CPL: Rust

Exercise Session

The goal of this exercise session is to make you familiar with Rust and its alternative approach to memory safety: the ownership/borrowing system.

There are three important sources of information about Rust:

- **The Rust Book (2022 edition)**: `https://doc.rust-lang.org/book/foreword.html`
  For when you can't remember what Ownership is or if you don't know how Rust does Error Handling.
- **Rust by Example**: `https://doc.rust-lang.org/stable/rust-by-example/`
  For when you want to know what the syntax of a *struct* is or how to define *methods*.
- **The Rust API documentation**: `http://doc.rust-lang.org/std/`
  For when you want to know what methods are available for a `Vec` or what the interface is of `Iterator`.

Running Rust

You will of course have to write Rust code in this exercise session. There are two ways to run Rust code:

1. The online editor on `https://play.rust-lang.org/`. No need to install Rust or to find an editor supporting Rust. Figure 1 shows what the online editor looks like.
2. Install Rust using **rustup**, see `https://www.rustup.rs/`. There are plugins for most editors, you can find them at `https://areweideyet.com/`.

Unless you have already installed Rust on your laptop, we recommend you use **option 1** today, as it is also the only option when working in the PC labs where the PCs don't have Rust installed. Whichever way you choose to run Rust code; write the "Hello world"-program in figure 1 yourself and make sure it executes correctly.

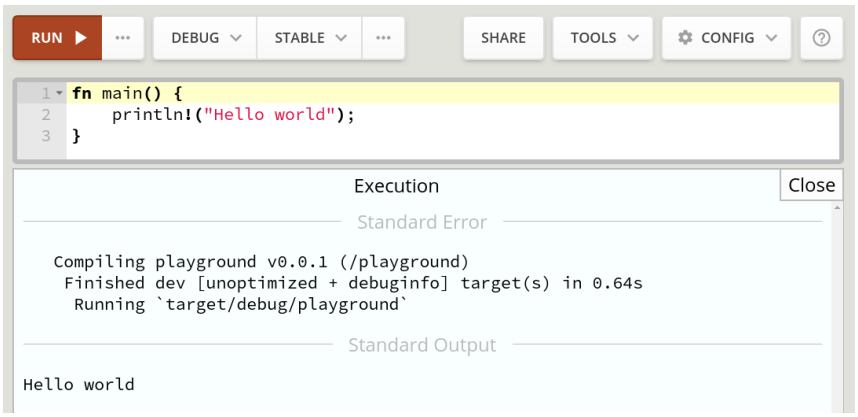

Figure 1: Hello world in the online editor

Exercise 1

The first exercise is just a warm-up to make you familiar with the syntax of Rust. Since you're all graduating soon, you will probably be going to job interviews. In some job interviews you will be asked to solve some programming puzzles to prove that you know how to program. A very famous programming puzzle used in job interviews is **FizzBuzz**. The author of the puzzle claims that a majority of CS graduates can't solve this simple problem. Prove them wrong!

> Write a program that prints the numbers from 1 to 100 (inclusive). But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

The output should look something like the following (you may print each entry on a separate line if you wish).

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 ...
```

Hint: look up `Range` in the Rust API and have a look at the section in the Rust book on loops.

Exercise 2

This exercise focuses on Ownership and Borrowing. Recall the example given in the lecture:

Rust

```rust
fn owner() {
  let mut vec = Vec::new();
  vec.push(1);
  print_length(vec);
  vec.push(2);
}
fn print_length(vec: Vec<i32>) {
  println!("Length: {}", vec.len());
}
```

We create a vector `vec`, add 1 to it, then try to print its length, and finally add 2 to it. This code doesn't compile (see the errors below), because the ownership of `vec` moved from `owner` to `print_length`. The `vec` is also deallocated in `print_length` as it is no longer used. This means that we cannot add 2 to the vector as it no longer exists.

```
L4 note: value moved here
```

```
L5 error: value used here after move
```

We solved this by giving `print_length` an **immutable borrow** of `vec`, written with an `&`. Note that an `&` was added to the signature of `print_length`, but also at the place where `vec` is borrowed to `print_length`. The correct code is given below.

Rust

```rust
// Change the name from `owner' to `main' to compile & run it.
fn owner() {
  let mut vec = Vec::new();
  vec.push(1);
  print_length(&vec);
  vec.push(2);
}
fn print_length(vec: &Vec<i32>) {
  println!("Length: {}", vec.len());
}
```

Now it's your turn. Open `ex2.rs`: this is a file with some Rust code that does all sorts of things with a vector. The code won't compile because the borrowing annotations are missing.

> Add `&` and `&mut` to indicate immutable and mutable borrows at the right places in the argument lists so that the code compiles. You are **not allowed to mod-**

ify the bodies of the functions except for the `main` function. Try to give the minimal borrow necessary, e.g. don't give a `&mut` when a `&` would suffice.

Try to think first about what the code is doing, because adding random `&`'s and `&mut`'s by trial and error can cause some confusing compile errors.

The expected output of the program is:

```
First element: -1
-1 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 10
Aaaand it's gone
```

Hints

- The first compile error you will get is: `type 'i32' cannot be dereferenced`, this is caused by `**i += 1;` and the `print!` of `**i`. The two dereferences are intentional in both instances, so don't try to change this! Dereferencing is overloaded in Rust, this means that you can use `*` to dereference different kinds of indirections. The first dereference is used to get from the immutable or mutable borrow to the thing that was borrowed. The second dereference fetches the contents from the `Box`.
- Read the sections on Ownership and References and Borrowing in the Rust book.

Exercise 3

This exercise focuses on Ownership and Borrowing, Algebraic Data Types and Pattern Matching, and Error Handling. The goal of this exercise is to build an interpreter for a simple language with arithmetic ($+, \times, -$ on `u32` numbers) and variables (represented as strings). Figure 2 shows the syntax of the language.

x	$::=$	$foo, bar, \ldots$	Variables
n	$::=$	$0, 1, \ldots$	Numbers
$\odot$	$::=$	$+, \times, -$	Binary operators
e	$::=$		Expressions
	$\mid$	n	Literal number
	$\mid$	x	Variable
	$\mid$	$x = e$	Assignment
	$\mid$	$e_1 \odot e_2$	Operator application
	$\mid$	$e_1; e_2$	Sequence

Figure 2: Syntax of the language

Some example programs:

$x = 1; x$	$\rightarrow 1$
$x = 3$	$\rightarrow 3$
$foo = 1; bar = 14; foo + bar$	$\rightarrow 15$
$1 * 10 - 2$	$\rightarrow 8$
x	$\rightarrow$ undefined variable: x
$prod = 2 * 3; sum = 4 + prod; sum$	$\rightarrow 10$

To make things easier for you, the result of an assignment is the value assigned to the variable. Note that using a variable to which no value is assigned causes an error.

Now open `ex3.rs`. This is a file with a skeleton for you to fill in. The documentation above each method specifies its behaviour. Replace all occurrences of the `unimplemented!` macro (which just gives an error) with your code and complete the definitions of `Expr` and `Env`. Write a `main` method that tests some of the example programs above.

Don't modify the type signatures of the functions! You are free to add additional functions, methods, data types, ...

Hints

- For help with defining the `Expr` enum, have a look at the section about Enums in the Rust book. Read the sections on Enums and Pattern Matching for more about pattern matching.
- When you get `error[E0072]: recursive type 'Expr' has infinite size`, you will probably have to use `Box`. Try to understand why there was a problem in the first place by clicking on the error code `[E0072]`.
- To represent the `Env`, have a look at the built-in collections in the API documentation. When you have found the right collection (click on one of the red links at the

bottom of the page), you will find some examples on how to use it (don't forget the `use` declaration at the top).

- Strings are a bit special in Rust: there is `&str`, an immutable borrow to a string. Every literal string that occurs in your program is such a `&str` (this makes sense, because you should not be able to mutate literal strings). There is also `String` which is an **owned** string that you can mutate. In this exercise we will use `String`. To convert a `&str` to a `String`, use the `to_owned()` method, e.g., `"test".to_owned()`. To convert a `String` to a `&str`, just use an immutable borrow (`&`).
- There are many methods for `Result` and `Option` that will make it easier to work with them, e.g. `map`, `ok_or`, `map_err`, `and_then`, etc. The `?` syntax is also very useful, see this example. Also have a look at the section about Error Handling in the Rust book.
- You can use the `format!` macro to format strings similar to how you print using `println!`, see this example.

Exercise 4

In this exercise you will implement the Iterator trait. The iterator you will write will not be an ordinary one. It iterates over `Strings` which, when printed below each other, will produce an ASCII depiction of a Christmas tree. The character used to print the top of the tree and the total height of the tree (excluding the top and trunk) are parameters to the iterator. The following code will produce the output below:

Rust

```
for i in ChristmasTree::new('$', 6) {
  println!("{}", i);
}
```

```
     $
     *
    ***
   *****
  *******
 *********
***********
    |||
```

To make it easier to understand which strings are exactly generated by the iterator, here is the example again but with all spaces replaced by dots and double quotes around each item produced by the iterator.

```
".....$"
".....*"
"....***"
"...*****"
"..*******"
".*********"
"***********"
"....|||"
```

The height of this tree is 6, excluding the top and trunk. The trunk is always three characters wide and one high. The number of asterisks, i.e. the maximal width of the tree is $2h - 1$ where h is the height of the tree. It should be easy to figure out how many spaces and asterisks to print given the line number.

Now open `ex4.rs`. This is a file with a skeleton for you to fill in. The documentation above each method specifies its behaviour. Replace all occurrences of the `unimplemented!` macro (which just gives an error) with your code and complete the definition of the `ChristmasTree` struct. You can use the `main` method to try out some examples (play around with the parameters).

Hints

- A `String` can be thought of as a more efficient `Vec` for `chars` (if you're familiar with Java: think of a `StringBuffer`), in other words, you can easily append characters to a `String` using its `push` method.
- You will probably need a field in the struct to remember on which line in the tree the iterator is.

Exercise 5

This exercise focuses on using an Iterator, Pattern Matching and Error Handling. In this exercise you will write an RPN or stack calculator.

The RPN calculator expects an input string representing the calculation to perform and will return a `Result<f32, String>` which is either the solution of the calculation (`f32`) or an error message describing what went wrong (`String`).

Some examples:

```
“1”                 => Ok: 1.0
“3 -2.5 *”          => Ok: -7.5
“1 2 + 10 *”        => Ok: 30.0
“9 7 - 4 + 2 * 3 /” => Ok: 4.0
“1 2 + +”           => Err: only one element on the stack
“1 2 3 +”           => Err: more than one element on the stack
“”                  => Err: no element on the stack
“1 2 $”             => Err: unknown operator: $
```

Now open `ex5.rs`. This is a file with a skeleton for you to fill in. The documentation above each method specifies its behaviour. Replace all occurrences of the `unimplemented!` macro (which just gives an error) with your code and complete the definition of the `rpn_calc` function. You can use the `main` method to try out some examples.

The algorithm you must implement in `rpn_calc`:

- For each input token (tokens are separated by whitespace)
  - If it is a number: push it onto the stack
  - If it is an operator (+, -, * or /):
    * Pop 2 operands from the stack (**Errors** possible)
    * Apply the operator to the two operands
    * Push the result on the stack
  - If it is neither: **Error**: return an `Err` with “unknown operator: X” as message where ‘X’ is the operator
- When there are no more tokens
  - If there is only one value on the stack: return it
  - If there are none: **Error**: return an `Err` with “no element on the stack” as error message
  - If there is more than one: **Error**: return an `Err` with “more than one element on the stack” as error message

Hints

- You are free to add additional functions, methods, data types, ... It may be useful to have a method that pops two elements from the stack, applies a binary operator to them and pushes the result on the stack.

- Strings are a bit special in Rust: there is `&str`, an immutable borrow to a string. Every literal string that occurs in your program is such a `&str` (this makes sense, because you should not be able to mutate literal strings). There is also `String` which is an **owned** string that you can mutate. In this exercise we will use `String`. To convert a `&str` to a `String`, use the `to_owned()` method, e.g., `"test".to_owned()`. To convert a `String` to a `&str`, just use an immutable borrow (`&`).
- There are multiple ways to iterate over the contents of a string, you can iterate over its bytes, characters, lines, ... Maybe there is another iterator that might be useful for this exercise? You can always split tokens manually if you want to. Consult the API for `String`.
- Strings can be converted to numbers using the `parse` method, see the API for `String`.
- There are many methods for `Result` and `Option` that will make it easier to work with them, e.g. `map`, `ok_or`, `map_err`, `and_then`, etc. The `?` syntax is also very useful, see this example. Also have a look at the section about Error Handling in the Rust book.
- You can use the `format!` macro to format strings similar to how you print using `println!`, see this example.