University of Puerto Rico

Mayagüez Campus

College of Engineering

Department of Electrical and Computer Engineering

Bachelor of Science in Computer Engineering

TestIt!

Michael Delgado

David Carrión

Estefanía Torres

Christopher Vegerano

CIIC 4030-096

Dr. Wilson Rivera

May / 18 / 2019

Introduction

Software developers are constantly testing programs for a right implementation that satisfies their needs. The unit testers are one great tool for testing programs in various programming languages. These tools provide the programmer with fast, precise and useful code examination. Creating these testers can be a challenge for newcomers in the field of programming. Therefore, our goal is to develop a simple and useful implementation for testing. Our language will facilitate the elaboration of custom testers by any beginner or experienced programmer. Developing testers for methods or functions often used for learning is a beneficial aspect for students or a beginner to identify if they have implemented the functions correctly. It will also help coders with more experience, when dealing with long lines of code containing many methods that need to be checked. Our language will help by detecting the methods automatically so the user just need to provide the expected output or type a keyword.

Language reference manual

Tokens

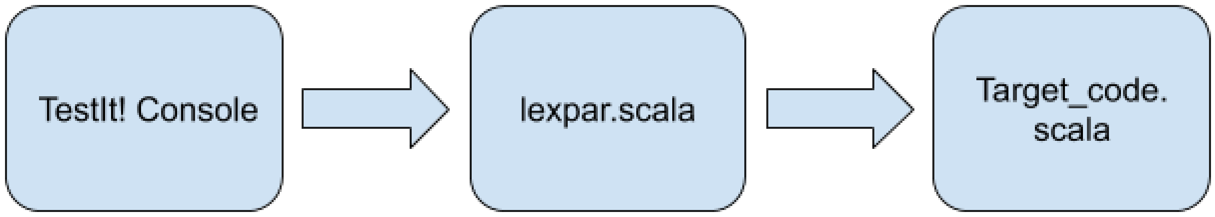
```
Int = "[0-9]+"
Word = "[a-zA-Z]{1}"
Id = "[a-zA-Z]{1}" | "[a-zA-Z][0-9]"
Delimiter := ("(" | ")" | "[" | "]" | "," | ";")
Operator = ( "<=" | ">=" | "=" | "!=" | "<" | ">")
Bool = ("true" | "false")
```

Grammar

```
Id = (forbidden ~> "[a-zA-Z]{1}".r ~ rep(forbidden ~> "[a-zA-Z]{1}".r |
"[0-9]+".r))
Term = ( "[" ~ mylist ~ "]"
   | Null
   | empty
   | bool
   | int
   | ("'" ~> word <~ "'"))
Proplist = ("'" ~ word ~ "'" ~ rep("," ~> "'" ~ word ~ "'")) | (int ~
rep("," ~> int))
mylist = rep(proplist)
propparamlist = (term ~ rep("," ~> term))
Plist = rep(propparamlist)
exp = (("test" ~ "method" ~ id ~ "(" ~ term ~ ")")
 | ("test" ~ "method" ~ id ~ "(" ~ plist ~ ")")
 | ("create" ~ "void" ~ "tester" ~ id ~ "(" ~ ")")
 | ("execute" ~ id ~ "(" ~ ")")
 | "testAll" )
```

Language development

Translator architecture



Module Interface

The Main.scala runs continuously in the console accepting commands as input from the user and sending them to the parser. With the help of the scala parsing library, the parser in lexpar.scala process this input corroborating the grammar and syntax. Now, depending on the input, the lexpar.scala calls for required methods from the class that the user wants to test. Calling upon a method was the biggest challenge for this project. We used scala reflection, creating a caller for invoking the method. Reflection in scala works differently as in other languages so our parser has some limitations, in that you have to send AnyRef. We decided to include methods with only one parameter for the time being. A class MyCases.scala was created as an example class to test methods, but the program works for any class.

Development Environment

- Scala 2.11: Language used
- Scala Parsing Library: Lexing and Parsing tools
- Intelli J: IDE

Test Methodology

To test the lexer and parser, various texts (created by the team) with right and wrong grammar examples were used. For the program functionality a bottom-up approach of Incremental Integration Testing was used to test each module individually and then integrate them one after the other to test all of them. Every test run revealed new problems and after fixing everything up, the program run successfully with no other discovered bugs.

Language tutorial

1. Make sure Scala 2.11 is installed.
2. Download the .zip file from the Github repository provided.
3. Unzip this file and import the project using your preferred IDE.
4. Run the main class (Main.scala) in the source folder of the project.
5. Input the commands in the console as instructed in the video tutorial or refer to Various Commands below.
6. To close, press q.

Various commands:

>> testAll

This commands test all of the declared methods in your scala class, it prompts the user for the parameters for each method, then the expected output, and proceeds to tests your methods to see if they pass or fail.

>> test method methodName(parameter)

Tests that specific method in this case *methodName* with the parameters inside the parentheses, and prompts the user for the expected output and tests the method.

\>>create void tester checkMult()

Creates a txt file where the user can customize a tester. To write the tester you must go to src\main\customTesters and you will find the file you created. The benefit of this as opposed to the test method command is that in can tackle a range of values. For example if I want my method to output true if the result is larger than (>) 3, something that is not possible in the expected output of the test method.

Syntax for void tester:

```
checkMult():
multiply([1,2,3])
true if (this < 7)
else false
```

The first line is already provided for the user when you create the file. In the second line you must specify the method that you are going to test with the parameters inside the parentheses. On the third line you must specify the words true if and within the parentheses specify the condition. You can simply write the word true if you want it to always return true, same as with false, or put a condition. The word this represents the output of the method. On the fourth line you must write else false.

```
>>execute checkMult():
Custom Tester: PASSED the test
Or
Custom tester: FAILED the test
```

Test Program

Example of testAll: (not shown completely due to long output)

```
>> testAll
Method: firstCharInStr
Specify parameters:
>>Sky
Expected output:
>>S
Your output:
S
Correct! The method 'firstCharInStr' passed the test

Method: revStr
Specify parameters:
>>der
Expected output:
>>rde
Your output:
Error with method

Method: add5Int
Specify parameters:
>>2
```

```
Expected output:
>>7
Your output:
7
Correct! The method 'add5Int' passed the test
-----------------------------------------------------------------------------------------------------------
```

Conclusion

With TestIt!, unit testing is easier in Scala by simplifying the creation of custom testers, testing common methods results (like sorting, reverse check, min and max finder, etc.) with just one keyword of expected output, and testing every single method in a file. These features makes this process way more easier in some cases, however this project could be further expanded in a future iteration. One way to expand the language capabilities is to support its keyword system (to test common methods) by adding testers like the basic collections operations search, add and remove, and much more. Another expansion option is to support other languages in which testing is not an easy and simple task. These are the ideas the team had during development, but there are lots more testing aids that could be implemented. Testing is a hard and hated part of programming, so every step taken towards making it easier and less tedious is a feature most programmers will enjoy.