Graphic Era Hill University
BHIMTAL CAMPUS

Term work of

Project Based Learning (PBL) of Compiler Design

Submitted in fulfillment of the requirement for the VI semester

Bachelor of Technology

By

Aashish Gupta

Arpita Rajput

Garima Aithani

Garima Bisht

Under the Guidance of Mr. Aviral Awasthi

Assistant Professor

Dept. of CSE

GRAPHIC ERA HILL UNIVERSITY, BHIMTAL CAMPUS
SATTAL ROAD, P.O. BHOWALI DISTRICT- NAINITAL-263132

2024 – 2025

Graphic Era Hill University
BHIMTAL CAMPUS

CERTIFICATE

The term work of Project Based Learning, being submitted by Aashish Gupta (2261051) s/o Mr. Mukesh Gupta, Arpita Rajput (2261116) d/o Mr. Rajesh Singh, Garima Aithani (2261206) d/o Mr. H.C Singh and Garima Bisht (2261207) d/o Mr. N.S Bisht to Graphic Era Hill University Bhimtal Campus for the award of Bonafide work carried out by us. They had worked under my guidance and supervision and fulfilled the requirements for the submission of this work report.

(Mr. Aviral Awasthi)
Faculty-in-Charge

(Dr. Ankur Singh Bist)
HOD, CSE Dept.

Graphic Era Hill University
BHIMTAL CAMPUS

STUDENT'S DECLARATION

We, Aashish Gupta, Arpita Rajput, Garima Aithani, Garima Bisht hereby declare the work, which is being presented in the report, entitled Term work of Project Based Learning of Compiler Design in fulfillment of the requirement for the award of the degree Bachelor of Technology (Computer Science) in the session 2024 - 2025 for semester VI, is an authentic record of our own work carried out under the supervision of Mr. Aviral Awasthi (Graphic Era Hill University, Bhimtal). The matter embodied in this project has not been submitted by us for the award of any other degree.

Date: …….

..……..…….

(Full signature of students)

Table of Contents

Chapters

Introduction

Project Overview

The Python GUI file implements a user-friendly interface for the Mini C Compiler using Tkinter and ttkbootstrap, providing a modern and visually appealing environment to interact with the compiler. Users can open C source files, write or edit code directly, and initiate compilation by invoking the external compiler executable. The GUI captures and displays compilation messages with color-coded formatting for easy readability and automatically generates and loads the Abstract Syntax Tree (AST) image from the compiler's Graphviz output. The design includes robust error handling and retry logic to ensure the AST image is properly loaded, enhancing the overall user experience. This front end effectively bridges the gap between the underlying compiler logic and the user, making the compilation process transparent and accessible.

Objectives

Python GUI :Provide an intuitive and interactive interface that facilitates the process of compiling

Lexical Analysis (Lexer)˙ : Develops a lexer to tokenize raw source code into a sequence of meaningful symbols (tokens).

Parsing and Grammar: Implement a recursive descent parser to validate token sequences against the language grammar and build a corresponding Abstract Syntax Tree (AST).

AST and Semantic Analysis: Construct an efficient and modular AST structure to represent program logic.

Interpreter Implementation :Create an interpreter that walks the AST and executes operations in realtime.

Testing and Validation : Write extensive test cases to validate lexer, parser, and interpreter correctness.

Technologies Used

Programming Language: C and Python

Input Processor: Lexical Analyser using Lex(Flex) Build Tools:

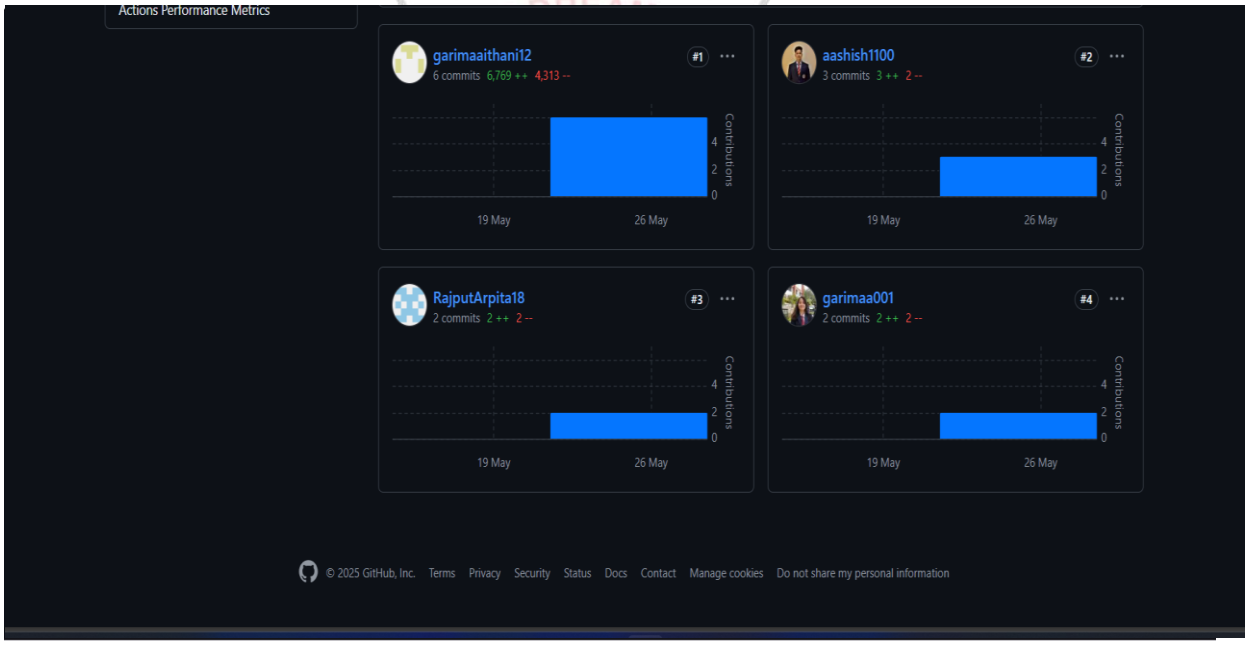
Flex(lex), Lex(Yacc), Bison, GCC, Python Interpreter, Graphviz

Operating System Compatibility: Windows, Linux, and macOS.

Key Components:Lexer, Parser, Psuedo-Assembly Code Generator, Python GUI Frontend, AST.

Github


Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Pulse
Contributors
Community
Community Standards
Traffic
Commits
Code frequency
Dependency graph
Network
Forks
Actions Usage Metrics
Actions Performance Metrics
Contributors
Contributions per week to main, excluding merge commits
Period: All
Contributions: Commits
Commits over time
Weekly from 18 May 2025 to 25 May 2025
19 May
26 May
Contributions
garimaaithani12
6 commits 6,769 ++ 4,313 --
#1
aashish1100
3 commits 3 ++ 2 --
#2


Actions Performance Metrics
garimaaithani12
6 commits 6,769 ++ 4,313 --
#1
aashish1100
3 commits 3 ++ 2 --
#2
RajputArpita18
2 commits 2 ++ 2 --
#3
garimaa001
2 commits 2 ++ 2 --
#4
19 May
26 May
Contributions
© 2025 GitHub, Inc.
Terms
Privacy
Security
Status
Docs
Contact
Manage cookies
Do not share my personal information

Compilation Workflow

Phases, Compilation and Execution:

1. lexer.cpp

Tokenizes raw input. Contains FSM logic for identifying keywords, identifiers, numbers, operators. Returns a stream of Token objects. Tokens include identifiers, literals, symbols like +, -, =, etc.

```
Token Lexer::getNextToken() {
    while (currentChar != '\0') {
        if (isspace(currentChar)) {
            skipWhitespace();
            continue;
        }

        if (isalpha(currentChar)) return identifier();
        if (isdigit(currentChar)) return number();
        if (currentChar == '+') { advance(); return Token(TokenType::PLUS,
        // ... more symbols

        error("Unknown character");
    }

    return Token(TokenType::EOF, "");
}
```

2. parser.cpp

Converts tokens to AST (Abstract Syntax Tree). Uses recursive descent parsing techniques. Constructs nodes for statements, expressions, control structures. Includes grammar for if, while, assignment, etc.

```
class BinaryExpr : public Expr {
public:
    BinaryExpr(std::unique_ptr<Expr> left, Token op, std::unique_ptr<Expr>
        : left(std::move(left)), op(op), right(std::move(right)) {}

    Value evaluate(Environment& env) override {
        Value l = left->evaluate(env);
        Value r = right->evaluate(env);
        switch (op.type) {
            case TokenType::PLUS: return l + r;
            case TokenType::MINUS: return l - r;
            // Additional operators...
        }
    }

private:
    std::unique_ptr<Expr> left;
    Token op;
    std::unique_ptr<Expr> right;
};
// Additional methods and members can be added as needed
```

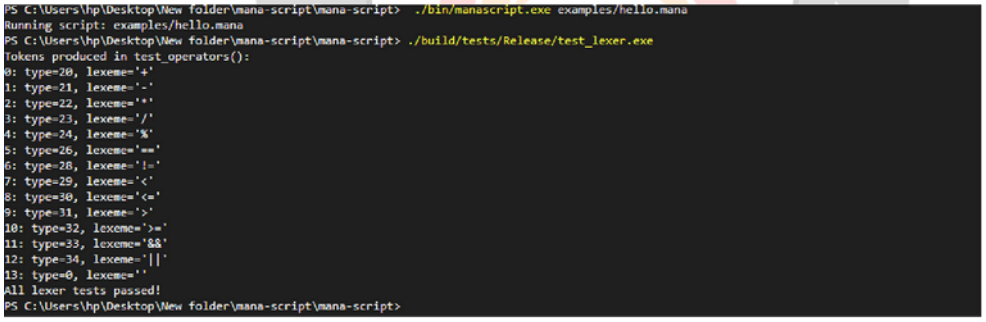
3. interpreter.cpp

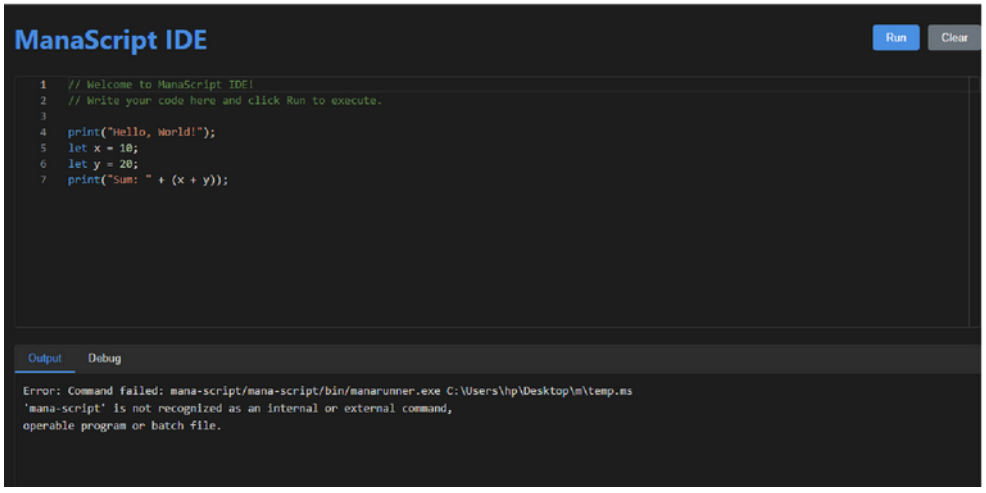
Walks the AST and executes code. Visitor pattern style visit() for each node type. Supports variables, expressions, control flow. Executes trees: expressions, conditionals, loops.

```cpp
int Interpreter::visitBinaryOp(BinaryOpNode* node) {
    int left = visit(node->left.get());
    int right = visit(node->right.get());
    if (node->op.type == TokenType::PLUS) return left + right;
    // ... other ops
}
int Interpreter::visitUnaryOp(UnaryOpNode* node) {
    int operand = visit(node->operand.get());
    if (node->op.type == TokenType::MINUS) return -operand;
    // ... other ops
}
```

Execution:

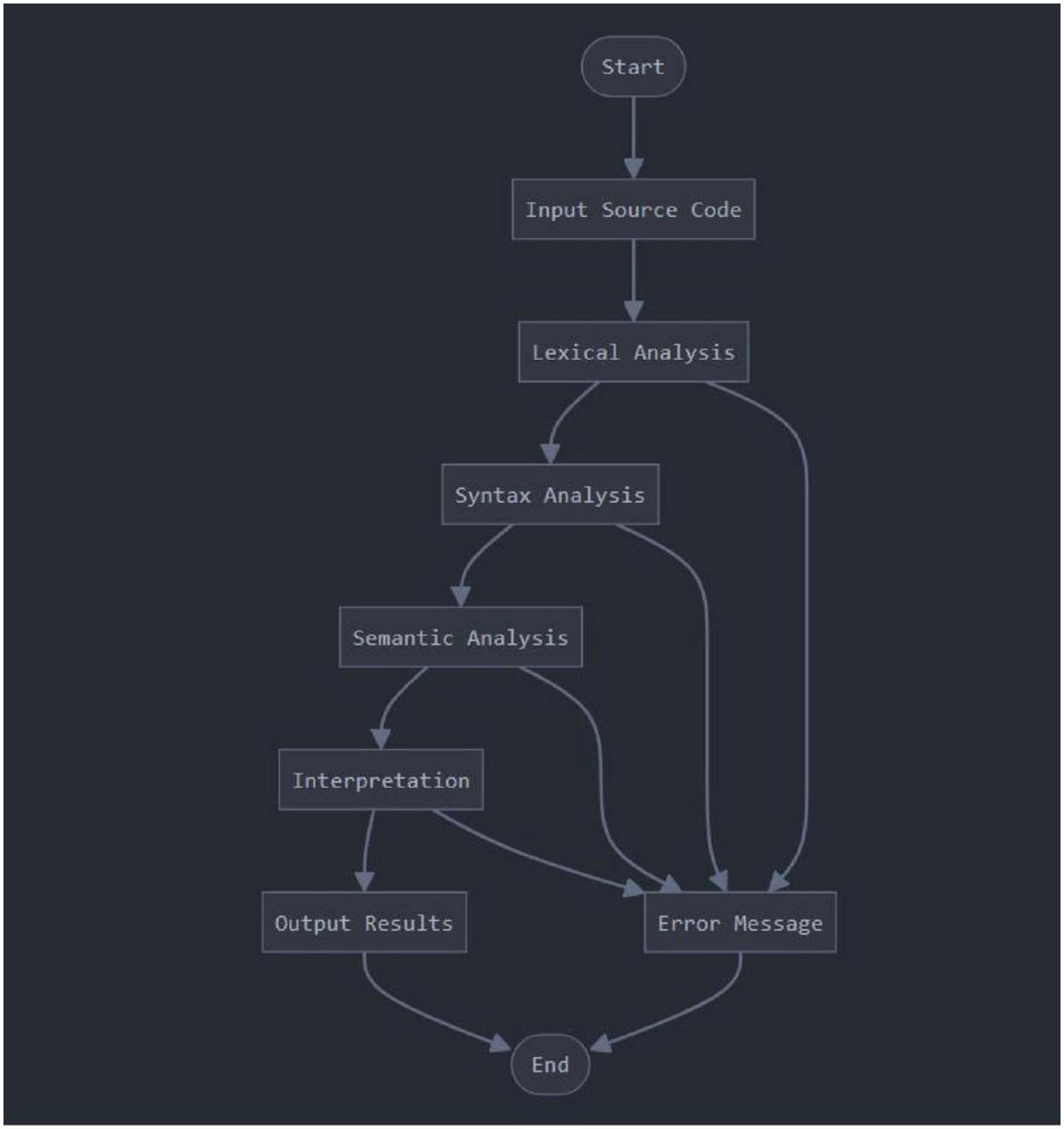
Example code in a custom IDE



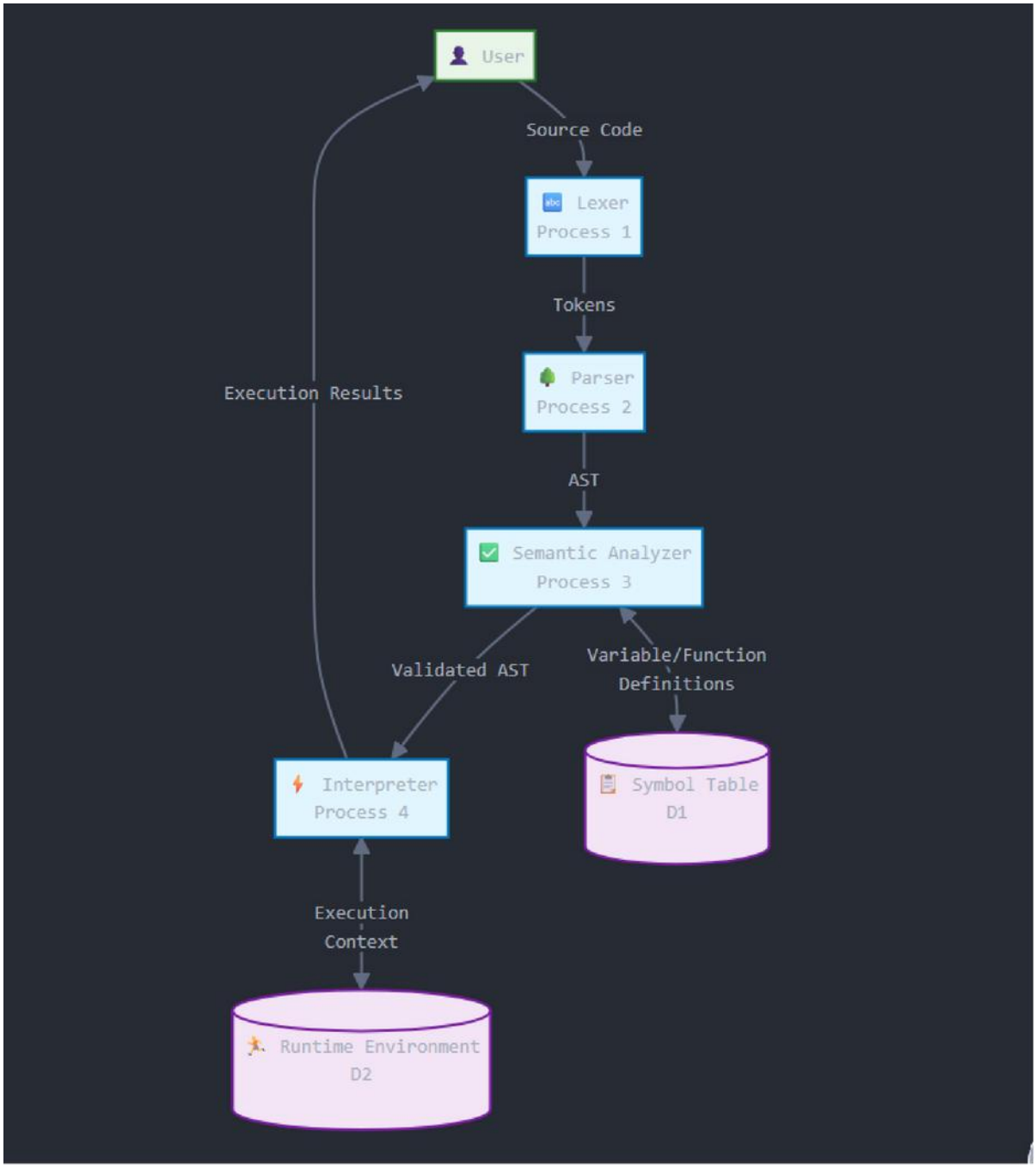


System Design

Flow Chart:



Data Flow Diagram:


User
Source Code
Lexer
Process 1
Tokens
Parser
Process 2
Execution Results
AST
Semantic Analyzer
Process 3
Validated AST
Variable/Function
Definitions
Interpreter
Process 4
Symbol Table
D1
Execution
Context
Runtime Environment
D2

Features

4.1 Lexical and Syntax Analysis: Utilizes a hand-written lexer and recursive descent parser to tokenize and parse the input source code into a well-defined Abstract Syntax Tree (AST).

4.2 Intermediate Code Generation: Produces three-address code (3AC), which uses temporaries for arithmetic expressions.

4.3 Semantic Validation: Implements rigorous semantic checks such as type checking, variable scoping, and declaration verification, enhancing language safety and reducing runtime errors.

4.4 Runtime Interpretation: Real-time code execution is powered by a robust interpreter that evaluates the validated AST and generates immediate results without intermediate compilation.

4.5 Symbol Table Management: Maintains a structured symbol table to track variable definitions, scopes, and types during the parsing and execution phases.

4.6 Test Coverage: Comes with a comprehensive suite of unit and integration tests covering all major components, ensuring reliability and correctness.

Conclusion

The Mini C Compiler project successfully demonstrates the fundamental principles and practical implementation of a compiler for a simplified subset of the C programming language. By integrating classical compiler construction tools such as Lex and Yacc with modern software development techniques using Python and Tkinter, this project bridges the gap between theoretical concepts and hands-on application. Throughout the development, key compiler phases were meticulously implemented—from lexical analysis to parsing, semantic validation, intermediate code generation, and final pseudo-assembly output. The inclusion of AST visualization using Graphviz not only enhances the transparency of the compilation process but also serves as an invaluable educational aid, helping users visualize the internal structure of source code.

Moreover, the Python-based GUI adds significant value by providing an accessible, interactive platform where users can input code, initiate compilation, and view detailed logs and outputs in real time. This user-centric design empowers learners and developers to experiment with code, observe the effects of compilation stages, and gain deeper insight into how high-level programming constructs are transformed into low-level instructions. Error handling and user feedback mechanisms ensure that the tool is robust and user-friendly, while the modular structure facilitates further enhancements and experimentation.

Overall, the Mini C Compiler project exemplifies how combining well-established compiler theory with practical programming and UI development can produce a powerful educational resource. It not only reinforces understanding of compiler internals but also cultivates skills in language design, parsing, code generation, and software engineering. This foundation paves the way for exploring more advanced compiler optimizations, supporting additional language features, or extending the GUI for richer user experiences in future iterations.