

JAVA COMPILER: A MODERN TOOL FOR STUDENT CODING AND TEACHER EVALUATION

Kavitha C.R*, Dudekula Sai Mallesh, Chandradeep Reddy Edurinty, Gangineni Divya Venkata Taraka Rama Rao,
Department of Computer Science and Engineering,
Amrita School of Computing, Bengaluru, Amrita Vishwa Vidyapeetham, India
cr_kavitha@blr.amrita.edu, bl.en.ucse21051@bl.students.amrita.edu
bl.en.ucse21045@bl.students.amrita.edu, bl.en.ucse21056@bl.students.amrita.edu

Abstract— This paper presents a Java compiler which will help a teacher in testing students' solutions written for home assignment. The Java programs developed by students are typed online through an HTML interface where they are analysed for lexical properties and syntax and their performance profile is generated. The fraud detection defines improper tokens during the assembling process and checking syntax errors during the time complexity analysis. Furthermore, the code summarizer determines cases of printing of answers directly in the code, whereby teachers can easily identify such. The results are presented well, and it will help teachers when evaluating code for correctness and efficiency. The objective of this project is two-fold; on one hand, code evaluation will be improved for faster results for educators and on the other hand, students will have practice tools for interactive learning.

Keywords— *Java Compiler, Python Backend, Lexical Analysis, Syntax Checking, Abstract Syntax Tree (AST), Code Summarizer.*

I. INTRODUCTION

Programming languages have been the driving force to the progress of computer science, therefore, creating software that is powerful and efficient. Java is absolutely the most widely-used programming language as it is platform independent, has numerous libraries, and has a big user base. It is thus, significant for students and developers to understand how Java programs are written, compiled, and executed.

Modern software systems are getting more and more complicated, so besides others, the knowledge of programming languages and compilers has become a must for that need. The compiler translates programmer's code into a series of instructions of program for a computer.

These instructions follow the sequence of operations: breaking the code into smaller units, checking if the structure is correct, and finally generating the machine code. The steps are highly important to ensure that the program is correctly executed and in an efficient manner.

This Java compiler, the paper claims, is one of a kind in the manner of helping teachers and students with their course material. It, in turn, analyses the Java code, finds out mistakes, and the code is reported as the correction.

Besides, it shows that the functions of a compiler are fulfilled, it additionally details concepts like whether the student plagiarizes by either his/her code answers being directly printed summary of the code is shown, the structure of the abstract syntax tree (AST), and, finally, it provides performance indicators of time complexity, space complexity. Therefore, these features help both students and teachers have more lean conversations.

The project is very useful to teachers as it provides all the desired details, thus, the feedback on student work is easier. The learners can improve their personal programs by the given comment and they also can figure out exactly how their code is compiled and executed. This approach is not only of assistance for Java learnership but it also clarifies for a student the compiler functionalities during the learning process.

In this paper, we describe how a compiler was made, the project gives a practical approach to teaching the concept of compilers, and it is useful in building the connection between theoretical and practical knowledge of programming.

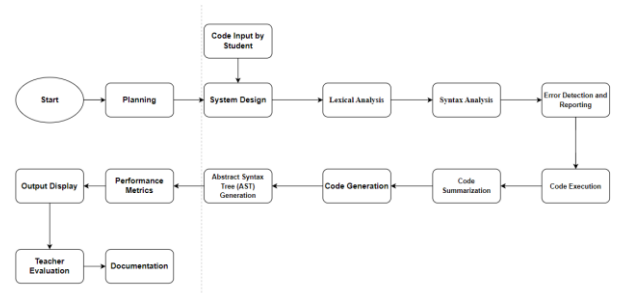
II. LITERATURE SURVEY

Mak *et al.* [1] highlights significant challenges in maintaining Java projects, particularly regarding snapshot breakage, where only 38.13% of snapshots are compliant due to various errors, as noted by Tufano *et al.* and replicated by Hassan *et al.* Additionally, research indicates that continuous integration (CI) failures often mirror these issues, emphasizing the importance of effective debugging methods. Techniques such as summarizing build failure messages and automating fixes for dependency issues have been proposed to aid developers in addressing these challenges. Hayataka *et al.* [2] reviews existing literature on optimizing Java bytecode, noting Clausen's work on the Cream optimizer, which enhances performance by removing dead code and loop invariants. It also references Tip *et al.*'s method for reducing application size through bytecode rewriting, emphasizing its relevance to performance. The review identifies a gap in addressing bytecode replacement concerning JVM behaviors, which this paper aims to fill. Additionally, it highlights how minor differences in bytecode can significantly

affect JIT compiler behavior and execution speed. Mark *et al.* [3] on code style errors in Java highlights the significance of maintaining a coherent coding standard, especially in collaborative environments where multiple developers contribute to the same codebase. Tools like PMD and SonarLint are commonly used to identify and rectify style violations, focusing on issues such as unused variables and unnecessary object creation. Furthermore, the complexity of accurately identifying these style issues is underscored by the challenges of purity analysis, which assesses the side effects of methods. Overall, the development of tools that can automatically refactor code to adhere to clean coding practices is essential for fostering better programming habits among developers. Lu *et al.* [4] paper systematically investigates 333 unique bugs in popular Java decompilers, revealing that a significant portion arises from the stages of entity loading, optimization, and code generation. Notably, over 80% of these bugs manifest as exceptions, syntactic errors, or semantic errors, with type inference issues being particularly complex to resolve. Additionally, the introduction of JD-Tester, a differential testing framework, demonstrates the importance of rigorous testing in identifying and addressing these decompilation challenges. Midya *et al.* [5] highlights the CWE taxonomy for categorizing vulnerabilities, the Juliet test suite as a vital resource for testing, and reviews five static analysis tools: Facebook Infer, SonarQube, SpotBugs, Find Security Bugs, and PMD. Comparing with prior research, the authors emphasize their broader coverage of weakness categories and the need for robust test suite validation. Limitations of static analysis, like false positives/negatives, are also noted, setting the foundation for the study's empirical evaluation. Jia *et al.* [6] paper studies bug detection in the Java Virtual Machine's (JVM) Just-In-Time (JIT) compiler, where complex runtime optimizations can introduce critical bugs. Researchers, led by Haoxiang Jia, analyzed 9,000+ bugs from OpenJDK's bug tracker and found many are triggered only under specific optimization settings, revealing limitations in traditional testing. To address this, they developed JOpFuzzer, a new testing tool that explores both seed inputs and optimization options, using machine learning to guide fuzzing toward likely bug-triggering configurations. JOpFuzzer outperformed existing tools, finding 41 bugs in OpenJDK, highlighting its effectiveness and potential for future JVM testing improvements. Utting *et al.* [7] compares outputs across implementations to uncover semantic bugs, is highlighted for its effectiveness in compilers, including JVM and Solidity. The authors use this approach to validate formal models of GraalVM IR against Java, confirming model accuracy with test suites. Challenges like fixed-width arithmetic surfaced, reflecting the

difficulty of matching formal models to real implementations. Insights gained from this study offer guidance for future modeling of complex systems to improve compiler reliability. Siri *et al.* [8] paper optimizations for array handling in Java runtimes, particularly with AArch64 architecture. Integrating Eclipse OMR with OpenJ9 enables architecture-specific enhancements, leveraging AArch64's NEON technology for SIMD operations, which boosts array copying performance. Key optimizations include the array copy Evaluator method in the JIT compiler, which in lines `System.arraycopy` to reduce method call overhead and improve memory efficiency. The paper also addresses garbage collection considerations for safe array copying. Using Bumble Bench and Perf, the authors benchmarked these optimizations, offering insights into efficient array handling on AArch64. Zang *et al.* [9] reviews key compiler testing techniques like grammar-based and mutation-based fuzzing, which JATTACK enhances by incorporating developer insights. It highlights JATTACK's use of template programs with specified "holes" to explore complex execution paths and its template-based approach, leveraging developer intuition to generate targeted test cases. The Author introduces a Java-embedded DSL for defining abstract syntax trees, broadening the search space for test cases. Overall, JATTACK's approach offers a unique, developer-driven method for JIT compiler testing. Gamboa *et al.* [10] reviews key developments in Liquid and Refinement Types. Originally applied in ML to enhance compile-time error detection, Liquid Types use logical predicates, with applications across languages like C, Ruby, and JavaScript. Java implementations, however, faced challenges due to rigid type hierarchies. A survey of Java developers on Liquid Java's syntax revealed limited familiarity with Refinement Types, emphasizing the need for user-centered design and educational resources. This underscores the importance of making Liquid Types more accessible for Java developers.

III. METHODOLOGY



Fig(a): Methodology of the Model

The Java compiler works step-by-step with different parts, all of which focus on very important parts during compilation. The very important parts include the

Lexical Analysis, Syntax Parsing, Semantic Analysis, Intermediate Code Generation, Code Optimization, Execution and Output Generation, and Performance Metrics. All these parts can make sure the compiler works efficiently and at educational level.

User Interface for Code Input: The project begins with a straightforward front end in which students can directly input or simply copy and paste their Java code. This is the component where they input their code for analysis by the system they are using.

Lexical Analysis: After the code has been entered the first step is to segment the code into smaller components called tokens. This process is termed as lexical analysis. Different authors explain lexical analysis in different ways but all point to the same concept; converting of textual data into a format that is usable by a computer.

Tokens act as different parts of the code which comes in the form of keyword, variable, operator etc. It is only of great importance in enabling the analyst to comprehend the structural form of the code.

Syntax Checking: Once the code is tokenized it verifies if the code conforms to Java grammar rules and regulation. This is referred to as syntax checking. Errors in code structure are likely to be detected in the system and the specific errors to be corrected leave a message to the user.

Code Execution: Once you see the check mark next to the script name, you know the code is good and the next thing is to run the script. The system processes the Java code and gives out the result which are the implementation of the code written.

Code Summarization: At this stage there is the code summarizer. It scans the code and manages to identify such habits like direct printing the answers. If such attempts are present, it informs the teacher as shown in the result below. This feature ensures that teachers can tell whether a particular student is trying to falsify their results by having the actual answer coded in the app.

Time Complexity Calculation: System also checks the time complexity of the algorithm. It indicates how well the solution i.e. the efficiency with which the code runs are one of the ways to assess the performance of the solution. It is crucial for knowing how lengthy the coding will become if larger inputs are entered.

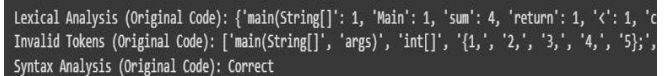
Outcome Presentation: After all the steps are taken, the system sets out the results for teachers in a meaningful way. Teachers see the output, errors, and time

complexity from the code summarizer as well as the feedback from the code summarizer. Therefore, teachers can easily access the improvement areas and in the assessments, students' work is evaluated more easily.

IV. RESULTS

The mini-Java compiler was tested carefully and produced results at each stage without error. Hence, it demonstrated the accuracy, efficiency, and learning potential of this compiler concept.

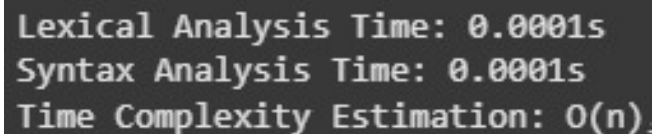
In the first stage, lexical analysis successfully demarcated the Java code into tokens; keywords, variables, and operators were split out very well. In the second stage, the syntax parser checked whether the code constructed according to its rules is syntactically correct, offering helpful error messages in case the rules were not followed.



```
Lexical Analysis (Original Code): {'main(String[]': 1, 'Main': 1, 'sum': 4, 'return': 1, '<': 1, 'c
Invalid Tokens (Original Code): ['main(String[]', 'args)', 'int[]', '{1,', '2,', '3,', '4,', '5');',
Syntax Analysis (Original Code): Correct
```

Fig(b): Lexical and Syntax Analysis of the input code

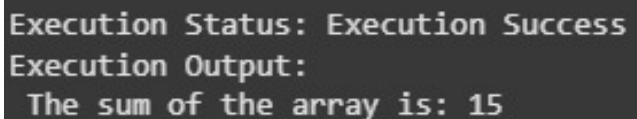
To test the efficiency of processing and compiling Java programs, mini-Java compiler was tested. Measurements were taken at every step of the compilation process.



```
Lexical Analysis Time: 0.0001s
Syntax Analysis Time: 0.0001s
Time Complexity Estimation: O(n).
```

Fig(c): Lexical and Syntax Analysis execution time

During lexical analysis as shown in Fig(b), the system did very well in making input Java code meaningful to be output to the users in terms of tokens. This way, users can see how source code is broken down into its constituent parts for further processing.



```
Execution Status: Execution Success
Execution Output:
The sum of the array is: 15
```

Fig(d): Execution Status of the input code

During syntax analysis, the compiler accepted those programs with correct syntax and moved them to the next phase while flagging errors with clear and actionable messages in such a way that would help users quickly identify and correct their mistakes.

```

Execution Status: Compilation Error
Error Details:
Main.java:13: error: not a statement
    sum += nums[i];12
                  ^
Main.java:13: error: ';' expected
    sum += nums[i];12
                  ^
2 errors

Lexical Analysis Time: 0.0001s
Syntax Analysis Time: 0.0001s

```

Fig(e): Output of the Error code

The efficient execution of the code demonstrated the compiler's capability to translate input Java programs to executable machine-level instructions. This paper displayed the output of the program, thus confirming the correctness of the code compiled.

```

Summary of code:
This code defines a class named 'Main'.
This program contains a main method, which is the entry point of the program.
The program defines the following methods: 'if', 'for', 'for', 'for', 'for'.
The program declares the following variables: i (type: int), j (type: int).
The program uses loops to iterate over data.
The program contains print statements to display the output.
The program uses conditional statements (e.g., 'if' statements) for decision-making.

```

Fig(f): Summary of the input code

This can be observed from Fig. (f) where the code summarizer successfully analyses the input Java program and generated a structured summary. It identifies the class name (Main) for the main class, the main method in the class (that becomes the entry point) and the methods and variables (if, for, i, and j) used in the code.

It also highlights the summable programming constructs such as loop for iteration and conditional statements for taking decision. It also has print statements used to show outputs, which teachers can see, to show what this program does, and to help them to evaluate the logic and the intent behind the code.

```

int[][] matrix1 = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

int[][] matrix2 = {
    {9, 8, 7},
    {6, 5, 4},
    {3, 2, 1}
};

```

Fig(g): Input of two matrices

We show the functionality of the proposed Java compiler in evaluating matrix operations from Fig. (g). The input consists of two matrices, 'matrix1' and

'matrix2', each defined as 3x3 arrays. 'matrix1' contains sequentially increasing integers, while 'matrix2' contains sequentially decreasing integers. These matrices are processed successfully by the compiler which implements correct lexical and syntactic analysis. It runs the program, validates and process the content of the program and generate the required output.

The example presented here illustrates that the compiler can be intelligent about multidimensional arrays, as multithinking the compiler is a practical thing to do in educational settings.

The system estimated and published the time complexity and space complexity linked to the compiled program, which thus provided users with useful information about their codes' computational requirements.

```

Execution Status: Execution Success
Execution Output:
Sum of the two matrices:
10 10 10
10 10 10
10 10 10

```

Fig(h): Output of Sum of Matrix in the backend

The output of the matrix operations defined in Fig. (g) is given in Fig. (h). The input matrices were processed successfully by the compiler, the syntax was validated and the program ran without error.

Everything works, we can see that multidimensional arrays are properly treated; the compiler is producing precise results as well. But again, this shows how the compiler can execute, analyze matrix-based computations, which effectively proves its usefulness in teaching and understanding programming concepts.

All-rounded methodology ensures that every stage of compilation is marked by transparency and improving clearness, which improves both practical and academic value of the project.

```

Enter Java Code:
public class Main {
    public static void main(String[] args) {
        // Example 2D matrices
        int[][] matrix1 = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };
        int[][] matrix2 = {
            {9, 8, 7},
            {6, 5, 4},
            {3, 2, 1}
        };
        // Ensure both matrices are of the same dimensions
        if (matrix1.length != matrix2.length || matrix1[0].length != matrix2[0].length) {
            System.out.println("Error: Matrices must be of the same dimensions.");
            return;
        }
        // Calculate the sum of the two matrices
        int[][] sumMatrix = new int[matrix1.length][matrix1[0].length];
        for (int i = 0; i < matrix1.length; i++) {
            for (int j = 0; j < matrix1[0].length; j++) {
                sumMatrix[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }
        // Print the sum matrix
        System.out.println("Sum of the two matrices:");
        for (int i = 0; i < sumMatrix.length; i++) {
            for (int j = 0; j < sumMatrix[0].length; j++) {
                System.out.print(sumMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}

Analysis Results:
Lexical Analysis: Correct
Syntax Analysis: Correct
Execution Status: Execution Success
Output: Sum of the two matrices:
10 10 10

```

Fig(i): Front-End of the Compiler

As the input code provided was correct, nothing is displayed in results, as the errors should not be shown. This frontend instead shows just successful processing and execution, and summarized outputs. It is user friendly, that is, the students can interact well with the compiler and lead to valuable knowledge about the code compilation process.

We used an abstract syntax tree to represent the structure of Java code graphically at a stage after syntactic analysis is done. The abstract syntax tree is unique because it represents the source program in a two-dimensional fashion and relates the various entities involved, such as expressions, statements, and method calls.

The mini-Java compiler performed well on both small and large programs; it run them correctly and without any delay or errors. In cases of coding mistakes, the output with error messages were comprehensible and help to fix mistakes fast. In most ways, the mini-Java compiler was an excellent tool in understanding how a compiler works and the steps involved in processing a programming language.

This project offers a Java compiler tool that allows our evaluation of student code using lexical analysis, syntax checking and code execution. Like that, code summarization by the integrated code solves the issue

VI. REFERENCES

- 5