Term work of

Project Based Learning (PBL)
of
Compiler Design

Submitted in fulfillment of the requirement for the VI semester

Bachelor of Technology

By

Ayush Debnath

Manu

Mohd. Adnan

Manish Singh Rathaur

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

CERTIFICATE

The term work of Project Based Learning, being submitted by Ayush Debnath (2261124) s/o Mr. Prabir Debnath, Manu (2261354) s/o Mr. Mohan Lal, Mohd. Adnan (2261367) s/o Yusuf Ali and Manish Singh Rathaur (2261648) s/o Khushal Singh Rathaur 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)	**(Dr. Ankur Singh Bist)**
Faculty-in-Charge	**HOD, CSE Dept.**

Graphic Era Hill University
BHIMTAL CAMPUS

STUDENT'S DECLARATION

We, Ayush Debnath, Manu, Mohd. Adnan and Manish Singh Rathaur, 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

Chapter **Pages**

Introduction

Project Overview

MANA Script is a custom-built high-level scripting language designed and implemented in C++. It features a complete interpreter pipeline, including lexical analysis, parsing, AST generation, and runtime evaluation. The language supports variables, control structures, functions, and object-oriented programming. It offers a REPL environment for interactive use and is structured to be easily extensible for future features like JIT compilation and type inference. With a modular architecture and robust testing, *MANA Script* showcases a practical application of compiler theory and systems design using core C++ principles.

Objectives

Language Design: Define a simple and readable syntax for the scripting language.

Lexical Analysis (Lexer): Develop 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 real-time.

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

.

Technologies Used

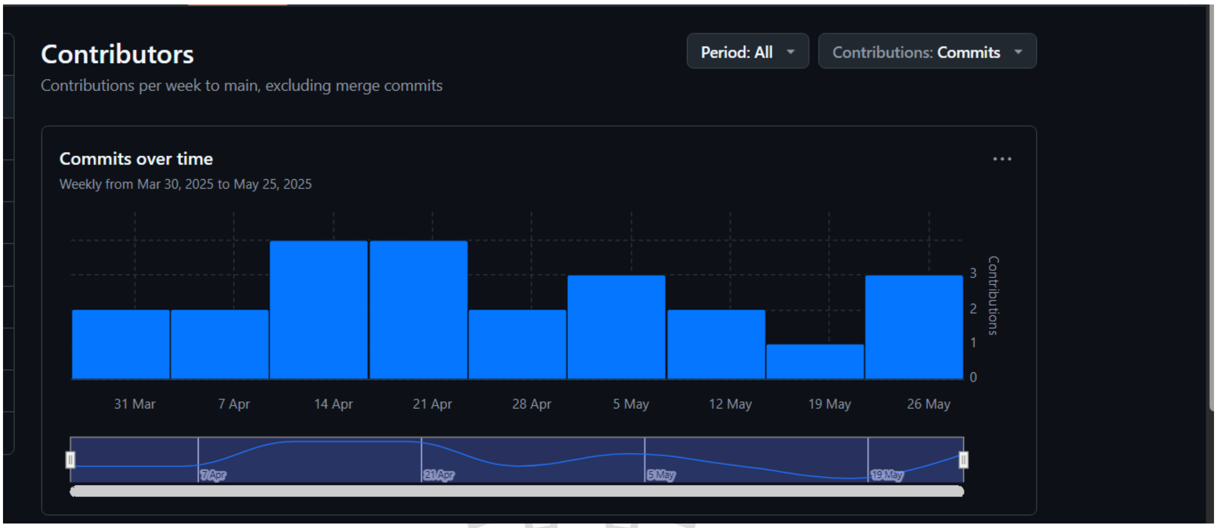
Programming Language: C++17

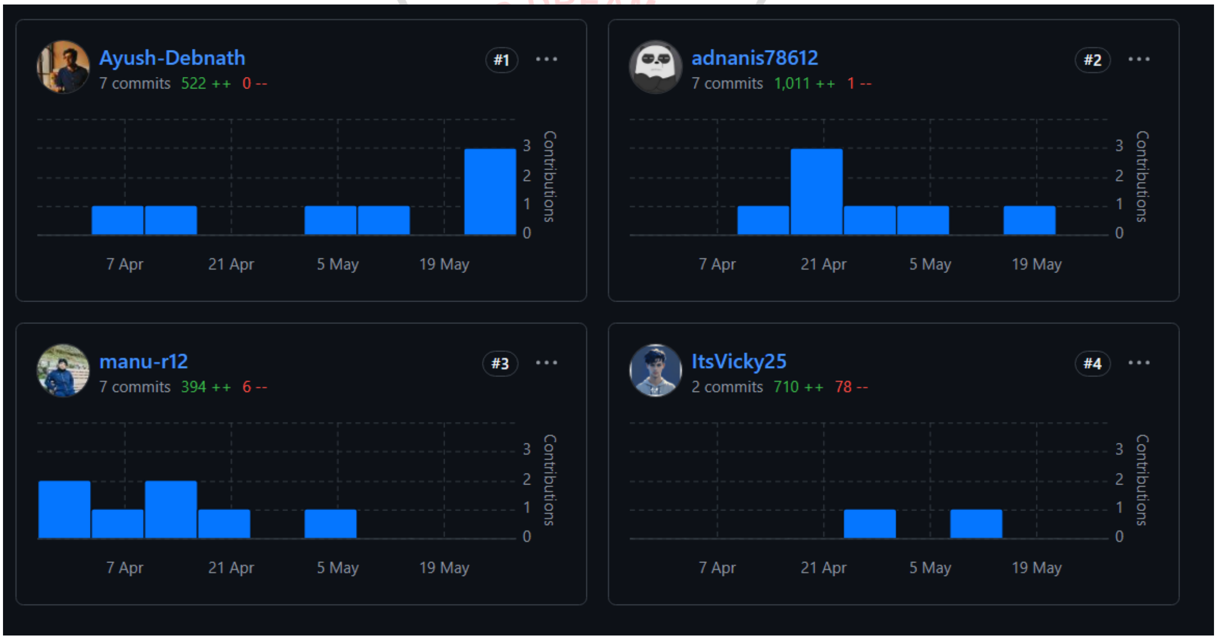
Input Processor: Handwritten Lexer and Recursive Descent Parser

Build Tools: CMake

Operating System Compatibility: Cross-platform – Compatible with Windows, Linux, and macOS.
Key Components: Lexer, Parser, Interpreter, Object System, REPL Interface.

Github


Contributors
Contributions per week to main, excluding merge commits
Period: All
Contributions: Commits
Commits over time
Weekly from Mar 30, 2025 to May 25, 2025
31 Mar
7 Apr
14 Apr
21 Apr
28 Apr
5 May
12 May
19 May
26 May
Contributions
0
1
2
3


Ayush-Debnath
7 commits 522 ++ 0 --
#1
adnanis78612
7 commits 1,011 ++ 1 --
#2
manu-r12
7 commits 394 ++ 6 --
#3
ItsVicky25
2 commits 710 ++ 78 --
#4
7 Apr
21 Apr
5 May
19 May
Contributions

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

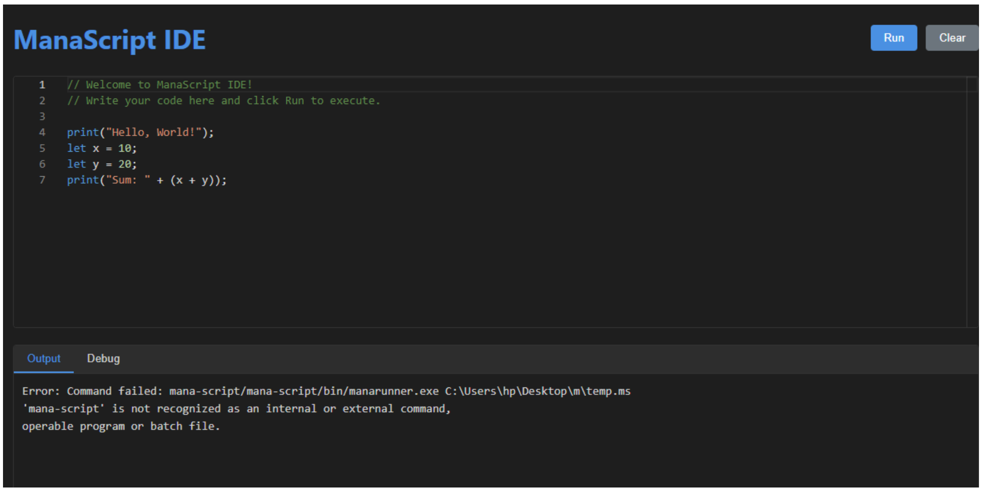
Walks the AST and executes code. Visitor pattern style visit() for each node type. Supports variables, expressions, control flow. Executes trees: expressions, conditionals, loops.

```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:

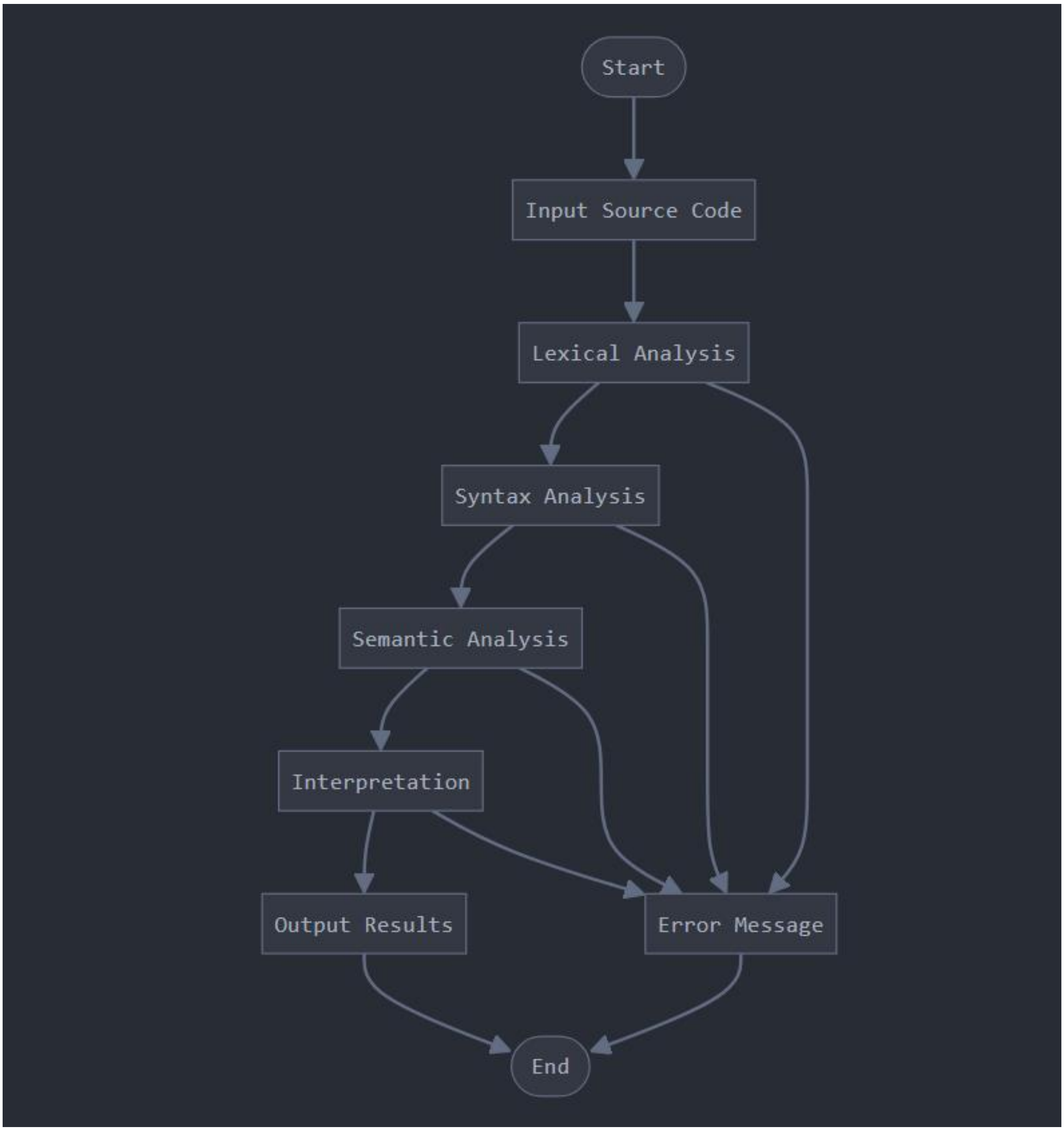
Example code in a custom IDE

```
PS C:\Users\hp\Desktop\New folder\mana-script\mana-script>  ./bin/manascript.exe examples/hello.mana
Running script: examples/hello.mana
PS C:\Users\hp\Desktop\New folder\mana-script\mana-script> ./build/tests/Release/test_lexer.exe
Tokens produced in test_operators():
0: type=20, lexeme='+'
1: type=21, lexeme='-'
2: type=22, lexeme='*'
3: type=23, lexeme='/'
4: type=24, lexeme='%'
5: type=26, lexeme='=='
6: type=28, lexeme='!='
7: type=29, lexeme='<'
8: type=30, lexeme='<='
9: type=31, lexeme='>'
10: type=32, lexeme='>='
11: type=33, lexeme='&&'
12: type=34, lexeme='||'
13: type=0, lexeme=''
All lexer tests passed!
PS C:\Users\hp\Desktop\New folder\mana-script\mana-script>
```

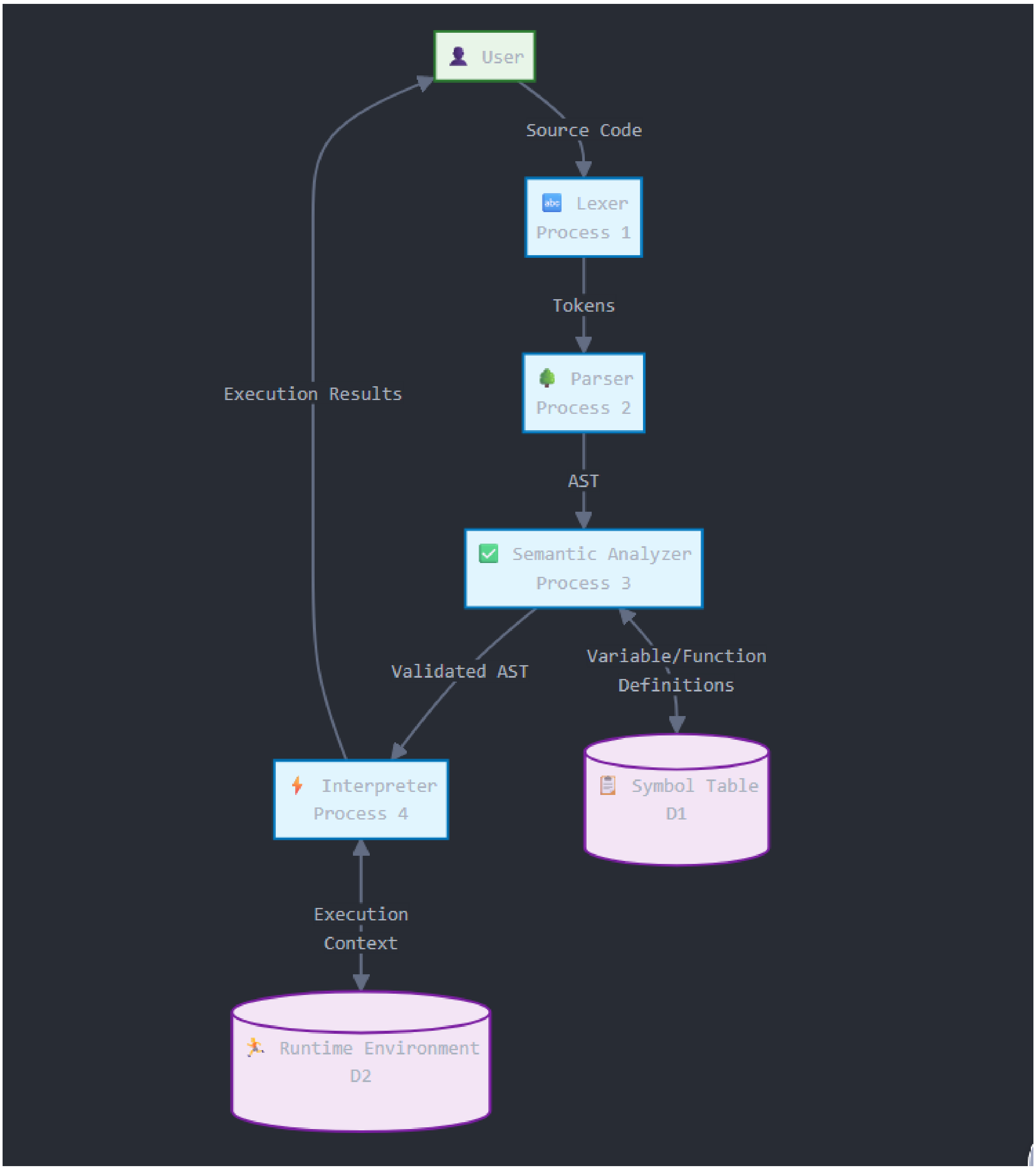


System Design

Flow Chart:



Data Flow Diagram:



Features

4.1 Custom Scripting Language: MANA Script introduces a bespoke programming language designed with simplicity and readability in mind, ideal for both educational use and lightweight scripting tasks.

4.2 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).

4.3 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 development of MANA Script has been a comprehensive exercise in understanding the intricate process of language design and interpreter construction. Starting from fundamental concepts like lexical analysis and syntax parsing to building an abstract syntax tree and executing code through a tree-walk interpreter, this project successfully encapsulates the core principles of compiler and interpreter technology.

Throughout the project lifecycle, the team navigated numerous technical challenges that tested both our theoretical knowledge and practical programming skills, particularly in C++. The modular and clean architecture of MANA Script facilitates future enhancements and scalability, making it a solid foundation for further exploration into advanced language features and optimizations.

While the current implementation focuses on essential programming constructs and interpreter functionality, it effectively demonstrates the viability of a custom scripting language tailored to specific user needs. The project has not only reinforced key concepts in compiler theory but has also highlighted the importance of code maintainability, systematic testing, and collaborative development in producing robust software.

Moreover, MANA Script serves as an educational platform for both developers and learners interested in language implementation, offering a clear, extendable, and well-documented codebase. The insights gained from this endeavor pave the way for future improvements such as enhanced error handling, bytecode compilation, and richer language features, which will significantly expand the language's applicability and performance.

In conclusion, MANA Script embodies a successful balance between academic rigor and practical implementation. It stands as a testament to our team's dedication, technical proficiency, and innovative spirit. As a project, it not only meets its initial objectives but also lays the groundwork for continued development, ultimately contributing to the broader field of programming language research and development.