functions (in order of implementation)	Test Inputs	Test Outputs
unMaybe	unMaybe (Just 3)	3
getCipherFromTuple	getCipherFromTuple [('a','b',0)] 0 getCipherFromTuple [('a','b',0), ('b','c',1),('a','c',2)] 2	‘b’ ‘c’
uniqueSteckerPair	uniqueSteckerPair ('a','b') [('a','b'),('c','d')] uniqueSteckerPair ('a','b') [('b','c'),('d','e')] uniqueSteckerPair ('a','b') []	Just [('a','b'),('c','d')] Nothing Just [('a','b')]
advanceRotors’	advanceRotors' (rotor1, rotor2) (0,0,0) 1 advanceRotors' (rotor1, rotor2) (0,0,0) 5 advanceRotors' (rotor1, rotor2) (0,16,4) 1	(0,0,1) (0,1,5) (1,17,5)

convertSteckerList	convertSteckerList [((0,0,0), (Just [('a','b')]))]	Just ((0,0,0),[('a','b')])
toTuples	toTuples [('a','b'),('b','c'),('c','d')]	[('a','b',0),('b','c',1),('c','d',2)]
nothingBool	nothingBool Nothing nothingBool (Just 1)	True False
findStecker		
breakEnigma		

Critical Reflection

Working in the functional programming paradigm has helped me think differently about how to approach problems and write many programmes more efficiently than in other programming languages. In Haskell each function usually calculates just one thing which helps with readability and improves the maintainability of a project. That is a practise I will endeavour to transfer to my usual object-oriented programming as dividing computation into smaller methods could improve the quality of my code.

Programming in Haskell has also allowed me to gain very valuable experience in using recursion. It is common practise in other programming languages and is widely used to efficiently solve a variety of problems and, over the course of this project, I have learnt how to recognise problems that are suited to recursive computation so that I can make the best use of it.

As mentioned in one of the lectures, Haskell can be a great asset for use in coding interviews because of its high efficiency and relatively small developer-base. This may give me an advantage in the future, so I am to have had an introduction to it in this module.