

# Rust-eze

CMPT 331 - Spring 2023 | Dr. Labouseur

Josh Seligman | [joshua.seligman1@marist.edu](mailto:joshua.seligman1@marist.edu)

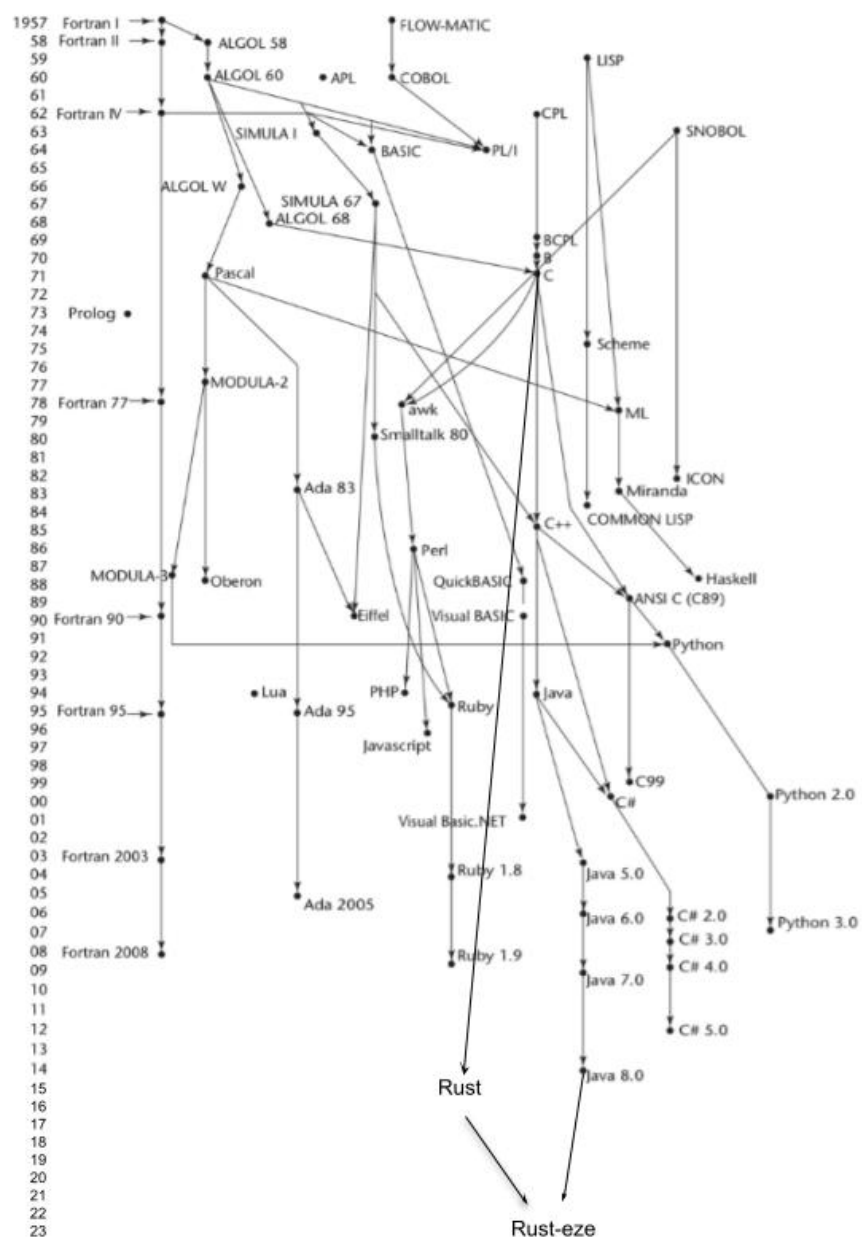
May 10, 2023

# 1 INTRODUCTION

Rust-eze is modern, object-oriented, type-safe programming language. It inherits the best parts of Rust and Java to provide a fast and memory-safe language for the object-oriented paradigm, while also bringing in some influence from Pascal. Rust-eze does, however, differ from its parent languages in the following ways:

1. Rust-eze is an object-oriented language, which means there are no structs and only classes/objects and enums.
2. Like Java, but unlike Rust, Rust-eze is statically typed, so all variables must be explicitly defined with their respective types.
3. Similar to Rust, but unlike Java, Rust-eze uses a system of borrowing and ownership so only one variable can point to a given place in memory at a time. This prevents the need for a garbage collector as variables are automatically dropped and the memory is freed when they go out of scope.
4. Unlike Java, but like Rust, Rust-eze is compiled into the native binary, so there is no need for a JVM or an intermediate bytecode representation of Rust-eze programs.
5. Similar to Rust, all instance variables must be initialized within the constructor.
6. Similar to both parent languages, all classes belong to a module (Java package). However, more similar to Rust, the module is inferred based on the relative file location and does not have to be explicitly defined within the file.

## 1.1 GENEALOGY



## 1.2 HELLO WORLD

```
1 model HelloWorld
2 start
3     pub fn main(Vec<String> args)
4         start
5             println("Hello world!");
6         finish main
7 finish HelloWorld
```

Listing 1: HelloWorld.rez

## 1.3 PROGRAM STRUCTURE

The key organizational concepts in Rust-eze are as follows:

1. Every file contains a single **model** (equivalent to a class), which should be the same name as the file minus the .rez file extension.
2. All instance members are by default private unless they are declared with the **pub** keyword.
3. Instance variables are declared within the **specs** block and must be initialized within the constructor, which is a function that is the same name as the model.
4. Local variables are immutable by default unless they are declared with the **mut** keyword.
5. The entry point for all Rust-eze programs is the main method that is located in one of the models of the project.

The program below defines a new **model** called **Lightning** that contains 3 instance variables: a String that is public called *name*, a public integer called *age*, and a private integer called *miles*. Each of these instance variables are initialized within the constructor. Each **Lightning** has 2 member functions: *drive* and *say\_it*. *drive* is a public method as it is defined with the **pub** keyword. Since it modifies an instance variable, it must take it a mutable reference to the object in addition to the number of miles being driven. Inside of the method, a new local variable called *new\_mileage* is declared without the **mut** keyword, meaning that it is immutable and is a constant with the value it is initialized with. The other instance method, *say\_it*, is also public and does not modify the instance variables, which is why it needs an immutable reference to **self**.

```
1 model Lightning
2 start
3     specs
4     start
5         pub String name;
6         pub int age;
7         int miles;
8     finish specs
9
10    pub fn Lightning(String my_name, int my_age)
11        start
12            self.name := my_name;
13            self.age := my_age;
14            self.miles := 0;
15        finish Lightning
16
17    pub fn drive(&mut self, int num_miles)
18        start
19            int new_mileage := self.miles + num_miles;
20            self.miles := new_mileage;
21        finish drive
22
```

```

23     pub fn say_it(&self)
24     start
25         println("Kachow!");
26         finish say_it
27 finish Lightning

```

Listing 2: Lightning.rez

Next, the Lightning model is imported to a new file called *Main.rez* and is initialized within the main method. Since the **mut** keyword is used, we can use the mutable method of *drive* on the new object as well as directly modify its public instance variables. After *the\_lightning*'s name is printed, we call its *say\_it* method and then call the *drive* method with an input of 42 miles to increase the object's mileage.

```

1 import garage.Lightning;
2
3 model Main
4 start
5     pub fn main(Vec<String> args)
6     start
7         mut Lightning the_lightning := new Lightning("McQueen", 17);
8         println(the_lightning.name);
9
10        the_lightning.say_it();
11        the_lightning.drive(42);
12    finish main
13 finish Main

```

Listing 3: Main.rez

## 1.4 TYPES AND VARIABLES

There are two kinds of variables in Rust-eze: *value types* and *reference types*. Variables of value types directly contain their data, while variables of reference types store references to their data or objects in memory. Due to the ownership system, only one variable of a reference type can point to a particular place in memory at a given time. See Section 3 for details.

## 1.5 VISIBILITY

In Rust-eze, visibility of methods and instance variables is defined as either public or private. Everything is default private unless explicitly stated to be public. Once a variable or method is public, it may be accessed outside of the model in which it is defined.

## 1.6 STATEMENTS DIFFERING FROM RUST AND JAVA

Statement	Example
Assignment statement	<pre> mut int x := 5; x = 3; int y := x + 2; </pre>

If statement	<pre> int x := 7; if x &gt; 3 &amp;&amp; x &lt; 9 start     println("Hello there") else     println("Kachow") finish if </pre>
For loop	<pre> for (mut int i in range(0, 10, 1)) start     println(i) finish for </pre>

## 2 LEXICAL STRUCTURE

### 2.1 PROGRAMS

A Rust-eze program consists of one or more source files. A source file is an ordered sequence of (probably) Unicode characters.

Conceptually speaking, a program is compiled using five steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis (parsing), which translates the stream of tokens into a concrete syntax tree (CST).
4. Semantic analysis, which converts the CST into an abstract syntax tree (AST) and is passed to the semantic analyzer for type checking and the borrow checker to make sure all variables referenced are active owners of data.
5. Code generation, which converts the AST into executable code for the target platform and CPU architecture.

### 2.2 GRAMMARS

This specification presents the syntax of the Rust-eze programming language where it differs from Rust and Java.

#### 2.2.1 LEXICAL GRAMMAR (TOKENS) WHERE DIFFERENT FROM RUST AND JAVA

```

<assignment operator> → :=
<block begin> → start
<block end> → finish
<print> → println
<visibility modifier> → pub |  $\epsilon$ 

```

<mutability modifier> → mut |  $\epsilon$

### 2.2.2 SYNTACTIC ("PARSE") GRAMMAR WHERE DIFFERENT FROM RUST AND JAVA

<model definition> → model <model name> <block begin> <statements> <block end> <model name>  
<specs definition> → specs <block begin> <spec definition> <block end>  
<spec definition> → <visibility modifier> <type> <spec name>; <spec definition> |  $\epsilon$   
<variable declaration> → <mutability modifier> <type> <variable name> <assignment operator> <expression>;  
<for loop> → for (mut <type> <var name> in <expr>) <block begin> <statements> <block end>

## 2.3 LEXICAL ANALYSIS

### 2.3.1 COMMENTS

Rust-eze supports two forms of comments: single-line and multi-line comments. Single-line comments start with the characters // and extend to the end of the line in the source file. Multi-line comments begin with /\* and end with \*/ and may span multiple lines. Comments do not nest.

## 2.4 TOKENS

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

Tokens:

- identifier
- keyword
- integer literal
- real literal
- character literal
- string literal
- operator or punctuator

### 2.4.1 KEYWORDS DIFFERENT FROM RUST AND JAVA

**New keywords:** range, model, specs, start, finish

**Removed keywords:** use, class, match, do, private, byte, short, str, boolean, static, public, void, i32, float

## 3 TYPE SYSTEM

### 3.1 TYPE RULES

### 3.2 VALUE TYPES (DIFFERENT FROM RUST AND JAVA)

### 3.3 REFERENCE TYPES (DIFFERENT FROM RUST AND JAVA)

## 4 EXAMPLE PROGRAMS

### 4.1 CAESAR CIPHER ENCRYPT

### 4.2 CAESAR CIPHER DECRYPT

### 4.3 FACTORIAL

### 4.4 SORT

### 4.5 PROGRAM 1

### 4.6 PROGRAM 2