CSCI 324 Term Project: An Analysis of the Rust Programing Language
By Mikas Marinos and Wesley Smith

This was a group project, the parts highlighted I wrote, jump to page 5

Historical Background of Rust

The programming language Rust was created by Graydon Hoare in 2006. Hoare was a programmer for Mozilla who found himself annoyed with the crashing and memory bugs that were found in C and C++. As a result, Hoare started working on a compiler that fixed the most common memory bug issues. This piqued the interest of employers at Mozilla who saw potential in the language to better implement web browsers. Soon Mozilla sponsored the project by devoting a small team to the project which progressively grew over time. The team set out to enforce memory safety by implementing strict rules on data and how it is moved, which created a memory system that would also protect concurrency. In 2013 Rust had perfected its memory system so they were able to dispose of its garbage collector. A garbage collector is a way to automatically manage memory which most higher-level languages use, however, it comes at the cost of performance. This allows Rust to be used where bare-bones languages like C, and C++ are needed yet still have the memory safety of languages like Javascript and Python. Rust's antecedents were C, C++, Javascript, ML, and Python. Rust is a multi-paradigm language with imperative, functional, and some object-oriented constructs [9].

Elements of the language

Rust breaks its keywords into three distinct categories, strict keywords which are keywords that cannot be used as identifiers, reserved keywords are keywords that are reserved for future use so Rust can be compatible with its future versions, and loose keywords which are keywords that can be used as identifiers. In total, Rust has upwards of 30 strict keywords, most of which are commonly used throughout most programming languages [7]. A few unique words to highlight are “crate” which is a part of the “extern crate” declaration which is used to import libraries into the program. The keyword “mut” designates a variable as mutable. “unsafe” is used to mark code that Rust’s borrow checker will ignore. This is needed for certain use cases like bindings to other languages where the compiler can not guarantee safety [4].

Rust includes the most common primitive data types found in other languages with some unique differences. Rust’s standard types shares similarities with a lot of other comparable languages like C or Java such as boolean values, char values, and strings. However, when it comes to integers and floating points Rust allows for greater control. Letters i, u, and f precede a number ranging from 8-128 each doubling. Rust assigns the letter ‘i’ to mean a signed integer, ‘u’ represents an unsigned integer, and lastly ‘f’ represents a floating point number. The number following the letters represents how many bits to use. Please note that for floating point integers there are only two values f32 and f64. In addition to these options there are options for isize and usize primitive types which use the target architecture’s register size [2]. In addition to this Rust has a fn primitive type. This type is a pointer to the actual code within the function, they can be called just like functions.

In addition to the primitive types, Rust has many different ways to structure data. The first is an array that operates similarly to most C-like languages, however, the array size must be a const known at compile time. Tuples operate similarly to an array but can contain different

data types. Structs are custom data types similar to tuples except the fields are named and it is possible to make certain values in them private. It operates very similarly to an object's data attributes in an object-oriented language. An enum is a data type that allows a user to define possible variants. Another of Rust's structured types is a slice, which is a pointer to a block of memory. Slices can be used with arrays, strings, etc. Slices are usually used to access portions of these data types and their size is determined at runtime.

Language Syntax

In terms of syntax, Rust shares a lot of similarities to antecedent programming languages such as Java and C. The reserved word ‘let’ Rust is used to assign a variable a value. Rust includes type inference, allowing the compiler to determine the variable type at compile time. Rust requires the programmer to declare mutable/changeable variables with the keyword “mut”. This forces users to think about how variables are used within programs and further promotes data safety and concurrency within Rust. Furthermore, this enables Rust’s compiler to more easily detect errors at compile time. Conditional controls are pretty similar to other languages, if, if else, and else if statements all work practically the same as in C++ or Java.

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

Rust has three different kinds of loops including “loop”, “while”, and “for”. “loop” just loops until “break” is called. All loops are expressions, the example below sets “result” to 20.

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

Rust’s while loops are pretty similar to C++ and Java, however, it does not need parentheses around the condition

```
    while number != 0 {
        println!("{number}!");

        number -= 1;
    }
```

Lastly, Rust's for loops are similar to Python's for loops and Java and C++'s for each loop. A variable is assigned to an element in an iterable data type and with each iteration the variable is changed to the next item until all items have been consumed.

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

Rust's functions operate pretty similarly to most C-like language functions. Nesting is allowed and the final expression in the function body becomes the return value. Short circuit returns are also allowed

```
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    // Corner case, early return
    if rhs == 0 {
        return false;
    }

    // This is an expression, the `return` keyword is not necessary here
    lhs % rhs == 0
}
```

Language Evaluation

Being a high level mostly imperative language, Rust's readability and writability are alright, similar to Java and C, however, its reliability excels, and perhaps is one of the highest in any programming language. Rust's readability is a mixed bag. Rust's syntactic similarities to other popular languages cause it to suffer from some of the same readability issues, such as some feature multiplicity and operator overloading, however, it does not allow for function overloading. Furthermore, Rust has a much larger amount of basic constructs, which can further these problems and create a learning curve for those new to computer science. An example of this could be found in something as simple as the data types for numbers in Rust. Rust gives an assortment of options for integer and floating point values which is hard to understand if not familiar with the language or new to programming in general. In addition to this, the sheer volume of number types would lead to an increase in operator overloading through basic operations like addition and subtraction. Rust's writability suffers from some of the same issues as its readability. It's very similar to other popular languages and the compiler offers very friendly error messages, but to understand how to fully take advantage of Rust, one must know the ins and outs of memory and how it is used. Rust makes up for these issues in its reliability. Rust

was built around the concept of reliability and as a result has excellent type-checking, exception handling, and restricted aliasing. Rust's compiler and memory safety is something that Rust was built around. As a result, the compiler will type check as well as restrict many problematic forms of aliasing. In addition to this, Rust's exception handling is excellent. Rust has two different types of exception handling, recoverable and unrecoverable or panics. Rust forces the programmer to handle any recoverable error, ensuring a reliable program. Panics shut the program down to ensure the program is not left in a bad state.

Overall, Rust is a great language, it can be complicated at times affecting Rust's readability and writability as all the compiler checks performed require the programmer to provide more information. Needless through all these compiler checks, Rust can provide much greater reliability while still maintaining great runtime performance. Rust's excellent reliability coupled with the fact its readability and writability are only slightly more complicated than competitor languages is the reason why Rust is one of the fastest-growing languages.

Simplicity	• No function overloading • Must explicitly declare any casting whether widening or narrowing the type. • Some feature multiplicity but not excessive. count++ does not exist, += does. "hello".to_string() equivalent to String::from("hello"). • Some feature multiplicity also exists for features that the compiler can not check memory safety. Ex unsafe and raw pointers, these are needed for a small set of uses like OS programming and bindings to other languages • Allows operator overloading • Large feature set, supports multiple coding paradigms
Orthogonality	• The functional constructs in rust are very orthogonal • Everything is an expression with a value, even void functions • Features such as error handling and the Result data type are built out of constructs already in the language such as Enums with Generics • Some imperative and object-oriented concepts exist that are not orthogonal
Data Types	• Very mature standard library with Vectors, Maps, Sets, and more • Good collection of primitives, unsigned and signed integers, Strings, etc.
Syntax Design	• Special words can not be used as vars • Standard amount of keywords
Support for Abstraction	• Supports functions, generics, objects

	• No Inheritance
Expressivity	• Support for many coding paradigms
Type Checking	• Strictly typed • Compiler can guess most types
Exception Handling	• Fully featured, with two classes of errors
Restricted Aliasing	• Rust borrow checker ensures that if more than one reference exists to a value only one can mutate it • No mutable references can exist when an immutable reference exists, guaranteeing immutable will never be changed

Simplicity, Orthogonality, Datetypes, and Syntax Design affect Readability. Those and Abstraction and Expressivity affect writability. All affect reliability [10].

Features Used by Our Common and Creative Program

Our common and creative program uses many features unique to rust. Both of our projects make use of Borrowing, exception handling, and use many elements taken from functional programming. Our creative program also uses features in Cargo, its package manager to easily integrate with a parser generator. Our common program snowman plays a game of hangman and allows adding words to a dictionary file. Our creative project, markdown, generates an HTML file from a latex-inspired markdown language.

Borrowing

Essential to Rust, perhaps its reason for existence is its approach to memory management. Among the classical languages, there are two main approaches, manual memory management, and garbage collection. Often lower-level languages will opt for manual memory management as it often offers better performance at the cost of memory safety, such as memory leaks. Garbage collection offers a solution at the cost of runtime performance. Rust offers a different solution to the problem, with borrowing and being able to determine the object's lifetime at compile time, Rust can offer the benefits of both.

With borrowing, rust ensures that all references point to a valid memory location, there is only one mutable reference at a time, and for any immutable references the value will not change.

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

All references must be marked “mut” if they can be changed, it will not compile if “&mut” is “&”.

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

This will not compile as Rust requires that if an immutable reference exists like r1 and r2, no mutable references can be created [3].

Exception Handling

In Rust, there are two classes of errors, recoverable and non-recoverable. With unrecoverable errors, panics are used and cause the program to halt and come to a stop. Recoverable errors are handled by returning an enum, called Result either containing the value of the desired operation or the error. The error is anything that has debug implications which provide a method to print the error data type to the console. By being an enum, we can then pattern matching the ok and the error values. Rust’s standard libraries include various shorthands to make this less tedious. The following is the definition of Result in the standard library.

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The next example, the function parse_version, returns a Result, either with an Ok containing the Enum Version, or a string with the error. Pattern matching is then used to determine how to respond if an error

```
#[derive(Debug)]
enum Version { Version1, Version2 }

fn parse_version(header: &[u8]) -> Result<Version, &'static str> {
    match header.get(0) {
        None => Err("invalid header length"),
        Some(&1) => Ok(Version::Version1),
        Some(&2) => Ok(Version::Version2),
        Some(_) => Err("invalid version"),
    }
}

let version = parse_version(&[1, 2, 3, 4]);
match version {
    Ok(v) => println!("working with version: {v:?}"),
    Err(e) => println!("error parsing header: {e:?}"),
}
```

One problem with this is it's very verbose. If this is required anytime an error is possible it would reduce readability and make people less likely to write functions that return Result, instead they may opt to just have it panic. To make it easier, Rust's standard libraries include methods to allow Result's to be handled easily. The two most common examples are .unwrap() and .expect(String). Unwarp will cause the program to panic if the value is an error, and expect will likewise cause the program to panic but it also panics with a string specified by the programmer to provide nicer errors. Rust also allows "?" to be used to propagate the errors down to lower functions [1].

We use exception handling in markdown as the parser generator returns a Result with either the html string or a struct as the error. As an example we use pattern matching to catch the error unknown command, and for others we use the debug implications to print the error struct.

Rust's Functional Design Influences

Rust includes some syntactical elements that allow writing in a functional style with ease. While not functions themselves, if statements, and pattern matching, are expressions and return a value.

```
let s = {64; print!("Hello"); "World"}; // s is set to "world"
println!(s);
```

The above statements print hello world as s is assigned to a list of statements, which the value of the last is used for assignment

Inside the grammar, markdown.lalrpop, once we determine a token is a command, the code used for determining what each command does uses pattern matching that returns a string.

```
match c.as_str() {
    "\\textbf" => Ok(format!("<strong>{p}</strong>")),
    "\\textit" => Ok(format!("<em>{p}</em>")),
    "\\latex"  => Ok(latex),
    "\\n" => Ok("<br>".to_string()),
    "\\\\" => Ok("\\".to_string()),
    error => Err(ParseError::User {error: Errors::UnknownCommand(error.to_string())}),
}
```

The parser generator puts c and p, the command, and the text respectively in scope. This match expression is also the last expression in a list of statements that precede it, as such its value is assigned to be of the whole block.

Cargo

While not directly a part of the language, Rust's package manager is inseparable from its regular use. Cargo performs essential functions such as dependency management, it sets the

many compiler flags needed to build the package, ensures builds are reproducible with different environments, and makes everything a package [5].

Cargo also sets out a directory structure for the program. Running “cargo new project_name” sets out the initial structure of the project. It creates a cargo.toml file which contains the metadata and dependencies needed to build the project while also creating a folder src where the main.rs resides. To add dependencies, they can be listed in cargo.toml. Among the many features, cargo allows for, one of them is build scripts which our project makes use of.

Our project uses a parser generator Larlpop, using a parser generator at first glance may seem intimidating, but with Cargo, it is fairly simple to include in the project. To integrate it into the project we use a feature known as build scripts. This feature allows rust code to be executed on the code we are building. To include the parser generator only a few lines need to be modified in the cargo.toml file, the file for markdown looks as follows.

```
[package]
name = "markdown"
version = "0.1.0"
edition = "2021"
build = "build.rs" # LALRPOP preprocessing
[dependencies]
lalrpop-util = { version = "^0.19", features = ["lexer"] }
regex = "1"
# Add a build-time dependency on the lalrpop library:
[build-dependencies]
lalrpop = "0.19.9"
```

Simple as that, the grammar provided in markdown.larlpop will be parsed and converted into a rust lr(1) parser at build time. Our parser builds a string, but it could just as easily build an AST. If we look into the grammar, we can see it contains Rust code; everything after the “=>” is a Rust expression. Having build scripts makes it easy to create languages for domain-specific purposes that may want to use rust code or for integrating with bindings from other languages [6].

```
LHS_Non_Terminal: type_of_attribute_it_synthesizes = {
    <value_of:RHS_Non_Terminal> => { rust expression that generates type_of_ attribute_it_synthesizes },
        ...other rules that generate LHS_Non_Terminal...
}
```

Syntax of Lalrpop’s Grammar

Conclusion and Strengths and Weaknesses of Rust

Rust has multiple advantages stemming from its memory safety and performance. With no runtime or garbage collector, Rust has very good performance, comparable to C and C++ [8]. In addition to this, Rust’s primary concern is memory safety, and because of this Rust is an extremely reliable language. Although Rust is a great language it does have some disadvantages. Firstly, Rust is slightly more complicated than comparable languages creating more of a learning curve for writing and reading the language. In addition, because of all the runtime checks, Rust has a slow compile time. Lastly, while its library support is fairly mature, it can not compete with languages like Python or C++ with the sheer momentum behind them. For

use cases desiring reliability and performance, Rust offers many unique features that make it an ideal choice.

References

1. Crate std; https://doc.rust-lang.org/std/result/ Accessed 2023 April 23.
2. Klabnik, S. and Nichols, C. *The Rust Programming Language. No Starch Press,* 9 (February 2023); https://doc.rust-lang.org/book/ch03-02-data-types.html Accessed 2023 April 23. Chapter 3.2
3. Klabnik, S. and Nichols, C. *The Rust Programming Language. No Starch Press,* 9 (February 2023); https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html. Accessed 2023 April 23. Chapter 4.2
4. Klabnik, S. and Nichols, C. *The Rust Programming Language. No Starch Press,* 9 (February 2023); https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html Accessed 2023 April 23. Chapter 19.1
5. The Cargo Book; https://doc.rust-lang.org/cargo/guide/why-cargo-exists.html Accessed 2023 April 23. Chapter 2.1
6. The Cargo Book; https://doc.rust-lang.org/cargo/reference/build-scripts.html Accessed 2023 April 23. Chapter 3.8
7. The Rust Reference; https://doc.rust-lang.org/stable/reference Accessed 2023 April 23.
8. The Computer Language 23.03 Benchmarks Game; https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html Accessed 2023 April 23.
9. Thompson, C. How Rust Went from a Side Project to the World's Most-Loved Programming Language. *MIT Technology Review*, 15 (February 2023); https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/ Accessed 2023 April 23.
10. Sebesta, R. *Concepts of Programming Languages.* 11th ed.*(* Pearson Education*,* 2016)