Mini Compiler

Submitted By:

Student Name	Student ID
Md Rasel Biswash	024231000510 1215
Md Rakib Hasan	024231000510 1395
Mst.Afiya Haque	024231000510 1630
Mst.Rukaia Zahan	024231000510 1390
Babor Ali	024231000510 1954

MINI COMPILER PROJECT REPORT This Report Presented in Partial Fulfillment of the course **CSE313 Compiler Design Lab in the Computer Science and Engineering Department**

DAFFODIL INTERNATIONAL UNIVERSITY

Dhaka, Bangladesh

August 14, 2025

DECLARATION

We hereby declare that this lab project has been done by us under the supervision of **Zannatul Mawa Koli, Lecturer** , Department of Computer Science and Engineering, Daffodil International University. We also declare that neither this project nor any part of this project has been submitted elsewhere as lab projects.

Submitted To:

14/08/25

Zannatul Mawa koli
Lecturer
Department of Computer Science and Engineering Daffodil International University

Submitted by

Md Rasel Biswash Md Rasel Biswash 0242310005101215 Dept. of CSE, DIU	
Md. Rakib Hasan Md Rakib Hasan 0242310005101395 Dept. of CSE, DIU	Mst. Afiya Haque Mst.Afiya Haque 0242310005101630 Dept. of CSE, DIU
Mst. Rukaia Zahan Mst.Rukaia Zahan 0242310005101390 Dept. of CSE, DIU	Babor Ali Babor Ali 0242310005101954 Dept. of CSE, DIU

Table of Contents

Chapter 1

Introduction

This chapter provides an overview of the Mini Compiler project, explaining its purpose, scope, and importance in the field of programming languages. It also outlines the main objectives, motivation, and features that guided the development process.

1.1 Introduction

This project presents a mini language compiler capable of handling floating-point numbers, variable assignment, arithmetic operations (+, -, *, /), conditional statements (if-else), and printing output. It uses Flex for lexical analysis, Bison for parsing, and implements functionality within the parser file

1.1 Motivation

The motivation behind building this compiler is to understand the core stages of language processing — lexical analysis, syntax analysis, semantic actions, and code execution — while

1.1 Objectives

- Implement lexical and syntax analysis using Flex and Bison.
- Support floating-point computations.
- Enable variable storage and retrieval.
- Provide basic decision-making using if-else.

1.1 Feasibility Study

The Mini Language Compiler is feasible as it uses free, open-source tools and requires no special hardware. Its limited scope ensures easy implementation on standard systems, making it cost-effective, technically achievable, and suitable for educational purposes.

1.2 Gap Analysis

Existing compilers are large, complex, and designed for full-scale programming languages, making them difficult for beginners to understand. The gap lies in the lack of simple, educational tools that clearly demonstrate the basic phases of compilation. The Mini Language Compiler addresses this by providing a lightweight, easy-to-understand system focused on core compilation processes for a small language.

1.3 Project Outcome

The Mini Language Compiler will produce a functional tool that can translate a small, predefined language into target code while demonstrating key compilation phases. It will enhance understanding of compiler design and serve as a practical learning resource.

Chapter 2

Proposed Methodology/Architecture

This chapter presents the proposed methodology and overall architecture of the Mini Compiler, detailing how each phase of the compilation process is implemented. It explains the step-by-step workflow, tools used, and the interaction between different modules to achieve the desired functionality.

2.1 Requirement Analysis & Design Specification

Requirement Analysis

- **Functional Requirements**:
  1. Accept source code written in the mini language.
  2. Perform lexical analysis to tokenize the input.
  3. Conduct syntax analysis to check grammatical correctness.
  4. Generate intermediate or target code.
  5. Display errors with proper messages.
- **Non-Functional Requirements**:
  1. Must run on standard operating systems (Windows/Linux).
  2. Should be lightweight and fast.
  3. Use open-source tools like Flex and Bison for development.

Design Specification

- **Architecture**: Follows the classic compiler design model with modules for Lexical Analysis, Syntax Analysis, Semantic Analysis (optional), and Code Generation.
- **Input**: Source file written in the mini language.
- **Output**: Tokens, parse tree (optional), and target/intermediate code.
- **Tools & Technologies**: Flex (lexical analysis), Bison (syntax analysis), C/C++ for integration, standard compiler tools for building and execution.

2.1.1 Overview

The Mini Language Compiler is a simplified compiler designed for educational purposes. It takes source code written in a small, predefined language as input, analyzes it through lexical and syntax phases, and generates intermediate or target code. The system demonstrates the core phases of compilation—tokenization, parsing, and code generation—in a clear and manageable way, providing a practical understanding of compiler design while being lightweight and easy to use

2.1.2 Proposed Methodology/ System Design

The Mini Language Compiler will be developed using a **modular, phase-based approach**, following the standard compiler architecture. The methodology includes:

Lexical Analysis: The source code is scanned to identify tokens such as keywords, identifiers, operators, and literals. Tools like **Flex** can be used for pattern matching.

Syntax Analysis (Parsing): Tokens are checked against the grammar rules of the mini language to ensure correct syntax. **Bison** or a similar parser generator can construct the parse tree.

Semantic Analysis (Optional): Checks for semantic consistency, such as type compatibility and variable declaration, depending on project scope.

Code Generation: Converts the validated input into intermediate code or a simple target representation.

8 **Error Handling**: At each phase, errors are detected and reported clearly to the user for easy debugging.

2.1.3 UI Design

```
Project Name: Mini Language Parser

Submitted By: Group-3
Name:                   ID:
Md Rasel Biswash        0242310005101215
Md Rakib Hasan          0242310005101395
Mst.Afiya Haque Nilasa  0242310005101630
Mst.Rukaia Zahan        0242310005101390
Babor Ali               0242310005101954

Section: 64_M1
Daffodil International University

Submitted To:
Zannatul Mawa Koli
Lecturer
Department of CSE
Faculty of Science and Information Technology
Daffodil International University

Submition Date: 14-08-2025

Write your code here(Note: To exicute ctrl+z and press enter):
```

2.2 Overall Project Plan

Objective:
Develop a compiler for a mini-language that performs **lexical, syntax, and semantic analysis** and optionally generates intermediate code.

Scope:

- Handle variables, arithmetic operations, conditionals, loops, and functions.
- Detect and report lexical, syntax, and semantic errors.

Phases & Timeline (8 weeks):

1. **Requirement Analysis** – Define language syntax, semantics, and project requirements (1 week)
2. **System Design** – Design compiler modules, DFDs, and flowcharts (1 week)
3. **Lexical Analysis** – Implement tokenizer using Flex (1 week)
4. **Syntax Analysis** – Implement parser using Bison (1 week)
5. **Semantic Analysis** – Type checking, scope management, symbol table (1 week)
6. **Intermediate Code Generation** – Generate three-address code (1 week)
7. **Testing & Debugging** – Test programs and fix errors (1 week)
8. **Documentation** – Prepare project report, user manual, and presentation slides (1 week)

Deliverables:

- Lexical & syntax analyzers
- Symbol table & semantic checker
- Intermediate code generator
- Test programs & results
- Project documentation

Tools:
C/C++, Flex, Bison, VS Code/Code::Blocks

Success Criteria:

- Accurate tokenization, parsing, and semantic checking
- Proper error detection and reporting
- Correct intermediate code generation
- Complete project documentation

Chapter 3

Implementation and Results

This chapter describes the practical implementation of the Mini Compiler, including the coding process, tools used, and integration of different modules. It also presents the results obtained from various test cases, showcasing the functionality and performance of the system.

3.1 Implementation

- **Lexical Analysis:** Implemented with **Flex** to tokenize keywords, identifiers, literals, and operators; detects lexical errors.
- **Syntax Analysis:** Implemented with **Bison** to build a parse tree; detects syntax errors.
- **Semantic Analysis:** Performs type checking, scope management, and symbol table integration; detects semantic errors.
- **Intermediate Code Generation (Optional):** Generates three-address code for execution.
- **Testing & Debugging:** Sample programs tested; errors fixed to ensure correct compilation.
- **Tools:** C/C++, Flex, Bison, VS Code/Code::Blocks.

3.2 Performance Analysis

- **Compilation Speed:** Measures time taken for lexical, syntax, and semantic analysis.
- **Accuracy:** Correct tokenization, parsing, and semantic checking; all errors detected.
- **Error Detection:** Evaluates how effectively lexical, syntax, and semantic errors are reported.
- **Memory Usage:** Monitors memory used by symbol tables, parse trees, and intermediate code.
- **Scalability:** Checks performance on small and large programs to ensure efficiency.
- **Outcome:** Compiler demonstrates fast compilation, accurate error detection, and efficient resource usage.

3.3 Results and Discussion

- The compiler successfully tokenizes, parses, performs semantic checks, and generates intermediate code.
- Lexical, syntax, and semantic errors are detected and reported accurately.
- Test programs with variables, arithmetic, conditionals, loops, and functions executed correctly.
- Compilation time is efficient and memory usage is reasonable.
- The compiler meets project objectives, demonstrating robust performance and scalability.

Chapter 4

Engineering Standards and Mapping

This chapter discusses the relevant engineering standards followed during the development of the Mini Compiler and how they were applied to ensure quality, maintainability, and consistency. It also maps the project's components and processes to established standards in software engineering and compiler design.

4.1 Impact on Society, Environment and Sustainability

4.1.1 Impact on Life: The mini-language compiler project improves understanding of compiler design and programming languages, enhances problem-solving and logical thinking, and provides practical experience with Flex, Bison, and C/C++ programming, building a strong foundation for advanced software development and computer science studies.

4.1.2 Impact on Society & Environment: The mini-language compiler promotes efficient software development and learning, contributing to technological advancement. Optimized algorithms reduce unnecessary computing, which can help minimize energy consumption and environmental impact.

4.1.3 Ethical Aspects: The mini-language compiler project highlights the importance of writing **clean and reliable code**, avoiding plagiarism, and responsibly using computing resources. It promotes **professional integrity**, accountability, and ethical practices in software development**.**

4.1.4 Sustainability Plan: The Mini Language Compiler is designed for maintainability with modular and well-documented code.
It ensures efficient use of resources and scalability for future enhancements. Comprehensive documentation supports smooth knowledge transfer and long-term sustainability.

4.2 Project Management and Team Work

Provide a cost analysis in terms of budget required and revenue model. In case of budget, you must show an alternate budget and rationales.

4.3 Complex Engineering Problem

4.3.1 Mapping of Program Outcome

In this section, provide a mapping of the problem and provided solution with targeted Program Outcomes (PO's).

Table 4.1: Justification of Program Outcomes

PO's	Justification
PO1	Applied compiler design concepts to develop the project.
PO2	Analyzed errors and designed solutions using Flex and Bison.
PO3	Designed and implemented a functional mini compiler system

4.3.2 Complex Problem Solving

In this section, provide a mapping with problem solving categories. For each mapping add subsections to put rationale (Use Table 4.2). For P1, you need to put another mapping with Table 4.2: Mapping with complex problem solving.

EP1 Dept of Knowledge	**EP2** Range of Conflicting Requirements	**EP3** Depth of Analysis	**EP4** Familiarity of Issues	**EP5** Extent of Applicable Codes	**EP6** Extent Of Stakeholder Involvement	**EP7** Inter-dependence
✓	✓	✓				✓

4.3.3 Engineering Activities

In this section, provide a mapping with engineering activities. For each mapping add subsections to put rationale (Use Table 4.3).

Table 4.3: Mapping with complex engineering activities.

EA1 Range of resources	**EA2** Level of Interaction	**EA3** Innovation	**EA4** Consequences for society and environment	**EA5** Familiarity
Used Flex, Bison, C/C++ compilers, and testing datasets.	Designed a compact, efficient, and testable compiler	Designed a compact, efficient, and testable compiler.	Promotes programming education without resource waste.	Implemented using widely taught compiler technologies.

Chapter 5

Conclusion

This chapter summarizes the overall achievements, findings, and experiences gained during the development of the Mini Compiler. It also reflects on the challenges faced, lessons learned, and the significance of the project in enhancing practical knowledge of compiler design.

5.1 Summary

The Mini Language Compiler performs lexical analysis, syntax parsing, and intermediate code generation.
It demonstrates core compiler concepts and ensures efficient, extendable design.
The project highlights teamwork, systematic development, and practical application of programming theory.

5.2 Limitation:

The Mini Language Compiler has limited support for complex language features and advanced optimizations.
It may not handle very large programs efficiently and lacks a full runtime environment.
Error reporting and debugging features are basic, requiring further enhancement for production-level use.

5.3 Future Work

In the future, the Mini Compiler can be extended to support more advanced programming constructs such as `for` loops, `while` loops, and other important control structures to make it closer to a fully functional compiler. Additional improvements could include performance optimizations, enhanced error detection and recovery mechanisms, and the integration of a runtime environment. Furthermore, adding debugging tools and a user-friendly graphical interface would significantly improve the usability and overall experience for end users.

References

1. Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Edition. Pearson Education India, 2007.
2. Kleinberg, Jon, and Eva Tardos. *Algorithm Design*. Pearson Education India, 2006.
3. Levine, John R., Tony Mason, and Doug Brown. *Lex & Yacc*. 2nd Edition. O'Reilly Media, 1992.
4. Appel, Andrew W. *Modern Compiler Implementation in C*. 2nd Edition. Cambridge University Press, 2002.
5. Louden, Kenneth C. *Compiler Construction: Principles and Practice*. Cengage Learning, 1997.