César Liesens - 39161400
Nima Farnoodian - 68372000

LINGI2132
Languages & Translators
Project - Part I

12/03/2021

1 Introduction

Welcome to the Rime programming language! This language took inspiration from various other languages, such as Java, Javascript, Scala, C#, Python, Oz... Currently, the range of possibilities offered by Rime is rather bare-bones, though we hope to develop it further in the future. Moreover, many if not most of the features discussed here are not fully enforced by the parser (or even at all). They of course require semantic analysis to be functional and implemented.

2 The Rime Programming Language

This section covers an exhaustive list of all the currently available features of the language, the first of which is dynamic typing. Indeed, why bother declaring and explicitly mentioning types when the compile can do it for you? The primitive types currently supported are `int` for 32-bit signed integers, `bool` for booleans (`TRUE` or `FALSE`) and `string` for character strings (delimited by double quotes). In addition, the type `void` is part of the language, though it serves little use at the moment; there is also a `NULL` literal, used to mark the absence of a value.

The language supports all the usual arithmetic (`+ - * / %`), comparison (`> >= < <=`), equality (`== !=`) and other boolean operators (`&& ||`), as well as two unary operators (`- !`). All of these should hopefully be self-explanatory.

Identifiers (for both functions and variables) can start with an underscore or any letter, followed by a sequence of letters, digits or underscores.

For comments, we chose the simple route of the well-known C-style comments, i.e. single-line comments are marked by `//`, and multi-line comments delimited with `/* ... */`.

Rime uses the well known = binary operator for assignment, with support for shorthand assignments through the `+= -= *= /= %=` operators.

Onto more interesting things! As we felt that languages with dynamic typing sometimes led to problems caused by reassigning variables, and losing track of types because of poorly named identifiers and other issues, we decided to introduce two keywords to differentiate variable declaration and definition (which can be done separately). The keywords `val` and `var` respectively allow defining a single-assignment or a multiple-assignment variable. We thought that having two distinct, short keywords would be better than forcing one default behavior and having a single keyword (such as `const`) to denote single- or multiple-assignment variables. This means that the choice is only up to you, whether you want to go for a functional approach and use `val` everywhere you can, or go the simpler but potentially more error-prone `var` way.

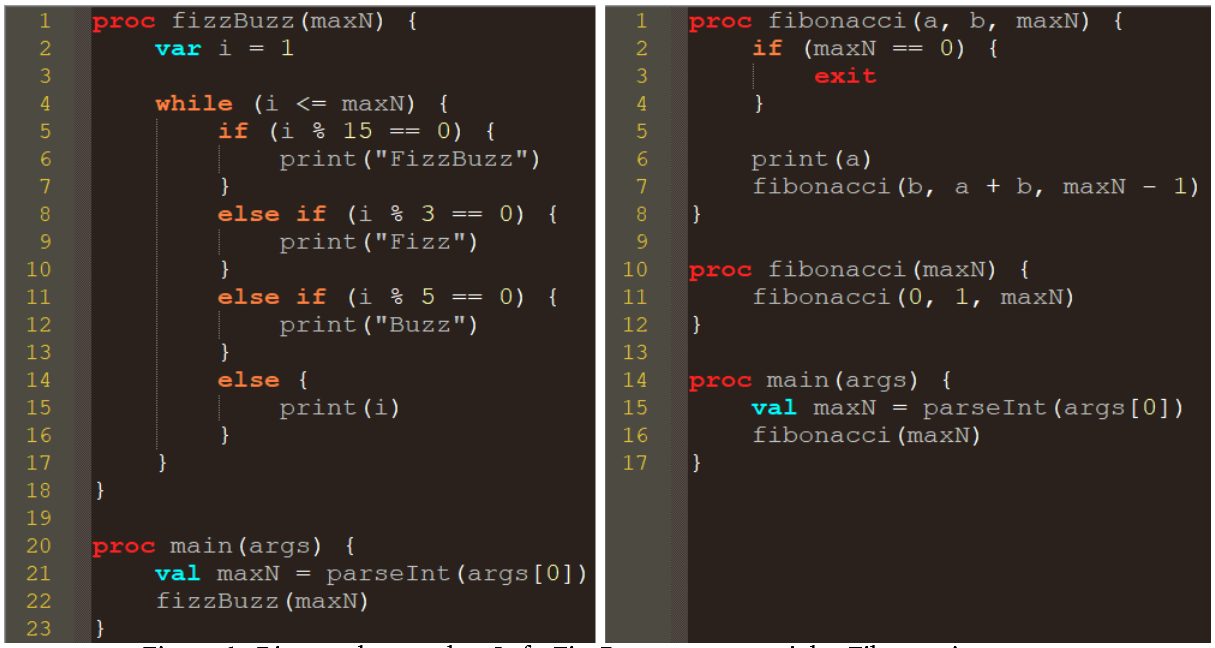
Functions can be defined using two keywords, `proc` and `func`, respectively used for procedures and functions (i.e. procedures that return a value); some examples are provided further below. Function calls are done simply by writing an identifier followed by a list of comma-separated arguments inside parentheses.

When it comes to control statements, Rime supports the usual C-style `while` loop, `if`, `else` and `else if` statements. All four of those require surrounding their nested code block to be surrounded by braces, though they can be written on a single line. The `exit` and `return` keywords can be used respectively inside a `proc` or `func`, to simply exit the procedure execution or return a value from a function. Furthermore, for a Rime program to be valid, it must contain an entry point, which is defined as a function with signature `proc main(args)`. One important thing to note is that, currently, no statements (conditionals, loops or functions definitions) can have an empty body. This may or may not change in the future, we are not totally decided on this yet.

The language will have native support for arrays, sets and maps. They can be created using the one of the three supported initializers, `array = [1, 2, 3]`, `set = {1, 2, 3}` or `map = {1:1, 2:2, 3:3}`, or with an initial size of zero with the use of `[]`, `set()` or `map()` respectively. Map and array access are done by simply writing an identifier followed by an expression inside square brackets.

The two functions (mentioned in the assignment description) `print` and `parseInt` are used within the code examples, but weren't defined in the grammar as they would only be calls to Java's `System.out.println` or `Integer.parseInt`, and will be added in the next part.

Finally, statements and code lines do not need to be suffixed by any special character or token. We recommend using `camelCase` for all identifiers, and ending your files with the `.rime` extension. The two code samples below should be easy to understand to anyone familiar with C-family languages, and give a better idea of what Rime programs look like.



```
proc fizzBuzz(maxN) {
    var i = 1

    while (i <= maxN) {
        if (i % 15 == 0) {
            print("FizzBuzz")
        }
        else if (i % 3 == 0) {
            print("Fizz")
        }
        else if (i % 5 == 0) {
            print("Buzz")
        }
        else {
            print(i)
        }
    }
}

proc main(args) {
    val maxN = parseInt(args[0])
    fizzBuzz(maxN)
}
```

```
proc fibonacci(a, b, maxN) {
    if (maxN == 0) {
        exit
    }

    print(a)
    fibonacci(b, a + b, maxN - 1)
}

proc fibonacci(maxN) {
    fibonacci(0, 1, maxN)
}

proc main(args) {
    val maxN = parseInt(args[0])
    fibonacci(maxN)
}
```

Figure 1: Rime code samples. Left, FizzBuzz program; right, Fibonacci program.

3 The Future

Overall, this first part of the project was quite engaging (maybe even *fun*?) and very time-consuming. There are many additional features we would like to implement in our language, though this will mostly depend on how difficult and, once again, time-consuming the semantic analysis part is. Many of those features come from the realm of functional programming; higher-order functions, lambda expressions, support for `map`, `filter` & `reduce/fold` operations... We would also like to introduce floating point numbers into the language, as well as more functionality for working with collections such as lists natively, similarly to the Oz language.