Universität Bielefeld
Technische Fakultät

AG Genominformatik
Dr. Lucas Robidou

The Rust programming language
Summer 2024 / 2025

Exercises

1 Modelling

1. A Pokémon is a creature that has a positive amount of maximum HP[1], a positive amount of current HP, and:

   - possibly some non-volatile status
   - possibly some volatile status

   Non-volatile statuses are Burn, Freeze, Paralysis, and Poison (when Poisoned, a Pokémon can be Badly poisoned, or not). Volatile statuses are Infatuation, Confusion, Curse, and Flinch.
   Write a Pokémon struct.

2. A transaction with your debit card might look like this:

   (a) The seller asks your bank to confirm that you have, say, M euros in your account.
   (b) The seller asks for N euros ($N \leq M$) from your bank account.
   (c) The provider of your card will wait a few days before moving the money.
   (d) The money is moved from your bank to the seller's bank.
   (e) The seller's bank credits the money to the seller's account.

   - Name the states above.
   - Name the transitions between these states.
   - Instead of using an enum to model the states, write a struct for each possible state of a transaction (each struct should contain the amount of money being moved).
   - For each such struct, implement a method (with a meaningful name) that returns a struct representing the next state. Note that the transition from state (a) to state (b) might fail.

   What are the advantages of using structs over an enum in this case?

   - Implement a `new` method for the first struct, and an `archive` method for the last one. The archive method returns a new struct, called `Archive`, which simply contains the amount of money moved. Make your `Archive` serializable and deserilazable.
   - Add an `IBAN` struct, that contains a bank account ID, represented as a String.
   - Make every state contain the two IBANs of the transaction. Update your methods.

2 Your first iterator

Iterators are structs that implement a trait, `Iterator`. This trait requires you to implement a method `next`, which returns an `Option`. At each iteration step, the `next` method is called. If the `next` method returns Some, the iteration continues. If the `next` method returns None, the iteration stops.
Let's start simple. Write an `IntegerIterator` struct that contains a single value of type i32. Implement the Iterator trait so that the iteration never finishes and always returns the next integer (0, 1, 2, etc.). Use the template below:

[1] Apparently, it means "Hit Point", not "Health Point". My life is a lie...

```
impl Iterator for IntegerIterator {
    type Item = i32;  // the Iterator trait requires us to provide the output type here

    fn next(&mut self) -> Option<i32> { // the output type has to match Item
        // your logic here: always return Some
    }
}
```

Implement a `new` method. Now, you can do:

```
for i in IntegerIterator::new() {
    // print it
    // (use CTRL+C to stop the program)
}
```

Write another iterator, `Range`, that consists of a start, an end, and a step. Write a `new` method for the struct. Implement `Iterator` so that the user can iterate from start to end with a step. Write a function `range` that returns a `Range`, so that you can do:

```
for i in range(0, 15, 2) {
    // ...
}
```

3 Lifetime

- Fix the compiler errors about the struct.

  ```
  struct Book {
      author: &str,
      title: &str,
  }
  ```

- Fix the compiler error by updating the function signature.

  ```
  fn longest(x: &str, y: &str) -> &str {
      if x.len() > y.len() {
          x
      } else {
          y
      }
  }
  ```