Table of Contents

Mini Compiler

Abstract

This project presents a Mini Compiler system developed using Python and the Flask web framework. The main goal of this project is to demonstrate the fundamental working principles of a compiler in a simple and understandable way. The mini compiler takes a small custom programming language as input, performs lexical analysis, checks syntax rules, executes arithmetic expressions, and displays either execution results or meaningful error messages. A web-based dashboard is designed using HTML, CSS, and JavaScript to make the system interactive and user-friendly. This project is mainly intended for educational purposes to help understand compiler design concepts.

Introduction

A compiler is a system software that translates high-level programming language source code into machine-level code through several defined phases. These phases analyze, verify, and process the source code step by step to produce correct output.

As shown in Fig. 1.1, the main phases of a compiler include lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation. Each phase performs a specific task, such as token generation, syntax checking, meaning validation, and final code production. This Mini Compiler project implements a simplified version of these phases to help beginners understand the basic working principles of a compiler.

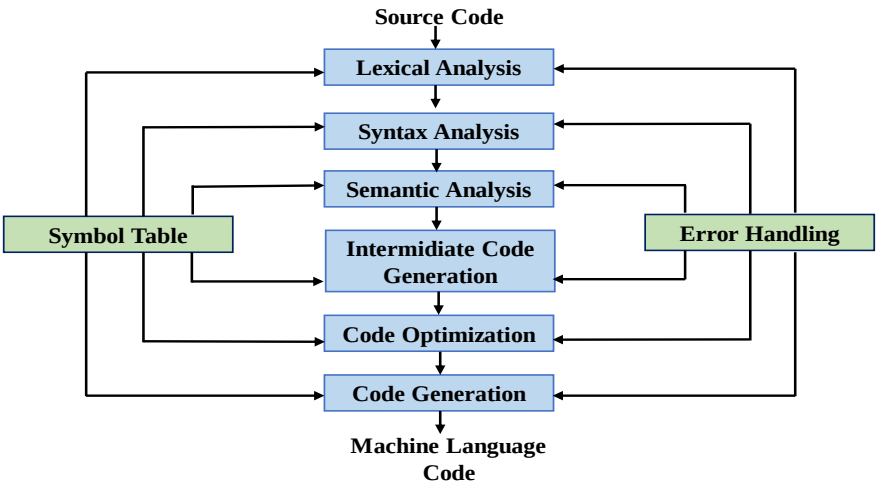


Fig. 1.1: Phases of Compiler

Objectives of the Project

The main objective of this project is to design and implement a simple compiler for learning purposes. This project aims to understand how source code is processed step by step inside a compiler. Another objective is to implement a lexical analyzer that can generate tokens from user input and detect invalid characters. The project also aims to perform syntax checking and runtime execution of expressions. Additionally, creating a clean and interactive web-based interface using Flask and HTML is an important objective of this project.

Supported Language Syntax

The Mini Compiler supports a very small and simple programming language. The language allows variable assignment using the '=' operator and supports **arithmetic operations** such as addition, subtraction, multiplication, and division. Each statement must end with a semicolon (;). The PRINT keyword is used to display values. Variables must be defined before use. Example supported syntax includes statements like assigning values to variables and printing expressions. This limited syntax helps keep the compiler logic simple and easy to understand.

System Architecture

The system follows a clear and sequential architecture. First, the user writes source code in the web interface. The code is then sent to the Flask backend server. The backend passes the code to the Lexer, which performs lexical analysis and generates tokens. These tokens are then processed by the Interpreter, which checks syntax rules, evaluates expressions, and executes print statements. Finally, the execution result or error messages are sent back to the frontend and displayed to the user. This architecture clearly demonstrates the internal flow of a compiler.

Lexical Analyzer

The lexical analyzer is responsible for reading the input source code character by character and converting it into meaningful **tokens.** In this project, the Lexer identifies tokens such as identifiers, numbers, assignment operators, arithmetic operators, parentheses, semicolons, and the PRINT keyword. It also keeps track of line and column numbers to report precise error messages. If an invalid character is found, the lexer generates a lexical error. This phase ensures that only valid tokens are passed to the next stage of compilation.

Interpreter and Execution Phase

The interpreter processes the list of tokens generated by the lexer. It executes **statements** sequentially and stores variable values in a dictionary. When an assignment statement is encountered, the interpreter evaluates the arithmetic expression and assigns the result to the variable. For PRINT statements, the interpreter evaluates the expression and stores the output. If a variable is used before being defined, a runtime error is generated. This phase represents the execution stage of the mini compiler.

Error Handling

Error handling is an important feature of this project. The compiler can detect and report different types of errors. **Lexical errors** occur when an invalid character is found. **Syntax errors** occur when required elements such as semicolons or assignment operators are missing. **Runtime errors** occur when undefined variables are used. **Mathematical errors** are also handled when an expression cannot be evaluated. All error messages include line numbers to help users easily identify and fix problems in their code.

Web Interface Design

The web interface is designed using HTML, CSS, and JavaScript. It consists of two main panels: one for source code input and another for compiler output. The interface is clean, visually appealing, and easy to use. JavaScript is used to send the source code to the backend using the Fetch API. The output panel displays either execution results or formatted error messages. This web-based approach makes the compiler interactive and accessible through a browser.

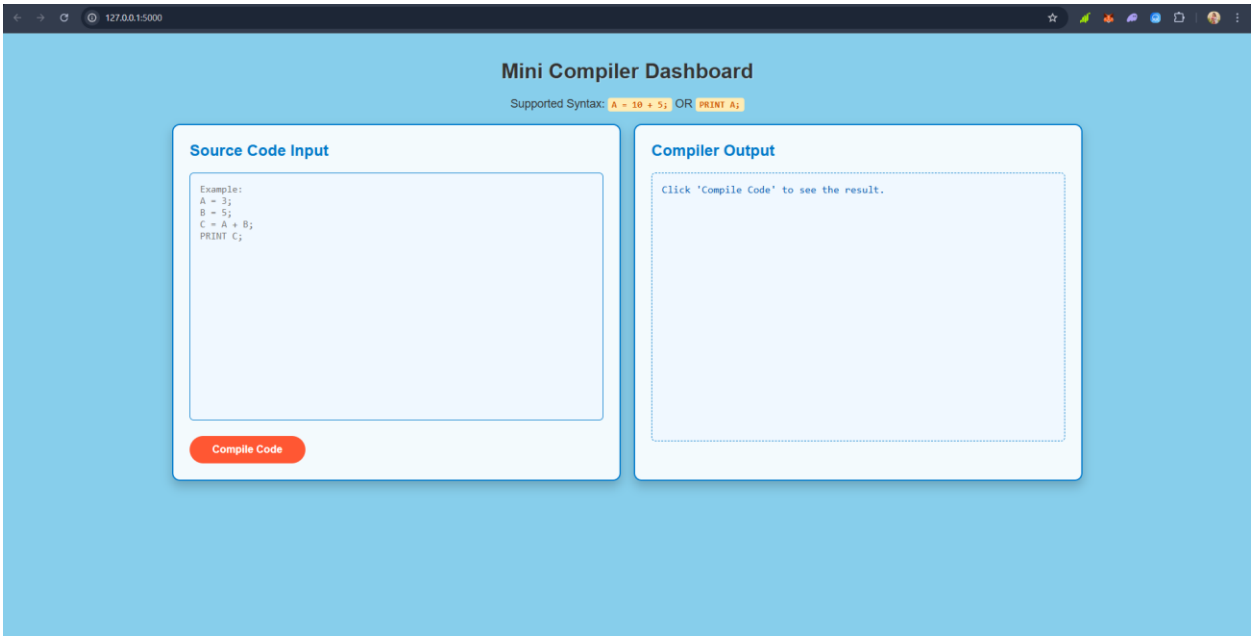
Backend Implementation Using Flask

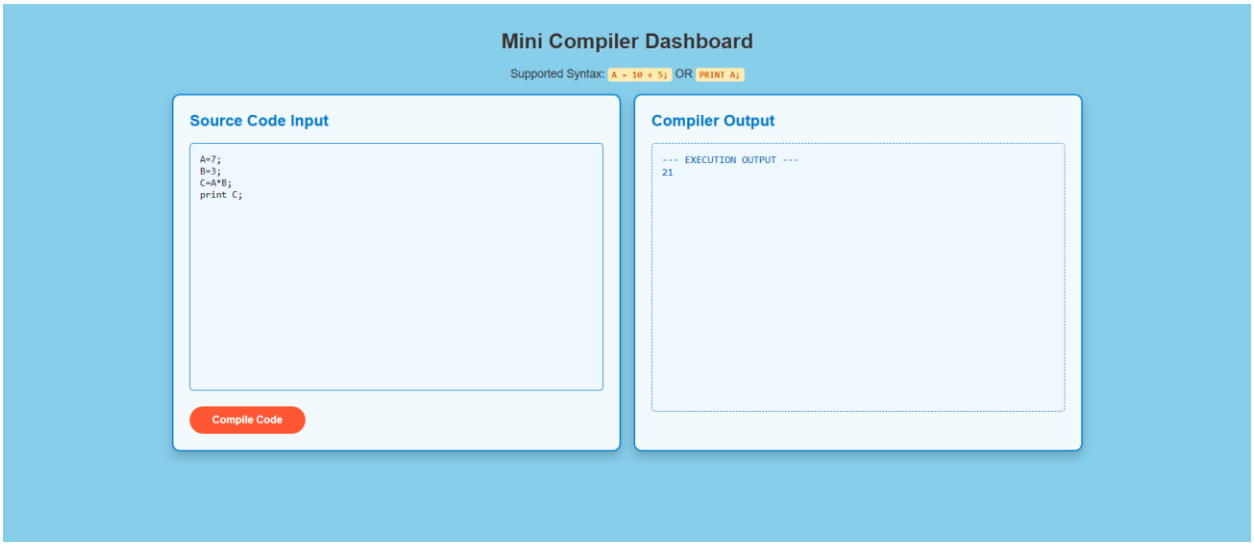
Flask is used as the backend framework for this project. The Flask server handles **HTTP** requests from the frontend. When the user clicks the compile button, the source code is sent to the '/compile' route as a JSON request. The backend processes the code using the lexer and interpreter and sends the result back as a JSON response. Flask plays a key role in connecting the frontend and backend components of the system.

Output and Results

The Mini Compiler successfully executes valid source code and displays the correct output. If the code contains errors, the compiler clearly reports them with descriptive messages. Screenshots of successful execution and error outputs are included to demonstrate the working of the system. These outputs prove that the compiler works as expected for supported language features.

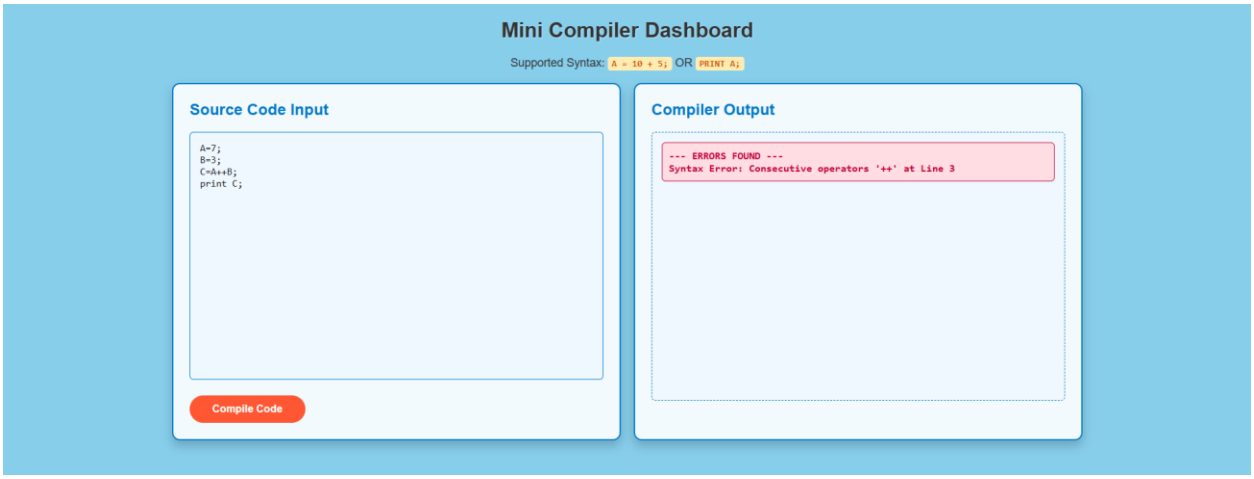
Output:



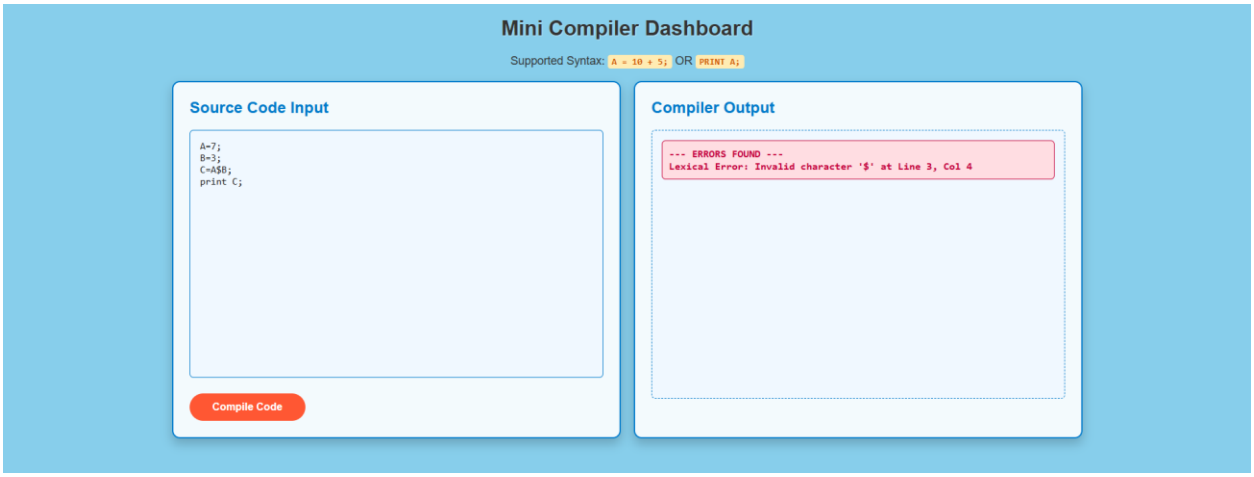


Errors

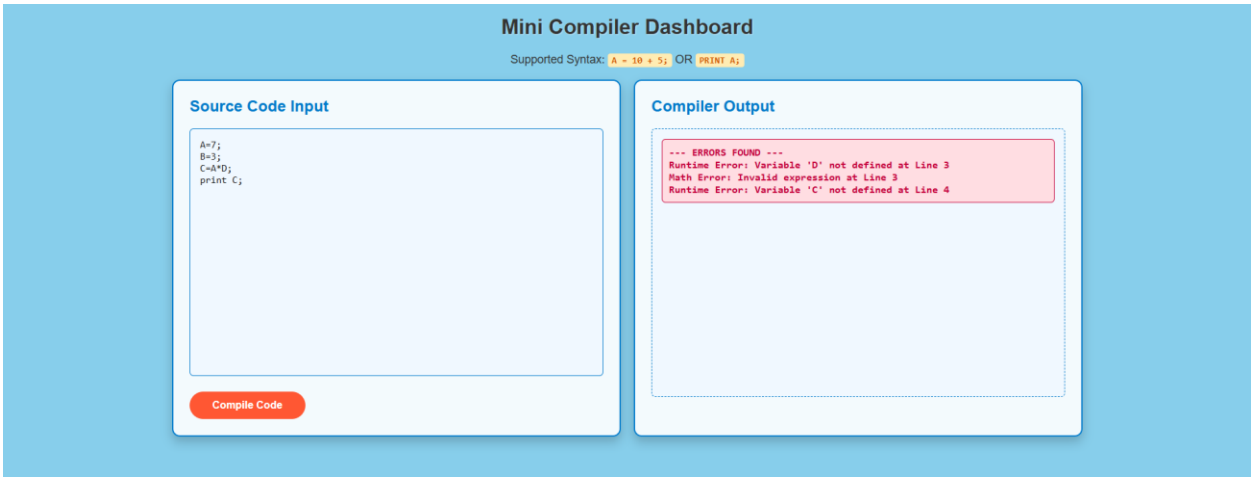
Syntax Error: Missing semicolon.



Lexical Error: Invalid character.



Runtime Error: Undefined variable.



Working Diagram or Flowchart

The flowchart illustrates the working mechanism of the Mini Compiler. The user inputs source code, which is sent to the Flask server for lexical analysis. If a lexical error is detected, an error message is displayed and execution stops. Otherwise, the tokens are passed to the interpreter for execution. The system then checks for syntax or runtime errors. If no error is found, the output is generated and displayed on the web interface.

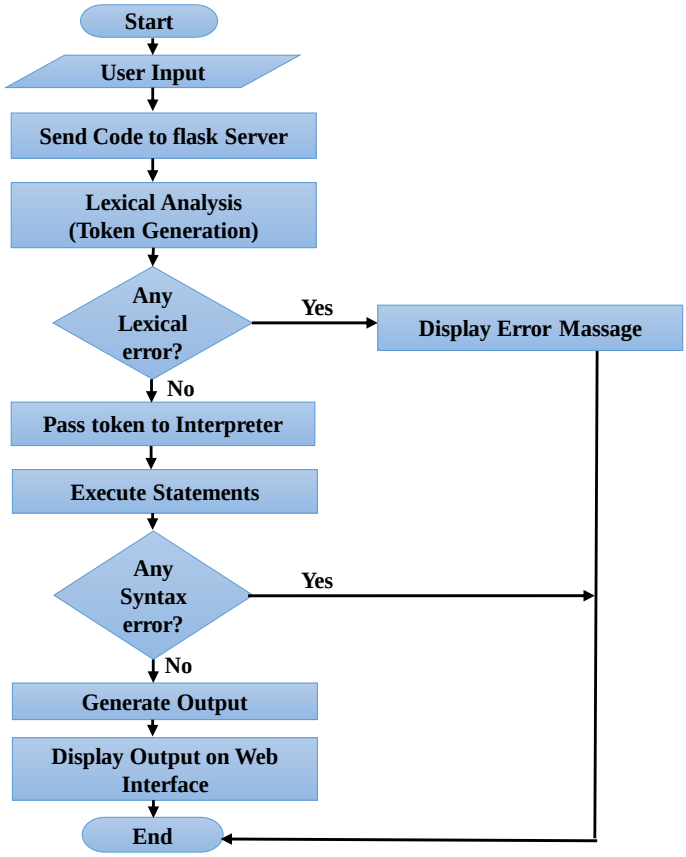


Fig. 1.2: Working Procedure Flowchart of Mini Compiler

Limitations of the Project

Despite its usefulness, the Mini Compiler has some limitations. It supports only a very small set of language features and does not include loops or conditional statements. The interpreter uses Python's eval function, which is not suitable for large or secure systems. There is no optimization phase, and the compiler does not generate machine-level code. These limitations are acceptable since the project is designed for learning purposes.

Future Improvements

There are many possible future improvements for this project. Additional language features such as conditional statements and loops can be added. A proper parser and abstract syntax tree can be implemented instead of direct evaluation. Syntax highlighting and improved error messages can enhance the user experience. Security can be improved by replacing eval with a custom expression evaluator. These improvements can make the compiler more powerful and robust.

Conclusion

In conclusion, the Mini Compiler project successfully demonstrates the basic concepts of compiler design in a simple and practical way. By implementing lexical analysis, interpretation, error handling, and a web-based interface, the project provides a strong foundation for understanding how compilers work internally. This project is an effective educational tool for students learning compiler design and programming language concepts.

References

Python Official Documentation, Flask Official Documentation, and standard Compiler Design textbooks were used as references during the development of this project.

Books:

Allen I. Holub, *Compiler Design in C*, Prentice Hall, 1990.

Keith D. Cooper and Linda Torczon, *Engineering a Compiler*, 2nd Edition, Morgan Kaufmann, 2011.

Online Resources:

GeeksforGeeks, "Compiler Design Tutorials," https://www.geeksforgeeks.org/compiler-design-tutorials/

TutorialsPoint, "Compiler Design," https://www.tutorialspoint.com/compiler_design/index.htm

Research Papers:

"Design and Implementation of a Simple Compiler for Educational Purpose," *IEEE Conference Paper*, 2015.

"A Mini Compiler for Teaching," *ACM Digital Library*, 2012.