Ostfalia
Hochschule für angewandte Wissenschaften

Informatik

Lennart Schrader, 70466182

Till Hajek, 70459970

Weitere Programmiersprache

Dokumentation Implementierung einer Enigma-Rotorschlüsselmaschine

Ostfalia Hochschule für angewandte Wissenschaften
– Hochschule Wolfenbüttel

Erste/-r Prüfer/-in: Prof. Dr. Hans Grönniger

Eingereicht am 06.06.2021

Inhalt

1 Einleitung

Die Enigma-Rotorschlüsselmaschine wurde bereits 1918 vom deutschen Elektroingenieur Arthur Scherbius erfunden[1]. Ihre große Bekanntheit erlangte die Enigma jedoch durch die Verwendung zur Verschlüsselung des Nachrichtenverkehrs und Funksprüchen des deutschen Militärs, der Wehrmacht, in der Zeit des Nationalsozialismus. Neben der Wehrmacht nutzten auch die Polizei sowie Geheimdienste und andere Institutionen des dritten Reiches verschiedene Modelle der Enigma.
Bis zum Ende des zweiten Weltkrieges wurden schätzungsweise ~40.000 Enigma-Maschinen ausgeliefert[2]. Obwohl Enigma-Nachrichten erstmals im Januar 1940 durch das britische Kryptologenteam in Bletchley Park, um Alan Turing herum entschlüsselt wurden[3], kamen Variationen der Enigma dennoch im Koreakrieg 1965 zum Einsatz[4].
Die Verschlüsselung durch die Enigma erfolgt Zeichen für Zeichen, mögliche Eingaben sind lediglich die 26 Buchstaben des lateinischen Alphabets. Die Eingabe eines Buchstabens auf der Tastatur löst ein elektrisches Signal aus, welches an die erste bewegliche Walze weitergeleitet wird. Diese Walze hat für jeden Buchstaben einen entsprechenden Kontakt auf der eingehenden Seite, weitergeleitet wird jedoch ein anderer Buchstabe. In einer Enigma sind typischer Weise drei solcher Walzen verbaut. Die erste Walze dreht sich um eine Position weiter. Wurde die erste Walze vollständig rotiert, so wird die nächste Walze um eine Position gedreht, das Gleiche gilt für die zweite und dritte Walze. Somit haben zwei identische Eingaben eine sich unterscheidende Verschlüsselung.
Nachdem eine Eingabe dreimal substituiert wurde, wird Diese einmalig „statisch“ verschlüsselt und, in umgedrehter Reihenfolge, durch die drei Walzen gesendet.
Die endgültige Verschlüsselung wird dem Benutzer in Form von aufleuchtenden Lampen mit entsprechenden Buchstaben angezeigt. Eine zusätzliche Maßnahme zur Erhöhung der Sicherheit ist die Verwendung eines sogenannten „Steckbrettes“, durch welches die Ein- und Ausgaben initial bzw. final verschlüsselt werden. Hierbei handelt es sich um ein simples Umleiten der Signale durch das Stecken eines Kabels.
Der Schlüssel für die Verschlüsselung und Entschlüsselung der Nachrichten ist beim Verwenden einer Enigma gleich, es handelt sich demnach um eine symmetrische Verschlüsselung. Dieser ergibt sich aus der Verwendung, Reihenfolge und Stellung der Walzen, der statischen Verschlüsselung nach und vor der Walzensubstitution und der genutzten Steckverbindungen.

1.1 Motivation

Im Rahmen der Lehrveranstaltung “Weitere Programmiersprache SoSe 2021“ haben wir, Lennart Schrader und Till Hajek, uns dafür entschieden, eine Simulation einer in Kaptiel 1 beschriebenen Enigmaverschlüsselungsmaschine in der Programmiersprache Haskell zu implementieren.
Die Aufgabenstellung durfte selbst gewählt werden, jedoch sollte auf die Implementierung einer GUI verzichtet werden. Wir beide fanden das Thema Verschlüsselung sehr interessant und sind dadurch auf die Idee gekommen eine Verschlüsselungsmaschine nachzubauen. Da Haskell eine funktionale Programmiersprache ist, waren wir davon überzeugt, dass Haskell sich für eine solche Implementierung anbieten würde.

1.2 Kurzbeschreibung

Unser Programm simuliert die Verschlüsselung einer Nachricht durch eine Enigma. Wie genau diese Verschlüsselung erfolgt ist in Kapitel 1 beschrieben und kann für besseres Verständnis auch der Abbildung 1, aus dem Anhang entnommen werden. Eine grafische Nutzeroberfläche ist nicht implementiert, die Ausführung des Programms erfolgt über die Kommandozeile.

[1] https://www.mpoweruk.com/enigma.htm [abgerufen 30.05.2021, 23:20]
[2] https://cryptocellar.org/enigma/e-history/konski&krueger-production.pdf [abgerufen 30.05.2021, 23:20]
[3] https://www.wissen.de/wie-der-code-der-legendaeren-enigma-maschine-geknackt-wurde [abgerufen 30.05.2021, 23:20]
[4] https://web.archive.org/web/20160825110549/http://scienceblogs.de/klausis-krypto-kolumne/files/2016/08/Koreafunk-1964_65-Chiffrierdienst.pdf [abgerufen 30.05.2021, 23:20]

2 Erläuterungen zur Implementierung

Die Implementierung ist in zwei Modules unterteilt: **components.hs** und **interface.hs.**
In components.hs werden die einzelnen Bestand- bzw. Bauteile der Enigma definiert, Funktionen, welche die Mechanismen einer Enigma-Maschine abbilden sollen, sowie die letztendliche Verschlüsselung sind ebenso enthalten. Das interface.hs Module importiert die benötigten Bauteile und wenige zusammengesetzte Funktionen aus components.hs. Weiter beinhaltet das Interface neben der Main-Funktion auch Definitionen und Funktionen, welche für die Überprüfung der Benutzereingaben nötig sind.

2.1 components.hs

Für alle relevanten Komponenten wurden Typen definiert.
Alphabete sind stets Listen aus 26 Chars – den Buchstaben des lateinischen Alphabetes.
Da die Komponente der Walze etwas komplizierter ist als die Umkehrwalze oder das Plugboard, ist für die Walzen ein eigener Datentyp angelegt worden. Dieser besteht aus zwei Alphabeten und einem Char, dem „umspringbuchstabe“, dieser simuliert die Kerbe einer Walze, welche eine vollständige Rotation signalisiert. Die Typen Plugboard und Umkehrwalze gleichen zwar einer Liste aus Tupeln, allerdings erfüllen sie unterschiedliche Funktionen. Jede Enigmaverschlüsselung benötigt eine Kombination aus drei Walzen, hierfür wurde der Typ Walzenkombi angelegt.

```haskell
alphabet :: [Char]
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

-- Data & Types
data Walze = Walze {rand_alphabet:: String, klar_alphabet :: String, umspringbuchstabe ::Char}
    deriving(Show)

type Walzenkombi = (Walze, Walze, Walze)

type Umkehrwalze = [(Char, Char)]

type Plugboard = [(Char,Char)]

-- Walzen
walze1 :: Walze
walze1 = Walze "VZBRGITYUPSDNHLXAWMJQOFECK" alphabet 'K'
```

Das Drehen der Walzen wird mit der dreheWalze-Funktion realisiert. Beim Drehen wird das „zufällige“ rand_alphabet um eine Position verschoben. Das Eingabe- bzw. klar_alphabet bleibt unverändert. So werden auch gleiche Eingaben unterschiedlich verschlüsselt. Anhand des Umspringbuchstaben kann bestimmt werden, ob die zweite und dritte Walze gedreht werden müssen. Die checkObDrehen-Funktion überprüft hier die Umspringbuchstaben und den Kopf der rand_alphabet-Liste. Es werden beim Aufruf der Funktion dreheAlleWalzen also nur die Walzen bewegt.

```haskell
-- Walzenfunktionen
dreheWalze :: Walze -> Walze
dreheWalze (Walze rand_alphabet klar_alphabet umspringbuchstabe) = Walze rand_alphabet_neu klar_alphabet umspringbuchstabe
        where rand_alphabet_neu = tail rand_alphabet ++ [head rand_alphabet]

checkObDrehen :: Walze -> Bool
checkObDrehen walze = umspringbuchstabe walze == head (rand_alphabet walze)

dreheAlleWalzen :: (Walze, Walze, Walze) -> (Walze, Walze, Walze)
dreheAlleWalzen (w1, w2, w3) = (dreheWalze w1,
                                if checkObDrehen w1 == True
                                   then dreheWalze w2
                                       else w2,
                                if checkObDrehen w2 == True
                                   then dreheWalze w3
                                       else w3)
```

Beim Tauschen der Buchstaben durch die Walzen, wird in einer gezippten Liste (aus rand_alphabet und klar_alphabet) nach dem Tupel gesucht, dessen zweites Element dem eingegebenen Buchstaben

entspricht. Daraufhin wird das erste Element des Tupels zurück gegeben. Jeder Eingabe wird mit der uebersetzenMitKombi-Funktion immer durch alle Walzen nacheinander übersetzt.

```
sucheInWalze :: [(Char,Char)] -> Char -> Char
sucheInWalze [] _ = '!' -- Fehlerindikator
sucheInWalze tupelliste c | snd (head tupelliste) == c = fst (head tupelliste)
                          | otherwise = sucheInWalze (tail tupelliste) c

uebersetzen :: Char -> Walze -> Char
uebersetzen c walze =  sucheInWalze (zip random klar) c
                    where random = rand_alphabet walze
                          klar = klar_alphabet walze

uebersetzenMitKombi :: Char -> Walzenkombi -> Char
uebersetzenMitKombi c (w1, w2, w3) = uebersetzen (uebersetzen (uebersetzen c w1) w2) w3

zurueckuebersetzen :: Char -> Walze -> Char
zurueckuebersetzen c walze = sucheInWalze (zip klar random) c
                           where random = rand_alphabet walze
                                 klar = klar_alphabet walze

zurueckuebersetzenMitKombi :: Char -> Walzenkombi -> Char
zurueckuebersetzenMitKombi c (w1, w2, w3) = zurueckuebersetzen (zurueckuebersetzen (zurueckuebersetzen c w1) w2) w3
```

Nach dem „dynamischen“ Wandeln durch die Walzen, wird eine Eingabe noch einmal „statisch“ durch die Umkehrwalze übersetzt. Die Übersetzung bzw. Umkehrung ist statisch, da die Umkehrwalze nicht drehbar ist. Das Prinzip der umkehren-Funktion gleich dem der sucheInWalze-Funktion.

```
-- Umkehrwalzen
umkehrwalze1 :: Umkehrwalze
umkehrwalze1 =   zip   "EJMZALYXVBWFCRQUONTSPIKHGD" alphabet

umkehrwalze2 :: Umkehrwalze
umkehrwalze2 =   zip  "IMETCGFRAYSQBZXWLHKDVUPOJN" alphabet

umkehrwalze3 :: Umkehrwalze
umkehrwalze3 = zip  "YRUHQSLDPXNGOKMIEBFZCWVJAT" alphabet

-- Umkehrwalzenfunktion
umkehren :: Char -> Umkehrwalze -> Char
umkehren c umkehrwalze | snd(head umkehrwalze) == c = fst (head umkehrwalze)
                       | otherwise = umkehren c (tail umkehrwalze)
```

Nach dem Umkehren wird die Eingabe erneut durch alle drei Walzen hindurch zurückübersetzt. Beim Zurückübersetzen werden die Walzen jedoch nicht gedreht.
Werden bei der Benutzung der Enigma keine Steckverbindungen angegeben, so bleibt das Plugboard eine leere Liste. Durch die Angabe eines Strings können die Verbindungen angegeben werden. Zwei aufeinander folgende Buchstaben geben hier eine Steckverbindung an. Dementsprechend muss die Länge des Eingabe-Strings durch zwei teilbar sein, da so gewährleistet ist, dass jeder angegeben Buchstabe auch einen entsprechenden Buchstaben erhält, mit dem er getauscht wird. So werden alle Eingaben initial nach der Eingabe und final vor der Ausgabe ver- bzw. entschlüsselt.

```
-- Plugboard/Streckbrett
plugboard :: Plugboard
plugboard = []

--Plugboardfunktionen
toList :: Char -> Char -> Plugboard
toList c1 c2 = [(c1, c2)]

changePlugboard :: String -> Plugboard -> Plugboard
changePlugboard [] board = []
changePlugboard (x:y:xs) board = toList x y ++ toList y x ++ changePlugboard xs board

initialesVerschluesseln :: Char -> Plugboard -> Char
initialesVerschluesseln c [] = c
initialesVerschluesseln c board | c == fst(head board) = snd (head board)
                                | otherwise = initialesVerschluesseln c (tail board)
```

Das Verschlüsseln ist eine Abfolge der zuvor abgebildeten und beschriebenen Funktionen auf einen Char mit einem Plugboard in der Reihenfolge:

initialesVerschluesseln→uebersetzenMitKombi→umkehren→zurueckuebersetzenMitKombi→ initialesVerschluesseln

```
verschluessle :: Char -> Walzenkombi -> Umkehrwalze -> Plugboard-> Char
verschluessle c (w1, w2, w3) u p = initialesVerschluesseln (zurueckuebersetzenMitKombi
                                                            (umkehren
                                                                (uebersetzenMitKombi
                                                                    (initialesVerschluesseln c p)(w1, w2, w3)) u) (w3, w2, w1)) p

verschluessleString :: String -> Walzenkombi -> Umkehrwalze -> Plugboard -> String
verschluessleString [] (w1, w2, w3) u p = []
verschluessleString (x:xs) (w1, w2, w3) u p = (verschluessle x (w1, w2, w3) u p) : (verschluessleString xs (dreheAlleWalzen(w1, w2, w3)) u p)
```

2.2 interface.hs

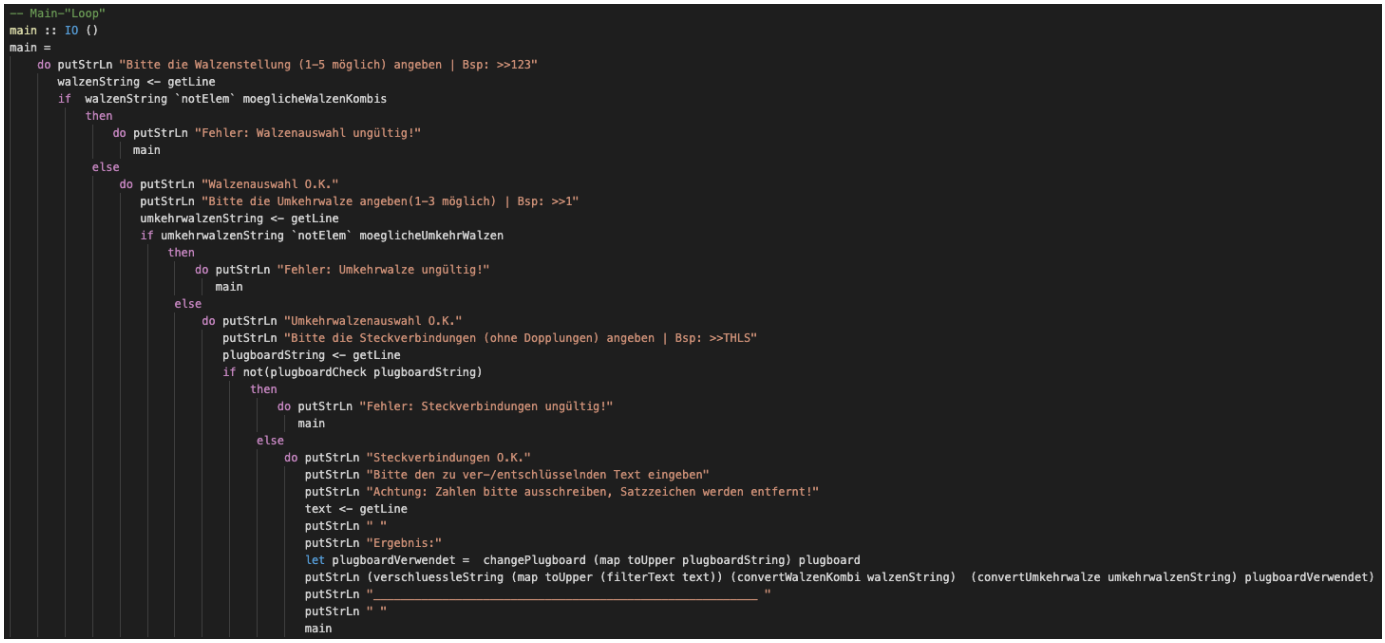
Im Interface-Module werden die zur Verschlüsselung der Nachricht benötigten Komponenten auf Basis der Eingabestrings erzeugt. Das Plugboard ist hierbei eine Ausnahme, ein neues Plugboard wird auf Basis des vorhandenem, leeren und mit der changePlugboard-Funktion definiert.

```
--Funktionen um Eingabestrings in Components zu übersetzen
convertWalze :: Char -> Walze
convertWalze '1' = walze1
convertWalze '2' = walze2
convertWalze '3' = walze3
convertWalze '4' = walze4
convertWalze '5' = walze5
convertWalze  _  = error "Bitte eine andere Walze (1-5) wählen"

convertWalzenKombi :: String -> Walzenkombi
convertWalzenKombi (x:y:xs) = (convertWalze x, convertWalze y, convertWalze (head xs))

convertUmkehrwalze :: String -> Umkehrwalze
convertUmkehrwalze (x:xs) | x == '1' = umkehrwalze1
                          | x == '2' = umkehrwalze2
                          | x == '3' = umkehrwalze3
                          | otherwise = error "Bitte eine andere Umkehrwalze (1-3) wählen"
```

Um Fehler zu vermeiden und die Robustheit des Programms zu erhöhen, werden nur gültige Eingaben zugelassen. Die Überprüfung erfolgt durch Listen, welche gültige Eingaben definieren. Im Falle des Plugboards gibt es eine Prüfmethode (Prüfen auf Länge, falsche Zeichen, doppelte Eingaben). Entspricht die Nachricht nicht den Vorgaben der Enigma (nur Buchstaben, kein Leerzeichen, keine Sonderzeichen), so werden diese aus der Nachricht herausgefiltert.

```
--Alle gültigen Eingabemöglichkeiten für Umkehrwalzen
moeglicheUmkehrWalzen:: [String]
moeglicheUmkehrWalzen = ["1","2","3"]

--Überprüfen der Eingabe des Plugboards
plugboardCheck :: String -> Bool
plugboardCheck [] = True
plugboardCheck string = even (length string) && nurBuchstaben string && keineDuplikate (sort string)

--Hilfsfunktion für plugboardCheck
keineDuplikate :: String -> Bool -- False wenn es Duplikate gibt
keineDuplikate [] = True
keineDuplikate (x:y:xs) | x == y = False
                        | x /= y && length xs == 0 = True
                        | otherwise = True && keineDuplikate (y:xs)

--Hilfsfunktion für plugboardCheck

nurBuchstaben:: String -> Bool
nurBuchstaben [] = True
nurBuchstaben (x:xs) | x `notElem` ['A'..'Z'] ++ ['a'..'z']= False
                     | otherwise = True && nurBuchstaben xs

--Textfilter welcher um ungewollte Zeichen zu entfernen
filterText :: String -> String
filterText string = [x | x <- string, x `notElem` ",.?!-:;1234567890ß</>_ +*#ÜÄÖüäö@$%&()=\""]
```

In der Main-Funktion des Interface-Modules werden die einzugebenden Parameter abgefragt. Erfolgt eine ungültige Eingabe des Benutzers, so erfolgt ein rekursiver Aufruf der Main-Funktion. Der Nutzer muss die Verschlüsselung erneut beginnen. Sind alle eingaben gültig, so wird dem Benutzer die verschlüsselte Nachricht ausgegeben. Die Main-Funktion wird erneut aufgerufen, somit läuft das Programm bis zur Beendigung in einer Endlosschleife.



```haskell
-- Main-"Loop"
main :: IO ()
main =
    do putStrLn "Bitte die Walzenstellung (1-5 möglich) angeben | Bsp: >>123"
       walzenString <- getLine
       if  walzenString `notElem` moeglicheWalzenKombis
           then
               do putStrLn "Fehler: Walzenauswahl ungültig!"
                  main
           else
               do putStrLn "Walzenauswahl O.K."
                  putStrLn "Bitte die Umkehrwalze angeben(1-3 möglich) | Bsp: >>1"
                  umkehrwalzenString <- getLine
                  if umkehrwalzenString `notElem` moeglicheUmkehrWalzen
                      then
                          do putStrLn "Fehler: Umkehrwalze ungültig!"
                             main
                      else
                          do putStrLn "Umkehrwalzenauswahl O.K."
                             putStrLn "Bitte die Steckverbindungen (ohne Dopplungen) angeben | Bsp: >>THLS"
                             plugboardString <- getLine
                             if not(plugboardCheck plugboardString)
                                 then
                                     do putStrLn "Fehler: Steckverbindungen ungültig!"
                                        main
                                 else
                                     do putStrLn "Steckverbindungen O.K."
                                        putStrLn "Bitte den zu ver-/entschlüsselnden Text eingeben"
                                        putStrLn "Achtung: Zahlen bitte ausschreiben, Satzzeichen werden entfernt!"
                                        text <- getLine
                                        putStrLn " "
                                        putStrLn "Ergebnis:"
                                        let plugboardVerwendet =  changePlugboard (map toUpper plugboardString) plugboard
                                        putStrLn (verschluesslString (map toUpper (filterText text)) (convertWalzenKombi walzenString)  (convertUmkehrwalze umkehrwalzenString) plugboardVerwendet)
                                        putStrLn "_____ "
                                        putStrLn " "
                                        main
```

3 Überlegung zu einer möglichen Implementierung in Python

Python bietet viele Möglichkeiten und Features, die eine Implementierung weiter erleichtern würden. Dies ist insbesondere der Fall, wenn man zuvor nur mit imperativen Programmiersprachen gearbeitet hat. Allerdings gibt es aber auch Teile der Umsetzung in Haskell, welche ohne großartige Veränderungen, in eine Python-Implementierung übernommen werden könnten.

3.1 Datentypen

Da die Datentypen List, Tupel und String auch in Python verfügbar sind, ist es durchaus möglich die Bauteile der Enigma ähnlich, wenn nicht gleich, in Python zu implementieren. Entscheidet man sich für eine Umsetzung, bspw. der Walzen als eine Liste von Tupeln, hätte man jedoch in Python den Vorteil, dass diese Veränderungen erlauben. Anstatt in Listen von Tupeln zu suchen, könnte man sich jedoch auch den in Python enthaltenen Datentyp **dict** (eine Schlüssel-Wert-Zuordnung) zu Nutze machen. Auch können in Python von Datentypen Variablen erstellt und Werte zugwiesen werden. So ist es möglich bestimmte Ergebnisse zwischenzuspeichern, wodurch auch die Arbeit mit Indizes vereinfacht wird.

3.2 Objektorientierung

Python ermöglicht den Fokus auf eine objektorientierte Umsetzung der Implementierung einer Enigma. Hier könnte man etwa so vorgehen, dass die einzelnen Komponenten in Klassen gekapselt werden. Die Klassen wiederum hätten dann Attribute, wie beispielsweise die Alphabete bei den Walzen. Ebenso würden sie die entsprechenden eigenen Klassenmethoden enthalten. In einer Mainmethode können die Klassen dann als Instanz verwendet werden.

3.3 Kontrollstrukturen

Während die in Haskell vorhandenen IF-ELSE-Verzweigungen nahezu identisch übernommen werden könnten, bietet Python in Form von While- und Zählerschleifen weitere Kontrollstrukturen, welche sich für den Umgang mit möglichen iterierbaren Datenstrukturen, wie Alphabeten, anbieten würden. Als Alternative zum rekursiven Funktionsaufruf, welcher in vielen Haskell-Funktionen verwendet wird, haben Schleifen den Vorteil, das während eines Durchlaufs verschiedene Funktionen ausgeführt werden können.

3.4 Typisierung

Ein großer Unterschied zwischen Python und Haskell ist die Typisierung. Während Haskell auf eine statische Typisierung setzt, nutzt Python eine dynamische Typisierung. Dieser Umstand sorgt zwar dafür, dass wenn man die Typen-Interferenzen in Haskell einmal behoben hat, meist keine anderen Probleme am Code auftauchen, allerdings geht auch viel Zeit beim Beheben dieser Fehler verloren. In Python hingegen kann sich mehr mit der eigentlichen Funktionalität beschäftigt werden und Fehler in dieser fallen schneller auf. Das erleichtert grade die Umsetzung eines Projektes bei einer vielseitigen Thematik wie der Enigma.

4 Fazit

Mit Haskell als Programmiersprache konnten wir erfolgreich eine Enigma-Rotorschlüsselmaschine implementieren. Mit ca. 150-200 Zeilen (Formatierung, zusätzliche Walzen und Lesbarkeit, Kommentare) Code, verteilt auf zwei Module ist die Implementierung unserer Einschätzung nach sehr kompakt. Die Besonderheiten und Eigenschaften von Haskell, wie beispielsweise Pattern Matching und Lazy Evaluation erwiesen sich als vorteilhaft beim Schreiben von Funktionen und ermöglichen eine leichtere und verständlichere Arbeit.

4.1 Anmerkungen & Schwierigkeiten Lennart

Bei diesem Projekt gab es mehrerer Hürden bzw. Schwierigkeiten: Es war enorm wichtig die Funktionsweise der Enigma vollständig zu verstehen. Damit Funktionen so Funktionieren, musste man immer darauf achten, dass die angegebenen Typen der Funktion stets eingehalten wurden(!), sobald dies Begriffen worden war fiel es deutlich einfacher Funktionen zu schreiben. Da die meisten Fehlermeldungen sich auch Type-Errors beziehen war das „Debuggen“ des Codes eine echte Herausforderung. I/O mit Haskell funktioniert, ist in anderen Programmiersprachen allerdings deutlich einfacher umzusetzen.

4.2 Anmerkungen & Schwierigkeiten Till

Ebenso wie Lennart bin ich der Meinung, dass der erste und entscheidende Schritt das Verstehen der Funktionsweise einer Enigma war. Hier ist uns auch immer wieder, bereits während der Umsetzung in Code, immer wieder aufgefallen, dass wir etwas an manchen Stellen missverstanden hatten und nach erneuter Klärung des Verständnisses den Code entsprechend anpassen mussten. Wenn die Funktionen allerdings erst einmal klar waren, wurde auch relativ schnell klar wie man diese in Haskell umsetzen wollte. Hier gibt es im Zweifel dann auch gar nicht so viele Möglichkeiten wie man denkt, da bereits Funktionen wie etwa Pattern Matching vorhanden sind, und uns so ein relativ klarer Weg zur Lösung vorgegeben wurde.

5 Anhang

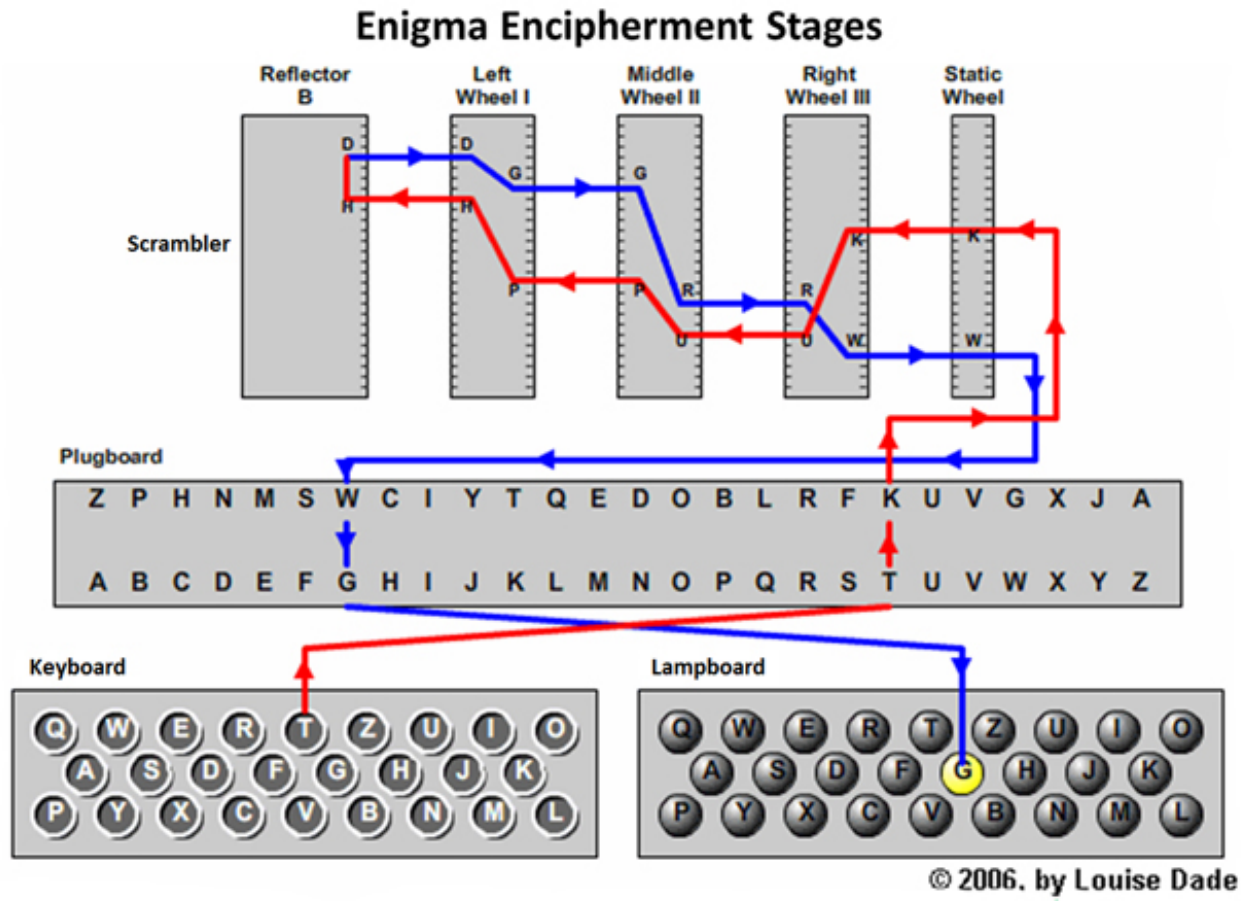


Abbildung 1https://www.mpoweruk.com/enigma.htm Enigma Encipherment Stages [abgerufen 30.05.2021, 23:20]