International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

Code Nerve - A Compiler Visualizer

Vaishnavi Punalkar

Assistant Professor

Suryadatta College of Management Information Research & Technology

Pune, India

Ayush Lokhande

Student, MSC Computer Science

Suryadatta College of Management Information Research & Technology

Pune, India

ABSTRACT

Compilers are integral to software development, yet their internal processes remain abstract and challenging for learners. This paper presents a Compiler Visualizer, an interactive educational tool that demonstrates the compilation pipeline from lexical analysis to code generation through real-time visual feedback. By converting token streams, parse trees, and semantic checks into tangible visuals, the tool simplifies the understanding of compiler phases, supports error tracing, and enhances debugging skills. Designed for students, educators, and self-learners, the visualizer reduces the steep learning curve of compiler design and fosters deeper comprehension of language processing concepts.

Keywords: Compiler, Compiler Visualization, Lexical Analysis, Parsing, Syntax Tree, Semantic Analysis, Intermediate Representation, Code Generation, Error Tracing, Educational Tool.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

I. INTRODUCTION

Compilers are essential components in software development, responsible for translating high-level programming code into executable machine instructions through several stages, including lexical, syntactic, and semantic analysis, intermediate representation, optimization, and final code generation. Although these processes are fundamental, understanding how they function internally can be challenging when approached only through textbooks or command-line utilities.

A Compiler Visualizer addresses this challenge by providing an interactive environment that demonstrates each stage of compilation step by step. Users can enter code, observe tokenization, follow the creation of parse trees, examine semantic checks, and review intermediate code transformations. By converting the abstract operations of a compiler into intuitive visual representations, the tool makes complex concepts easier to grasp, strengthens debugging abilities, and improves conceptual understanding.

This methodology benefits learners, instructors, and independent students alike by reducing the difficulty of mastering compiler design. It also offers a practical foundation for introductory compiler projects by supporting grammar testing, error tracing, and the visualization of symbol tables and scope management. Beyond educational purposes, it can also help developers debug domain-specific languages or experiment with new language design concepts.

In summary, the Compiler Visualizer serves as both an instructional and a self-learning resource, turning intricate compiler theory into an engaging and accessible interactive experience.

1.1 Key Contributions

The key features and contributions of this work include:

- **Comprehensive End-to-End Visualization:** The system provides an integrated overview of the entire compilation pipeline, from source code input to virtual machine execution, offering a unified understanding of all stages.
- **Dynamic and Clear AST Representation:** An optimized layout algorithm ensures a clean, interactive parse tree visualization that supports zooming, panning, and code highlighting to link source and structure.
- **Assembly-to-Binary Correlation Module:** The tool includes a feature that clearly demonstrates how assembly instructions correspond to their binary and hexadecimal equivalents, helping users connect human readable code to its machine-level form.
- **Reversible Execution (Time-Travel Debugging):** The built-in Virtual Machine enables users to move forward and backward through execution states using a timeline interface, enhancing comprehension of runtime behavior and program flow.

II. Research Problem

Compilers translate high-level source code into machine instructions through several phases: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization and target code generation. These internal processes are abstract and hard for learners to understand when studied only through textbooks or command-line tools.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

Existing tools cover only parts of the pipeline.

- **ANTLR** supports syntax analysis and only partly supports lexical analysis.
- **JFLAP** supports syntax analysis only partially.
- **Python Tutor** and online IDEs (Replit, OnlineGDB) show step-by-step execution and variable states but do not show how the compiler itself works.
- None of the surveyed tools offer semantic analysis, intermediate representation, or an end-to-end view of the compilation pipeline.

There is therefore no single interactive, learner-oriented tool that visualizes the full compilation process, from source code to virtual machine execution, with error tracing at each stage. This gap makes compiler design steeper to learn and weakens students' debugging and conceptual understanding.

Problem statement: How can the abstract, multi-phase working of a compiler be turned into an interactive, step-by-step visual experience that lets students, educators and self-learners trace source code through every stage of compilation and execution?

III. Research Objectives

1. To design and develop a Compiler Visualizer that demonstrates the compilation pipeline from lexical analysis to code generation and virtual machine execution through real-time visual feedback.
2. To visualize the output of each compiler phase: token streams, parse trees (AST), semantic checks and symbol tables, intermediate representation, and assembly code.
3. To implement a decision-based error-handling mechanism that halts the pipeline when an error occurs and shows line-specific feedback, so users see where and why a process fails.
4. To build an interactive, non-overlapping AST visualization using an adaptive layout algorithm, with zooming, panning and source-to-structure code highlighting.
5. To provide an Assembly-to-Binary correlation module that maps assembly instructions to their hexadecimal and binary encodings, linking human-readable code to machine-level form.
6. To develop a custom stack-based Virtual Machine with reversible execution (time-travel debugging), where state snapshots let users step forward and backward through program execution.
7. To support learners and educators in grammar testing, error tracing, and understanding symbol tables and scope management, and to help developers experiment with domain-specific languages and new language design concepts.
8. To validate the tool through a case study (a C program that calculates the square of an integer) and to plan a formal user study for measuring its pedagogical effectiveness.

IV. Literature Review

4.1 Foundations of compiler design. The Dragon Book (Aho, Lam, Sethi & Ullman) is the standard reference on compiler design. It describes the structured sequence of phases: lexical analysis, parsing, semantic validation, intermediate representation, optimization and code emission. It explains these phases in text, but their internal workings stay abstract for learners who study only from books or non-visual command-line interfaces. This is the main motivation for the present work.

4.2 ANTLR. Parr and Quong (1995) introduced ANTLR, a Predicated-LL(k) parser generator. In the paper's comparison, ANTLR supports syntax analysis and partially supports lexical analysis. It has no semantic analysis, intermediate representation or code generation, no end-to-end pipeline view, and it is not designed for learners. It is a language-processing tool, not a teaching aid.

4.3 JFLAP. Rodger et al. (2011) describe efforts to increase the use of JFLAP in courses. JFLAP is designed for learners and partially supports syntax analysis and step-by-step execution. However, it lacks lexical analysis,

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

semantic analysis, intermediate representation, code generation, memory or variable state views, and an end-to-end pipeline view.

4.4 Python Tutor. Guo (2013 presented Online Python Tutor, an embeddable web-based program visualization tool for CS education. It is learner-friendly and offers step-by-step execution and memory/variable state views. It visualizes program execution only, not the compiler phases (lexical, syntax, semantic, IR or code generation.

4.5 Online IDEs (Replit, OnlineGDB. Online IDEs provide step-by-step execution and variable state views, and code generation only partially. They are not designed for learners, and they give no visibility into lexical, syntax or semantic analysis, intermediate representation, or the full pipeline.

4.6 Recent work on compiler education. Two recent works, listed in the references, address visualization in compiler education:

- *VisOpt – Visualization of Compiler Optimizations for Computer Science Education* (SIGCSE TS '25).
- *Trace-Based Bytecode Interpreter Visualization for Compiler Construction Education* (Zenodo, 2025).

Both confirm growing interest in visual approaches to teaching compilers. The titles indicate they focus on specific parts of the topic, namely optimization visualization and bytecode interpretation, rather than the whole pipeline.

V. LITERATURE SURVEY

Compilers are essential components of software engineering, translating high-level source programs into executable machine code through a structured sequence of phases such as lexical analysis, parsing, semantic validation, intermediate representation generation, optimization, and final code emission. Despite their foundational role, the internal mechanisms of compilers often appear abstract and difficult to comprehend when studied solely through textbooks or non-visual command-line interfaces.

A **Compiler Visualizer** addresses this limitation by providing an interactive environment that demonstrates each stage of compilation in a stepwise, observable manner. Users can input source code, examine the generated token streams, visualize the construction of parse trees, review semantic analysis and symbol-table updates, and trace the transformation of intermediate representations. By transforming abstract compiler processes into tangible visual representations, the tool enhances comprehension, facilitates debugging, and supports deeper conceptual understanding.

This approach benefits students, educators, and self-learners by offering a hands-on framework for experimenting with grammars, identifying errors, and exploring scope and symbol management. Additionally, it assists developers working with domain-specific languages or investigating new language design concepts by enabling real-time inspection of both compile-time and runtime behaviors. In essence, the Compiler Visualizer serves as both an educational aid and a practical debugging companion, transforming complex compiler theory into an engaging, accessible, and interactive learning experience.

Key Contributions

- **Comprehensive End-to-End Visualization:** Provides an integrated view of the complete compilation pipeline—from source code input to virtual machine execution—highlighting interactions and data flow between individual stages.
- **Interactive and Optimized AST Representation:** Implements an enhanced layout and interaction model that generates clear, non-overlapping parse trees with full support for zooming, panning, and source-tostructure highlighting.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

- **Assembly-to-Binary Correlation Module:** Introduces a specialized component that maps assembly mnemonics to their corresponding hexadecimal and binary encodings, bridging the gap between humanreadable code and machine-level instructions.
- **Reversible Execution (Time-Travel Debugging):** Integrates a virtual machine capable of state snapshotting and timeline-based navigation, allowing users to step forward and backward through program execution to better analyze runtime behavior.

Comparison of the Literature survey

Feature	ANTLR	JFLAP	Python Tutor	Online IDE (Replit, OnlineGDB)
Lexical Analysis	Partial	Not Present	Not Present	Not Present
Syntax Analysis	Present	Partial	Not Present	Not Present
Semantic Analysis	Not Present	Not Present	Not Present	Not Present
Intermediate Representation	Not Present	Not Present	Not Present	Not Present
Code Generation	Not Present	Not Present	Not Present	Partial
End to End Pipeline view	Not Present	Not Present	Not Present	Not Present
Designed for learners	Not Present	Present	Present	Not Present
Step by step code execution	Not Present	Partial	Present	Present
Memory/ Variable state view	Not Present	Not Present	Present	Present

VI. SYSTEM ARCHITECTURE

Compilers are essential components of software engineering, translating high-level source programs into executable machine code through a structured sequence of phases such as lexical analysis, parsing, semantic validation, intermediate representation generation, optimization, and final code emission. Despite their foundational role, the internal mechanisms of compilers often appear abstract and difficult to comprehend when studied solely through textbooks or non-visual command-line interfaces.

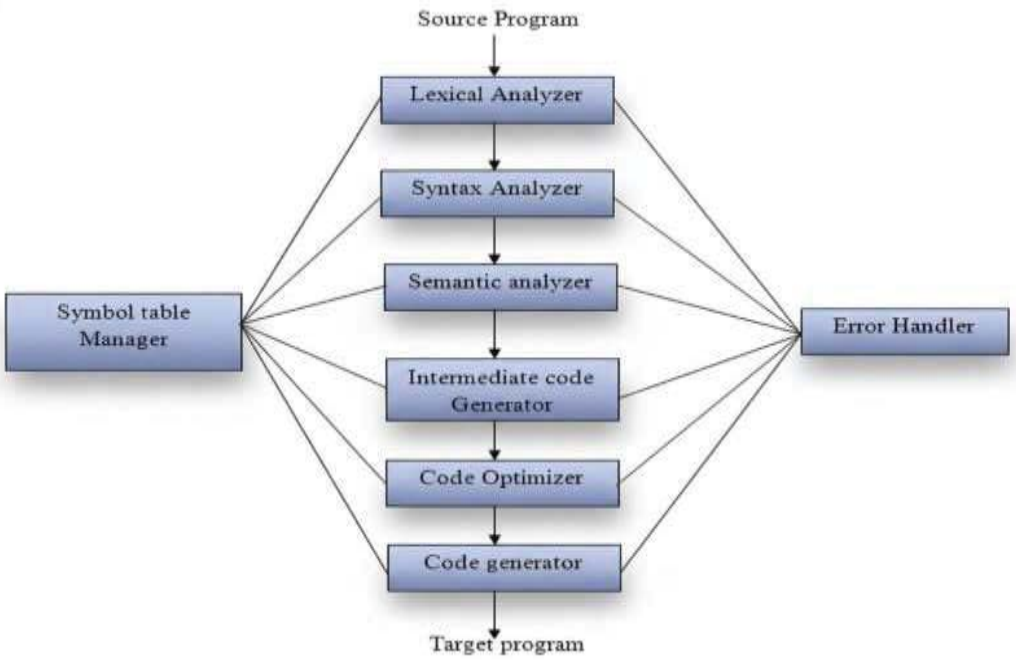
6.1 Compiler Visualizer Software

A **Compiler Visualizer** addresses this limitation by providing an interactive environment that demonstrates each stage of compilation in a stepwise and observable manner. Users can input source code, examine the generated token streams, visualize the construction of parse trees, review semantic analysis and symbol-table updates, and trace the transformation of intermediate representations. By transforming abstract compiler processes into tangible visual representations, the tool enhances comprehension, facilitates debugging, and promotes deeper conceptual understanding.

This approach benefits students, educators, and self-learners by offering a hands-on framework for experimenting with grammars, identifying errors, and exploring scope and symbol management. Furthermore, it assists developers working with domain-specific languages or exploring new language design concepts by enabling real-time inspection of both compile-time and runtime behaviors. In essence, the Compiler Visualizer serves as both an educational aid and a practical debugging companion, transforming complex compiler theory into an engaging, accessible, and interactive learning experience. **Compiler Diagram**

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026



VII. METHODOLOGY

The proposed Compiler Visualizer adopts a phase-wise decision-oriented architecture that models the internal stages of a compiler and represents them through real-time visualization. The methodology is structured around five major components: input acquisition, compilation phase processing, decision framework, visualization, and feedback.

1. **Input Acquisition:** The system accepts source code from the user through an integrated editor. Along with this, grammar rules, symbol definitions, and session parameters are retrieved to initialize the compilation pipeline.

2. **Compilation Phase Processing:**
   - i. **Lexical Analysis:** The code is decomposed into tokens. Invalid character sequences are flagged as lexical errors.
   - ii. **Syntax Analysis:** Using parsing techniques, a hierarchical parse tree is generated. Any grammar mismatches are identified as syntax errors.
   - iii. **Semantic Analysis:** Logical consistency is ensured by validating type rules, declarations, and scope. Violations at this stage are reported as semantic errors.
   - iv. **Intermediate Code Generation (ICG):** A machine-independent representation such as three-address code is generated. Errors at this stage indicate failed translation.
   - v. **Optimization:** Redundant instructions are removed and code efficiency is improved, though this remains an optional phase within the project scope.
   - vi. **Target Code Generation:** Pseudo-assembly or simplified binary code is generated for visualization.

3. **Decision Framework:** At each phase, the system evaluates the success or failure of the operation. If an error is encountered, the pipeline halts and the error is immediately visualized with line-specific feedback. If no errors occur, the process transitions seamlessly into the next phase. Successful execution across all phases results in a complete compilation flow.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

4. **End Process and Iteration:** The compilation process concludes with either a successful target code visualization or an error report. Users may iteratively edit the code and rerun the process.

7.1 Lexical and Syntax Analysis

The lexical analysis phase breaks down the input source code into a sequence of tokens using predefined regular expressions. Each token is identified by its type, value, and position within the source code. The detailed token specifications are presented in **Table 2**.

7.1.1 Regular Expression Specification for Lexical Tokens

Token Type	Regular Expression	Example
Keyword	\b(int|return|if|else)\b	int, if
Identifier	[a-zA-Z_]\w*	main, x, result
Integer	\d+	5, 10
Operator	==|<=|>=|!=|&&|||[+\-*/<>=]	>=
Punctuation	[(){};,]	(, }

In Table 7.1.1 Regular Expression Specification for Lexical Tokens, a recursive descent parser processes the token stream to build a hierarchical Abstract Syntax Tree (AST), which represents the grammatical structure of the code. To ensure the AST is readable, an adaptive layout algorithm is used. The algorithm first performs a post-order traversal to calculate the required horizontal width of each subtree based on its content, preventing node overlap. It then performs a pre-order traversal to assign final coordinates, resulting in a clean, aesthetically pleasing visualization.

7.2 Virtual Machine and Instruction Set The target for the compiler is a custom stack-based Virtual Machine (VM) designed for simplicity and ease of analysis. The VM features an instruction pointer, a call stack for managing function calls, and separate memory segments for global and local variables. The custom assembly language, or instruction set, is detailed in Table 4.2.1. **4.2.1 Instruction set for Custom virtual machine**

Mnemonic	Opcode (Hex)	Argument	Description
STORE	$0x10$	dest, src	Copies value from dest to src
EVAL	$0x20$	dest, a, op, b	Computes $a \text{ op } b$ and store the result in dest
JUMP_IF_FALSE	$0x30$	cond, target	Jump to instruction address target if cond is 0
JUMP	$0x31$	target	Unconditionally jumps to instruction address target
PUSH_ARG	$0x40$	value	pushes the value onto argument buffer for a function call
CALL_OP	$0x50$	func, dest	calls function func and stores the return value in dest
RETURN	$0x60$	value	Ends the current function, returning value
LABEL	$0x70$	name	Marks a location in the code with a name

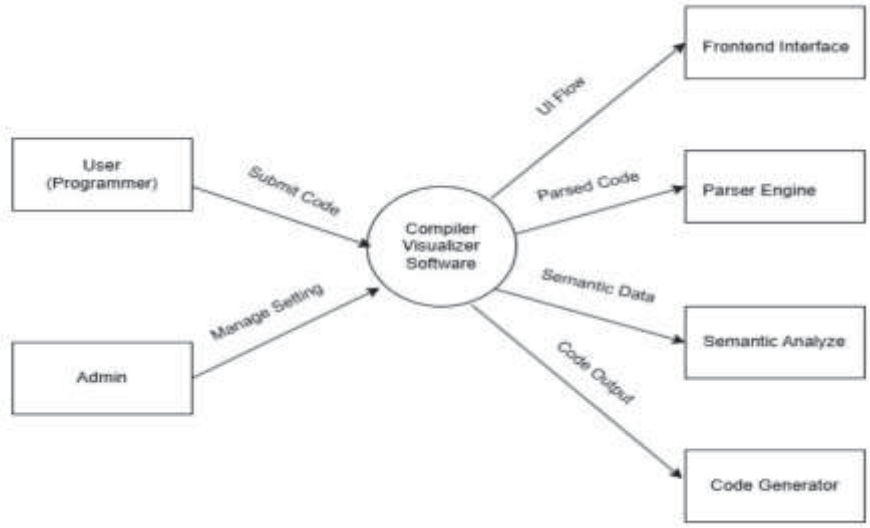
(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

Each assembly instruction is encoded into a specific byte format for execution by the VM. This encoding is made visible in the Assembly ↔ Binary mapping view. For example, the EVAL instruction uses a 17-byte format, as detailed in Table 4.2.2

7.2.2 Binary Instruction Format

Bytes	Field	Data Type	Description
0	Opcode	uint8	$0x20$ for the EVAL instruction.
1-4	Destination ID	uint32	32-bit hash of the destination variable name.
5-8	Operator ID	uint32	32-bit hash of the operator string (e.g., "+").
9-12	Operand A ID	uint32	32-bit hash of the first operand's name/value.
13-16	Operand B ID	uint32	32-bit hash of the second operand's name/value.



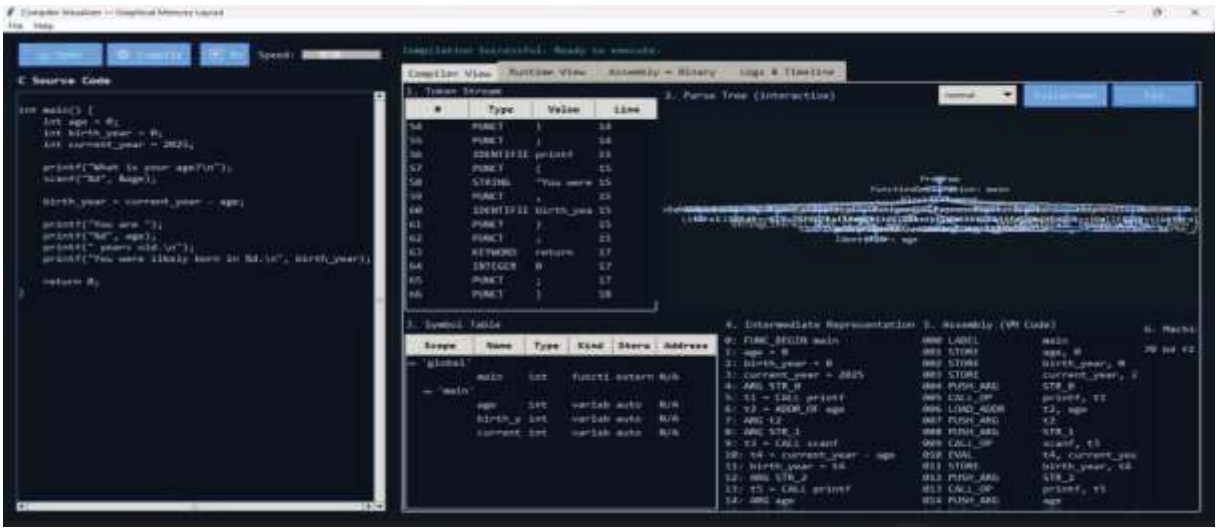
To enable the time-travel debugging feature, the VM's entire state (instruction pointer, call stack, and memory) is captured as a snapshot after each instruction is executed. These snapshots are stored in a list, allowing the user to move the timeline slider to restore the VM to any previous state, providing a powerful mechanism for analyzing program flow.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

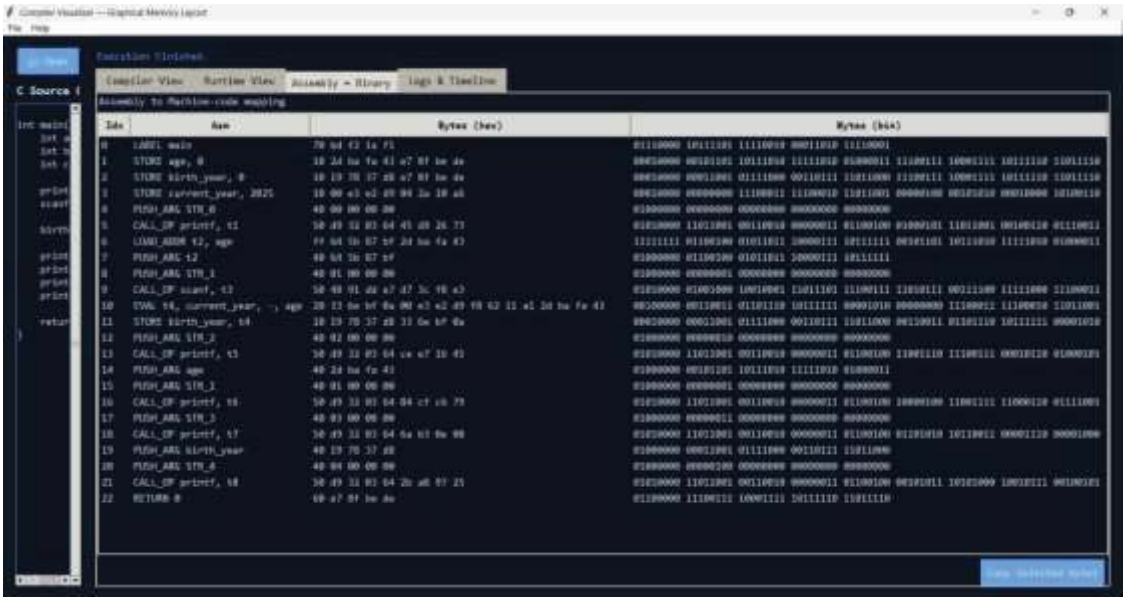
VIII. RESULTS AND DISCUSSION

To demonstrate the functionality of the Compiler Visualizer, a case study was performed using a C program designed to calculate the square of a user-provided integer. The tool's output provides a comprehensive view of both the compilation stages and the program's runtime behavior.



8.1 Compiler phase view

Figure 1 shows the main interface after the successful compilation of the sample program. The screen is logically divided, presenting the user with multiple, simultaneous perspectives. The source code editor is on the left, while the right panel contains tabbed views for each compiler artifact. The "Compiler View" tab is active, displaying the Token Stream (1), the interactive Parse Tree (2), the Symbol Table (3), the Intermediate Representation (4), and the final VM Assembly code (5). This unified display allows a learner to directly trace a variable like square from its declaration in the source code to its representation in the symbol table and its usage in the assembly instructions.

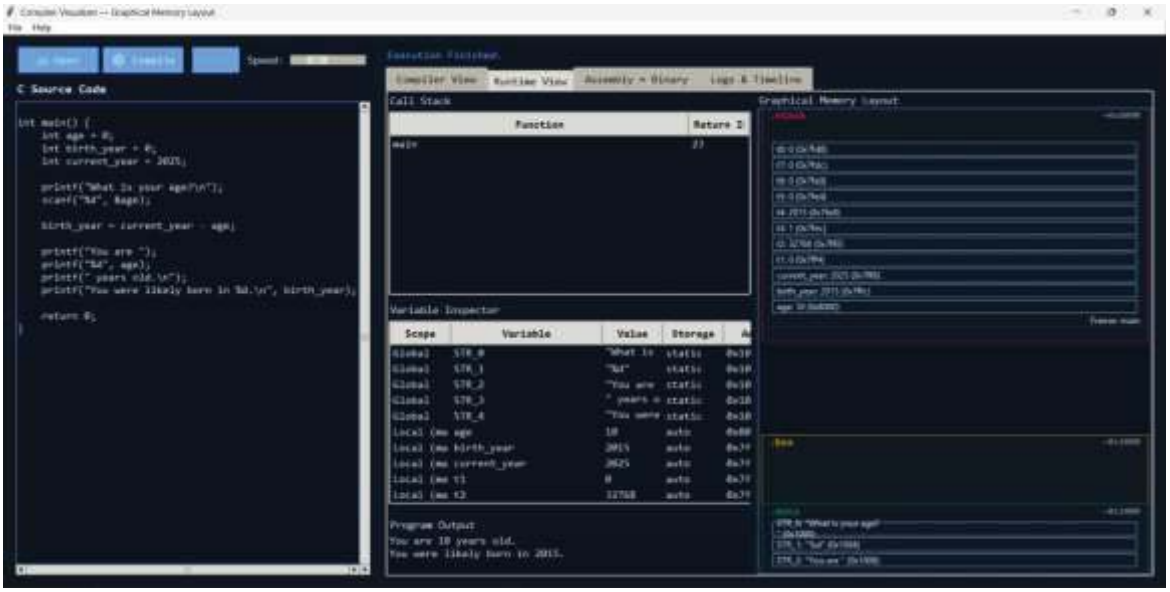


(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

8.2 Assembly to Binary

Figure 5.2 displays the "Assembly ↔ Binary" view. This table is a key pedagogical feature of the tool, as it demystifies the final code generation step. Each assembly instruction generated in the previous stage is listed alongside its raw machine code representation in both hexadecimal and binary formats. For instance, the CALL_OP instruction at index 4, which invokes the printf function, is shown to be encoded as the byte sequence 50 00 00 00 05 b5 b5 b5. This direct mapping makes the abstract concept of machine code tangible and understandable.



8.3 Runtime View

Figure 5.3 illustrates the "Runtime View" after the program has been executed. This view provides insights into the program's dynamic behavior. The Call Stack panel shows that the main function has completed, returning at instruction 15. The Variable Inspector lists all local variables and their final values and memory addresses at the end of execution. Finally, the Program I/O Console displays the interaction with the user, confirming the program's correct output. By observing these panels during step-by-step execution, a student can understand how function calls are managed and how variable values are updated in memory over time.

Collectively, these visual outputs provide a rich, multi-faceted learning experience. The tool successfully breaks down the "black box" of the compiler, allowing users to see the direct consequences of their code at every stage of the process, from syntax to execution.

IX. CONCLUSION

The proposed Compiler Visualizer provides a structured, interactive, and educational approach to understanding compiler design. By integrating a user-friendly frontend built with Python/Tkinter and a custom compiler backend, the system bridges visualization with processing. Each phase of compilation is systematically represented, enabling students to trace the transformation of source code in real time. The methodology emphasizes decision-based error handling, ensuring that users not only see where a process fails but also understand why. The visualizer thus stands as a valuable resource for academia and self-learners, promoting deeper comprehension and practical skill development in compiler construction.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

9.1 Future Work

The modular architecture of the visualizer supports future scalability and enhancements. Future work will focus on three key areas. First, the C language support will be expanded to include more complex features such as loops, arrays, and pointers. Second, a dedicated visualization module for compiler optimizations will be added to illustrate techniques like constant folding and dead code elimination. Finally, a formal user study will be conducted with computer science students to quantitatively measure the tool's pedagogical effectiveness and gather feedback for further refinement.

X. REFERENCES

[1] The "Dragon Book": Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools. Addison-Wesley.

[2] T. Parr and R. Quong, "ANTLR: A Predicated-LL(k) Parser Generator," Software: Practice and Experience, vol. 25, no. 7, pp. 789-810, 1995.

[3] S. H. Rodger, E. Wiebe, K. Min, K. C. O'Neal, and C. Rasala, "Increasing the Use of JFLAP in Courses," in Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11), New York, NY, USA: ACM, 2011, pp. 19-20.

[4] P. Guo, "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education," in Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE '13), New York, NY, USA: ACM, 2013, pp. 579-584.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed. Boston, MA, USA: Addison-Wesley, 2007.

[6] K. L. K, et al. (2025). "VisOpt – Visualization of Compiler Optimizations for Computer Science Education." In Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE TS '25).

[7] (Author, 2025). "Trace-Based Bytecode Interpreter Visualization for Compiler Construction Education." Zenodo. (July 2025).

(This work is licensed under a Creative Commons Attribution 4.0 International License.)