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 langue 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 and enums.

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.
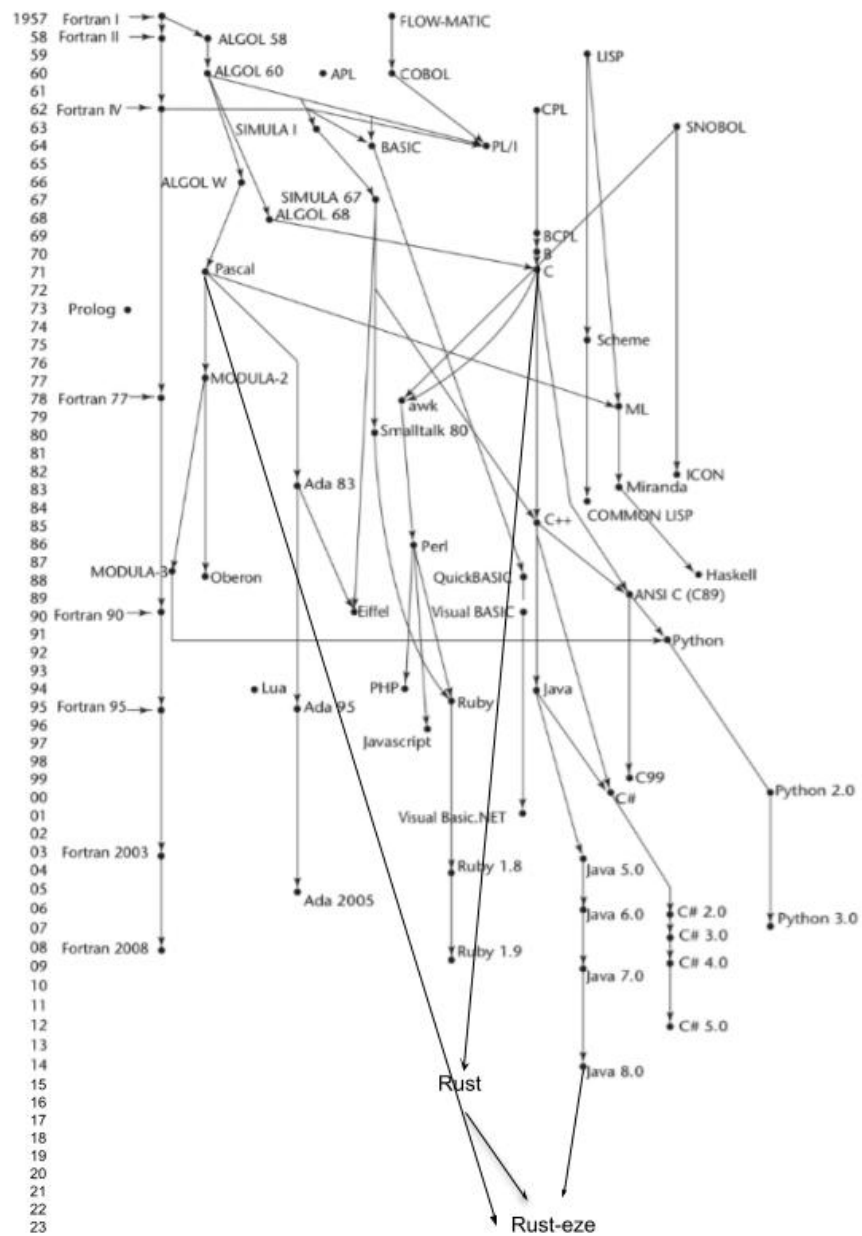
1.2 Hello World

```
1  model HelloWorld
2  start
3      ext fn main(Vec<String> args) -> void
4      start
5          println("Hello_world!");
6      finish main
7  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**.

```
1  model Lightning
2  start
3      specs
4      start
5          ext String name;
6          ext i32 age;
7          i32 miles;
8      finish specs
9
10     ext fn Lightning(String my_name, i32 my_age)
11     start
12         self.name := my_name;
13         self.age := my_age;
14         self.miles := 0;
15     finish Lightning
16
17     ext fn drive(&mut self, i32 num_miles) -> void
18     start
19         i32 new_mileage := self.miles + num_miles;
20         self.miles := new_mileage;
21     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

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:

$S \vdash e1 : T$
$S \vdash e2 : T$
T is a primitive type
────────────────────
$S \vdash e1 := e2 : T$

$S \vdash e1 : T$
$S \vdash e2 : T$
T is a primitive type
────────────────────
$S \vdash e1 == e2 : bool$

$S \vdash e1 : T$
$S \vdash e2 : T$
T is a primitive type
────────────────────
$S \vdash e1 != e2 : bool$

$S \vdash e1 : T$
$S \vdash e2 : T$
T is a numeric primitive type
────────────────────
$S \vdash e1 > e2 : bool$

$S \vdash e1 : T$
$S \vdash e2 : T$
T is a numeric primitive type
────────────────────
$S \vdash e1 < e2 : bool$

$S \vdash e1 : T$
$S \vdash e2 : T$
T is a numeric primitive type
────────────────────
$S \vdash e1 + e2 : T$

S ⊢ e1 : String
S ⊢ e2 : T with a String representation
————————————————————————————
S ⊢ 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: ```\nString x := "hello";\n```
Vec<T>	A dynamic, homogeneous array of values of type T, which can be any type. Example: ```\nVec<i32> nums1 := [0, 0, 7];\nmut Vec<i32> nums2 := new Vec<i32>(2);\nnums2[0] := 42;\nnums2[1] := 99;\n```
Tuple	A collection of values of different types that is fixed in size. Example: ```\nTuple<i32, bool, i32> my_tuple := (7,\n false, 3);\nmut Tuple<String, i32> the_tuple :=\n new Tuple<String, i32>();\nthe_tuple.0 := "jOSh";\nthe_tuple.1 := 2;\n```

4 Example Programs

4.1 Caesar Cipher Encrypt

```
1  // Model definition for the Caesar cipher
2  model CaesarCipher
3  start
4      // Encryt a string based on the shift amount
5      ext fn encrypt(&self, &String in_str, i32 shift_amt) -> String
6      start
7          // Get the real shift amount and create a new vector of characters
8          i32 real_shift := shift_amt % 26;
9          Vec<char> out_vec := new Vec<char>();
10
11         // Loop through the entire string
12         for mut i32 i in range(0, in_str.len(), 1)
13         start
14
15             // Get the character at the given index as an integer
16             // (u8) casts the expression on the right of it to be an u8
17             mut u8 cur_char := (u8) in_str.char_at(i);
18
19             if cur_char >= 97 && cur_char <= 122
20             start
21                 // Convert the lowercase letters to uppercase letters
22                 cur_char := cur_char - 32;
23             finish if
24
25             // Only modify uppercase letters
26             if cur_char >= 65 && cur_char <= 90
27             start
28                 // Perform the shift
29                 cur_char := cur_char + real_shift;
30
31                 // This is the difference for wraparound for Z
32                 mut i32 diff := cur_char - 90;
33                 if diff > 0
34                 start
35                     // Perform the Z wraparound
36                     cur_char := 65 + diff - 1;
37                 else
38                     // Now compute the A wraparound if there was no Z wraparound
39                     diff := 65 - cur_char;
40
41                     if diff > 0
42                     start
43                         cur_char := 90 - diff + 1;
44                     finish if
45                 finish if
46             finish if
47
48             // Get the final character as the appropriate type
49             char final_char := (char) cur_char;
50
51             // Push it to the end of the vector (vector will resize as needed)
52             out_vec.push(final_char);
53         finish for
54
55         // Join the elements of the vector together to form a string
56         // by using the string representation of each of the elements
57         return out_vec.join("");
58     finish encrypt
59
60     ext fn main(Vec<String> args) -> void
```

```
61      start
62          CaesarCipher cipher := new CaesarCipher();
63
64          // Create a new string
65          String x := "Kachow";
66
67          // Perform encrypt and pass an immutable reference to x
68          String y := cipher.encrypt(&x, 95);
69
70          // Should print Kachow
71          println(x);
72          // Should print BRTYFN
73          println(y);
74      finish main
75  finish model
```

Listing 4: CaesarCipher.rez

4.2 CAESAR CIPHER DECRYPT

```
1  model CaesarCipher
2  start
3      // Returns a decrypted string based on the shift amount
4      ext fn decrypt(&self, &String in_str, i32 shift_amt) -> String
5      start
6          // Decrypt is the same as encrypt with negative shift amount
7          // Encrypt is defined in Section 4.1
8          return self.encrypt(in_str, -shift_amt);
9      finish encrypt
10
11      ext fn main(Vec<String> args) -> void
12      start
13          CaesarCipher cipher := new CaesarCipher();
14
15          String x := "Kachow";
16          String y := cipher.encrypt(&x, 95);
17          String z := cipher.decrypt(&y, 95);
18          // Kachow
19          println(x);
20          // BRTYFN
21          println(y);
22          // KACHOW
23          println(z);
24      finish main
25  finish model
```

Listing 5: CaesarCipher.rez (enhanced)

4.3 FACTORIAL

```
1  // Define a model for the factorial program
2  model FactorialProgram
3  start
4      // Return the result of factorial(num)
5      ext fn factorial(&self, i32 num) -> i32
6      start
7          if num <= 0
8          start
9              // factorial(0) = 1
10             // and anything < 0 is invalid, so return 1
11             return 1;
```

```
12          else
13              // factorial(n) = n * factorial(n - 1)
14              return num * self.factorial(num - 1);
15          finish if
16      finish factorial
17
18      ext fn main(Vec<String> args) -> void
19      start
20          FactorialProgram fp := new FactorialProgram();
21          // Should be 120
22          println(fp.factorial(5));
23
24          // Both should be 1
25          println(fp.factorial(0));
26          println(fp.factorial(-1));
27      finish main
28  finish model
```

Listing 6: FatorialProgram.rez

4.4 QUICKSORT

```
1   // Import the Random model for use later
2   import std.util.Random;
3
4   // Define a model for the classic SortsAndShuffles file from algorithms
5   model SortsAndShuffles
6   start
7       // This is the wrapper function for the quicksort implementation
8       ext fn quicksort(&self, &mut Vec<i32> data) -> void
9       start
10          // Call quicksort on the entire vector
11          self.quicksort_with_indices(data, 0, data.len() - 1);
12      finish quicksort
13
14      // Helper quicksort function
15      // Private because only accessible within the model
16      fn quicksort_with_indices(&self, &mut Vec<i32> data, i32 start_index, i32 end_index) ->
              void
17      start
18          // Recursion base case to end if we have an array of size 0 or 1
19          if start_index >= end_index
20          start
21              return;
22          finish if
23
24          // Declare a pivot index that will be changed in a few lines
25          mut i32 pivot_index := 0;
26
27          if end_index - start_index < 3
28          start
29              // Pivot index is the start index because will need one more level
30              // of recursion regardless of the pivot
31              pivot_index := start_index;
32          else
33              // Otherwise declare a new random object
34              Random ran := new Random();
35
36              // Get the first pivot option
37              i32 pivot_choice_1 := ran.randInt(start_index, end_index + 1);
38
39              // Get the second pivot option, but only use it if different from
40              // 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_index, end_index, pivot_index);

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

    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_index, i32 end_index, 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_index - 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_index, end_index, 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
```

```
101                    i32 temp := *data[i];
102                    *data[i] := *data[last_low_partition_index];
103                    *data[last_low_partition_index] := temp;
104                finish if
105            finish for
106
107            // Move the pivot to the appropriate location
108            *data[end] := *data[last_low_partition_index + 1];
109            *data[last_low_partiton_index + 1] := pivot;
110
111            // Return the location of the pivot to distinguish the 2 partitions
112            return last_low_partition_index + 1;
113        finish partition
114
115        // Knuth shuffle
116        ext fn knuth_shuffle(&self, &mut Vec<i32> data) -> void
117        start
118            // Create the random object
119            Random ran := new Random();
120
121            // Iterate through the entire array
122            for mut i32 i in range(0, *data.len(), 1)
123            start
124                // Get a random index
125                i32 swap_index := ran.randInt(0, *data.len()));
126
127                // Swap the 2 elements
128                i32 temp := *data[i];
129                *data[i] := *data[swap_index];
130                *data[swap_index] := temp;
131            finish for
132
133        finish knuth_shuffle
134
135        ext fn main(Vec<String> args) -> void
136        start
137            // Initialize a vector with initial capacity for 10 elements
138            mut Vec<i32> my_arr := new Vec<i32>(10);
139            // Fill the vector with values 0 - 9
140            for mut i32 i in range(0, 10, 1)
141            start
142                my_arr[i] := i;
143            finish for
144
145            SortsAndShuffles sas := new SortsAndShuffles();
146
147            // Shuffly the vector
148            sas.knuth_shuffle(&mut my_arr);
149            println(my_arr.to_string());
150
151            // Sort the vector
152            sas.quicksort(&mut my_arr);
153            println(my_arr.to_string());
154        finish main
155
156 finish model
```

Listing 7: SortsAndShuffles.rez

4.5 SELECTION SORT

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

```
 3
 4  model SortsAndShuffles
 5  start
 6      // Create a public function for doing a selection sort
 7      ext fn selection_sort(&self, &mut Vec<i32> data>) -> void
 8          // Loop through all but the last element because an array of length 1
 9          // is already sorted
10          for mut i32 i in range(0, *data.len() - 1, 1)
11          start
12              // Assume first element is smallest
13              // Can directly assign here because i32 is a value type, so the value
14              // gets stored rather than the actual reference
15              mut i32 smallest_index := i;
16
17              // Go through the rest of the array
18              for mut i32 j in range(i + 1, *data.len(), 1)
19              start
20                  // If we have a smaller element, save the new lower index
21                  if *data[smallest_index] > *data[j]
22                  start
23                      smallest_index := j;
24                  finish if
25              finish for
26
27              // Move the smallest element in place
28              i32 temp := *data[i];
29              *data[i] := *data[smallest_index];
30              *data[smallest_index] := temp;
31          finish for
32      finish selection_sort
33
34      ext fn main(Vec<String> args) -> void
35      start
36          // From Section 4.4
37          mut Vec<i32> my_arr := new Vec<i32>(10);
38          for mut i32 i in range(0, 10, 1)
39          start
40              my_arr[i] := i;
41          finish for
42
43          SortsAndShuffles sas := new SortsAndShuffles();
44
45          // Shuffle (from Section 4.4)
46          sas.knuth_shuffle(&mut my_arr);
47          println(my_arr.to_string());
48
49          // Sort
50          sas.selection_sort(&mut my_arr);
51          println(my_arr.to_string());
52      finish main
53  finish model
```

Listing 8: SortsAndShuffles.rez (enhanced)

4.6 OBJECT-ORIENTED PROGRAMMING WITH TRANSFORMERS

```
1  model Owner
2  start
3      specs
4      start
5          // Every owner has a name
6          ext String name;
7      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          self.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