

CSE 6324 - 004 - Advance Topic in Software Engineering

Project Title: Smart Contract Analysis tool - Using ANTLR

Team 5 - Iteration 2

Team Members -

- | | |
|-------------------------------|--------------|
| 1. Shubham Arun Malankar | (1002031033) |
| 2. Nageshwar Ramkumar Jaiswal | (1002033432) |
| 3. Ravi Prakasha | (1002026832) |
| 4. Navyashree Budhihal Mutt | (1001965572) |
| 5. Rushikesh Mahesh Bhagat | (1001911486) |

Project Plan:

Iteration 2: Building a tool to detect vulnerabilities and error using ANTLR

In this iteration, our team set out to build a tool from scratch that could effectively detect vulnerabilities and errors in smart contracts. We recognized the importance of such a tool in the analysis of smart contracts, as it would enable developers to identify and mitigate potential issues before deployment.

However, developing such a tool proved to be a challenging task. One of the most significant challenges we encountered was selecting the optimal programming language to use in smart contract analysis. We were aware that different programming languages possess varying strengths and weaknesses when it comes to analyzing smart contracts. Thus, we conducted extensive research to identify the most suitable programming language for our project.

Our research was based on the available resources for generating parse trees, which are essential in smart contract analysis. After thorough investigation, we concluded that ANTLR was the most powerful parse generator available for reading, processing, executing, or translating structured text or binary files.[\[1\]](#) As such, we utilized ANTLR to develop a tool that could detect both vulnerabilities and errors in smart contracts.

To achieve this, we found a G4 file, which acts as a grammar file and helps in generating three Java files - lexer.java, parser.java, and visitor.java. [\[16\]](#) We pass the Solidity file into the lexer file, which generates a token that is then passed to the parse file as input. [\[8\]](#) [\[9\]](#) The output of the parse file is an Abstract Syntax Tree (AST), which is then passed to the visitor file. [\[10\]](#) From there, our tool processes the source unit code, and the final output is a report of vulnerabilities and errors detected by our tool.

In summary, our team's primary objective in this iteration was to develop a tool from scratch that could detect vulnerabilities and errors in smart contracts. We encountered challenges in selecting the optimal programming language for smart contract analysis, but our extensive research led us to choose ANTLR as the most suitable parse generator. With the use of ANTLR, we developed a tool that generates a report of vulnerabilities and errors detected in smart contracts, ultimately aiding in their analysis and security.

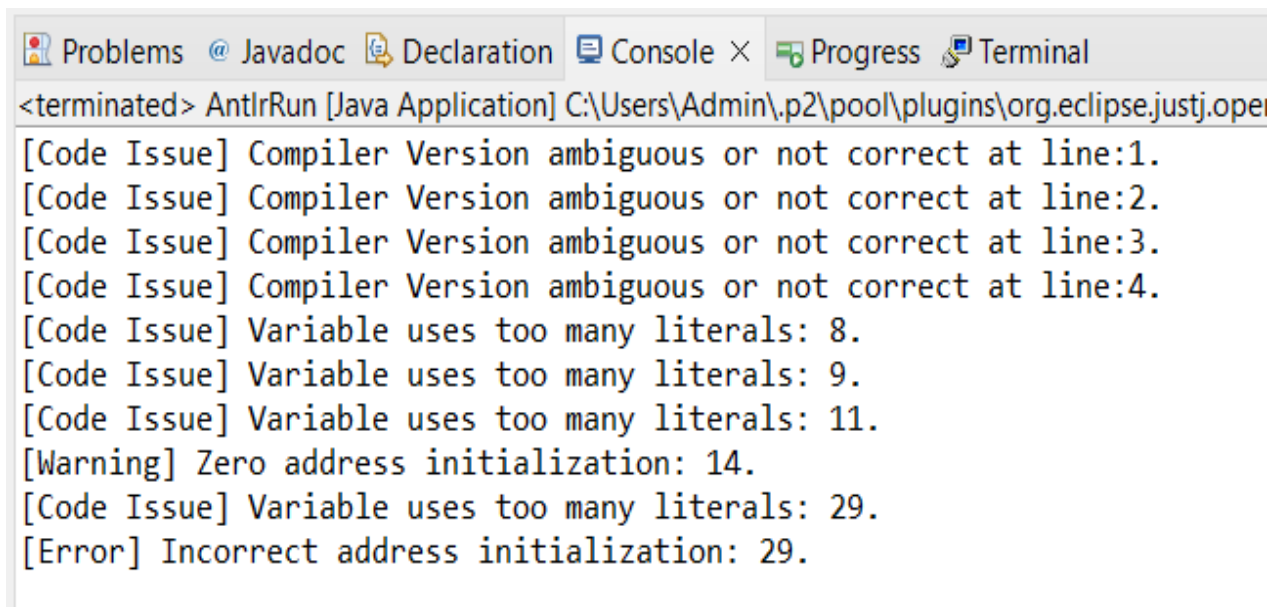
Specification and design:

Inputs:

1. The input is a solidity file. Go to Package Explorer and right click on the project.
2. Click on Run as > Run Configurations.
3. Inside Arguments > Select Variables > Select File Prompt
4. Click on Run and select the folder tests.
5. Choose the required solidity test file for static analysis.

Output:

1. In the below screenshot we the prompt shows that we have multiple definitions for solidity file version. Also, we have assigned incorrect address and there are variables with zero address assignment.



```
<terminated> AntlrRun [Java Application] C:\Users\Admin\p2\pool\plugins\org.eclipse.justj.open  
[Code Issue] Compiler Version ambiguous or not correct at line:1.  
[Code Issue] Compiler Version ambiguous or not correct at line:2.  
[Code Issue] Compiler Version ambiguous or not correct at line:3.  
[Code Issue] Compiler Version ambiguous or not correct at line:4.  
[Code Issue] Variable uses too many literals: 8.  
[Code Issue] Variable uses too many literals: 9.  
[Code Issue] Variable uses too many literals: 11.  
[Warning] Zero address initialization: 14.  
[Code Issue] Variable uses too many literals: 29.  
[Error] Incorrect address initialization: 29.
```

Fig 1: Output showing different vulnerabilities.

Data structures used:

- ArrayList: List interface in Java is implemented concretely by ArrayList. As new or deleted components are added, the resizable array can dynamically expand or contract. By using their index, elements may be retrieved, and ArrayList offers several practical techniques for modifying the list, including add, remove, and set. Because of its simplicity and quick access to its members, the ArrayList data structure is well-liked. [\[11\]](#)
- Iterator: In Java, an iterator interface offers a mechanism to loop around a group of elements, such as a List or Set. It offers ways to retrieve the following element in the collection and determine whether there are still more elements in the collection to iterate through. Iterators may be used to iteratively delete entries from a collection as well as to traverse collections in a forward direction. The Java Collections Framework's core component and a widely used interface in Java is the iterator interface. [\[11\]](#)

Implemented Feature:

- Check ambiguous or incorrect version of solidity file. Compiling smart contracts with an outdated version of Solidity can potentially introduce security vulnerabilities. [\[12\]](#) Newer versions of Solidity contain security enhancements and bug fixes are not present in older versions. To reduce the risk of vulnerabilities, it is important to use the latest version of Solidity and adhere to best practices for smart contract development.
- A zero-address vulnerability in Solidity smart contracts occurs when the contract code does not handle the zero Ethereum address (0x0). [\[7\]](#) Attackers can exploit this vulnerability to execute malicious codes, steal funds or disrupt the normal operation of the contract. We fixed this vulnerability by checking the address value assigned to variables.
- If a smart contract has a function that transfers tokens to a recipient's address, and the contract code assigns an incorrect address to the recipient variable, the tokens may be sent to an unintended address or lost permanently. We fixed this error by performing input validations.
- The "too many digits" vulnerability is a type of vulnerability that can occur in smart contracts written in Solidity. [\[15\]](#) This vulnerability arises when a contract's code fails to properly handle numbers with many digits, which can lead to unexpected behavior or even the loss of funds. This vulnerability particularly arises when there are too many zeros in literal. Therefore, we check literal for consecutive zeros and validate if it is valid hexadecimal address.

Code and Tests:

The code changes done as part of this iteration are available through our repository:

Repository link - <https://github.com/shubhammalankar/ASE-CSE-6324-Team-5-Slither>

Check Iteration 2 folder for the source code.

Steps involved in running the tool:

1) Install Eclipse IDE:

1. Go to any browser and type Eclipse. [2]
2. Click on the Submit button.
3. Select the appropriate version of eclipse according to your operating system.
4. Install Eclipse by clicking on the downloaded .exe file.
5. Click on the run button.
6. Click on the install button.
7. Click on the launch button.
8. Create a new java project.

2) Install Java:

1. Install Java from Oracle Java Downloads page. [3]
2. Click on the appropriate installer for the operating system and download it.
3. Run installation file.
4. Close the exit wizard after receiving the "Successfully Installed" message.

3) Set Environment Variables for Java in Windows:

1. Add Java to System Variables. [14]
2. Open the Start menu and search for environment variables.
3. Select the Edit the system environment variables result.
4. In the System Properties window, under the Advanced tab, click Environment Variables.
5. Under the Stem variables category, select the Path variable and click Edit:
6. Click the New button and enter the path to the Java bin directory.
7. Click OK to save the changes and exit the variable editing window.

4) Download Antlr :

1. Download Antlr jar from Antlr website. [4]

To set up Antlr path for Windows:

1. Search for environment variable and copy the path where antlr is downloaded

2. Click on the new environment variable.
3. Give the variable name as ANTLR_HOME and the path copied to variable value.
4. Select Path and click edit and create a new entry as %ANTLR_HOME%\bin
5. Click on ok

5) To set up build-path for the project: [\[13\]](#)

1. Right click on the project click on Build Path and configure build path.
2. Click on Libraries and Add Jars.
3. Then select the jar within the lib folder of the project.
4. Go to source and select on the project and expand to select native library location.
5. Double click on it and add the lib folder and select ok.
6. Apply and Close for the last step.

Please find the attached screenshots for the above-mentioned steps.

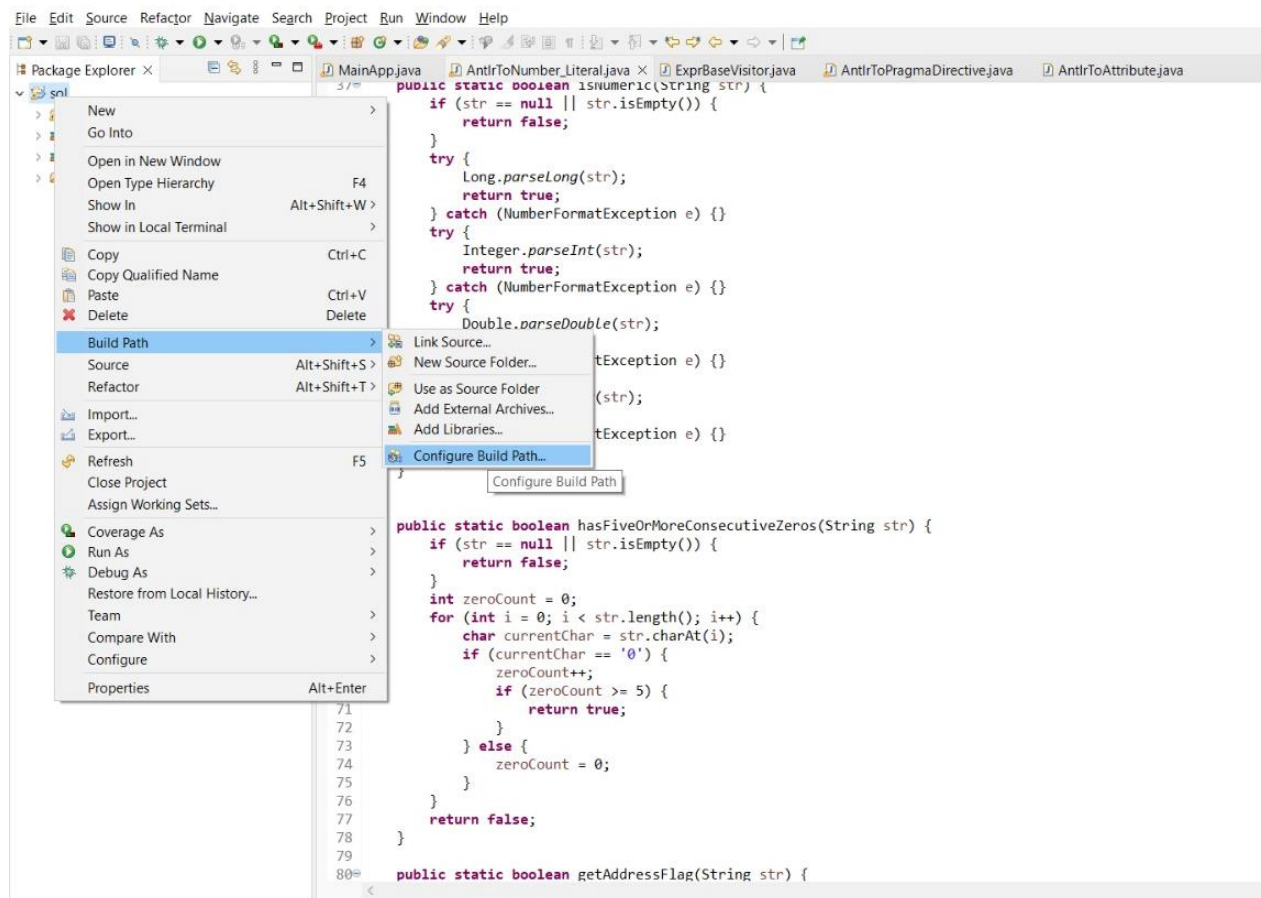


Fig 2: Screenshot showing configuring Build Path.

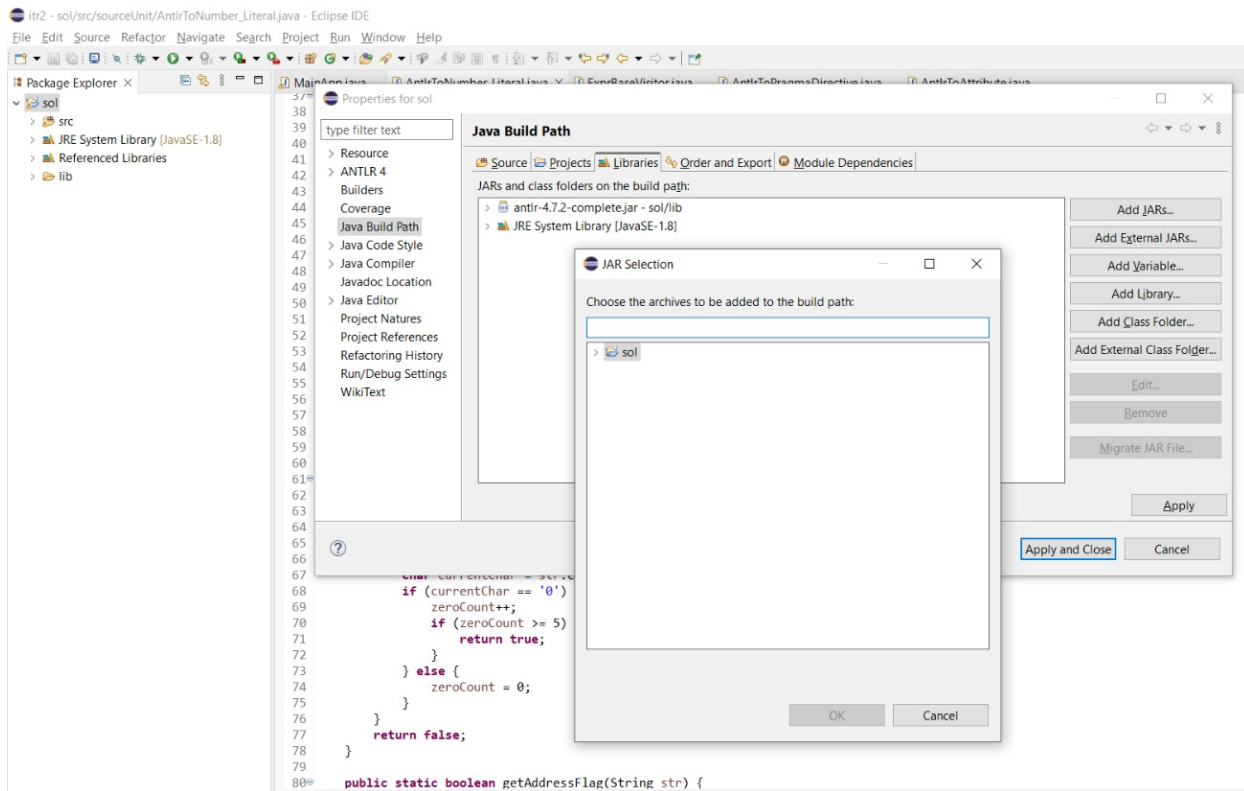


Fig 3: Screenshot showing Build Path window.

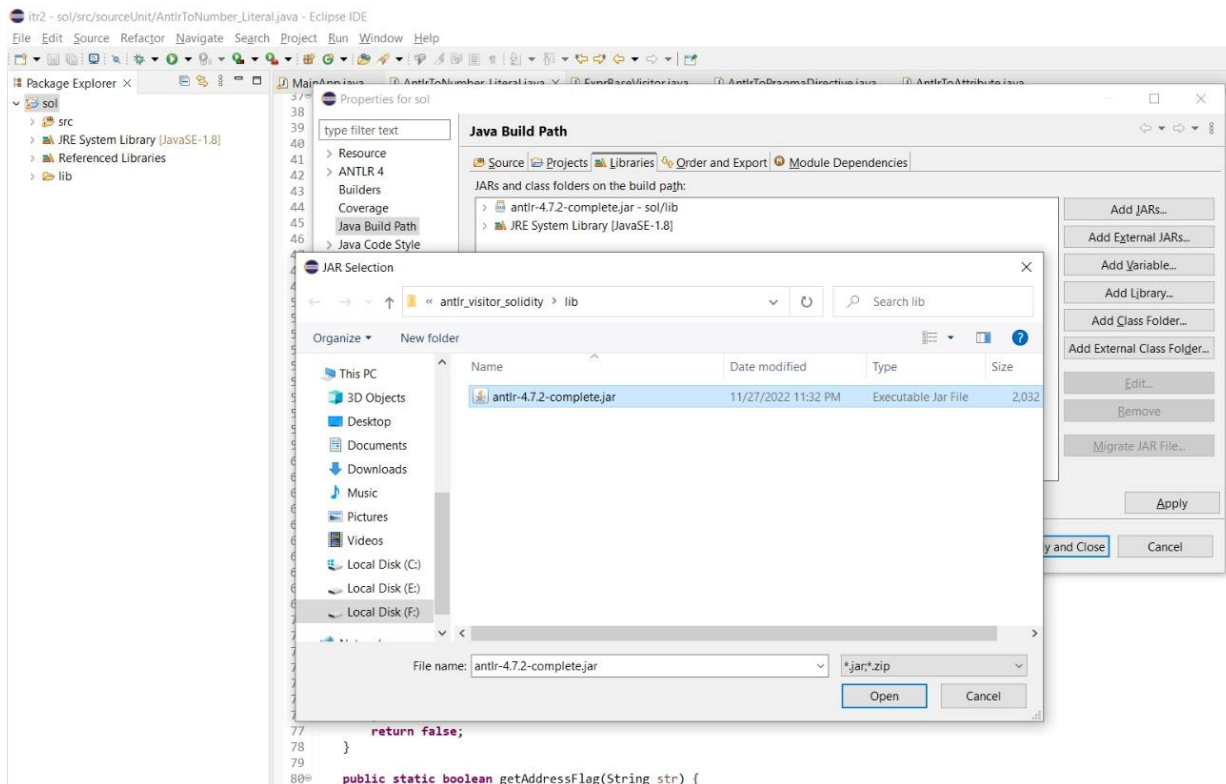


Fig 4: Screenshot showing antlr.jar selection.

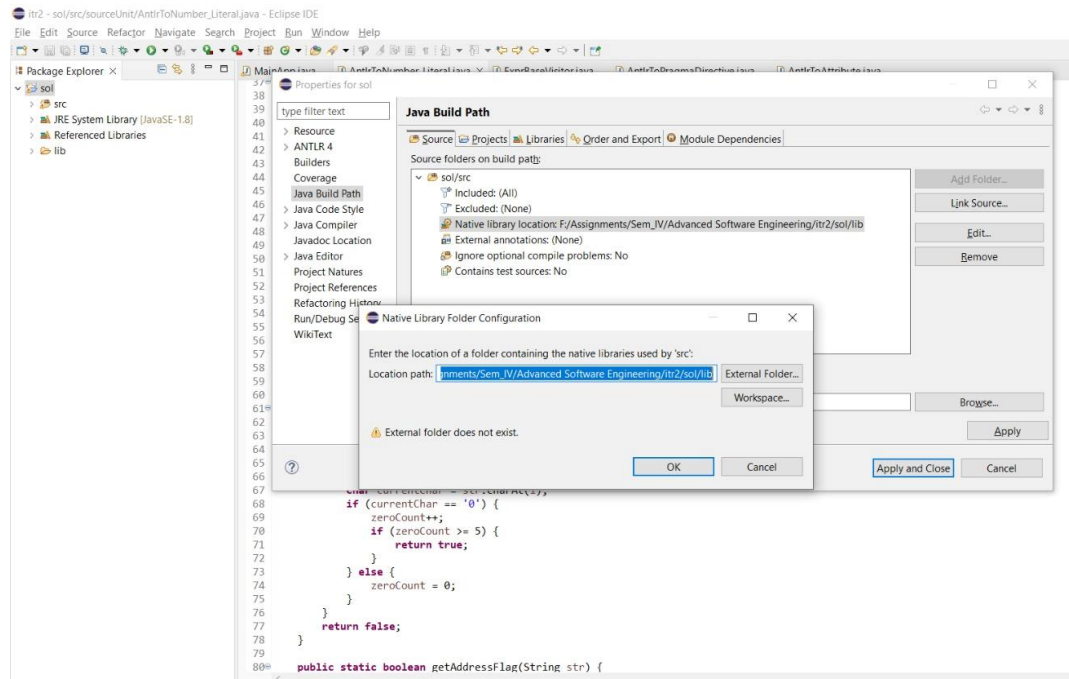


Fig 5: Screenshot adding jar to native library.

6) To generate parser and lexer classes for g4 file: [\[13\]](#)

1. Right click on the antlr folder in the project folder outside of eclipse and open command prompt.
2. Type the command `antlr4 -no-listener Expr.g4`.
3. Then compile all the java classes within the project structure.
4. Enter the command `javac *.java`.

Please find the attached screenshots for the above-mentioned steps.

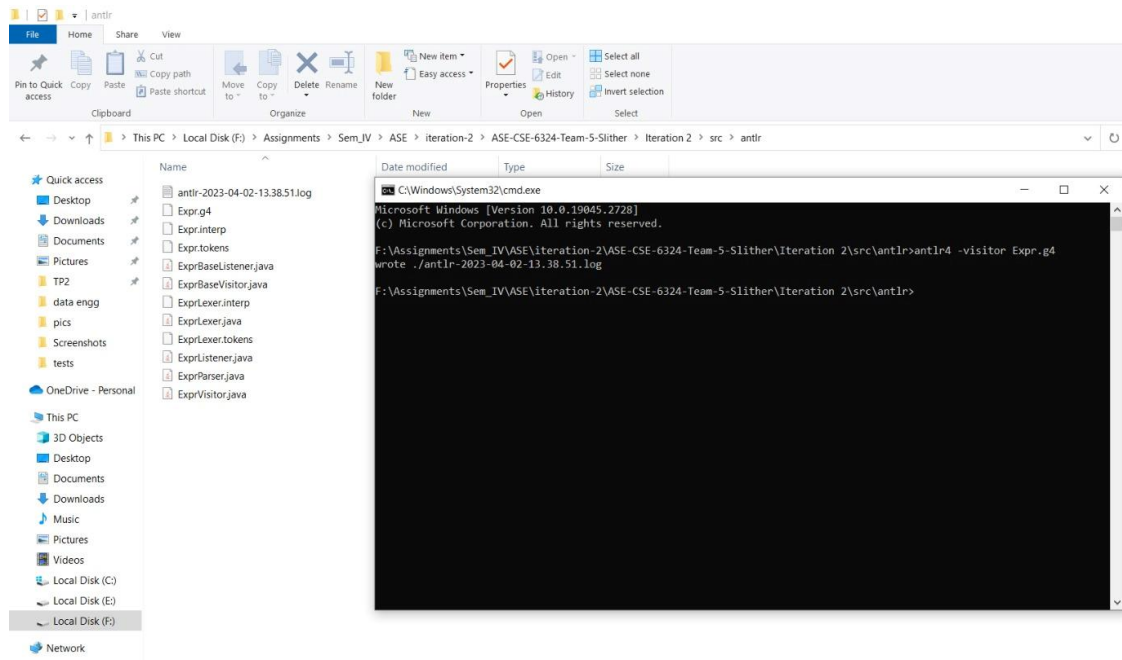


Fig 6: Screenshot showing command executed and java files generated.

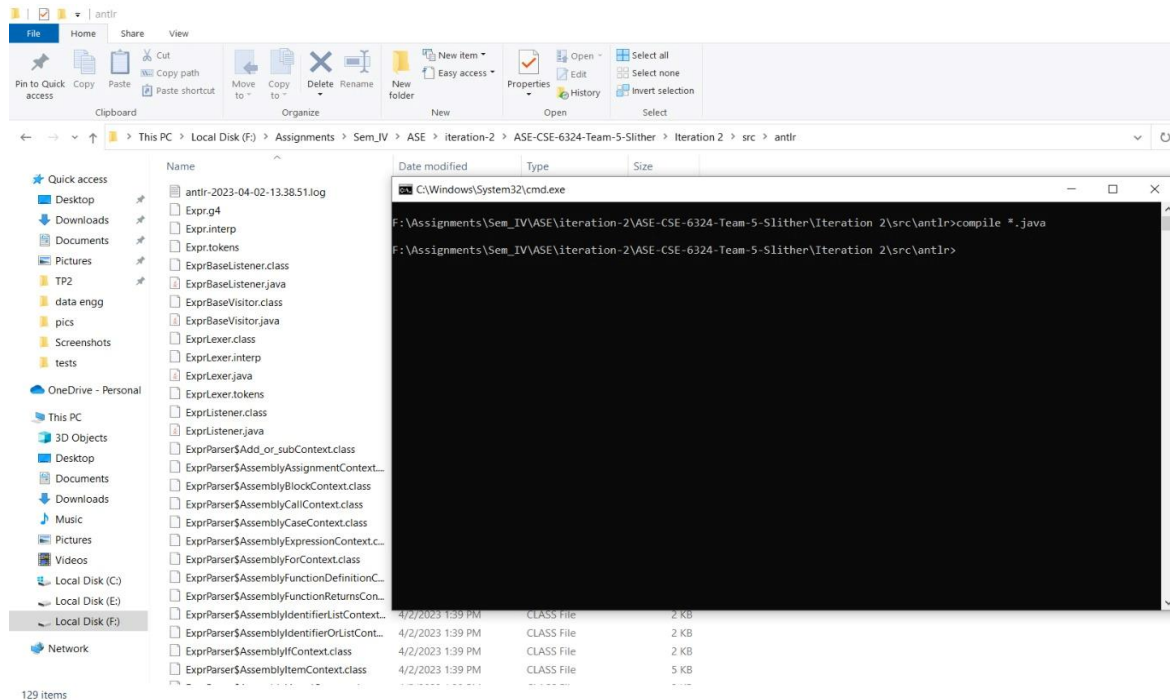
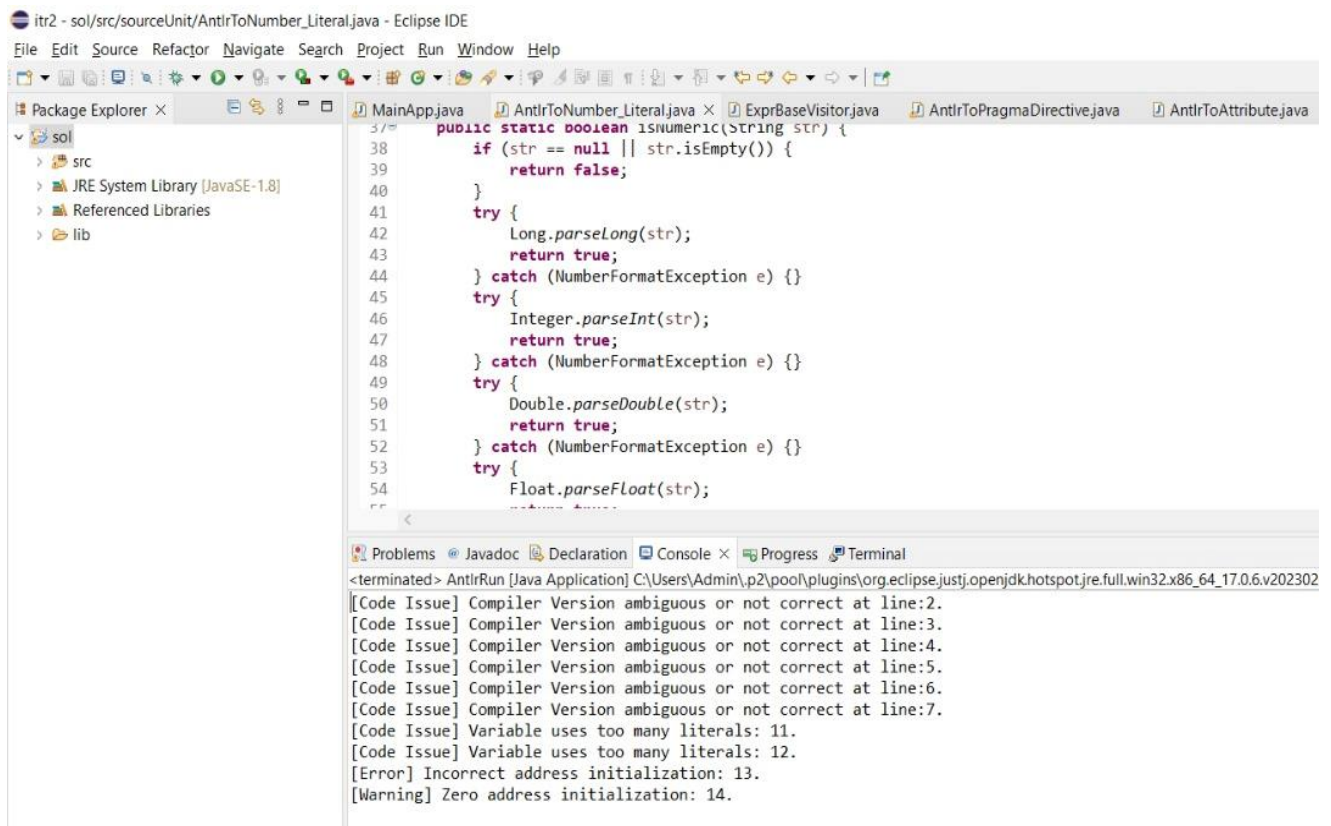


Fig 7: Screenshot showing java compilation.

7) To check for vulnerabilities within smart contracts:

1. Download the project from the repository. [\[5\]](#)
2. Import the project and follow the above environment setup mentioned for the above project.
3. Right Click on the project and run as configurations
4. Then in the Arguments Tab under program arguments click on variables
5. In variables select file_prompt and run
6. Then select the solidity test file from tests folder



The screenshot shows the Eclipse IDE interface. The top toolbar includes icons for File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The Package Explorer on the left shows a project named 'sol' with subfolders 'src' and 'lib'. The main editor displays a Java file named 'AntlrToNumber_Literal.java' with the following code:

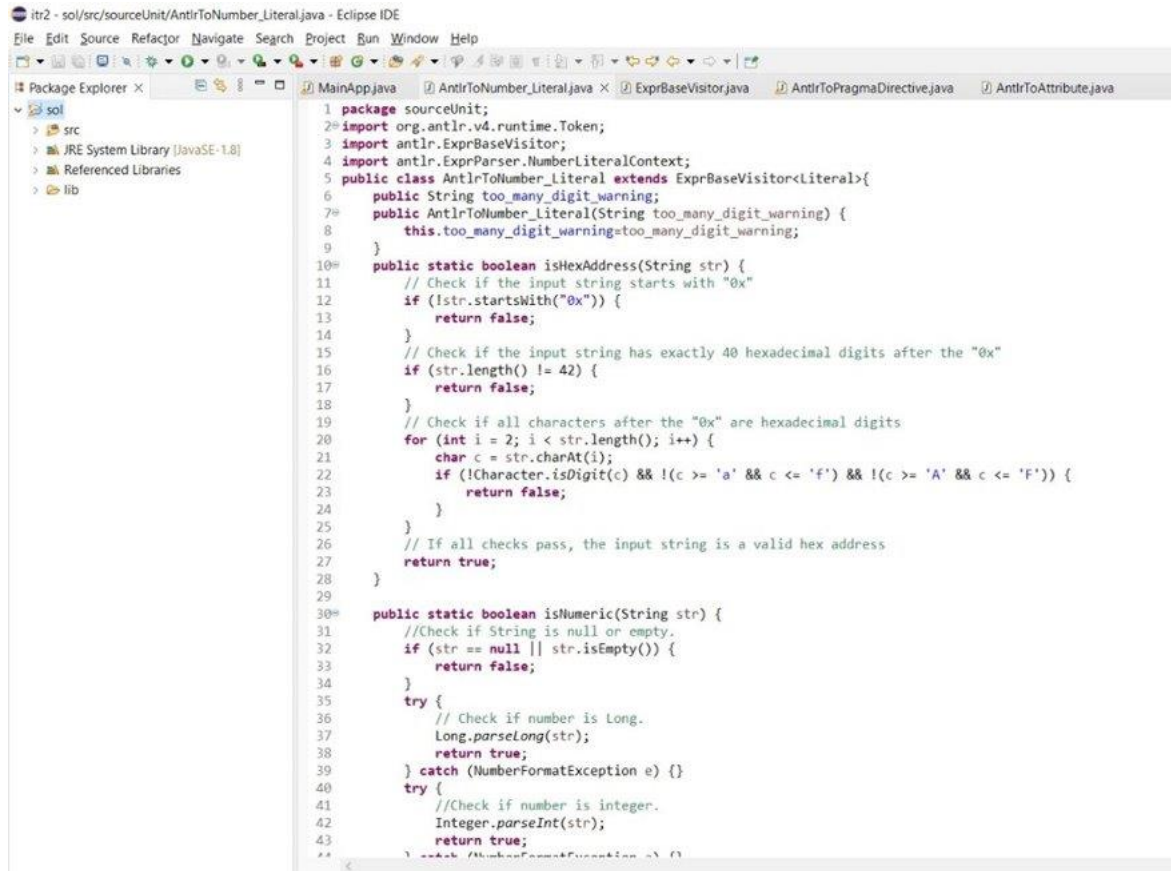
```
37 public static boolean isnumeric(String str) {
38     if (str == null || str.isEmpty()) {
39         return false;
40     }
41     try {
42         Long.parseLong(str);
43         return true;
44     } catch (NumberFormatException e) {}
45     try {
46         Integer.parseInt(str);
47         return true;
48     } catch (NumberFormatException e) {}
49     try {
50         Double.parseDouble(str);
51         return true;
52     } catch (NumberFormatException e) {}
53     try {
54         Float.parseFloat(str);
55         return true;
56     } catch (NumberFormatException e) {}
57 }
```

The bottom console shows the output of the AntlrRun [Java Application] command. The output includes several warnings and errors:

```
<terminated> AntlrRun [Java Application] C:\Users\Admin\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v202302
[[Code Issue] Compiler Version ambiguous or not correct at line:2.
[Code Issue] Compiler Version ambiguous or not correct at line:3.
[Code Issue] Compiler Version ambiguous or not correct at line:4.
[Code Issue] Compiler Version ambiguous or not correct at line:5.
[Code Issue] Compiler Version ambiguous or not correct at line:6.
[Code Issue] Compiler Version ambiguous or not correct at line:7.
[Code Issue] Variable uses too many literals: 11.
[Code Issue] Variable uses too many literals: 12.
[Error] Incorrect address initialization: 13.
[Warning] Zero address initialization: 14.
```

Fig 8: Screenshot showing the output showing vulnerabilities.

Code Screenshots:



```
1 package sourceUnit;
2 import org.antlr.v4.runtime.Token;
3 import antlr.ExprBaseVisitor;
4 import antlr.ExprParser.NumberLiteralContext;
5 public class AntlrToNumber_Literal extends ExprBaseVisitor<Literal>{
6     public String too_many_digit_warning;
7     public AntlrToNumber_Literal(String too_many_digit_warning) {
8         this.too_many_digit_warning=too_many_digit_warning;
9     }
10    public static boolean isHexAddress(String str) {
11        // Check if the input string starts with "0x"
12        if (!str.startsWith("0x")) {
13            return false;
14        }
15        // Check if the input string has exactly 40 hexadecimal digits after the "0x"
16        if (str.length() != 42) {
17            return false;
18        }
19        // Check if all characters after the "0x" are hexadecimal digits
20        for (int i = 2; i < str.length(); i++) {
21            char c = str.charAt(i);
22            if (!Character.isDigit(c) && !(c >= 'a' && c <= 'f') && !(c >= 'A' && c <= 'F')) {
23                return false;
24            }
25        }
26        // If all checks pass, the input string is a valid hex address
27        return true;
28    }
29
30    public static boolean isNumeric(String str) {
31        //Check if String is null or empty.
32        if (str == null || str.isEmpty()) {
33            return false;
34        }
35        try {
36            // Check if number is Long.
37            Long.parseLong(str);
38            return true;
39        } catch (NumberFormatException e) {}
40        try {
41            //Check if number is integer.
42            Integer.parseInt(str);
43            return true;
44        } catch (NumberFormatException e) {}
45    }
```

Fig 9: Screenshot for zero-address check and too-many digits detection

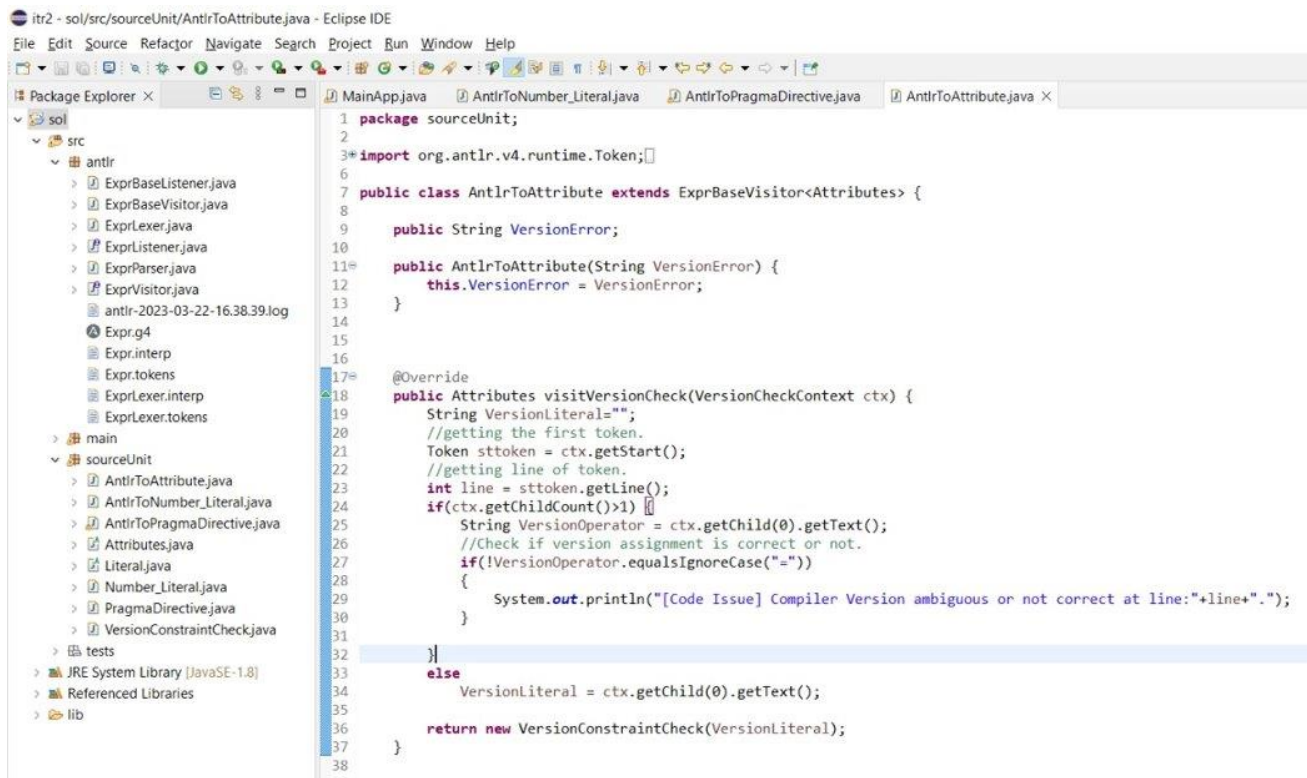


Fig 10: Screenshot for ambiguous version detection in solidity

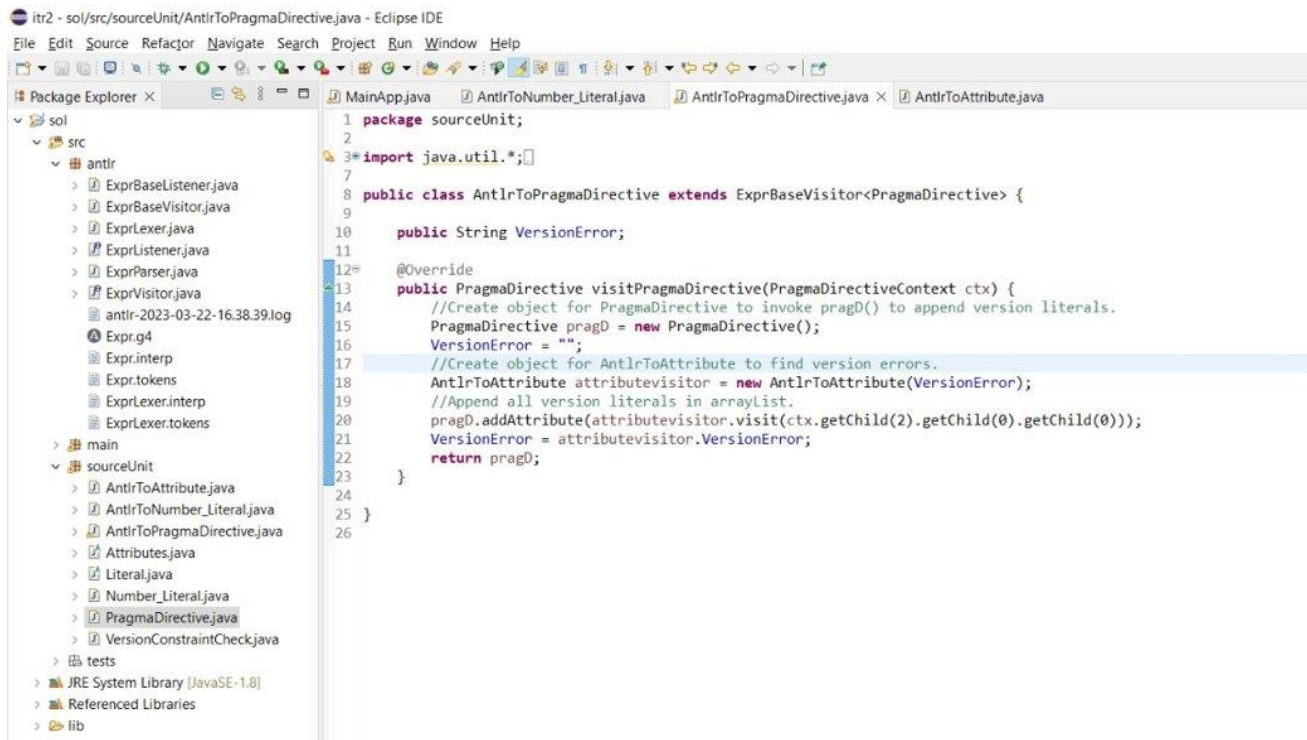


Fig.11 Screenshot to check for ambiguous version detection in solidity.

Risk:

Risk 1 - Inadequate documentation for Antlr

It could be challenging for team members to work together efficiently on Solidity code projects if ANTLR isn't properly documented. Further it is difficult to maintain and update Solidity code in the future if code maintenance is not performed timely. [\[1\]](#)

Risk Exposure - Improper code maintenance cause confusion, inconsistencies, or flaws in the code, which could result in system failures or security breaches, whereas collaborating risks involves delays, mistakes, or misunderstandings that can affect the project's overall success. R.E.: $0.15 * 2 \text{ (hours)} = 0.3 \text{ (Potentially 36 minutes more)}$.

Mitigation - To mitigate improper code maintenance risk we can make use of version control to manage changes to Antlr generated code. Further we can implement high quality code review which will help potential maintenance and collaboration issues early.

Risk 2 - How to visit actual nodes of Abstract Syntax Tree.

Due to less experience of Antlr, we were facing difficulty in accessing AST. It was necessary to understand the working abstract syntax tree. We were facing issues in getting actual values of nodes.

Risk Exposure - Needed to invest more of our time in visiting each node and understand the functionality.

R.E.: $0.55 * 3 \text{ (hours)} = 1.65 \text{ (Potentially 5 hours more)}$.

Mitigation - We debugged each output of the result and got our target literal on which we further worked upon.

Risk 3 - Finding Test Contract.

Finding test contracts which will align to the test cases and meet requirements such as ambiguous version definitions, check zero address, also incorrect address assignment.

Risk Exposure - Will not be able to proper functionality testing of our tool.

R.E.: $0.50 * 2$ (hours) = 1 (Potentially 2 hours more).

Mitigation - We found a reliable github repository which consists of test contract related to crypto currency, e-governance which aided our testing requirements. [\[6\]](#)

Risk 4 - Antlr producing different results for different versions of JDK

We implemented the code in eclipse IDE in java, we need to match the version of Antlr jar file with jdk version. Antlr jar required specific version of java jdk. [\[1\]](#)

Risk Exposure - Parse tree was not generating.

R.E.: $0.6 * 2$ (hour) = 1.2 * (Potentially 2 hours and 30 minutes more).

Mitigation - While setting the project environments, we need to take care of all the versions of jdk(Java Development Kit) and jar files are compatible with each other.

Risk 5 - Setting up Antlr

To run Antlr globally, we need to set OS environment variables so that we can compile grammar (G4 file) expression file. G4 file is used to generate lexer file and parser file which is compiled again using Antlr. This process is time-consuming and tedious and needs to be followed accordingly. [\[1\]](#)

Risk Exposure - Our main component to write the logic is lexer and parse file. Incorrect setup of Antlr will not generate the outcome i.e., the lexer and parse file. Even if the files are generated and the path for Antlr compiler does not match then we won't get resulting class files.

R.E.: $0.2 * 3$ (hour) = 0.6 (Potentially 2 hours more).

Mitigate - We followed the process as mentioned on online resources and generated the files required to build Abstract Syntax Tree. [\[13\]](#)

Customers and Users:

- It is recommended for all smart contract developers in the industry to utilize Solidity smart contract analysis tools to thoroughly analyze their contracts and ensure their security and functionality
- Local companies utilizing Ether and in charge of their own smart contract management should think about using Solidity smart contract analysis tools to raise the caliber and dependability of their smart contracts.
- Darshan Ujjini Mallikarjuna: A blockchain enthusiast who has hands-on experience with our static analysis tool and tested smart contracts using that tool and gave frequent feedback.

References:

- [1] <https://wwwantlr.org/>
- [2] <https://www.eclipse.org/>
- [3] <https://www.oracle.com/java/technologies/downloads/>
- [4] <https://wwwantlr.org/download.html>
- [5] <https://github.com/shubhammalankar/ASE-CSE-6324-Team-5-Slither/tree/master/Iteration%202>
- [6] <https://github.com/OpenZeppelin/openzeppelin-contracts>
- [7] <https://blackadam.hashnode.dev/zero-address-check-the-danger>
- [8] <https://dev.to/lefebvre/compiler-101---overview-and-lexer-3i0m>
- [9] <https://www.baeldung.com/java-antlr>
- [10] <https://stackoverflow.com/questions/29971097/how-to-create-ast-with-antlr4>
- [11] <https://www.geeksforgeeks.org/arraylist-in-java/>
- [12] <https://ethereum.stackexchange.com/questions/96743/build-with-multiple-solc-versions>
- [13] https://www.youtube.com/watch?v=pa8qG0l10_I
- [14] <https://stackoverflow.com/questions/32241179/setting-up-environmental-variables-in-windows-10-to-use-java-and-javac>
- [15] <https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits>
- [16] <https://github.com/antlr/grammars-v4/blob/master/solidity/Solidity.g4>