Rust-eze

CMPT 331 - Spring 2023 | Dr. Labouseur

Josh Seligman | joshua.seligman1@marist.edu

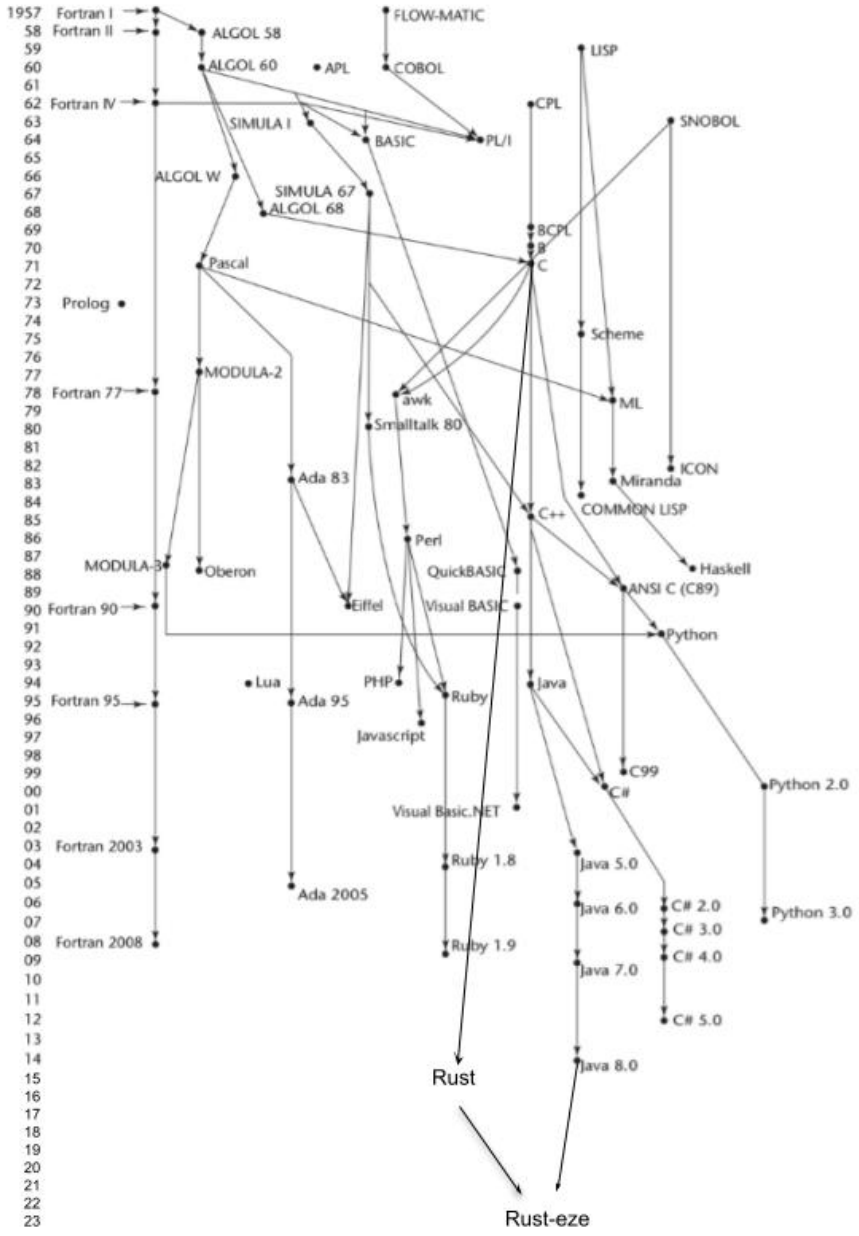
May 10, 2023

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 languge for the object-oriented paradim, 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

```
model HelloWorld
start
    pub fn main(Vec<String> args) -> void
    start
        println("Hello world!");
    finish main
finish model
```

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**.

```
model Lightning
start
    specs
    start
        pub String name;
        pub i32 age;
        int miles;
    finish specs

    pub fn Lightning(String my_name, i32 my_age)
    start
        self.name := my_name;
        self.age := my_age;
        self.miles := 0;
    finish Lightning

    pub fn drive(&mut self, i32 num_miles) -> void
    start
        i32 new_mileage := self.miles + num_miles;
        self.miles := new_mileage;
    finish drive

```

```
23      pub fn say_it(&self) -> void
24      start
25          println("Kachow!");
26      finish say_it
27  finish model
```

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) -> void
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 model
```

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	`mut i32 x := 5;`
	`x = 3;`
	`i32 y := x + 2;`

If statement	`i32 x := 7;`
	`if x > 3 && x < 9`
	`start`
	`  println("Hello there")`
	`else`
	`  println("Kachow")`
	`finish if`

For loop	`for (mut i32 i in range(0, 10, 1))`
	`start`
	`  println(i)`
	`finish for`

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 | ϵ

<mutability modifier> → mut | ϵ

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> | ϵ
<variable declaration> → <mutability modifier> <type> <variable name> <assignment operator> <epression>;
<for loop> → for (mut <type> <var name> in <expr>) <block begin> <statements> <block end>
<function definition> → <visibility modifier> fn <function name> (<parameter list>) -> <return type> <block begin> <statements> <block end>
<parameter list> → <type> <parameter name>, <parameter list> | ϵ

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, double, long, float, int, as

3 Type System

Rust-eze uses a strong static type system. This means that the Rust-eze compiler will catch type mismatch errors at compile time through early binding compile-time type checking.

3.1 Type Rules

The type rules for Rust-eze are as follows:

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T \quad T \text{ is a primitive type}}{S \vdash e1 := e2 : T}$$

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T \quad T \text{ is a primitive type}}{S \vdash e1 == e2 : \text{bool}}$$

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T \quad T \text{ is a primitive type}}{S \vdash e1 != e2 : \text{bool}}$$

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T \quad T \text{ is a numeric primitive type}}{S \vdash e1 > e2 : \text{bool}}$$

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T \quad T \text{ is a numeric primitive type}}{S \vdash e1 < e2 : \text{bool}}$$

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T \quad T \text{ is a numeric primitive type}}{S \vdash e1 + e2 : T}$$

$$\frac{S \vdash e1 : \text{String} \quad S \vdash e2 : \text{String}}{S \vdash e1 + e2 : \text{String}}$$

3.2 Value Types

Data Type	**Description**
i8, i16, i32, i64	Signed integers that store up to X bits, where X is the number after "i" in the data type.
u8, u16, u32, u64	Unsigned integers that store up to X bits, where X is the number after "u" in the data type.

f32, f64	Floating point numbers that store up to X bits, where X is the number after "f" in the data type.
bool	Boolean value that can be either false or true.
char	Character value that stores the data for a single character.

3.3 Reference Types

Data Type	**Description**
String	A sequence of characters
Vec<T>	A dynamic, homogeneous array of values of type T, which can be any type.
Tuple	A collection of values of different types that is fixed in size.

4 Example Programs

4.1 Caesar Cipher Encrypt

```
model CaesarCipher
start
    pub fn encrypt(&self, &String in_str, i32 shift_amt) -> String
    start
        i32 real_shift := shift_amt % 26;
        Vec<char> out_vec := new Vec<char>();

        for (mut i in range(0, in_str.len(), 1))
        start
            mut i8 cur_char := (i8) in_str.char_at(i);

            if cur_char >= 97 && cur_char <= 122
            start
                cur_char := cur_char - 32;
            finish if

            if cur_char >= 65 && cur_char <= 90
            start
                cur_char := cur_char + real_shift;

                mut i32 diff := cur_char - 90;
                if diff > 0
                start
                    cur_char := 65 + diff - 1;
                else
                    diff := 65 - cur_char;

                    if diff > 0
                    start
                        cur_char := 90 - diff + 1;
                    finish if
                finish if
            finish if
```

```
34
35              char final_char := (char) cur_char;
36              out_vec.push(final_char);
37          finish for
38
39          return out_vec.join("");
40      finish encrypt
41
42      pub fn main(Vec<String> args) -> void
43      start
44          CaesarCipher cipher := new CaesarCipher();
45
46          String x := "Kachow";
47          String y := cipher.encrypt(&x, 95);
48          println(x);
49          println(y);
50      finish main
51  finish model
```

Listing 4: CaesarCipher.rez

4.2 Caesar Cipher Decrypt

```
1  model CaesarCipher
2  start
3      pub fn decrypt(&self, &String in_str, i32 shift_amt) -> String
4      start
5          return self.encrypt(in_str, -shift_amt);
6      finish encrypt
7
8      pub fn main(Vec<String> args) -> void
9      start
10         CaesarCipher cipher := new CaesarCipher();
11
12         String x := "Kachow";
13         String y := cipher.encrypt(&x, 95);
14         String z := cipher.decrypt(&y, 95);
15         println(x);
16         println(y);
17         println(z);
18     finish main
19 finish model
```

Listing 5: CaesarCipher.rez (enhanced)

4.3 Factorial

```
1  model FactorialProgram
2  start
3      pub fn factorial(&self, i32 num) -> i32
4      start
5          if num <= 0
6          start
7              return 1;
8          else
9              return num * self.factorial(num - 1);
10         finish if
11     finish factorial
12
13     pub fn main(Vec<String> args) -> void
14     start
```

```
        FactorialProgram fp := new FactorialProgram();
        println(fp.factorial(5));
        println(fp.factorial(0));
    finish main
finish model
```

Listing 6: FatorialProgram.rez

4.4 Quicksort

```
import std.util.Random;

model Sorts
start

    pub fn quick_sort(&self, &mut Vec<i32> data) -> void
    start
        self.quick_sort_with_indices(data, 0, data.len() - 1);
    finish quick_sort

    fn quick_sort_with_indices(&self, &mut Vec<i32> data, i32 start_index, i32 end_index) ->
         void
    start
        if start_index >= end_index
        start
            return;
        finish if

        mut i32 pivot_index := 0;

        if end_index - start_index < 3
        start
            pivot_index := start_index;
        else
            Random ran := new Random();

            i32 pivot_choice_1 := ran.randInt(start_index, end_index + 1);
            mut i32 pivot_choice_2 := ran.randInt(start_index, end_index + 1);
            while pivot_choice_2 == pivot_choice_1
            start
                pivot_choice_2 := ran.randInt(start_index, end_index + 1);
            finish while

            mut pivot_choice_3 := ran.randInt(start_index, end_index + 1);
            while pivot_choice_3 == pivot_choice_1 && pivot_choice_3 == pivot_choice_2
            start
                pivot_choice_3 := ran.randInt(start_index, end_index + 1);
            finish while

            if *data[pivot_choice_1] <= *data[pivot_choice_2] && *data[pivot_choice_1] >= *
                data[pivot_choice_3]
            start
                pivot_index := pivot_choice_1;
            else if *data[pivot_choice_1] <= *data[pivot_choice_3] && *data[pivot_choice_1]
                >= *data[pivot_choice_2]
                pivot_index := pivot_choice_1;
            else if *data[pivot_choice_2] <= *data[pivot_choice_1] && *data[pivot_choice_2]
                >= *data[pivot_choice_3]
                pivot_index := pivot_choice_2;
            else if *data[pivot_choice_2] <= *data[pivot_choice_3] && *data[pivot_choice_2]
                >= *data[pivot_choice_1]
                pivot_index := pivot_choice_2;
            else
```

```
                pivot_index := pivot_choice_3;
            finish if
        finish if

        i32 partition_out := self.partition(data, start, end, pivot_index);

        self.quick_sort_with_indices(data, start, partition_out - 1);
        self.quick_sort_with_indices(data, partition_out + 1, end);

    finish quick_sort_with_indices

    fn partition(&self, &mut Vec<i32> data, i32 start, i32 end, i32 pivot_index) -> i32
    start
        mut i32 pivot := *data[pivot_index];
        *data[pivot_index] := *data[end];
        *data[end] := pivot;

        mut i32 last_low_partition_index := start - 1;

        for mut i in range(start, end, 1)
        start
            if *data[i] < pivot
            start
                last_low_partition_index := last_low_partition_index + 1;

                i32 temp := *data[i];
                *data[i] := *data[last_low_partition_index];
                *data[last_low_partition_index] := temp;
            finish if
        finish for

        *data[end] := *data[last_low_partition_index + 1];
        *data[last_low_partiton_index + 1] := pivot;

        return last_low_partition_index + 1;
    finish partition

finish model
```

Listing 7: Sorts.rez

4.5 Program 1

4.6 Program 2