Rust Programming Language

By: Michael Plekan

Introduction

The Rust programming language is a pretty new language relatively to C or Java, only being first created in 2015. My interest in it comes from the combination of features to improve writing speed by shorthand syntax and the execution speed on the same level as C. The overall look and feel is drastically different from Python and Java with colons and arrows being used quite frequently in the syntax. For this project I go over a lot of the major features of Rust and compare them to Java, Python, or C. I then go into a few features in more detail in the deep dive section. Lastly, I talk about the jacobi iteration program I made during this project.

Language Overview

This section goes over a bunch of the language features with some examples of the features showing off how they might be used. This section is to give readers a general idea of Rust and some of the differences.

Data Types

Primitives

Rust has the same standard primitive types but the syntax for them is different. In C there is int, float, long, etc, whereas in Rust use i,u, or f followed by 8,32,64,128, or size. The letter specifies whether it is a signed integer, unsigned integer or floating point number and the number specifies the number of bits to use or use the architecture default. This departure from C can be confusing at first but it tells the programmer more information about the variable than the normal naming convention does. This can be nice when debugging overflow errors or trying to reduce memory use to a minimum. I think this is a good thing for both readability and writability after getting used to it.

Strings and str

Rust has two types of strings. The first is String which is stored in a vector which is the Rust version of an Arraylist or dynamically sized array. This storage method allows them to be growable. The second type of string is str which seems to be the more simple almost C like string. The best part about strings in rust is the amount of functions that are available to modify them.

```
string.chars().filter(|x|*x=='l').count()
```

Just within this simple example the programmer can split the string into chars, filter it and count

the number of occurrences that passed the filter. This is only skimming the surface of what Rust has built in to work with and modify strings.

Custom Types

There are two main custom types in Rust, the struct and the enum. The struct type is a lot like the structs in C. They can hold multiple named fields. The fields don't have to match types. Where Rust really differs is with the enum type. They can be used like C enums but they can also do so much more. You can have structs as options in the enum. You can also specify that you want to use the names within the enum without having to use the name of the enum. This would be very useful in the tokenizer and parser because we can use the names but not have to worry about what they equal.
One last feature for both of these is the ability to add functions to them.

```
enum Coins{
    Penny=1,
    Nickel=5,
    Dime=10,
    Quater=25,
    Dollar(i32)
}

impl Coins{
    fn to_dollars(&self)->f64{
        match self{
            Coins::Penny => return 0.01,
            Coins::Nickel => return 0.05,
            Coins::Dime => return 0.1,
            Coins::Quater => return 0.25,
            Coins::Dollar(v) => return *v as f64,
        }
    }
}
```

This example shows an enum called coins where there is a function to convert the values into the dollar amount. The self parameter is very similar to self in python. Overall this extra functionality is nice and can make the code more readable by removing extra words that don't need to be there,like in the tokenizer and parser example.

Modules

The modules in Rust are like classes in Java and python. They can hold functions, structs, and more. They have a hierarchical system and can manage permissions (private/ public) like Java.

Control Flow

Loops

We all know about the for and while loops and Rust has them like any other language but it also has some neat features for them as well. There is a loop keyword for infinite loops instead of having to do while(true) or a variation of that. You can also label your loops as shown in the example below.

```
'outer: while x<10 {//doesn't reach 10 b
    println!("Entered the outer loop");
    x+=1;
    'inner: loop {
```

I think this is a very useful feature because it not only allows readers to refer to different loops easier, it also lets you say exactly where you want a break to go. This improves both readability and writability. Another nice feature

of the loops is that you can assign a variable to the result almost like a function call.

```
let mut i = 2;
let solution:i32 = loop{
    if i >127 {
        break i;
    }
    i*=2;
};
```

I think this is an interesting feature because on one hand it shortens the code and makes it clear what you're trying to get from a loop but it can also make it harder to spot the assignment statements if you're not used to seeing them before loops.

If statements

The if statements are basically the same as any other language except for a future we saw in loops. This is the ability to assign a variable to the result of an if statement. It is sort of like a ternary operator taken to the extreme. This feature drastically increases the readability over other languages that can only match this using ternary operators.

If Let Statements

Let Else Statements

While Let Loop

Match statements

The match statements in Rust are alot like the switch statements of Java and C but like a lot of other features they have extra functionality. They can handle many different types of things, like strings, numbers, enums, etc. They can also match arrays, tuples, and structs. There are a lot of interesting things that can be done with them but that will be covered in the deep dive section.

Functions

Full Functions

Closures

Memory Management

Borrowing

Referencing

Deep Dive

Match statements

Topics:binding and guards and assignment of result

Option types

Topics: None vs Some, how this can be useful for make error tolerant code(easy error catching)

Ownership and Borrowing

Topics: To the best of my understanding explain how this deals with the memory management of Rust

Jacobi Program

While learning Rust, I built a jacobi iteration program in Rust. I made it using a bunch of different language features.

Struct

I created a struct for holding the data in each spot of the matrix.

```
struct Field {
    value: f64,
    permanent: bool,
}
```

The value field holds the numeric value and the permanent field is set true for spots that shouldn't be changed. This is very straightforward if you have made a struct in C. Where this differed is using the derive keyword as seen below.

```
#[derive(Debug, Clone, Copy)]
struct Field {
```

This keyword along with the fields inside are things that I haven't seen in other languages in this format. The derive is telling the compiler to build the functions or other things needed to implement Debug, Clone, and Copy. Debug is used for getting a simple print format of the struct. Clone and Copy are for moving the data around. This is apart of the language I didn't dive into that much but it definitely seems like it could be very useful.

Enums

I used an enum inorder to manage the start condition.

```
//create an enum for the start type
enum Start {
    Boundary,//all edges are 1
    Diagonal,//diagonal is 1
    Random,//random points are 1
}
```

There are 3 types of start positions that the program can do. The boundary puts ones along

all the edges and sets them all to permanent as the struct describes. The diagonal sets a line of ones going diagonally through the matrix and set to permanent. The last is just a random amount and at random positions get set to 1 and set to permanent. This enum lets me cleanly use the match later on.One last thing to mention is this line below.

```
fn main() {
    use Start::*;
```

This is very similar to C++ where you can say use a namespace for this section so you don't have to type it out everytime. The line above does this for the enum I created.

Macros

Match statement

For Loop

Conclusion

In the future there are so many more features and use cases for Rust that I want to explore. Even though I went through a lot of features in this paper, it just skims the surface. They have an entire book like C for the Rust language, if that gives an idea of how big the language is. Also there is the package manager which expands the language to do pretty much whatever you want. An area of Rust that I would like to explore more is the multithreaded appellation of Rust. I'm curious to figure out if the safety features of Rust stop it from getting some of the same performance results as C.

Citations:

Introduction - Rust By Example

The Rust Programming Language