GIST AI

BY ANDREW SIKOWITZ

OVERVIEW FOR A1

- Try not to be intimidated by the writeup length
- Deliverables:
  - [enigma.ml] – the functions you must implement are all documented in this file
  - [enigma_test.ml] – the groups of tests you must write are listed in this file
  - Also: test-driven development, pair programming, and git

- This is all you have to do!
  - The majority of the writeup is **help** on these deliverables, not extra work
  - One step of the writeup at a time, in order

TEST-DRIVEN DEVELOPMENT (TDD)

- Write tests based on a function's specification, *then* implement it
  - Better understand what you are supposed to implement
  - Your implementation is based on the tests, not the other way around

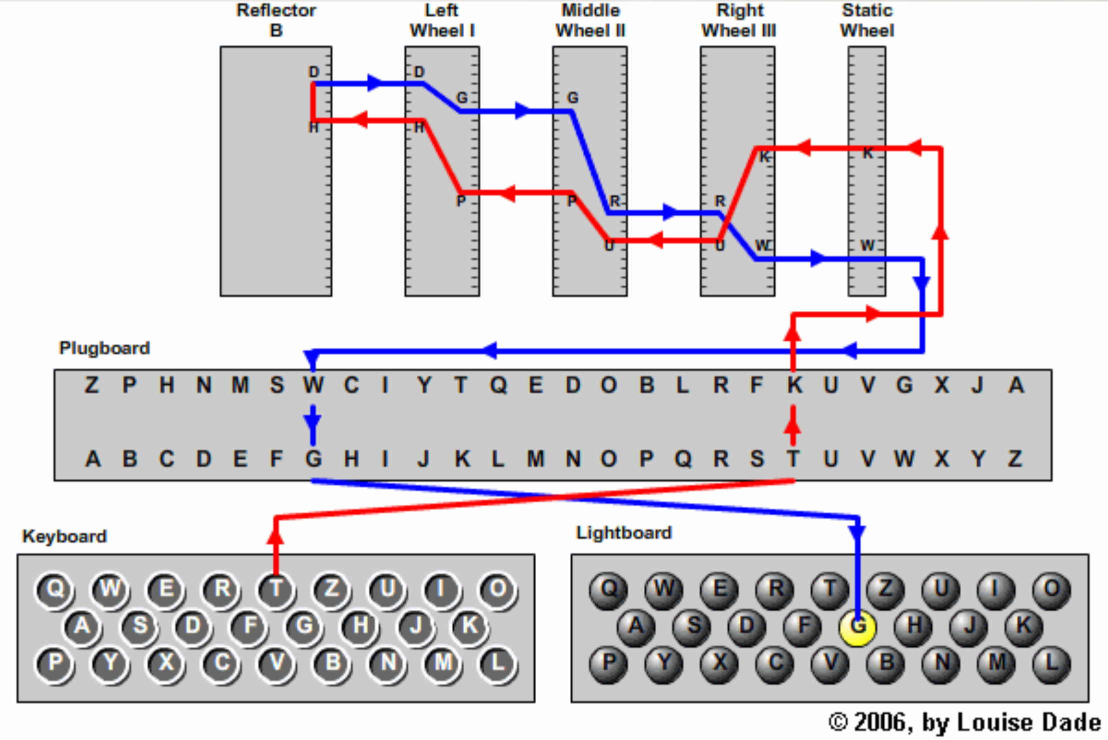
- Try it out with other functions in this assignment, as you have to write tests for them
  - Hint: There's several test cases for different functions throughout the writeup

Before getting started…

- The Makefile is a bit different this time:
  - `make build`: Generate compiled bytecode for [enigma.ml] (in the _build directory)
  - `make test`: Generate compiled bytecode for [enigma_test.ml], then run the tests
- Make sure you create a **private** GitHub repo!
  - Use the Cornell GitHub, as your partner may not have unlimited private repos
- Make sure to run both `git config` commands – they will make your life easier
- This session will not go over pair programming or how to use git

ENIGMA MACHINE OVERVIEW OVERVIEW

- You press a letter; a (potentially) different letter lights up
- Pressing a letter triggers an electric current through wiring
- Wiring depends on Enigma machine state: plugboard, rotors, and reflector
- Wiring also depends on rotor "top letter" (offset), which can change after a letter is pressed



SUBSTITUTION CIPHER

- Several components of the Enigma machine implement a "substitution cipher"
- Essentially a one-to-one mapping between letters
- We encode the mapping as a 26-character string
  - Represent letters based on their alphabetical indices – letter 0 = a, letter 1 = b, and so on
  - For each index i ($0 \le i \le 25$) of the string, letter i maps to the character at index i in the string
- Ex: "BCDEFGHIJKLMNOPQRSTUVWXYZA" maps each letter to the letter after it
  - A -> B,   B -> C,   J -> K,   Z -> A

SUBSTITUTION CIPHER IMPLEMENTED

- **Suppose** we want to write a function that implements a substitution cipher:
  - This is purely hypothetical – you do not need to do this!
  - ~~Maps one-to-one each letter to a different letter~~
  - Maps one-to-one each number between 0 and 25 (inclusive) to a different number in that range
- To mimic the assigned functions, let our function take arguments:
  - [wiring]: The 26-character string denoting the substitution cipher mapping
  - [input_pos]: The integer representation of the input letter
- Output: The integer representation of the output letter
- Ex: [calc_subst_cipher "BCDEFGHIJKLMNOPQRSTUVWXYZA" 5] = 6
  - The letter at position 5 (zero indexed) is 'G', which has index 6 in the alphabet (zero indexed)

[map_r_to_l] and [map_l_to_r]

- Likely the most complicated functions **to understand**
- Implement how current passes through rotors, for each direction
  - Rotors are like the reflector, except they can be rotated
  - When they are rotated, current that would normally enter at a certain position is **offset**, and current that would normally exit at a certain position is **offset in the opposite direction**
- Arguments:
  - [wiring]: The substitution cipher
  - [input_pos]: The integer representation of the input letter
  - [top_letter]: The letter at the "top" of the rotor, specifying the offset
- Output: The integer representation of the output letter

[map_r_to_l] and [map_l_to_r]: top_letter

- If top letter is ‘A’:
  - There is no offset – the rotor behaves just like the reflector
- If top letter is ‘B’:
  - Current that would normally enter at position 0 now enters at position 1
  - Current that would normally enter at position 2 now enters at position 3
  - Current that would normally enter at position 25 now enters at position 0
  - Current that would normally exit at position 25 now exits at position 24
  - Current that would normally exit at position 2 now exits at position 1
  - Current that would normally exit at position 0 now exits at position 25

[map_r_to_l] and [map_l_to_r]: Final Tips

- Rotor overview:
  - Current enters at some position
  - Then, it is offset based on [top_letter]
  - Then, it is rerouted based on the rotor's wiring (as with the reflector)
  - Then, it is offset back, based on [top_letter]
- Make sure to keep your numbers between 0 and 25
- Look at functions in the String module, and remember your [index] function
  - https://caml.inria.fr/pub/docs/manual-ocaml/libref/String.html
- Read the writeup carefully and/or make the Pringles can model ~~(or see it in office hours)~~
- Make sure to test your functions on the provided test cases.

[map_refl], [map_plug], and [cipher_char]

- [map_refl] is a simpler version of [map_r_to_l] – it has no offset
- [map_plug] takes in a list as input… what do you do with lists?
  - Also remember if a letter is not part of the [plugs], you return the same letter
- [cipher_char] is putting all the pieces you've built together

[step]

- Likely the most complicated function **to write**

- Suggestion: Recursively step one rotor at a time
  - Think about the order in which you want to iterate through the rotors
  - Think about what information you need in deciding whether to step a single rotor
  - Special cases for first and last rotors

[cipher]

- Combine [cipher_char] and [step]

- Look at the String module (again)
- [Char.escaped] or [String.make] to convert a character to a string

PIPELINING

- Use [e1 |> e2] to pass [e1] as the last input to the function [e2]
  - [e1 |> e2 |> e3] is equivalent to e3 (e2 e1)
  - Like passing an input through multiple consecutive functions
  - Often looks cleaner and makes more sense conceptually
- Example: Get the second to last element of a list (**insecurely** and inefficiently)
  - Do not use `List.hd` or `List.tl` in your own code!

```
List.hd (List.tl (List.rev lst)))
vs.
lst |> List.rev |> List.tl |> List.hd
```

"Take the list, reverse it, take its tail, then take the head of that"

PIPELINING (FORMATTING)

- For long chains, format as so:

```
e1
|>  e2
|>  e3
|>  e4
|>  e5
```

PIPELINING (ADVANCED)

- You can use a partially applied function as part of the pipeline
- The piped value is passed as the last argument

```
2 |> (-) 5 => 3
```

- Equivalent to `(-) 5 2 = 5 - 2 = 3`
  - You can use infix operators such as + and - as functions by putting parentheses around them
    - For multiplication, do (*), with spaces before and after *, to avoid comment syntax

RECORD SYNTAX

- For record: type person = {name: string; age: int; gpa: float}
- Define a new record: {name = "Andrew"; age = 21; gpa = 0.}
- You can use this like any expression in OCaml
  - let me = {name = "Andrew"; age = 21; gpa = 0.}
  - f {name = "Andrew"; age = 21; gpa = 0.} (* call function [f] with that record as input *)
- **Define a new record based on an existing record (very useful):**
  - {old_record with field1 = value1; field2 = value2; …}
  - Ex: let new_me = {me with gpa = 4.0}

    ```
    val new_me : person = {name = "Andrew"; age = 21; gpa = 4.}
    ```
  - Does not change the old record (it's immutable)!

DEEP PATTERN MATCHING

- You can often match complicated patterns in one go

```
let (p, q, {name; age}) = (3, 4, {name = "Andrew"; age = 21; gpa = 0.})

match ([1;2;3], 5) with
| (h1::h2::t, v) -> h1+h2+v
| _ -> 0
```

PATTERN MATCHING: WHEN

- Limit match cases based on a bool with the when keyword!

```
match [1; 2; 3] with
| h::t when h > 2 -> 0
| h::t when List.length t < 3 -> 1
| h::t -> 2
| [] -> 3
```