**CS474 – OBJECT ORIENTED LANGUAGES AND ENVIRONMENT**

**FALL ’19 – COURSE PROJECT**

**DOCUMENTATION**

**TEAM MEMBERS**

|  |  |  |
| --- | --- | --- |
| **NAME** | **NETID** | **UIN** |
| ASHWIN BALASUBRAMANI **(LEADER)** | abalas25 | 661357395 |
| SACHIN MANIMEKALAI BALAKRISHNAN | smanim2 | 678578693 |
| SHREYA SHRIVASTIVA | sshriv9 | 650062501 |

**PROBLEM STATEMENT**

To create an instrumentation program that takes the syntactically correct source code of some Java applications, and to use Eclipse Java Abstract Syntax Tree (AST) parser, to parse this application to an AST, and to use the Visitor builder pattern to compute the scopes and variables of the application and to insert corresponding instrumenting statement in the appropriate places of the code.

**LANGUAGES USED**

* Java (Version: 11.0.4) – for the program that is parsed
* Scala (Version: 2.13.1) – for all the main implementation of the problem statement

**REPOSITORIES IMPORTED**

* Eclipse Java Development Tools (JDT) Core (Version: 3.19.0)

<https://mvnrepository.com/artifact/org.eclipse.jdt/org.eclipse.jdt.core>

* Eclipse JDT Annotations for Enhanced Null Analysis (Version: 2.2.300)

<https://mvnrepository.com/artifact/org.eclipse.jdt/org.eclipse.jdt.annotation>

* The Apache Commons IO library contains utility classes, stream implementations, file filters, file comparators, endian transformation classes, and much more.

<https://mvnrepository.com/artifact/commons-io/commons-io>

**TO TEST THE APPLICATION**

The file launcher file in the test folder is ran with different java source codes, to get the instrumentation lines inserted to them along with the logging files.

Once the Java program is instrumented, **uncomment** the line 17 in the newly instrumented java code, to get the Trace files.

//template.logging(); //Uncomment statement after the code is instrumented

**LAUNCHING APPLICATION**

* The main launch file is in **src/test/scala/launcher.scala** location of the project folder. This will take the AddTwoNumbers.java file and insert logging statements into it.
* **SBT Build:** The application compiles and runs from the command line using the commands

sbt clean, sbt compile and sbt run.

* **IntelliJ:** The application was built is IntelliJ, hence, to run the application, go to the location **src/test/scala/launcher.scala** to get the results.
* After launching the application, the results are produced. The files created are

1. **\_old\_java\_file\_name.txt** – this file contains the original source code for which the instrumentation code is inserted and computed.
2. **java\_file\_name.java** – this file contains the source code with the instrumentation lines for the Template class, which will invoke the template.scala file.
3. **parser.scala** – this file has the code for all the parser implementations: reading the code, parsing it using AST parser, rewriting the AST with instrumentation nodes and unparsing the AST to get the file with instrumentation code.
4. **template.scala** – this file contains the template class code which uses the instrumented java code to create two other files, one with the logging statements and another file with a table of all variables and their bindings in the application’s scope.
5. **launcher.scala** – the main file that launches all the implementation.
6. **tracefile.txt** – file that contains the logging statements for the original java source code.
7. **tablefile.txt** – file that contains all the variables and their bindings in the application’s scope.

**EXPLANATION**

**parser.scala**

First, all the required packages are imported which includes,

* **java.io** – package to use the BufferedReader for file reading, creating new files, FIleInputStream, FileOutputStream etc.,
* **java.util** – for using ArrayList’s etc.,
* **scala.util.control** – to use the break statement to come out of loops
* **org.eclipse.jdt.core.dom.rewrite** – to rewrite the AST, so instrumentation lines can be added to the file
* **org.eclipse.jdt.core.dom** – to create AST, AST nodes, AST parser, visitor pattern, compilation unit, to parse various statements of the code
* **org.eclipse.jface.text** – to create document and iDocument
* **org.eclipse.text.edits.TextEdit** – to do elementary text manipulation operations
* **org.apache.commons.io.FileUtils** – to do general file manipulation utilities

The source code is stored in a file object. This file is copied into another file adding \_old\_ prefix to the existing file name, as the current file will be populated with the instrumented statements. After that, the source file is read using BufferedReader into an object and this string is turned into a character array, as AST Parser uses characters to parse the file.

A new AST parser is created using the JLS11 and this parser is set to be the compilation unit kind and the source is set with the Java source file character array. Then, several AST visitor classes are created for each type of statements required for the application such as expression, looping statements (if, while, for), assignment statement, method invocation etc., In each of the visitor class, the visit method is overridden to add the visited node matching the specified statement to a list. This node value is returned using a user defined function.

Now that all the nodes in the source code are visited using the visitor pattern, next step is to add instrumentation lines to this AST. This is done by using the AST rewrite function. It reads all the nodes, the line numbers of each lines, its starting position and counts the number of statements that matches for which the instrumentation code needs to be added. Because, not all the lines of a source file need to be added with an instrumentation line.

Now, an instrumentation line is built for the number of the statements available in the source code. This instrumentation line looks something like,

template.instrum(13, "If Statement", template.pair("AddTwoNumbers.main().i: ", String.valueOf(i)));

In this line, template is the name of the template class, instrum in the method in the template class which takes 3 parameters: line number, type of the statement, scope of the statement (class\_name.function\_name.variable\_name), value of the variable. The last two variables (scope and the value of the variable are given as a pair for easier computation and the instrum function takes these pairs as a string of arguments. These instrumentation statements are added after and before of each of the matched lines in the code. This is because the looping statements and the return statements, may sometimes be unreachable by the compiler, hence those statements are inserted before its original statement and others are inserted after. To rewrite the file, the output object must be converted to a document object and then written to it.

**template.scala**

All required packages are imported which includes,

* **java.io** – package to creating new files, FileWriter etc.,
* **java.util** – for using ArrayList’s etc.,
* **scala.util.control** – to use the break statement to come out of loops

For adding the logging statements, initially all the variable’s names, values, line numbers where they are present must be extracted from the original source code. For this, a separate argument class is written which extracts the names, values and line numbers from the function call to the instrum function, so that these values can be logged into the trace file.

Instrum function takes 3 arguments, the line number, type of statement and the string of arguments in which all the scopes of the variables and method used are taken. In this function, the logging file is built slowly, by calling the logging() function, which will in turn call the other function, which will use the argument class, extract all the required names and values of each statements and create a logging file.

**Output of trace file:**

Line Number: 6 Type of statement: Assignment Statement Parameters: AddTwoNumbers.main().sum: 10  
Line Number: 7 Type of statement: Assignment Statement Parameters: AddTwoNumbers.main().a: Ashwin  
Line Number: 8 Type of statement: Assignment Statement Parameters: AddTwoNumbers.main().b: Balasubramani  
Line Number: 10 Type of statement: Assignment Statement Parameters: AddTwoNumbers.main().i: false  
Line Number: 11 Type of statement: While Statement Parameters: AddTwoNumbers.main().i: false  
Line Number: 13 Type of statement: If Statement Parameters: AddTwoNumbers.main().i: false  
Line Number: 20 Type of statement: Return Statement Parameters: AddTwoNumbers.display().name: New Name  
Line Number: 16 Type of statement: Assignment Statement Parameters: AddTwoNumbers.main().c: New Name

**Output of table:**

AddTwoNumbers.main().sum:   
Present at Line Position: 6  
Type of Statement: Assignment Statement  
Value: 10  
 AddTwoNumbers.main().a:   
Present at Line Position: 7  
Type of Statement: Assignment Statement  
Value: Ashwin  
 AddTwoNumbers.main().b:   
Present at Line Position: 8  
Type of Statement: Assignment Statement  
Value: Balasubramani  
 AddTwoNumbers.main().i:   
Present at Line Position: 10  
Type of Statement: Assignment Statement  
Value: false  
 AddTwoNumbers.main().i:   
Present at Line Position: 11  
Type of Statement: While Statement  
Value: false  
 AddTwoNumbers.main().i:   
Present at Line Position: 13  
Type of Statement: If Statement  
Value: false  
 AddTwoNumbers.display().name:   
Present at Line Position: 20  
Type of Statement: Return Statement  
Value: New Name  
 AddTwoNumbers.main().c:   
Present at Line Position: 16  
Type of Statement: Assignment Statement  
Value: New Name

The same thing is done for the table that needs to be created with the variables and their bindings in the application’s scopes. This table will also have the path to a variable, its declaration in the line of the code. The table is printed in the format by using a separate print table function.

def pair(a: String, b:String): Seq[String] ={  
 *Seq*(a + ", " + b)  
}

Code that changes String to Seq[String] in order to achieve Java-Scala compatibility