# COM2108: Functional Programming – Autumn 2019
# Breaking the Enigma Code

Functional Programming Design Case Study

In this case study we'll simulate the breaking of the Enigma code. This is the basis of your third assignment. The first step is to understand how it was done.

## 1 The method

### 1.1 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.

Their task was to find (every day) 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...

### 1.2 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. 1, 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 263 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 1: A Bombe

### 1.3 Cribs and Menus

As explained in Assignment 2, codebreaking was dependent on having a Crib and a Menu to guide the search through the crib.

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

### 1.4 Codebreaking

Suppose we assume

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

We are going to find out if, for these choices, we can find a stecker which is compatible with the crib by using its menu.

- If we can't, we change the choice of initial offsets.
- If we run out of choices for initial offsets we change the Enigma configuration (we won't code this step).

**The crib has implications for the stecker**, as illustrated below (same example as in the specification for assignment 2):

| 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 | J | E | R | R | | T | | | | | | T | | | | | | Q | | | | | |
| out | T | J | Q | S | | W | | | | | | | | | | | | B | | | | | |
| cipher | R | W | I | V | T | Y | R | E | S | X | B | F | O | G | K | U | H | Q | B | A | I | S | E |

Suppose the menu is `[1,0,5,21,12,7,4,3,6,8,18,16,9]`

- We 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 [('E','E')]
- Suppose the enigma encodes E to J at the menu start position, 1, with the given offsets. Then ('J','W') must be added to the Steckerboard.
- Following the menu, this means that at position 0 the input to the Enigma is J and, if its output is T, we add ('T','R') to the stecker.
- At the next menu position, 5, suppose the Enigma encodes T to W. We try to add ('W','Y') to the Steckerboard but we can't do that because we already have ('J','W').
- So the initial steckerboard `[('E','E')]` doesn't work out and we try the next, say `[('E','F')]`.
- If we reach the end of the menu without finding a contradiction we have a potential solution (i.e. the initial offsets and the steckerboard which is compatible with the menu)
- If all 26 initial Steckerboard pairs have been tried we change the initial offsets and try again.

# 2 Designing the Haskell simulation

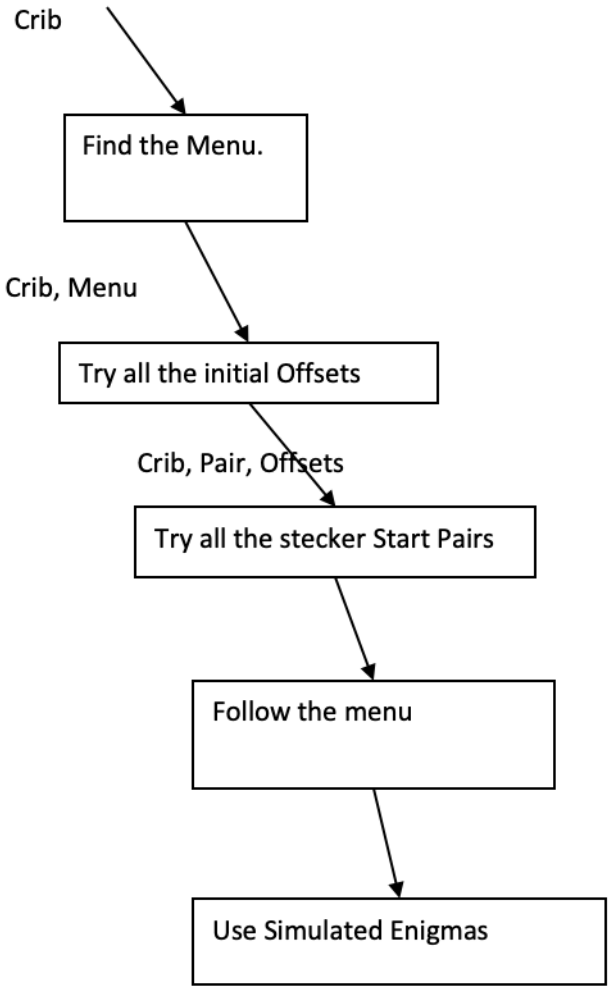
The complete programme will be Bombe.hs and it will import Enigma.hs (and maybe other stuff)

We design top-down. As we go we specify the functions that are needed and the datatypes they will operate on. In Haskell, this comes down to writing the type definitions and leaving comments to explain what the functions will compute. We leave the coding until later.

To simplify things we'll assume that the choice of rotors and reflector is always the same: referring to the assignment 2 spec 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 might like to try removing these restrictions.

## 2.1 Overview

First get a rough feel for the whole application:




The key steps being:

1. Given the Crib, find the Menu

2. With the Crib and Menu, try each initial offset setting in turn until one works (return this set of offsets) or we run out.
3. With the Crib, Menu and Offsets, try each Start Pair until one works or we run out.
4. Follow the implications using the menu and crib, building the Steckerboard. If we reach the end of the menu, success, but failure if there is a steckerboard contradiction.
5. Now use the simulated Enigmas that result to find the encoding for a given letter at a given point.

## 2.2 Top-down design

### 2.2.1 breakEnigma: The Top Level

We'll call the top-level function **breakEnigma**. Given a **Crib** it will search for a solution.

A solution will consist of the initial **Offsets** and a Steckerboard which is compatible with the Crib. We may or may not find a solution so the return type should be a **Maybe**:

```
breakEnigma ::  Crib -> Maybe (Offsets, Steckerboard)
```

we have the **Crib** and **Menu** datatypes from assignment 2.

**breakEnigma** will first call **longestMenu** to find the longest menu in the Crib.

**breakEnigma** will try out different initial **Offsets** in turn, starting at (0,0,0), until one succeeds or it reaches (25,25,25), which indicates failure.

We need an auxiliary function which takes the **Crib**, the **Menu** and the current **Offsets**, tries them out, returns the result if a solution is found and otherwise recurses with the next **Offset** choice: this suggests

```
breakEA :: Crib -> Menu -> Offsets -> Maybe (Offsets, Steckerboard)
```

but we can be a bit smarter: call the first position on the Menu the **Start**. This identifies a letter – the one in the plain at the start index – which we are going to use in the first pair in the **Steckerboard** we are trying to find. We are then going to make an assumption about what it is steckered to: this will form an initial **Steckerboard**. In the example in 1.4 above we assumed it would be unsteckered, so the initial Steckerboard would be `[('E', 'E')]`. Every time **breakEA** recurses to try a new choice of offsets it will start with the same initial **Steckerboard**, so we might as well define this in breakEnigma and pass it in:

```
breakEA :: Crib -> Menu -> Steckerboard -> Offsets -> Maybe (Offsets, Steckerboard)
```

### 2.2.2 findStecker: Given the Offsets, search for a Steckerboard

**breakEA** will call a function **findStecker** whose responsibility is to search for a **Steckerboard** which matches the **Crib** given a particular set of **Offsets**. It might succeed or fail:

```
findStecker :: Crib -> Menu -> Steckerboard -> Offsets -> Maybe Steckerboard
```

**findStecker** is given the 1-pair initial **Steckerboard** `[(x,y)]`. It arranges to explore the implications which follow from this, given the **Crib**, **Menu** and **Offsets**. If a contradiction is found, **findStecker** recurses to try the next assumption `[(x,(y+1) mod 26)]` until one works or all 26 possibilities have been tried.

### 2.2.3 followMenu: build the Steckerboard from the Menu

The recursive function to do this, followMenu, will look like so:

```
followMenu :: Crib -> Menu -> Steckerboard -> Offsets -> Maybe Steckerboard
```

If

- the index at the head of the menu is $i$ ,
- the plain Char there is $p$,
- the stecker says $p$ is steckered to $q$
- the Enigma encodes $q$ to $r$ at position $i$ with the current offsets (use your code from assignment 2)
- the ciphered char at $i$ is $c$

the implication is that we should add ($r$, $c$) to the stecker, using **steckerAdd**.

If that addition is compatible with the existing stecker , **followMenu** recurses with the rest of the menu and the new stecker.

If the menu is empty the stecker represents a solution.

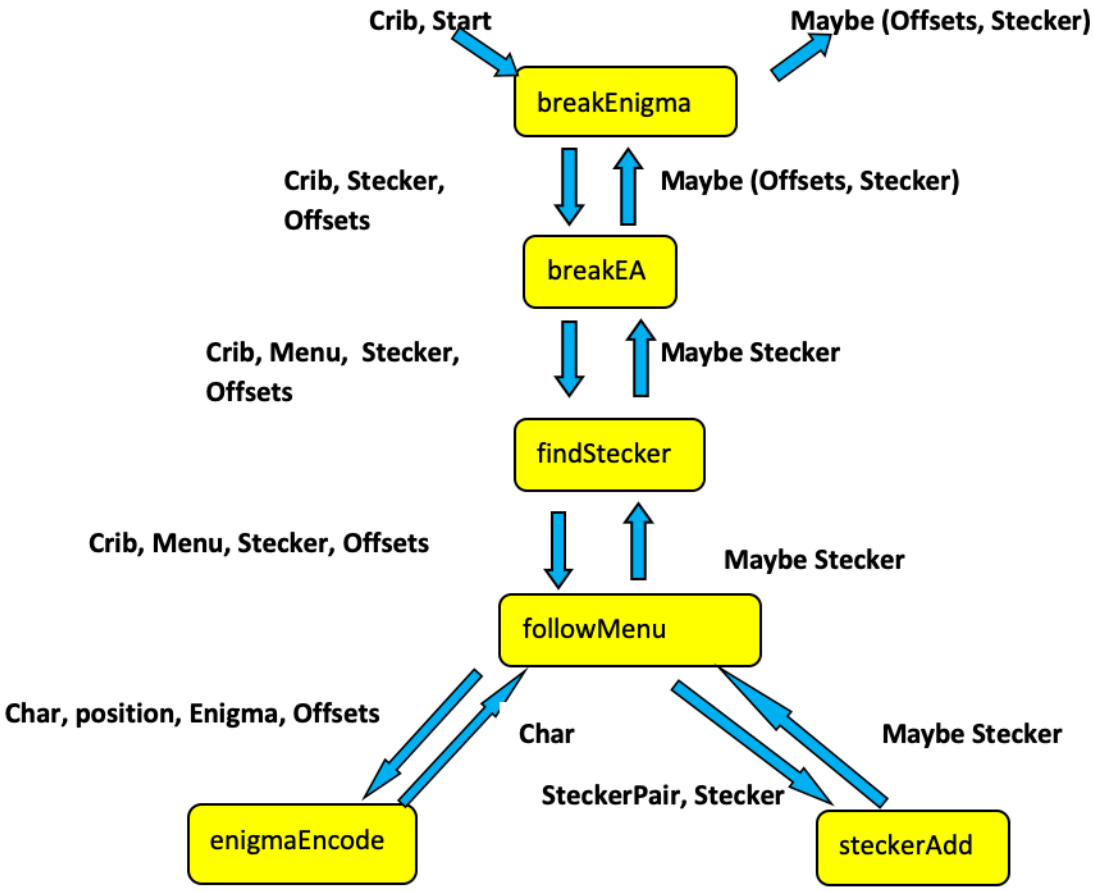
### 2.2.4 Adding to a Steckerboard

```
steckerAdd  :: SteckerPair -> Steckerboard -> Maybe Steckerboard
```

Succeeds, returning a new **Steckerboard**, if the **SteckerPair** is already in the **Steckerboard**, or there are no existing entries for either Char.

### 2.3 Structure Diagram

*What calls what, what arguments are passed in, what is returned.*




## 3 Implementation and Testing

Now we can start coding, from the bottom of the structure upwards, testing as we go.

We have **enigmaEncode** already.

- We start coding with **steckerAdd**,
  - Then we can deal with **followMenu**,
    - Then **findStecker**.
      - And so on up the coding hierarchy.

Note that we are coding a search, one that could go on a long time. We should therefore invent test cases for **breakEnigma** for which the solution will be found quickly, e.g. with the initial rotor positions (0,0,1) not (25,24,24). We can easily construct such test cases with **encodeMessage** for steckered enigmas.