Rust-eze

CMPT 331 - Spring 2023 | Dr. Labouseur

Josh Seligman | joshua.seligman1@marist.edu

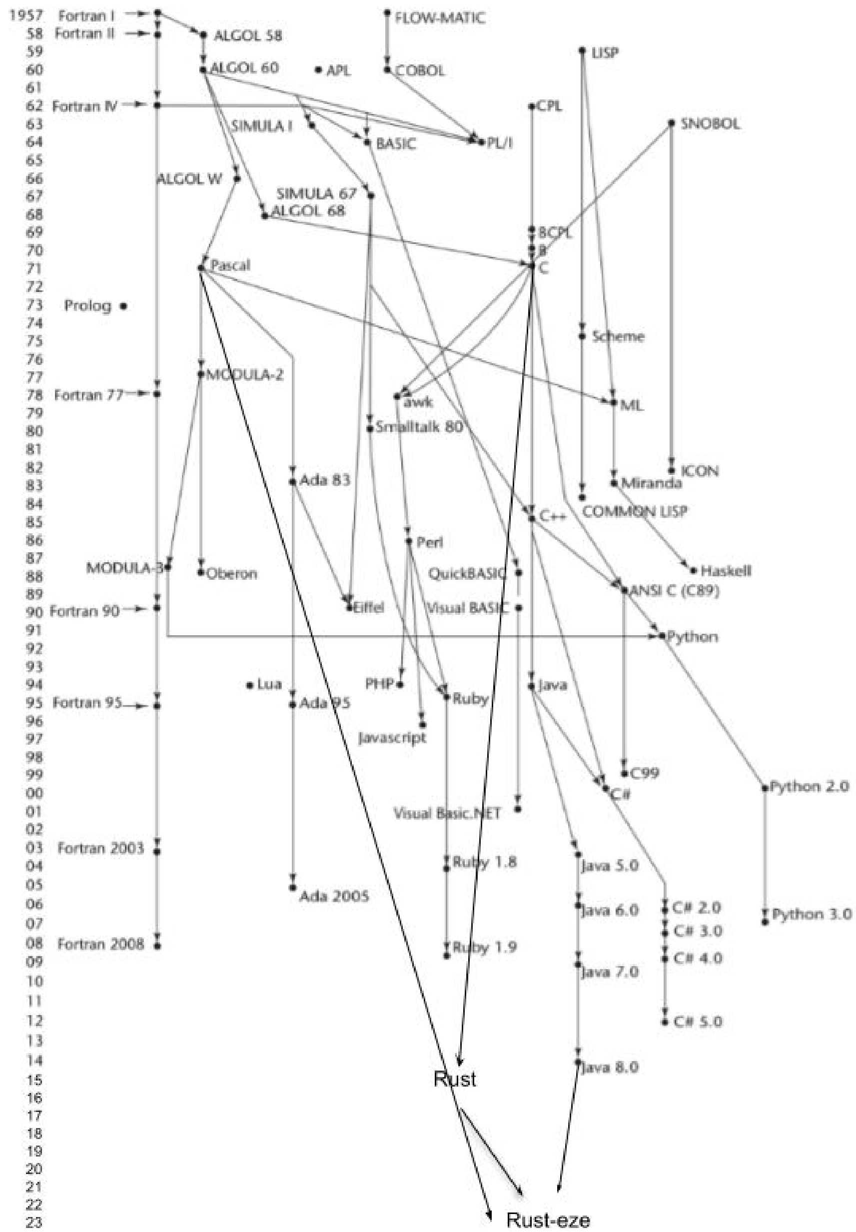
May 10, 2023

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

```

```
    pub fn say_it(&self) -> void
    start
        println("Kachow!");
    finish say_it
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.

```
import garage.Lightning;

model Main
start
    pub fn main(Vec<String> args) -> void
    start
        mut Lightning the_lightning := new Lightning("McQueen", 17);
        println(the_lightning.name);

        the_lightning.say_it();
        the_lightning.drive(42);
    finish main
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;`

<table>
<tr><td>If statement</td><td><pre>
i32 x := 7;
if x > 3 && x < 9
start
    println("Hello there")
else
    println("Kachow")
finish if
</pre></td></tr>
<tr><td>For loop</td><td><pre>
for (mut i32 i in range(0, 10, 1))
start
    println(i)
finish for
</pre></td></tr>
</table>

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> <parent declaration> <block begin> <statements> <block end> <model name>
<parent declaration> → extends <parent model name> | ϵ
<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, 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:

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

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

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

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

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

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

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

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
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 definition for the Caesar cipher
model CaesarCipher
start
    // Encryt a string based on the shift amount
    pub 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
            // (i8) casts the expression on the right of it to be an i8
            mut i8 cur_char := (i8) 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

    pub 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
    pub 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

    pub 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)
    pub 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

    pub 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
    pub 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
            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

            // Get the third pivot option but only use it if different from
            // both of the other options
            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

            // Get the median of the pivots and set the appropriate pivot index
            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

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

        // Perform quicksort on the 2 sides of the partition
        self.quicksort_with_indices(data, start, partition_out - 1);
```

```
        self.quicksort_with_indices(data, partition_out + 1, end);

    finish quicksort_with_indices

    // Delare a private function for sorting the array that returns the index
    // for the partition
    fn partition(&self, &mut Vec<i32> data, i32 start, i32 end, i32 pivot_index) -> i32
    start
        // Move the pivot to the end of the array
        i32 pivot := *data[pivot_index];
        *data[pivot_index] := *data[end];
        *data[end] := pivot;

        // Keep track of where the low partition starts
        mut i32 last_low_partition_index := start - 1;

        // Go through the sub array, excluding the last index because the pivot
        // value is in the end index
        for mut i in range(start, end, 1)
        start
            if *data[i] < pivot
            start
                // Make space for the low value
                last_low_partition_index := last_low_partition_index + 1;

                // 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
    pub 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

    pub 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
    pub 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

    pub 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
        pub String name;
    finish specs

    // Define a new owner with the given name
    pub fn Owner(String my_name)
    start
        self.name := my_name;
    finish Owner
finish model
```

Listing 9: Owner.rez

```
import garage.Owner;

model Transformer
start
    specs
    start
        pub String name;
        pub String car_name;
        i32 power;
        &Transformer leader;
        &Owner owner;
    finish specs

    // Define a constructor for the Transformer
    pub fn Transformer(String my_name, String my_car_name, i32 power)
    start
        self.name := my_name;
        self.car_name := my_car_name;
        self.power := power;

        // Leader and owner have to be set by setters, so null at first
        // Also follow the rule that all specs are initialized in the constructor
        self.leader := null;
        self.owner := null;
    finish Transformer

    // Returns the difference in power when attacking another transformer
    // > 0 means win
```

```
    // < 0 means loss
    // == 0 means draw
    pub fn attack(&self, &Transformer oponent) -> i32
    start
        mut i32 power_diff := self.power - *oppenent.get_power();

        &Transformer oppenent_leader := *oppenent.get_leader();
        if *oppenent_leader != null
        start
            // We will say the leader helps out if their comrade is being attacked
            power_diff := power_diff - *oppenent_leader.get_power();
        finish if

        return power_diff;
    finish attack

    // Increases the power of the transformer by the factor
    pub fn supercharge(&mut self, i32 factor)
    start
        self.power := self.power * factor;
    finish supercharge

    // Define a getter for the power spec
    pub fn get_power(&self) -> i32
    start
        return self.power;
    finish get_power

    // Define a getter for the leader spec
    pub fn get_leader(&self) -> &Transformer
    start
        return self.leader;
    finish get_leader

    // Define a setter for the leader
    pub fn set_leader(&mut self, &Transformer new_leader)
    start
        self.leader := new_leader;
    finish set_leader

    // Define a setter for the owner
    pub fn set_owner(&mut self, &Owner new_owner)
    start
        self.owner := new_owner;
    finish set_owner
finish model
```

Listing 10: Transformer.rez

```
import garage.Transformer;

// Autobot is a subclass/model of Transformer
model Autobot extends Transformer
start
    pub fn Autobot(String my_name, String my_car_name, i32 power)
    start
        // All specs are initialized in the super constructor, so nothing
        // needs to be explicitly initialized here
        self.super(my_name, my_car_name, power);
    finish Autobot

    // Function to "roll out"
    pub fn roll_out(&self)
    start
        println("Roll out: " + self.name);
```

```
    finish roll_out
finish model
```

Listing 11: Autobot.rez

```
import garage.Owner;
import garage.Transformer;
import garage.Autobot;

model MainTransformers
start
    pub fn main(Vec<String> args) -> void
    start
        // Create a new autobot for Bumblebee
        mut Autobot bumblebee := new Autobot("Bumblebee", "Chevy Camaro", 500);

        // Give Bumblebee an owner
        Owner sam := new Owner("Sam Witwicky");
        // This method is accessible because Autobot extends Transformer
        bumblebee.set_owner(&sam);

        Autobot optimus_prime := new Autobot("Optimus Prime", "Truck", 1000);
        // Give a reference to Optimus Prime for Bumblebee's leader
        bumblebee.set_leader(&optimus_prime);

        // Create Megatron
        Transformer megatron := new Transformer("Megatron", "Tank", 2000);

        // This will be 500 because Bumblebee gets assistance from Optimus Prime
        // but their combined power is too low
        println(megatron.attack(&bumblebee));

        // Should be -1000 because the difference in their power is 1000 and
        // Megatron has no leader
        println(optimus_prime.attack(&megatron));

        // Multiply Bumblebee's power by a factor of 3
        bumblebee.supercharge(3);

        // This will be -500 because Bumblebee is now at 1500 power
        // plus Optimus Prime's 1000 power
        println(megatron.attack(&bumblebee));

        // Optimus Prime has the function because he is an Autobot
        // Megatron does not because it is only defined in the Autobot model
        optimus_prime.roll_out();
    finish main
finish model
```

Listing 12: MainTransformers.rez