



UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

TESTING DOCUMENT

PROJECT NAME: ARCANE ARCADE

CLIENT: TONY VD LINDEN

TEAM NAME: TERABITES

TEAM MEMBERS

| | | | |
|-------------|------------|----------|----------|
| NG Maluleke | D Mulugisi | C Nel | LE Tom |
| 13229908 | 13071603 | 14029368 | 13325095 |

September 27, 2016

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 2 |
| 1.1 | Purpose | 2 |
| 1.2 | Scope | 2 |
| 1.3 | Test Environment | 2 |
| 1.4 | Assumptions and Dependencies | 2 |
| 2 | Test Items | 2 |
| 3 | Functional Features to be Tested | 3 |
| 4 | Test Cases | 3 |
| 4.1 | Language Features | 3 |
| 4.2 | | 3 |
| 4.3 | | 4 |
| 4.4 | | 5 |
| 5 | Execution Strategy | 5 |
| 6 | Item Pass/Fail Criteria | 5 |
| 7 | Detailed Test Results | 6 |
| 7.1 | Overview of Test Results | 6 |
| 8 | Conclusions and Recommendations | 6 |

1 Introduction

The project is called *Arcane Arcade*, which references the esoteric language users will have to use, as well as the gamification approach to try and make it as fun as possible.

1.1 Purpose

Unit testing is a crucial part of a continual software development approach and is necessary to ensure that all components properly work before integrating them into the system.

1.2 Scope

1.3 Test Environment

The tests are independent of the operating system and occur under Maven. The environment setup of the tests is the inclusion and import of the used frameworks. The tests are JUnit tests and are running under JUnit 4.12. Maven sets up the environment by providing all the needed dependencies which are included in the POM file.

1.4 Assumptions and Dependencies

Most of the unit testing is to test the compiler component of the system. The compiler consists of a lexer, parser, and visitor. The lexer and parser are automatically generated by ANTLR4 using a specified grammar file which specifies the lexer and syntax rules for the language. The visitor is manually implemented and is in charge of walking the parse tree in the correct order and executing the appropriate commands on each node.

Testing is done through JUnit test files and Maven automatically running the tests when deploying the project or running the *test* command through Maven.

2 Test Items

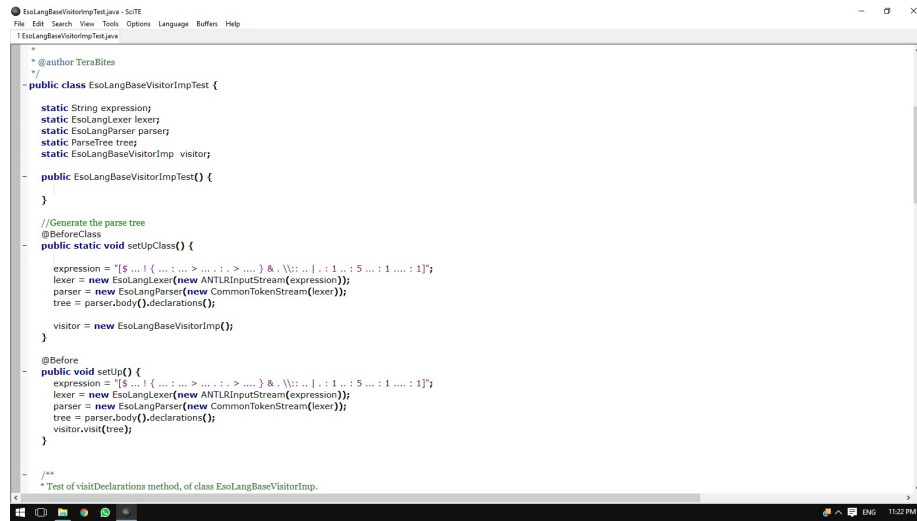
The lexer and parser are tested together by feeding a list with a syntactically correct and wrong statements in the implemented language. The lexer then tokenizes the input whereafter the parser parses the token stream and builds a parse tree. It is then tested that the parser reports the correct semantic errors.

The visitor is tested by having it run through a list of statements. The visitor is implemented in such a way that it holds a list of values that need to be printed. The test then simply looks at this list in the visitor object and compares it with a list of expected results.

3 Functional Features to be Tested

4 Test Cases

4.1 Language Features



```

*
* @author TeraBites
*/
public class ESoLangBaseVisitorImpTest {

    static String expression;
    static ESoLangLexer lexer;
    static ESoLangParser parser;
    static ParseTree tree;
    static ESoLangBaseVisitorImp visitor;

    public ESoLangBaseVisitorImpTest() {

    }

    //Generate the parse tree
    @BeforeClass
    public static void setUpClass() {
        expression = "[{ ... | { ... : ... > ... : > ... } & \\.\\w : \\w | : 1 .. : 5 ... : 1 .... : 1}]";
        lexer = new ESoLangLexer(new ANTLRInputStream(expression));
        parser = new ESoLangParser(new CommonTokenStream(lexer));
        tree = parser.body().declarations();
        visitor = new ESoLangBaseVisitorImp();
    }

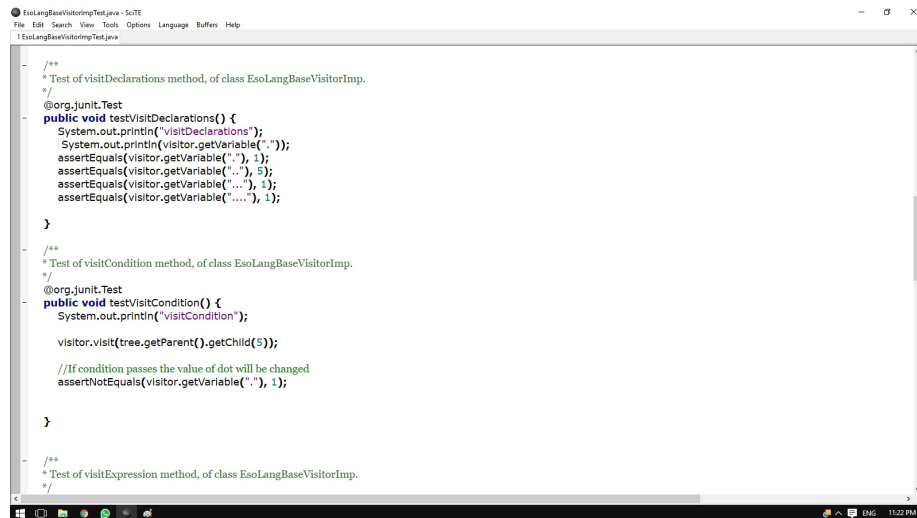
    @Before
    public void setUp() {
        expression = "[{ ... | { ... : ... > ... : > ... } & \\.\\w : \\w | : 1 .. : 5 ... : 1 .... : 1}]";
        lexer = new ESoLangLexer(new ANTLRInputStream(expression));
        parser = new ESoLangParser(new CommonTokenStream(lexer));
        tree = parser.body().declarations();
        visitor.visit(tree);
    }

    /**
     * Test of visitDeclarations method, of class ESoLangBaseVisitorImp.
     */
}

```

The two above above methods annotated with the @Before and the @Before-Class prepare the test environment for the next test methods by executing and restoring the states prior to running another test.

4.2



```

/**
 * Test of visitDeclarations method, of class ESoLangBaseVisitorImp.
 */
@org.junit.Test
public void testVisitDeclarations() {
    System.out.println("visitDeclarations");
    System.out.println(visitor.getVariable("."));
    assertEquals(visitor.getVariable("."), 1);
    assertEquals(visitor.getVariable("."), 5);
    assertEquals(visitor.getVariable("."), 1);
    assertEquals(visitor.getVariable("."), 1);
}

/**
 * Test of visitCondition method, of class ESoLangBaseVisitorImp.
 */
@org.junit.Test
public void testVisitCondition() {
    System.out.println("visitCondition");
    visitor.visit(tree.getParent().getChild(5));

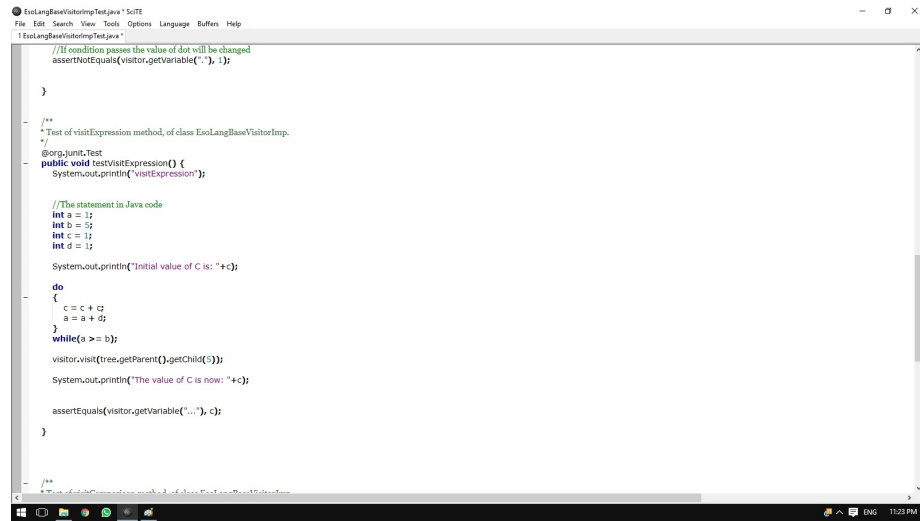
    //If condition passes the value of dot will be changed
    assertEquals(visitor.getVariable("."), 1);
}

/**
 * Test of visitExpression method, of class ESoLangBaseVisitorImp.
 */
}

```

Here we test to see if the assignments under declarations and working properly and if conditional statements are functioning as expected. To test if the methods are functioning as expected, we compare values from the methods and the values that we expect to find via the `assert()` method.

4.3

A screenshot of a Java IDE window titled 'EsoLangBaseVisitingTest.java'. The code is a JUnit test for the 'visitExpression' method of the 'EsoLangBaseVisitorImp' class. It includes a comment about a condition being changed, followed by a test method that prints the initial value of 'c', performs a loop of calculations (c = c + c, a = a + d), calls the visitor method, prints the new value of 'c', and finally asserts that the visitor's result matches the expected value of 'c'.

```
1 EsoLangBaseVisitingTest.java
// If condition passes the value of dot will be changed
assertNotEquals(visitor.getVariable("c"), 1);

}

/**
 * Test of visitExpression method, of class EsoLangBaseVisitorImp.
 */
@org.junit.Test
public void testVisitExpression() {
    System.out.println("visitExpression");

    // The statement in Java code
    int a = 1;
    int b = 5;
    int c = 1;
    int d = 1;

    System.out.println("Initial value of C is: " + c);

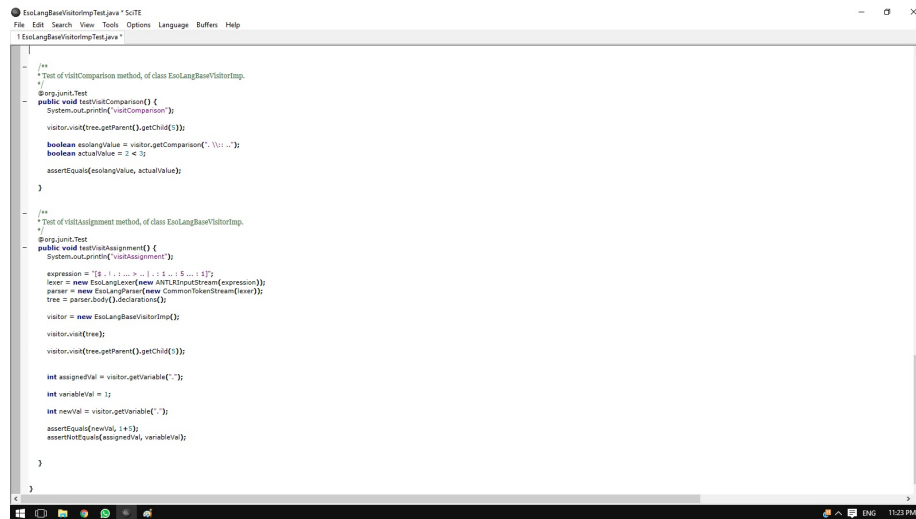
    do
    {
        c = c + c;
        a = a + d;
    }
    while(a >= b);

    visitor.visit(tree.getParent().getChild(5));
    System.out.println("The value of C is now: " + c);

    assertEquals(visitor.getVariable("c"), c);
}
```

An expressions which is ran in the esolang is also ran in java code and the end results of the two same but differently executed methods are compared for equality through the `assert` methods.

4.4



We test assignment and comparison by comparing expected values with the values we get from the esoteric language.

5 Execution Strategy

6 Item Pass/Fail Criteria

7 Detailed Test Results

7.1 Overview of Test Results

7.2 Functional Requirements Test Results

7.2.1 Authentication Module

7.2.2 User Management Module

7.2.3 Challenge Management Module

8 Conclusions and Recommendations

The body of our tests re-use the same variables which can have a negative effect on our testing as values get altered before the test so to combat this we used junit `@BeforeClass` and `@After` annotations to prepare the environment for accurate testing and to make sure that each execution begins with the appropriate criteria.

Exit criteria are observed and compared immediately before all test functions exit. Here the effects of the of the statements are compared and asserted as the expected values.