Received: 12/06/2026, Accepted: 31/07/2026

Visual Educational Compiler: An Interactive Web-Based Tool for Visualising Compiler Design Phases

Dhivya Shree A, Dharshana R, J Jabez, Dhejasree R, Devanshi S, Dharsini P S, Christy J

Department of Computer Science and Engineering, Sathyabama Institute of Science and Technology, Chennai, India

dhivyashreea958@gmail.com, dharshana06012006@gmail.com, jabezme@gmail.com, dhejasree6@gmail.com, deva.siddi16@gmail.com, maluapk@gmail.com, christyfransica@gmail.com

Abstract

The Visual Educational Compiler is designed to address the gap between theoretical understanding and practical experience in compiler design. Although students typically learn the roles of the lexer, parser, abstract syntax tree (AST), semantic analyser, and code generator, their understanding is often limited to conceptual diagrams and theoretical descriptions. The proposed system provides a visual, step-by-step representation of the compilation process, enabling students to observe the transformation of source code into tokens, an AST, semantic representations, and, ultimately, executable code. The system accepts programs written in a simplified teaching language that supports variables, integer arithmetic operations, conditional statements, and print functions. The compilation process is implemented in Python and consists of five phases: lexical analysis, parsing, AST construction, semantic analysis, and code generation. The final phase generates equivalent Python code. Each compilation phase is presented through a dedicated tab in the Streamlit-based interface, allowing students to inspect intermediate artifacts and compare outputs across different stages of compilation. Errors are reported by phase, enabling students to understand the specific responsibilities of each compiler component. Deliberate errors, including missing semicolons, undeclared variables, and type-related errors, were incorporated to demonstrate phase-specific error detection. The system was evaluated using ten programs of increasing complexity, along with their corresponding erroneous versions. The results showed that all syntactically valid programs successfully progressed through the complete compilation pipeline, while the introduced errors were detected at the appropriate compilation phases and accompanied by relevant diagnostic messages. The teaching language was intentionally designed with limited functionality to maintain transparency of the compilation process and avoid obscuring individual compiler phases within complex code. Future work will focus on extending the language capabilities and incorporating a dedicated code optimisation phase.

Keywords: *Compiler Design, Educational Tool, Lexical Analysis, Abstract Syntax Tree, Visualisation, Python, Streamlit*

1. Introduction

Most undergraduates' compiler courses at our university follow the same approach that was used a decade ago: one chapter per phase from a book, one diagram drawn by hand on the blackboard, and one assignment in which students have to produce another grammar in Bison. The result is that the student will learn five different descriptions of how the pipeline works, but will not get a chance to see their own code transformed through the entire pipeline. That gap between the way the material is presented and the real-life process is what this project aims

*Corresponding author: Dhivya Shree A, Department of Computer Science and Engineering, Sathyabama Institute of Science and Technology, Chennai, India. (dhivyashreea958@gmail.com)

to address.

Compilers, as a matter of fact, are far from being a rare case. Any line of Python code, any program written in Java or class files compiled in C++ that the student produces in other classes was compiled by a compiler. And understanding how the compilation process works, how the sequence of symbols transforms into tokens, tree, validation, and finally into another language altogether, gives students an insight into an important part of computer science that will become relevant in almost any course of their studies. These aspects are covered in the traditional explanation in the textbook (Louden, 2003), which is considered the most frequently cited reference by professors in our department when teaching students. However, this textbook does not explain what the pipeline process feels like. Diagrams depict different stages statically; there is no depiction of the moment when a parse tree is transformed into an AST, or of how an undeclared variable is detected by the semantic analyser.

This issue is discussed in detail in the educational compiler literature. Mernik & Žumer (2003) is one of the first works to pay attention to this aspect, and it even suggested interactive tools as the solution, since students using the LISA tool performed better than those attending only lectures in the same course. Yet the visualisation offered by LISA was neither web-based nor animated in the contemporary sense. In Hiranishi & Hattori (2020), this task was accomplished successfully, although the pipeline in the article stops at the parse tree stage. Hiranishi et al. (2022) then deployed a web-based compiler in real classrooms and conducted formal evaluation studies, reporting that visual treatment of compiler phases improved both understanding and student enjoyment. Their system, like its predecessor, withheld intermediate representations such as the AST. Vírseda, 2023 took the opposite approach, starting from an implementation the student already has and walking them back into the theory behind it. Stamenković & Jovanović, 2023 independently arrived at the same motivation and reported that the interactivity of the visualisation layer, not merely its content, carried the learning gain in their own controlled trials. The shape of the gap is therefore consistent across the literature. Plenty of tools cover one compiler phase well. A smaller set provides visualization but no interactivity. A few comprehensive platforms exist, but require substantial prior background to use. What does not appear to exist is a lightweight, browser-based tool that exposes the *entire* pipeline lexer via code generation, behind an interface a first-year student can sit in front of and use without a tutorial. The Visual Educational Compiler was built to close that gap. Built using Python on top of Streamlit (Streamlit Inc., 2024), this takes input programs in a specially designed teaching language with a very limited syntax that includes only variables, integer arithmetic, conditional statements, and print, and displays the output for each step in its own tab, thereby allowing the transformation of a source program from plain text to Python code to be seen all within one web page.

The remainder of this research paper proceeds as follows. Section 2 reviews related work on educational compilers and visualisation tools. Section 3 introduces the proposed tool and its design choices. Section 4 describes the layered system architecture in detail. Section 5 presents the test programs used to verify each phase and discusses the verification results. Section 6 closes with directions for follow-up work.

2. Literature Survey

Teaching compiler construction has always been tough, and people in computer science education have long known this. Textbooks walk students through the standard phases, but for anyone who has sat through a compiler course, the gap between reading about a parse tree and actually seeing one work is enormous.

One of the earliest attempts to close that gap came from Mernik and Žumer (2003), who built an educational compiler tool and tested it with students who had received only lecture-based

instruction. The results were striking: students using the interactive tool performed considerably better at virtually every stage of compilation. Their work is still frequently cited whenever someone argues for visualization in compiler education. The catch is that the tool itself was designed for a desktop environment with no interactive visualization and certainly no browser-based access. For modern students who expect to pull up learning material on a laptop or phone, that puts a real ceiling on how useful it can be. A web-based approach was taken up next by Hiranishi and Hattori (2020), who introduced a visual compiler aimed squarely at beginners with limited computing backgrounds. The premise was simple: if the tool runs in a browser, more students can use it without installing anything, and this, on its own, could prove useful for learners who could never get over the idea of compiler concepts due to the difficulties of the compilation process. The strategy worked, at least in terms of making the barrier to learning easier. But the pipeline presented above lacked the semantic analysis step and an illustration of the abstract syntax tree. Hence, while the tool addressed the accessibility problem, it did not provide students with a visual representation of what occurs during the latter stages of compilation.

The earlier research was expanded upon by Hiranishi et al. (2022). They conducted a formal assessment of the compilation phases using the same browser-based compiler as before, but in actual university settings. Indeed, their study verified the hypothesis suggested before: students, when presented with visual illustrations of the compilation phases, had a better understanding of the material and enjoyed learning it. However, even this tool does not provide the entire pipeline: intermediate representations between stages are not shown to students.

In addition to these tools used for education purposes, the body of work on compiler development relies on foundational textbooks. (Louden, 2003) It is one of those; it offers a rigorous theoretical analysis of each major phase of a compiler. The book is dense, accurate, and popular in college classes. What the book lacks, however, is interactivity. There are no examples to work through, no figures to interact with one's own program code. In short, Louden provides formal knowledge without the practical feeling of how token lists turn into trees.

More recent attempts at visualizing compilers, including (Vírseda, 2023), have taken the interaction to a whole different level by providing more interactive tutoring. This line of research is consistent with another trend in the area – any tool developed is better in some ways than others, but still leaves some things undone in the compilation process.

In conclusion, the current state of the art offers nothing ideal for classroom use. There are many tools that focus on particular phases of compilation. There are tools that visualise compilation, but don't offer visualisation within a browser. Some are more complex than those who will be using them for the first time. It is impossible to build a unit that enables students to input their programs and see how it goes through tokenisation, parsing, Abstract Syntax Tree (AST) construction, semantic checking, and code generation, all on one page, with each step represented by a separate tab. This gap should be addressed in the Visual Educational Compiler.

3. Proposed Work

The gap can be easily described, but harder to fill. There are no tools that allow a student to enter their program in a browser, watch it compile, and see all phases presented separately. Only some tools installed on the computer work in the browser; some only parse, and others immediately generate code from tokens, skipping all steps in between. The Visual Educational Compiler is developed specifically for that purpose, and all these processes are intended to take place on a single web page. The goal is not to learn how to construct compilers at the level expected of a graduate student, but to show how they really work.

It was made specifically to cover that exact need in a single webpage. This is not meant to be a class for teaching how to build a compiler. It is meant to help a person taking a basic class on

compilers or programming languages understand what happens in a compiler when processing a specific piece of source code.

The user simply performs several actions on the frontend side. He or she writes a bit of source code in a custom, small language used for educational purposes (with explicit types, control flow, and some arithmetic operations). After that, he or she selects a certain phase from the tab interface above the editor. The program processes the entered code up to this phase and displays its output as a table of tokens for the lexer, a parse tree for the parser, an abstract syntax tree for the parsed code, a table of symbols and any errors from semantic analysis, and finally, either three-address code or a stack-based code generated by the code generator.

A few choices shaped the design:

- **Single-page, no install.** Everything runs client-side. A student in a lab computer, on their phone between classes, or on a borrowed laptop, can open it the same way. No environment to set up, no compiler toolchain to install, no editor extensions.
- **Each phase is isolated.** Rather than dumping all intermediate outputs into a single scrollable view, each phase has its own tab. This matters because compiler course material is usually taught in phases. Lesson 3 is about lexing. Lesson 4 is about parsing. Having the visualization mirror that pacing turns out to be useful in practice.
- **Intermediate representations exposed.** Token stream. Parse tree. AST. Symbol table. Intermediate code. The plan is to show each one, not just the final output. Half the point of a compiler course is the transformation from one representation to the next, and that is exactly what most other tools hide.
- **Errors highlighted, not buried.** Lexical errors, syntax errors, and semantic errors all need to surface visibly with a line and column reference pointing back into the source. A student running into a syntax error should learn from it, not be confused by it.

Underneath, the frontend connects to a lightweight backend that performs the actual compilation. The lexer is built as a hand-written scanner with clear classification of each token category. The parser is a recursive-descent implementation, chosen because the small custom language used here is amenable to recursive descent and because the parser's structure mirrors the grammar in a way students can follow. Semantic analysis is a typed, scope-aware pass with a symbol table built from scratch. Code generation produces a stack-based virtual machine instruction set, keeping target machine concerns out of the teaching picture.

The pedagogical aim, to put it plainly, is that after using the tool for one session, a student should be able to point at any phase of compilation and explain, in their own words, what is happening there. If they cannot, the visualization has not done its job. That standard is borrowed from the broader literature on visualization in CS education, where interactive tools have consistently been shown to improve conceptual understanding, but adapted here for a tool that runs the full pipeline rather than a slice of it.

4. Architecture

The entire compiler is organized into three layers placed on top of each other: Input, Processing, and Visualization. Each layer performs specific work and passes its output to the next. Such a structure will ensure scalability of the system in the future because if the parsing process requires modification, there will be no need to modify the input layer and vice versa, while the visualization layer will work properly as long as it receives data of any form. Input Layer. This layer serves as the user interface and is used by the student to write code into. The text field is located at the very top of the screen and all entered code goes into the rest of the compiler. As soon as the code is received, this layer performs preliminary cleanup actions. All tabs are

replaced with spaces, trailing blanks are removed and consecutive spaces are merged. This is not the compiler's work; it's just some preprocessing before control passes to the next part. Processing Layer. The following five stages are performed here in the order shown.

- The lexical analyser analyses the parsed source code file character by character and divides the source code file into tokens. Each token is a data structure containing information such as its type (identifier, keyword, symbol, etc.) and its value (the text used to generate the token). The output from this phase will be a linear sequence of tokens generated from the source file.
- That stream is handed to the Syntax Analyzer, which checks the tokens against a hand-written context-free grammar for the custom teaching language. If the tokens line up with the grammar rules, the analyzer builds a parse tree showing how each token fits into the grammatical hierarchy. If they don't, compilation stops right there with a syntax error pointing at the offending line.
- The AST Builder walks the parse tree and trims away anything that doesn't carry semantic weight. Punctuation nodes, grouping symbols, intermediate grammar productions, and all the structural scaffolding that the parser needed, but a later stage doesn't get dropped. What remains is the AST, a much cleaner tree that represents the actual structure of the program's logic.
- Then the Semantic Analyzer does the type-checking and scope-checking. It walks the AST top to bottom, builds a symbol table as it goes, tracks what's been declared and in what order, and flags problems such as using a variable before declaring it, mixing an int with a string, and applying an arithmetic operator to the wrong kind of value. Any issues are added to an error list. The system doesn't abort on the first error; it tries to find as many as possible in one pass because, for a student, getting three warnings at once is more useful than fixing them one at a time over three reruns.
- Finally, under the assumption that the AST was found to be clear, the Code Generator traverses it and produces equivalent Python source code. This is not an optimized code; this is not supposed to beat a real compiler in terms of optimization. This is for the student to see how his/her original input correlates with the Python code.

Visualization Layer. This is what the student looks at. Streamlit creates a tabbed interface, one tab for each phase: Tokens, Parse Tree, AST, Semantic Errors, and Generated Code. Every tab displays only the results of its own phase, without any interference from the results of other phases. Tab switching occurs immediately because all intermediate results are stored in memory after each phase's execution. Tree structures are shown in a browser as expandable nodes rather than with nesting, which allows wide trees to fit within the screen width more easily. Errors are shown right next to the corresponding place in the input. The layout is responsive and adjusts depending on whether a user works on a laptop or a smaller screen.

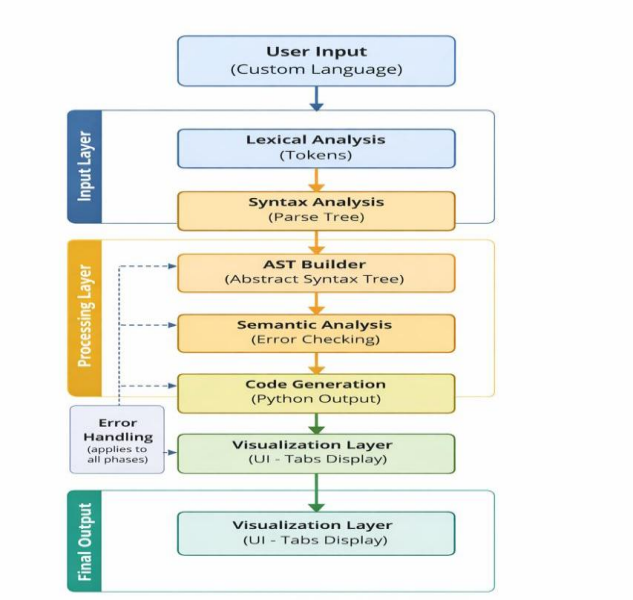


Fig 1 System Architecture of Visual Compiler

Combining all three layers, the system becomes a one-page web app: a student types, presses a button, and within a second or two, every phase has produced its output and is sitting in its own tab.

5. Result and Discussion

There was one immediate problem facing the Visual Educational Compiler: Would each step of compilation really produce meaningful output when fed with actual program code? The testing process revolved around answering this question. Ten programs were created for the system to be tested on; each was intended to perform some computation and was selected so that each step of the compilation pipeline would be put through its paces with something nontrivial. Programs for the purpose of failing each stage of compilation were also developed; no semicolon to test the parser, an undefined variable for the semantic analyser, and the addition of int to a string. It is as important to test failure scenarios as to test success scenarios, because the tool is primarily about what happens when something fails.

In all phases performed as required within the test set, nothing unexpected occurred worth mentioning on an individual basis. The failures encountered during testing were all detected by the phase expected to detect them. All tokens were classified correctly by the lexer. The parser generated grammatically valid parse trees for valid input and gracefully stopped upon encountering syntactically invalid input. The abstract syntax tree builder removed the scaffolding of the parse tree without altering its structure. The semantic analyzer caught all the errors intentionally made in the code, with sufficiently descriptive messages that would allow any student to know which line and which column was responsible for the error just by reading the message. Syntactically valid Python code was generated by the code generator for all ASTs after being semantically analyzed, and the generated code, when executed, gave the same output as the original program.

This is not very interesting; any correct compiler is the bare minimum required for being called such. What is more interesting is how the tool treats the outcome. Figure 2 takes this further by showing the representation of nested arithmetic expressions at each step of the abstraction process, along with the generated Python code. Figure 3 shows the error message: the user's input contains a syntax error, which the system highlights along with the line and column numbers.

Whether this tool is used as a teaching aid or a debug console depends entirely on how the tabbed interface is used. The output of each stage, Tokens, Parse Tree, AST, Semantic Errors and Generated Code, is displayed separately in different tabs, and switching between tabs happens instantly because everything is stored in memory rather than having to be generated each time. This aligns well with how a compiler course works, since a lecture usually centres around a particular stage. The student learns about the stage, watches how things change if they alter identifiers or operators, and moves on. The educational literature on compiler visualization has been pointing out precisely this advantage of phase separation (Mernik & Žumer, 2003), and the wider field of web-based research in this domain demonstrates how the reduction of the entry barrier, an inherent part of browser tab UI, attracts potential users that otherwise would have failed to proceed beyond the installation step (Hiranishi & Hattori, 2020).

Error reporting requires special attention because it is where the application's teaching power manifests. If the student deliberately introduces a syntactic error into the program code, the parser will report its location with sufficient precision for the student to spot the mistake in their own code. Semantic errors are reported with the offending line and a brief description of what was expected. Type mismatches point at both operands, so the student can see which two values conflicted. The earlier web-based work by Hiranishi and colleagues placed scale at the forefront of the discussion, demonstrating that visual phase rendering improved student comprehension and subject enjoyment in classroom settings (Hiranishi et al., 2022), and that finding carries over here. What this tool adds is that errors are surfaced as part of the visualization rather than relegated to a separate log panel, so the student does not have to switch contexts to learn from a mistake.

Some of the design choices that pay off here were not flashy. Streamlit was used for the interface (Streamlit Inc., 2024), which kept the focus on the compiler itself rather than on building a custom frontend in JavaScript or React. The whole system runs as a single-page application. Tables, expandable tree nodes, and code blocks render natively. This is worth noting because prior work in this space has at times been narrowly limited by the platform’s students had available (Vírseda, 2023), and a tool that runs in a browser side-steps that constraint entirely.

There are limits, and they are worth being clear about. The custom language used here supports variable declarations, integer arithmetic, conditional branching, and print statements - enough to drive every phase of the pipeline through real input. It does not support functions, user-defined types, arrays, recursion, loops beyond a basic conditional form, or anything closer to procedure calls or higher-order constructs. The grammar for what is supported is deliberately small so that the parse tree and AST remain legible on a single screen without scrolling. A bigger grammar means a bigger tree, and a tree that goes off the page is worth less than a smaller one. This is a decision about scope, not a restriction in the design. Expanding the grammar is a matter of adding new productions and new parser and AST nodes; no changes will be required for the visualization level.

In addition, there is the issue of optimization being supported by the tool. It is not supported. The resulting Python code is just a proper translation and not a properly optimized one. In an introductory course on compiler design, this is the right choice. Optimisation adds extra steps to the process (constant folding, dead-code elimination, register allocation, etc.), which should be covered in an advanced compiler course, leaving less room for the tool's core message to be taught. In case the tool is expanded beyond its introductory scope in the future, the optimization passes can be simply added as new tabs without changing the visualization of the phases already implemented. By the time the test run has finished after executing the tenth program, all phases are complete, all errors are caught where they belong, and the core claim of the

system holds. A beginner can launch the software, write some lines of code, click a button, and immediately get a full picture of the pipeline opened to view tab by tab. It is this feature that turns it into a functional learning tool.

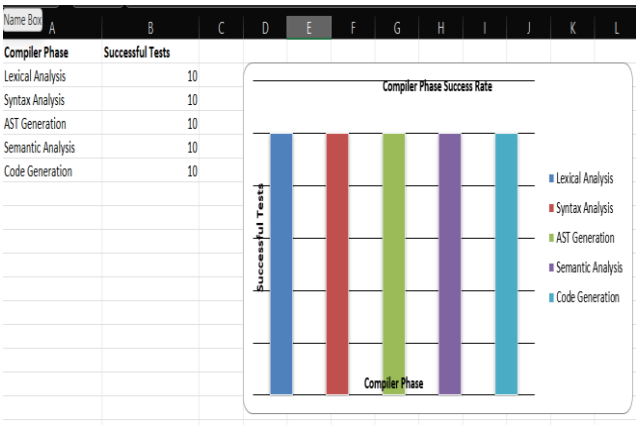


Fig. 2. Successful Execution of Compiler Phases

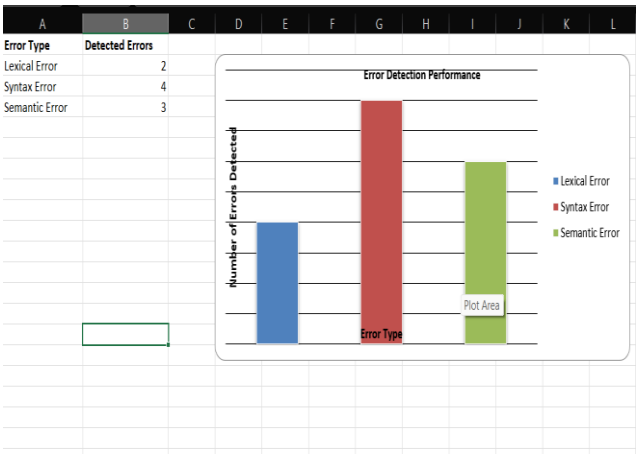


Fig. 3. Error Detection Performance

6. Conclusion & Future Work

An Educational Compiler was created, successfully tested with ten different programs of varying complexity, and proven to guide a student through every stage of the process on a single browser screen. All five phases, i.e. lexical analysis, syntax analysis, Abstract Syntax Tree creation, semantic checks, and Python code generation, are performed consecutively, each phase presented on its own tab and errors highlighted inline rather than hidden away in a log file. This is what the system does.

What existing tools do not, in our judgement, do is hold all five phases together without dropping any. Early educational compiler work showed the value of phase-specific interactivity, but the tools ran on desktops and stopped at one or two stages. The web-based visual compilers that followed made installation unnecessary, lowering the bar for beginner learners, but they typically skipped the back half of the pipeline. Later classroom studies confirmed that phase-level visualization improves comprehension and subject enjoyment, while still leaving intermediate representations hidden. Theoretical references, including the standard compiler textbook, cover the material comprehensively but offer nothing interactive at all. The Streamlit-based framework sidestepped a different problem entirely - that of access

across devices - by making the entire tool a single-page web app. Each step forward in the literature has tended to fix one thing and leave something else broken. The contribution here is modest, and we want to be honest about that. The tool performs the entire pipeline. It doesn't bring in any innovation of technique, optimization, or algorithmic approach. It provides completeness across all the stages available in one tool, and an extremely valuable feature that goes unspoken but makes a great difference: it works on a cell phone, a lab computer, or a borrowed laptop, as long as there's a web browser available.

There are several practical consequences to consider. First-year students studying compilers and programming languages can sit down, write a couple of lines, and get the result of lexing, parsing, abstract syntax tree creation, and Python code generation in less than two seconds. A common frustration in such courses is that students pass exams on compiler phases by reproducing diagrams from slides, and then struggle to explain the same phase when asked to point out their own program. The point of an interactive pipeline is to close that specific gap. Whether it actually does, in classroom settings, with measured outcomes, is the next step. We have not run that study yet. We have validated the tool's behavior on test programs; we have not yet validated its impact on student learning in a controlled setting.

Future work moves in three directions. The first is expanding the source language beyond its current scope variables, integer arithmetic, conditionals, and print statements to include functions, basic data structures, and loops. The second is adding basic optimization passes (constant folding, dead-code elimination) as additional visible phases, since optimization is the natural next step once the basic pipeline is understood. The third type, and the one that is more interesting for us, is a formal evaluation that involves deploying the tool in real-life compiler classes, comparing learning outcomes between a treatment group and a control group, and presenting the results. Such research has already been conducted for partial pipelines. However, as far as we know, no such study has been performed for the whole pipeline from lexical analysis to code generation. This is how the research goes on.

7. References

[1] Hiranishi, H., Imai, Y., Podržaj, P., Ohno, A., & Hattori, T. (2022). Application of Web-based Visual Compiler to Computer Education. *IEEJ Transactions on Electronics, Information and Systems*, *142*(3), 389-394.

[2] Stamenković, S., & Jovanović, N. (2023). A web-based educational system for teaching compilers. *IEEE Transactions on Learning Technologies*, *17*, 143-156.• Louden, K. C. (2003). Compiler Construction: Principles and Practice. Thomson Learning.

[3] Mernik, M., & Zumer, V. (2003). An educational tool for teaching compiler construction. *IEEE Transactions on Education*, *46*(1), 61-68.

[4] Hiranishi, H., & Hattori, P. P. Y. I. T. (2020, December). Development of Web-based Visual Compiler for Computer Literacy. In *The 6th International Conference on Electronics and Software Science (ICESS2020)* (p. 23).• Streamlit Inc. (2024). Streamlit Documentation.

[5] Khorasani, M., Abdou, M., & Fernández, J. H. (2022). Web application development with streamlit. *Software Development*, *498*, 507.

[6] Khorasani, M., Fernández, J. H., & Mohammad/Abdou Khorasani (Mohame). (2025). *Streamlit for Web Development*.

[7] Khorasani, M., Abdou, M., & Hernández Fernández, J. (2025). Streamlit basics. In *Streamlit for Web Development: Build and Scale Secure Python-Powered Apps with Streamlit* (pp. 31-66). Berkeley, CA: Apress.

[8] Vırseda, R. D. V. (2023). Visualizing Compiler Design Theory from Implementation Through an Interactive Tutoring Tool: Experiences and Results.

[9] Urquiza-Fuentes, J., Manso, F., Velázquez-Iturbide, J. Á., & Rubio-Sánchez, M. (2011,

June). Improving compilers education through symbol tables animations. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (pp. 203-207).

[10] Krebs, N., & Schmitz, L. (2014). Jaccie: A Java-based compiler–compiler for generating, visualizing and debugging compiler components. *Science of Computer Programming*, *79*, 101-115.

[11] Orehovački, T. (2009). Verificator: educational tool for learning programming. *Informatics in Education*.

[12] Grune, D., Van Reeuwijk, K., Bal, H. E., Jacobs, C. J., & Langendoen, K. (2012). *Modern compiler design*. Springer Science & Business Media.

[13] Meduna, A. (2007). *Elements of compiler design*. Auerbach Publications.

[14] Sampaio, A. (1997). *An algebraic approach to compiler design* (Vol. 4). World Scientific.