César Liesens - 39161400
Nima Farnoodian - 68372000

LINGI2132
Languages & Translators
Project - Part III

13/05/2021

1 The Rime Programming Language 3.0

Rime has, once again, evolved quite a bit since we've submitted part 2 of the project. We were not satisfied with what we had produced, as some features (mainly related to dictionaries) weren't working as intended or hadn't been implemented fully. We then decided to rewrite our `SemanticAnalysis` class from scratch before starting to write the interpreter, and made various changes to Rime along the way. The aim of this section is to give a brief summary of all those changes.

First, we've removed all shorthand assignment operators (e.g. `+=`); the language is still statically typed, and variables must still be assigned when they are declared. The entry point's signature is now `proc main([string] _args_)`, and the list of command line arguments, `_args_`, can be accessed from anywhere within a Rime program. Functions can now only be defined before the program's entry point (the `main` function). The previously built-in functions for creating empty lists, sets & dicts have been removed as they were redundant. Finally, a new statement, `pass`, has been added to the language to represent empty statements.

Additionally, all of the required features are working as intended (as far as we know and have tested), as well as the extra features we've implemented, and so are as the five test programs whose outputs are compared with their Java counterparts' in `rime.tests.interpreter.ExampleProgramTests` for arbitrary arguments (though they can be quite slow, especially the `Sort` program).

2 Language Features

2.1 Literals, Comments & Variable Definitions

```
val bool: boolVar = TRUE        // single assignment boolean variable
val int: intVar = -25           // single assignment integer variable
var string: stringVar = "abc"   // reassignable string variable

stringVar = "def"
```

```
/*
 * the code below is not supported by the language,
 * variables must be assigned when they are declared
 */

var int: x
x = 25
```

2.2 Expressions & Operators

```
// See 'rime.tests.interpreter.InterpreterTests.test_booleans_1()'
val int: x = 25
val int: y = 75
val bool: lessThan50 = (x < 50 && y < 50)
val bool: moreThan50 = !lessThan50                  // true
```

```
// See 'rime.tests.interpreter.InterpreterTests.test_arithmetic()'
val int: x = 11 % 6
val int: y = (x + 2 * 5) / 3
val int: z = -y * 5                                    // 25
```

```
// See 'rime.tests.interpreter.InterpreterTests.test_booleans_2()'
val int: x = -5
val int: y = 5
val bool: atLeastOnePositive = (x >= 1 || y >= 1)    // true
```

2.3 Lists

```
// See 'rime.tests.interpreter.InterpreterTests.test_lists_4()'
val [int]: numbers = [1, 2, 3, 4, 5]
numbers[2] = 99                  // numbers : [1, 2, 99, 4, 5]
```

```
// See 'rime.tests.interpreter.InterpreterTests.test_lists_5()'
var [int]: numbers = [int]
numbers = append(numbers, 1)     // numbers : [1]
numbers = append(numbers, 2)     // numbers : [1, 2]
numbers = append(numbers, 3)     // numbers : [1, 2, 3]
```

2.4 Dicts

```
// See 'rime.tests.interpreter.InterpreterTests.test_dicts_1()'
val {string:int}: ageByName = {"Tam": 12, "Tem": 26, "Tim": 30}
```

```
// See 'rime.tests.interpreter.InterpreterTests.test_dicts_2()'
val {string:int}: ageByName = {string:int}
ageByName["Tom"] = 15        // ageByName : {"Tom": 15}
ageByName["Tum"] = 44        // ageByName : {"Tom": 15, "Tum": 44}
ageByName["Tym"] = 22        // ageByName : {"Tom": 15, "Tum": 44, "Tym": 22}
```

2.5 Function Declarations and Calls

```
// See 'rime.tests.interpreter.InterpreterTests.test_helloWorld_2()'
proc sayHello() {
    print("Hello, World")
    exit
}
```

```
// See 'rime.tests.interpreter.InterpreterTests.test_functions_1()'
// Takes an integer parameter 'x' and returns the square of 'x'
func int square(int: x) {
    return x * x
}
```

```
func int sum(int: x, int: y) {
    return x + y
}

print(sum(5, 15)) // 20
```

```
proc f(bool: someCondition) {
    if (someCondition) { print("Hello!") }
    else { pass }
}
```

2.6 Control Statements

```
var int: i = 0

while (i < 5) {
    i = i + 1
} // i : 5
```

```
val int: x = -25
var int: y = 0

if (x >= 25) {
    y = +1
}
else {
    y = -1
} // y : -1
```

2.7 Command Line Arguments, Entry Point and Built-in Functions — `print` & `parseInt`

```
// Every valid Rime program must contain this declaration,
// and all functions must be defined beforehand
proc main([string] _args_) {
    // convert the first command line argument to an integer and print it
    // 'print' will try to convert its argument to a string before printing
    print(parseInt(_args_[0]))
}
```

3 Extra Language Features

3.1 String Concatenation

```
val string: x = "abc"
val string: y = "def"
val string: z = x + y       // z : "abcdef"
```

3.2 Sets

```
val {int}: numbers = {1, 2, 3, 4, 5}
```

```
var {int}: numbers = {int}
numbers = add(numbers, 1)   // numbers : {1}
numbers = add(numbers, 1)   // numbers : {1}
numbers = add(numbers, 1)   // numbers : {1}
numbers = add(numbers, 2)   // numbers : {1, 2}
numbers = add(numbers, 3)   // numbers : {1, 2, 3}
```

3.3 Single & Multiple Assignment Variables

```
var int: x = 100
x = 50                                  // this is fine
```

```
val int: x = 100
x = 50                                  // semantic analysis would find an error
```

3.4 Other Built-in Functions

```
var [int]: numbers = [1, 2, 3, 4, 5]

print(length(numbers))                  // 5
numbers = append(numbers, 99)           // numbers : [1, 2, 3, 4, 5, 99]
print(length(numbers))                  // 6
```

```
var {int}: numbers = {1, 2, 3, 4, 5}

print(contains(numbers, 99))            // false
numbers = add(numbers, 99)              // numbers : {1, 2, 3, 4, 5, 99}
print(contains(numbers, 99))            // true
```

4 Implementation Details & Language Semantics

The table below shows an overview of the main Rime types and their corresponding Java implementation.

Rime type	Java type
`int`	`Integer`
`bool`	`Boolean`
`string`	`String`
`{T}` — `set`	`HashSet<T>`
`[T]` — `list`	`ArrayList<T>`
`{K:V}` — `dict`	`HashMap<K, V>`

Table 1: Rime types and the type used in their Java implementation.

The semantics of Rime is, in our opinion at least, rather simple and intuitive. First, variables cannot be declared and defined separately; `var` variables can of course be reassigned after their definition, but they must always be given an initial value when declared. There is thus no need to worry about default values and their complications. Additionally, assignments are considered statements, not expressions, and therefore have no value.

Because of the somewhat hacky and inelegant way in which the command line arguments, `_args_`, were implemented, they are part of the root scope and are thus available from anywhere in a Rime program. Functions and blocks introduce their own scope, and function parameters cannot be reassigned and trying to do so will result in an error during semantic analysis.

Finally, we will end this section with a succinct list of operations that would result in errors during the semantic analysis process. Empty return statements, `returns` inside `proc` definitions and `exits` inside `func` definitions; binary and unary expressions with mismatched operand and operator types; mismatched variable and expression types in assignments; accessing array elements with non-integer

values; accessing a dictionary value with a wrongly typed key; referring to undefined variables or functions; collection initializations with mismatched types; functions calls with wrong number of arguments (or mismatched types).