Lily Sanchez and Jacob Valero

Rust Report - Assignment 3

Built-in Types

Rust has four single-value types: integers, floating point values, Booleans, and characters. Integers and floats can be customized when they are declared, but default to being signed and 32-bit (for integers) and 64-bit (for floats). Booleans are one byte in size, and characters are four bytes, and can include any Unicode character, including accented letters, foreign characters, and emojis.

Rust also has two built-in types that contain multiple values: tuples, which can contain values of different types, and arrays, which cannot. Both tuples and arrays have a fixed size upon initialization. The values are stored at indices 0–*n*-1, where n is the size of the tuple or array, and are accessed with the forms *variableName***.**index and *variableName***[**index**]**, respectively.

Variables

Variables in Rust are immutable by default, meaning that the programmer must specify if they want to be able to bind a new value to a variable upon its declaration, using the keyword **mut**. The names of immutable variables can, however, be reused with the keyword **let**, effectively replacing the previous reference with whatever the programmer has most recently bound to the variable. This is called “shadowing”. Unlike mutable variables, shadowed variables can be assigned new types when they are updated, letting the name *x* for example first reference a boolean, then later, a String, so long as the keyword **let** is used when *x* is bound to a String. For instance:

```
let x = true;
```

```
let x = “Hello”;
```

will assign the String value “Hello” to x, even though it had been a boolean previously. The name is simply reused.

Meanwhile:

```
let mut y = true;
```

```
y = “Hello”
```

will not compile. Mutable or not, variables cannot have their types changed without being redeclared.

Control Structures

Assignment: let <var> = <expression>

As shown above, initial assignment of variables, known as bindings in rust documentation, begin with **let**. For example:

```
let x = 5;
```

Similarly to Go, Rust has type inference and will infer the value of 5 as a 32bit Integer (i32). However we can also declare the typing of the variable as follows:

```
let x: i32 = 5;
```

Note that Rust does not allow for multiple assignments.

Conditional Statements - If-elif-else

Here is the general structure for if statements, which resemble other programming languages and is similar to Go in that the conditional statement does not need to be surrounded by parentheses while the block is enclosed in braces:

```
if <condition> {
     <statement(s)>
} elif <condition> {
     <statement(s)>
} else {
     <statement(s)>
}
```

Conditional statements - match, if let, while let

match works very similarly to switch statements from languages like Java, C, and Go. However unlike Java and C, there is no default fallthrough behavior.

```
match <variable> {
     <possible instance of the same type> => <expression>
     <possible instance of the same type> => <expression>
     ...
     _ => <expression> //This is the default case
```

For example:

```
let number = 5;
match number {
     1 => println!(“One”);
     5 => println!(“Five”);
     _ => println!(“Not one or five.”);
```

Rust also allows you to set the outcome of a match to a variable as such:

```
let result = match boolean {
     //etc.
}
```

if let and **while let** have similar functionality to match, except for that they allow for non-exhaustive case catching with **if let**, and adding while loop functionality to match with **while let**, allowing for more concise expressions when compared to their **match** counterparts. They follow the structure below:

```
if let <data> = <case> {
     <statement>
} else if { // Optional: Alternate case
     <statement>
} else { // Optional: Default failure case
     <statement>
}

while let <data> = <case> { // Think of this first line as the
     <statements>           // conditional in a while loop
}
```

If you're wondering why these control structures exist, **match** and its concise relatives **if let** and **while let** exist for rust's goal of speed, as simply checking for equality across the same type is faster than evaluating boolean expressions.

Loops - loop, while, and for loops

Rust provides three different types of loops defined by their keywords: **loop**, **while** and **for** loops. **loop** is an infinite loop, and **while** and **for** loops resemble the usage of many other programming languages where **while** continues until its condition is not true and **for** loops iterate over a range of values, which can include a data structure that is iterable, such as arrays. See templates below:

```
loop {
     <statements>
}
```

```
while <condition> {
      <statements>
}


for <variable> in <range or iterable> { //To specify range: x..y where
<statements>                          //x < y
}
```