Rust-eze

CMPT 331 - Spring 2023 | Dr. Labouseur

Josh Seligman | joshua.seligman1@marist.edu

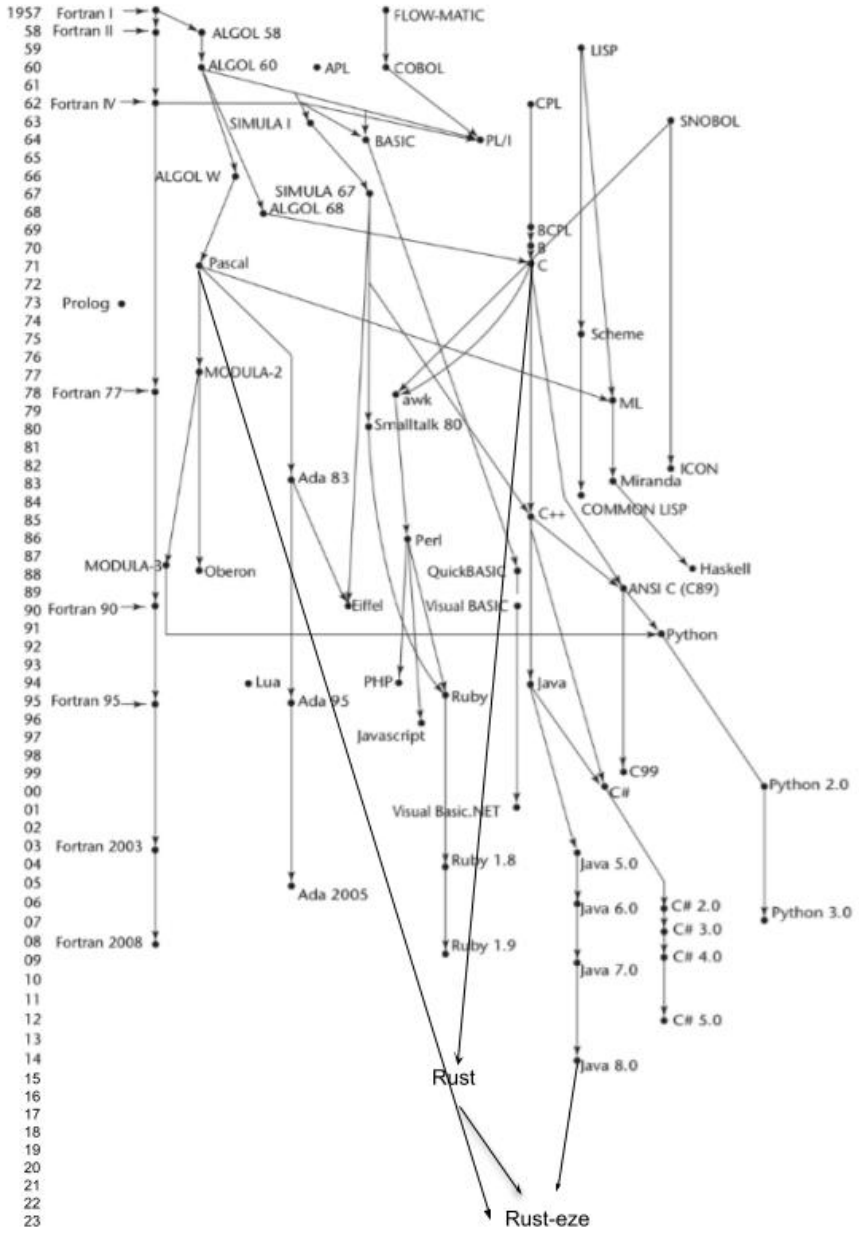
May 5, 2023

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 syntactic 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, like Java, means there are no structs and only classes/objects.
2. Like Java, but unlike Rust, Rust-eze requires that all variables 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 models (classes) belong to a garage (Java package). However, more similar to Rust, the garage 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
    ext 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 default to being in the interior (private) unless they are declared with the **ext** keyword to note their exterior (public) presence.
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: an exterior String called *name*, an exterior integer called *age*, and an interior 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 an exterior method as it is defined with the **ext** 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
        ext String name;
        ext i32 age;
        i32 miles;
    finish specs

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

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

```
22
23      ext 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 exterior specs. 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      ext 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 interior (private) or exterior (public). Everything is part of the interior unless explicitly stated to be in the exterior. 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`

While loop	`mut i32 i := 0;`
	`while i < 10`
	`start`
	`  println(i);`
	`  i := i + 1;`
	`finish while`

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 scope and 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> → ext | ϵ
<mutability modifier> → mut | ϵ

2.2.2 Syntactic ("parse") grammar where different from Rust and Java

<model definition> → model <model name> <parent declaration> <block begin> <statements>
<block end> model
<parent declaration> → extends <parent model name> | ϵ
<specs definition> → specs <block begin> <spec definition> <block end> specs
<spec definition> → <visibility modifier> <type> <spec name>; <spec definition> | ϵ
<variable declaration> → <mutability modifier> <type> <variable name> <assignment operator>
<expression>;
<if statement> → if <condition> <block begin> <statements> else <statements> <block end> if
<for loop> → for mut <type> <var name> in <expr> <block begin> <statements> <block end> for
<while loop> → while <condition> <block begin> <statements> <block end> while
<function definition> → <visibility modifier> fn <function name> (<parameter list>) -> <return type>
<block begin> <statements> <block end> <function name>
<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, ext, Tuple
Removed keywords: use, class, match, do, private, byte, short, str, boolean, static, public, double, long, float, int, as, pub, impl, struct

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:

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

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

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

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

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

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

$S \vdash e1 : String$
$S \vdash e2 : T$
T is either a primitive or is a reference type with a String representation

---

$S \vdash e1 + e2 : String$

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. Example: `String x := "hello";`

Vec<T>	A dynamic, homogeneous array of values of type T, which can be any type.
	Example:
	`Vec<i32> nums1 := [0, 0, 7];`
	`mut Vec<i32> nums2 := new Vec<i32>(2);`
	`nums2[0] := 42;`
	`nums2[1] := 99;`

Tuple	A collection of values of different types that is fixed in size.
	Example:
	`Tuple<i32, bool, i32> my_tuple := (7, false, 3);`
	`mut Tuple<String, i32> the_tuple := new Tuple<String, i32>();`
	`the_tuple.0 := "jOSh";`
	`the_tuple.1 := 2;`

4 Example Programs

4.1 Caesar Cipher Encrypt

```
// Model definition for the Caesar cipher
model CaesarCipher
start
    // Encryt a string based on the shift amount
    ext fn encrypt(&self, &String in_str, i32 shift_amt) -> String
    start
        // Get the real shift amount and create a new vector of characters
        i32 real_shift := shift_amt % 26;
        Vec<char> out_vec := new Vec<char>();

        // Loop through the entire string
        for mut i32 i in range(0, in_str.len(), 1)
        start

            // Get the character at the given index as an integer
            // (u8) casts the expression on the right of it to be an u8
            mut u8 cur_char := (u8) in_str.char_at(i);

            if cur_char >= 97 && cur_char <= 122
            start
                // Convert the lowercase letters to uppercase letters
                cur_char := cur_char - 32;
            finish if

            // Only modify uppercase letters
            if cur_char >= 65 && cur_char <= 90
            start
                // Perform the shift
                cur_char := cur_char + real_shift;

                // This is the difference for wraparound for Z
                mut i32 diff := cur_char - 90;
                if diff > 0
                start
                    // Perform the Z wraparound
                    cur_char := 65 + diff - 1;
                else
                    // Now compute the A wraparound if there was no Z wraparound
                    diff := 65 - cur_char;

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

            // Get the final character as the appropriate type
            char final_char := (char) cur_char;

            // Push it to the end of the vector (vector will resize as needed)
            out_vec.push(final_char);
        finish for

        // Join the elements of the vector together to form a string
        // by using the string representation of each of the elements
        return out_vec.join("");
    finish encrypt

    ext fn main(Vec<String> args) -> void
```

```
    start
        CaesarCipher cipher := new CaesarCipher();

        // Create a new string
        String x := "Kachow";

        // Perform encrypt and pass an immutable reference to x
        String y := cipher.encrypt(&x, 95);

        // Should print Kachow
        println(x);
        // Should print BRTYFN
        println(y);
    finish main
finish model
```

Listing 4: CaesarCipher.rez

4.2 Caesar Cipher Decrypt

```
model CaesarCipher
start
    // Returns a decrypted string based on the shift amount
    ext fn decrypt(&self, &String in_str, i32 shift_amt) -> String
    start
        // Decrypt is the same as encrypt with negative shift amount
        // Encrypt is defined in Section 4.1
        return self.encrypt(in_str, -shift_amt);
    finish encrypt

    ext fn main(Vec<String> args) -> void
    start
        CaesarCipher cipher := new CaesarCipher();

        String x := "Kachow";
        String y := cipher.encrypt(&x, 95);
        String z := cipher.decrypt(&y, 95);
        // Kachow
        println(x);
        // BRTYFN
        println(y);
        // KACHOW
        println(z);
    finish main
finish model
```

Listing 5: CaesarCipher.rez (enhanced)

4.3 Factorial

```
// Define a model for the factorial program
model FactorialProgram
start
    // Return the result of factorial(num)
    ext fn factorial(&self, i32 num) -> i32
    start
        if num <= 0
        start
            // factorial(0) = 1
            // and anything < 0 is invalid, so return 1
            return 1;
```

```
        else
            // factorial(n) = n * factorial(n - 1)
            return num * self.factorial(num - 1);
        finish if
    finish factorial

    ext fn main(Vec<String> args) -> void
    start
        FactorialProgram fp := new FactorialProgram();
        // Should be 120
        println(fp.factorial(5));

        // Both should be 1
        println(fp.factorial(0));
        println(fp.factorial(-1));
    finish main
finish model
```

Listing 6: FatorialProgram.rez

4.4 Quicksort

```
// Import the Random model for use later
import std.util.Random;

// Define a model for the classic SortsAndShuffles file from algorithms
model SortsAndShuffles
start
    // This is the wrapper function for the quicksort implementation
    ext fn quicksort(&self, &mut Vec<i32> data) -> void
    start
        // Call quicksort on the entire vector
        self.quicksort_with_indices(data, 0, data.len() - 1);
    finish quicksort

    // Helper quicksort function
    // Private because only accessible within the model
    fn quicksort_with_indices(&self, &mut Vec<i32> data, i32 start_index, i32 end_index) ->
        void
    start
        // Recursion base case to end if we have an array of size 0 or 1
        if start_index >= end_index
        start
            return;
        finish if

        // Declare a pivot index that will be changed in a few lines
        mut i32 pivot_index := 0;

        if end_index - start_index < 3
        start
            // Pivot index is the start index because will need one more level
            // of recursion regardless of the pivot
            pivot_index := start_index;
        else
            // Otherwise declare a new random object
            Random ran := new Random();

            // Get the first pivot option
            i32 pivot_choice_1 := ran.randInt(start_index, end_index + 1);

            // Get the second pivot option, but only use it if different from
            // the first
```

```
41              mut i32 pivot_choice_2 := ran.randInt(start_index, end_index + 1);
42              while pivot_choice_2 == pivot_choice_1
43              start
44                  pivot_choice_2 := ran.randInt(start_index, end_index + 1);
45              finish while
46
47              // Get the third pivot option but only use it if different from
48              // both of the other options
49              mut pivot_choice_3 := ran.randInt(start_index, end_index + 1);
50              while pivot_choice_3 == pivot_choice_1 || pivot_choice_3 == pivot_choice_2
51              start
52                  pivot_choice_3 := ran.randInt(start_index, end_index + 1);
53              finish while
54
55              // Get the median of the pivots and set the appropriate pivot index
56              if data[pivot_choice_1] <= data[pivot_choice_2] && data[pivot_choice_1] >= data[
                    pivot_choice_3]
57              start
58                  pivot_index := pivot_choice_1;
59              else if data[pivot_choice_1] <= data[pivot_choice_3] && data[pivot_choice_1] >=
                    data[pivot_choice_2]
60                  pivot_index := pivot_choice_1;
61              else if data[pivot_choice_2] <= data[pivot_choice_1] && data[pivot_choice_2] >=
                    data[pivot_choice_3]
62                  pivot_index := pivot_choice_2;
63              else if data[pivot_choice_2] <= data[pivot_choice_3] && data[pivot_choice_2] >=
                    data[pivot_choice_1]
64                  pivot_index := pivot_choice_2;
65              else
66                  pivot_index := pivot_choice_3;
67              finish if
68          finish if
69
70          // Perform the partition
71          i32 partition_out := self.partition(data, start_index, end_index, pivot_index);
72
73          // Perform quicksort on the 2 sides of the partition
74          self.quicksort_with_indices(data, start_index, partition_out - 1);
75          self.quicksort_with_indices(data, partition_out + 1, end_index);
76
77      finish quicksort_with_indices
78
79      // Delare a private function for sorting the array that returns the index
80      // for the partition
81      fn partition(&self, &mut Vec<i32> data, i32 start_index, i32 end_index, i32 pivot_index)
            -> i32
82      start
83          // Move the pivot to the end of the array
84          i32 pivot := data[pivot_index];
85          data[pivot_index] := data[end];
86          data[end] := pivot;
87
88          // Keep track of where the low partition starts
89          mut i32 last_low_partition_index := start_index - 1;
90
91          // Go through the sub array, excluding the last index because the pivot
92          // value is in the end index
93          for mut i in range(start_index, end_index, 1)
94          start
95              if data[i] < pivot
96              start
97                  // Make space for the low value
98                  last_low_partition_index := last_low_partition_index + 1;
99
100                 // Move it to its new spot in the array
```

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

        // Move the pivot to the appropriate location
        data[end] := data[last_low_partition_index + 1];
        data[last_low_partiton_index + 1] := pivot;

        // Return the location of the pivot to distinguish the 2 partitions
        return last_low_partition_index + 1;
    finish partition

    // Knuth shuffle
    ext fn knuth_shuffle(&self, &mut Vec<i32> data) -> void
    start
        // Create the random object
        Random ran := new Random();

        // Iterate through the entire array
        for mut i32 i in range(0, data.len(), 1)
        start
            // Get a random index
            i32 swap_index := ran.randInt(0, data.len()));

            // Swap the 2 elements
            i32 temp := data[i];
            data[i] := data[swap_index];
            data[swap_index] := temp;
        finish for

    finish knuth_shuffle

    ext fn main(Vec<String> args) -> void
    start
        // Initialize a vector with initial capacity for 10 elements
        mut Vec<i32> my_arr := new Vec<i32>(10);
        // Fill the vector with values 0 - 9
        for mut i32 i in range(0, 10, 1)
        start
            my_arr[i] := i;
        finish for

        SortsAndShuffles sas := new SortsAndShuffles();

        // Shuffly the vector
        sas.knuth_shuffle(&mut my_arr);
        println(my_arr.to_string());

        // Sort the vector
        sas.quicksort(&mut my_arr);
        println(my_arr.to_string());
    finish main

finish model
```

Listing 7: SortsAndShuffles.rez

4.5 Selection Sort

```
// Import random for the Knuth shuffle defined in Section 4.4
import std.util.Random;
```

```
model SortsAndShuffles
start
    // Create a public function for doing a selection sort
    ext fn selection_sort(&self, &mut Vec<i32> data) -> void
        // Loop through all but the last element because an array of length 1
        // is already sorted
        for mut i32 i in range(0, data.len() - 1, 1)
        start
            // Assume first element is smallest
            // Can directly assign here because i32 is a value type, so the value
            // gets stored rather than the actual reference
            mut i32 smallest_index := i;

            // Go through the rest of the array
            for mut i32 j in range(i + 1, data.len(), 1)
            start
                // If we have a smaller element, save the new lower index
                if data[smallest_index] > data[j]
                start
                    smallest_index := j;
                finish if
            finish for

            // Move the smallest element in place
            i32 temp := data[i];
            data[i] := data[smallest_index];
            data[smallest_index] := temp;
        finish for
    finish selection_sort

    ext fn main(Vec<String> args) -> void
    start
        // From Section 4.4
        mut Vec<i32> my_arr := new Vec<i32>(10);
        for mut i32 i in range(0, 10, 1)
        start
            my_arr[i] := i;
        finish for

        SortsAndShuffles sas := new SortsAndShuffles();

        // Shuffle (from Section 4.4)
        sas.knuth_shuffle(&mut my_arr);
        println(my_arr.to_string());

        // Sort
        sas.selection_sort(&mut my_arr);
        println(my_arr.to_string());
    finish main
finish model
```

Listing 8: SortsAndShuffles.rez (enhanced)

4.6 Object-Oriented Programming With Transformers

```
model Owner
start
    specs
    start
        // Every owner has a name
        ext String name;
    finish specs
```

```
8
9       // Define a new owner with the given name
10      ext fn Owner(String my_name)
11      start
12          self.name := my_name;
13      finish Owner
14  finish model
```

Listing 9: Owner.rez

```
1   import garage.Owner;
2
3   model Transformer
4   start
5       specs
6       start
7           ext String name;
8           ext String car_name;
9           i32 power;
10          &Transformer leader;
11          &Owner owner;
12      finish specs
13
14      // Define a constructor for the Transformer
15      ext fn Transformer(String my_name, String my_car_name, i32 power)
16      start
17          self.name := my_name;
18          self.car_name := my_car_name;
19          self.power := power;
20
21          // Leader and owner have to be set by setters, so null at first
22          // Also follow the rule that all specs are initialized in the constructor
23          self.leader := null;
24          self.owner := null;
25      finish Transformer
26
27      // Returns the difference in power when attacking another transformer
28      // > 0 means win
29      // < 0 means loss
30      // == 0 means draw
31      ext fn attack(&self, &Transformer oponent) -> i32
32      start
33          mut i32 power_diff := self.power - oppenent.get_power();
34
35          &Transformer oppenent_leader := oppenent.get_leader();
36          if oppenent_leader != null
37          start
38              // We will say the leader helps out if their comrade is being attacked
39              power_diff := power_diff - oppenent_leader.get_power();
40          finish if
41
42          return power_diff;
43      finish attack
44
45      // Increases the power of the transformer by the factor
46      ext fn supercharge(&mut self, i32 factor)
47      start
48          self.power := self.power * factor;
49      finish supercharge
50
51      // Define a getter for the power spec
52      ext fn get_power(&self) -> i32
53      start
54          return self.power;
55      finish get_power
```

```
56
57      // Define a getter for the leader spec
58      ext fn get_leader(&self) -> &Transformer
59      start
60          return self.leader;
61      finish get_leader
62
63      // Define a setter for the leader
64      ext fn set_leader(&mut self, &Transformer new_leader)
65      start
66          self.leader := new_leader;
67      finish set_leader
68
69      // Define a setter for the owner
70      ext fn set_owner(&mut self, &Owner new_owner)
71      start
72          self.owner := new_owner;
73      finish set_owner
74  finish model
```

Listing 10: Transformer.rez

```
1   import garage.Transformer;
2
3   // Autobot is a subclass/model of Transformer
4   model Autobot extends Transformer
5   start
6       ext fn Autobot(String my_name, String my_car_name, i32 power)
7       start
8           // All specs are initialized in the super constructor, so nothing
9           // needs to be explicitly initialized here
10          super(my_name, my_car_name, power);
11      finish Autobot
12
13      // Function to "roll out"
14      ext fn roll_out(&self)
15      start
16          println("Roll␣out:␣" + self.name);
17      finish roll_out
18  finish model
```

Listing 11: Autobot.rez

```
1   import garage.Owner;
2   import garage.Transformer;
3   import garage.Autobot;
4
5   model MainTransformers
6   start
7       ext fn main(Vec<String> args) -> void
8       start
9           // Create a new autobot for Bumblebee
10          mut Autobot bumblebee := new Autobot("Bumblebee", "Chevy␣Camaro", 500);
11
12          // Give Bumblebee an owner
13          Owner sam := new Owner("Sam␣Witwicky");
14          // This method is accessible because Autobot extends Transformer
15          bumblebee.set_owner(&sam);
16
17          Autobot optimus_prime := new Autobot("Optimus␣Prime", "Truck", 1000);
18          // Give a reference to Optimus Prime for Bumblebee's leader
19          bumblebee.set_leader(&optimus_prime);
20
21          // Create Megatron
```

```
22          Transformer megatron := new Transformer("Megatron", "Tank", 2000);
23
24          // This will be 500 because Bumblebee gets assistance from Optimus Prime
25          // but their combined power is too low
26          println(megatron.attack(&bumblebee));
27
28          // Should be -1000 because the difference in their power is 1000 and
29          // Megatron has no leader
30          println(optimus_prime.attack(&megatron));
31
32          // Multiply Bumblebee's power by a factor of 3
33          bumblebee.supercharge(3);
34
35          // This will be -500 because Bumblebee is now at 1500 power
36          // plus Optimus Prime's 1000 power
37          println(megatron.attack(&bumblebee));
38
39          // Optimus Prime has the function because he is an Autobot
40          // Megatron does not because it is only defined in the Autobot model
41          optimus_prime.roll_out();
42      finish main
43  finish model
```

Listing 12: MainTransformers.rez