COMSATS UNIVERSITY ISLAMABAD,

ATTOCK CAMPUS

Department Of Computer Science

Project Mini Compiler

Course Name:	**Compiler Construction**
Submitted To:	**Mr. Bilal Bukhari**
Submitted By:	**Noor Ul Ain**
Registration No:	**Sp22-Bcs-037**
Submitted By:	**Kulsoom Bano**
Registration No:	**Sp22-Bcs-059**

Table Of Content

Project Title: Mini Compiler

1. Abstract

This project presents the design and implementation of a modular mini compiler capable of translating high-level pseudo-code into low-level pseudo-assembly instructions. The compiler mimics the structure of a real-world compiler by integrating key compilation phases: lexical analysis, syntax analysis (parsing), semantic analysis, intermediate code generation, code optimization, and final target code generation.

Each phase is designed to be independently testable and clearly structured, making the compiler both educational and maintainable. The system also includes features such as symbol table management and error handling to simulate realistic compiler behavior. This project aims to enhance students' understanding of compiler architecture and the transformation of source code into machine-level representations.

2. Introduction

The Mini Compiler project is developed as part of the Compiler Construction course to demonstrate the core concepts and phases of a compiler. It simulates a simplified version of a real compiler by processing source code through lexical, syntax, and semantic analysis, followed by intermediate code generation, optimization, and target code generation.

The compiler is designed with a modular structure, allowing each phase to function independently for better understanding and maintenance. It also includes error handling and symbol table management to reflect standard compiler behavior.

This project serves as a learning tool, helping students visualize how high-level code is translated into low-level instructions, reinforcing both theoretical and practical aspects of compiler design.

2.1 Background

Compilers are essential tools in software development, enabling the translation of human-readable programming languages into machine-executable code. The design and construction of a compiler require a deep understanding of formal languages, automata

theory, and software engineering principles. This mini compiler project was initiated as part of the Compiler Construction course to provide hands-on experience in implementing core concepts of compilation.

The goal was to simulate how modern compilers work by dividing the compilation process into structured phases: lexical analysis, parsing, semantic analysis, intermediate code generation, optimization, and target code generation. Each phase mirrors the role of its counterpart in professional compilers such as GCC or Clang but is simplified for educational purposes.

By building a compiler from scratch using C#, students gain practical insight into tokenization, abstract syntax tree construction, symbol management, and code generation. This project not only reinforces theoretical knowledge but also enhances problem-solving and system design skills critical to compiler engineering.

2.2 Objectives

The primary objective of this mini compiler project is to design and implement a simplified yet functional compiler that transforms high-level pseudo-code into low-level pseudo-assembly instructions. This hands-on implementation aims to reinforce the theoretical concepts covered in the Compiler Construction course. The specific objectives include:

- To understand and implement the key phases of a compiler: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and target code generation.
- To design modular components for each phase that can be integrated into a complete compilation pipeline.
- To detect and report lexical, syntactic, and semantic errors effectively using a centralized error handling mechanism.
- To simulate symbol table creation and management for variable and identifier tracking.
- To demonstrate the transformation of user-written pseudo-code into a low-level form resembling assembly code for educational purposes.
- To provide a simple command-line interface for interaction and testing.

2.3 Scope

This project focuses on the implementation of a mini compiler using the C# programming language. It is intended for academic and educational use, specifically within the context of learning compiler design fundamentals. The scope of the project includes:

- **Input Language**: A simplified high-level pseudo-code with basic constructs such as variable declarations, assignments, arithmetic operations, and print statements.
- **Compilation Phases**: Full implementation of lexical analysis, parsing, semantic analysis, intermediate code generation, code optimization, and target code generation.
- **Symbol Table**: Basic support for tracking variable names, types, scopes, and values.
- **Error Handling**: Structured reporting of errors detected during lexical, syntactic, and semantic phases.
- **Output**: The output is pseudo-assembly code, not machine code or platform-specific binaries.
- **Limitations**: The compiler does not support advanced features such as object-oriented constructs, function calls, control flow structures (e.g., if-else, loops), or type inference.

This scope ensures the compiler remains focused and achievable within a semester-long academic project while providing a solid foundation for understanding real-world compiler design.

3. System Overview

This section provides a high-level overview of the architecture and main components of the mini compiler. The system is modular in design, enabling each compilation phase to function independently while contributing to the overall compilation pipeline.

3.1 Architecture

The architecture of the mini compiler follows the traditional multi-phase structure of a compiler. Each phase takes the output of the previous one as input, performing its designated function in the transformation of high-level pseudo-code into low-level pseudo-assembly instructions. The major phases include:

- **Input Handling**: Accepts pseudo-code as user input.
- **Lexical Analysis**: Tokenizes the input into a stream of valid symbols.
- **Syntax Analysis (Parsing)**: Validates the grammatical structure and builds a syntax tree.
- **Semantic Analysis**: Ensures the logical correctness of the code and populates the symbol table.
- **Intermediate Code Generation**: Converts the syntax tree into a platform-independent intermediate representation.
- **Code Optimization**: Refines the intermediate code to improve efficiency.
- **Target Code Generation**: Outputs pseudo-assembly code.

The compiler uses object-oriented principles for modularity, encapsulation, and maintainability. Each phase is implemented as a separate class or module, facilitating future expansion and testing.

3.2 Major Components

The mini compiler consists of the following core components:

- **CompilerController**
  Acts as the orchestrator that invokes each phase in the correct order based on user input.
- **LexicalAnalyzer**
  Responsible for scanning the input source code and breaking it into tokens. It identifies keywords, identifiers, literals, and operators.
- **Parser(SyntaxAnalyzer)**
  Constructs the Abstract Syntax Tree (AST) using the tokens and verifies that the structure of the code conforms to defined grammar rules.
- **SemanticAnalyzer**
  Checks for type consistency, undeclared variables, and other logical issues. It interacts with the Symbol Table to validate and store declarations.
- **IntermediateCodeGenerator**
  Translates the syntax tree into a linear sequence of intermediate instructions that are easy to optimize and analyze.

- **Optimizer**
  Performs basic optimizations such as constant folding, dead code elimination, and algebraic simplification on the intermediate code.
- **TargetCodeGenerator**
  Converts the optimized intermediate code into readable pseudo-assembly code.
- **SymbolTable**
  A data structure that stores information about variables such as name, type, scope, and value.
- **ErrorHandler**
  Centralized utility that logs and displays lexical, syntactic, and semantic errors with contextual information.

4. Detailed Module Description

This section provides an in-depth explanation of each functional module of the mini compiler. Each module is designed to perform a specific phase in the compilation pipeline, working in sequence to transform high-level pseudo-code into low-level pseudo-assembly code.

4.1 Input Handling

Purpose:
To accept the user's high-level pseudo-code input as raw text.

Functionality:

- Reads the source code from the console or a file.
- Stores the input in a class-level string variable (sourceCode).
- Prepares the code for lexical processing.

4.2 Lexical Analysis

Purpose:
To scan the raw source code and divide it into meaningful tokens.

Functionality:

- Detects identifiers, keywords, literals, operators, and delimiters.
- Outputs a List<Token> structure.
- Invalid tokens are passed to the ErrorHandler.

Class: LexicalAnalyzer
Output: List<Token>

4.3 Syntax Analysis

Purpose:
To validate the grammatical structure of the token sequence and build an abstract syntax tree (AST).

Functionality:

- Implements a recursive-descent or predictive parsing strategy.
- Constructs and returns an ASTNode tree.
- Detects syntax errors such as mismatched parentheses or missing operators.

Class: Parser
Output: ASTNode syntaxTree

4.4 Semantic Analysis

Purpose:
To ensure logical correctness of the code and populate the symbol table.

Functionality:

- Verifies variable declarations, type compatibility, and usage scopes.
- Updates the SymbolTable with new symbols.

- Reports issues like undeclared variables or type mismatches to ErrorHandler.

Class: SemanticAnalyzer
Dependencies: ASTNode, SymbolTable, ErrorHandler

4.5 Intermediate Code Generation

Purpose:
To convert the validated syntax tree into intermediate, platform-independent code.

Functionality:

- Traverses the AST and generates instructions like t1 = a + b.
- Uses three-address code for simplicity.
- Prepares code for optimization.

Class: IntermediateCodeGenerator
Output: List<string> intermediateCode

4.6 Optimization

Purpose:
To enhance the performance and reduce redundancy in the intermediate code.

Functionality:

- Applies basic optimizations such as:
    - Constant folding
    - Dead code elimination
    - Strength reduction
- Outputs a refined version of the intermediate code.

Class: Optimizer
Input/Output: List<string>

4.7 Target Code Generation

Purpose:
To convert optimized intermediate code into a low-level pseudo-assembly format.

Functionality:

- Converts three-address instructions to target code (e.g., MOV, ADD, SUB).
- Simulates register usage for operations.
- Handles print statements and assignments directly.

Class: TargetCodeGenerator
Output: Console or file output of pseudo-assembly

4.8 Error Handling

Purpose:
To capture, store, and report errors during all phases of compilation.

Functionality:

- Collects messages from lexical, syntax, and semantic phases.
- Provides functions to:
    - Add error with line number
    - Check if errors exist
    - Print all errors in a readable format

Class: ErrorHandler
Dependencies: Used by multiple compiler components

5. Symbol Table and Error Reporting

5.1 Symbol Table

Purpose:
The Symbol Table is a central data structure used during semantic analysis to store information about identifiers (variables, constants, etc.) used in the source code.

Functionality:

- Maintains a list of declared symbols, each with attributes like name, type, scope, and value.
- Supports insertion (Add) and lookup (Lookup) operations.
- Prevents redeclaration errors and helps validate scope and type rules.

Structure:
Each Symbol entry contains:

- Name – Identifier's name.
- Type – Data type (e.g., int, float).
- Scope – Contextual location (global, local).
- Value – Optional assigned value or placeholder.

Class:
SymbolTable
Dependencies:
Used by the SemanticAnalyzer for verification and by the Compiler for symbol inspection.

5.2 Error Reporting

Purpose:
Error reporting ensures that the user is informed about issues at various stages of compilation: lexical, syntax, and semantic.

Functionality:

- Collects and stores all error messages with line numbers.
- Errors are categorized by phase and severity.
- Offers functions to:

- AddError(string message, int line)
- HasErrors()
- PrintErrors()

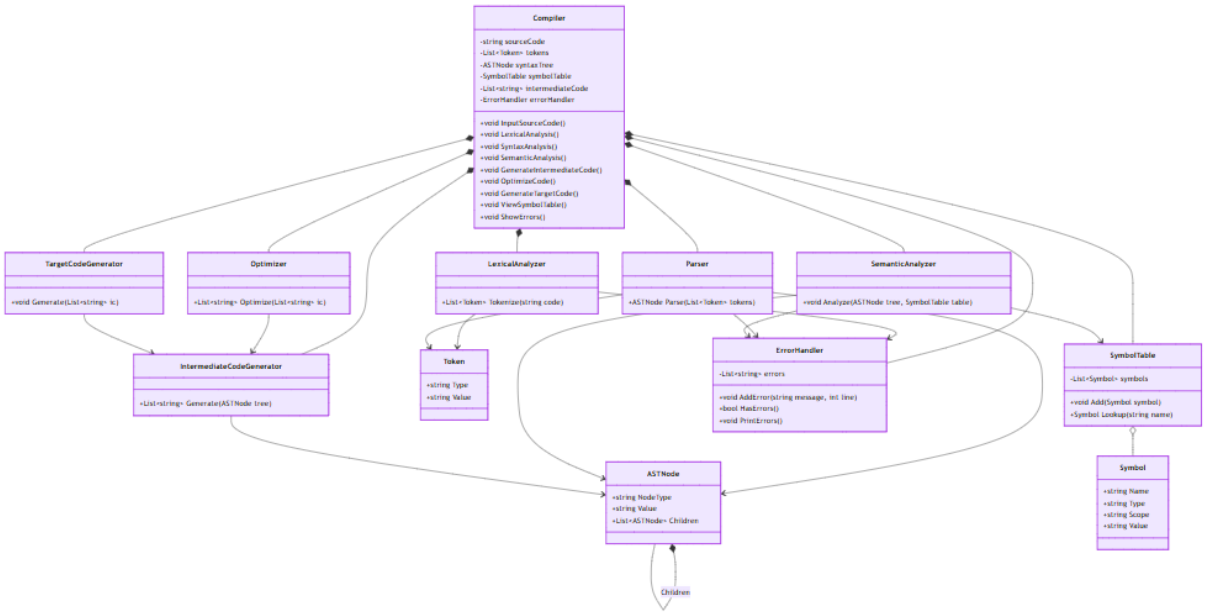
Class:
ErrorHandler
Dependencies:
Invoked by LexicalAnalyzer, Parser, and SemanticAnalyzer.

Benefits:

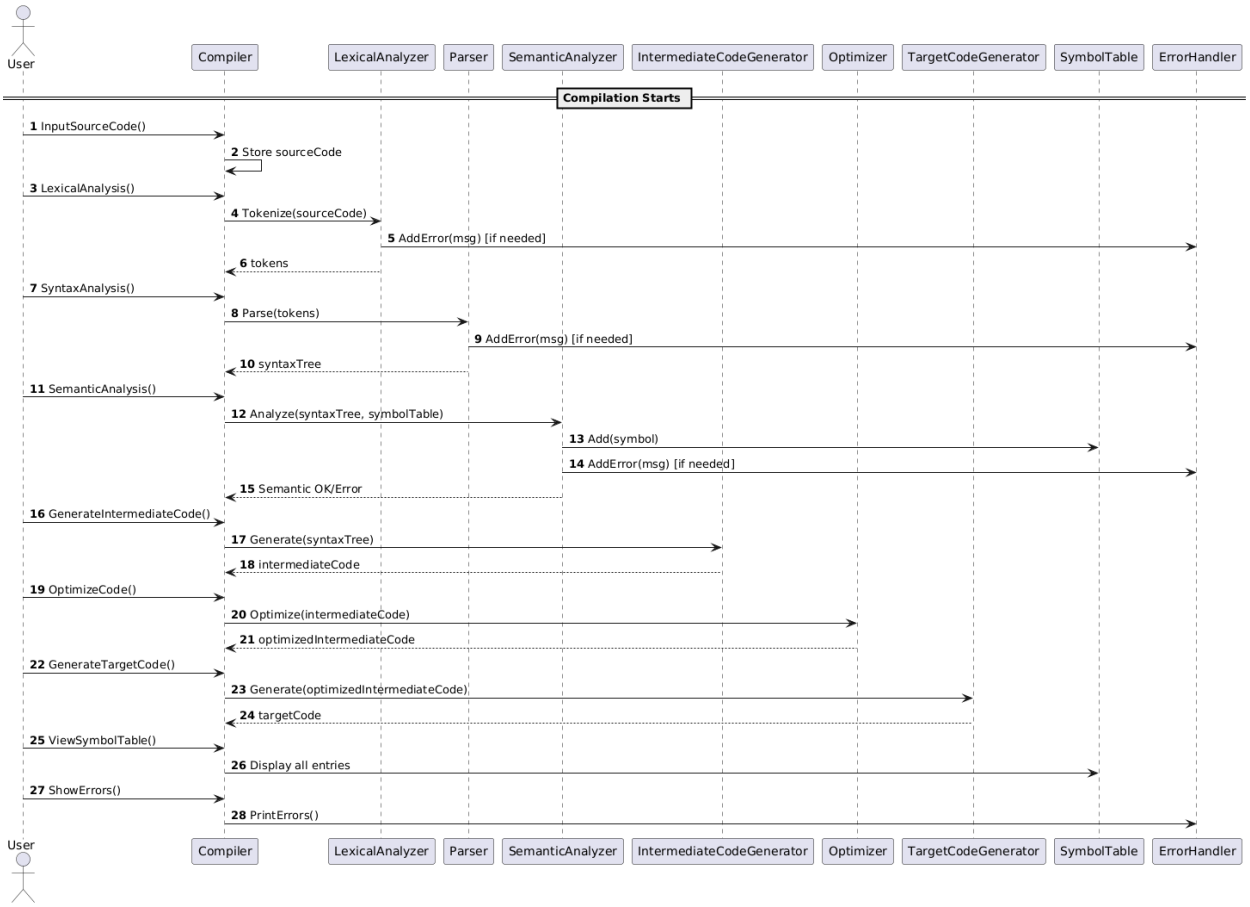
- Centralized error management.
- Encourages user-friendly feedback and debugging.
- Ensures the compiler continues scanning and reporting multiple issues in one run.

6. Diagrams

6.1 Class Diagram



6.2 Sequence Diagram



7. Testing and Validation

Purpose:

To ensure each phase of the mini compiler works as intended and handles both valid and invalid inputs gracefully.

7.1 Unit Testing

Each module was tested independently

Module	Test Case	Expected Result	Expected Result
LexicalAnalyzer	Input with mixed keywords and symbols	Token list with correct classifications	Passed
Parser	Invalid syntax	Syntax error reported	Passed
SemanticAnalyzer	Undeclared variable	Semantic error logged	Passed
IntermediateCodeGen	Valid syntax tree	Three-address code generated	Passed
Optimizer	Redundant assignment	Removed in optimized output	Passed
TargetCodeGenerator	Optimized code	Pseudo-assembly generated	Passed

7.2 Integration Testing

The complete pipeline was tested using full pseudo-code programs:

- Verified step-by-step execution from input to output.
- Ensured error propagation between stages was intact.

7.3 Validation

- Cross-checked symbol tables and intermediate code with hand-written expectations.
- Verified edge cases (missing semicolon, undeclared variables, etc.).
- Used negative testing to confirm error handling robustness.

8. Technologies Used

Technology	Purpose
Java (JDK 17+)	Core language used to implement the compiler.
IntelliJ IDEA	Primary IDE for code development and testing.
PlantUML	Used to create sequence diagrams.
Mermaid JS	Used to design class diagrams.
JUnit (optional)	For writing test cases (if applicable).
Git/GitHub	Version control and collaboration.
Markdown/Word	Documentation of project deliverables.

9. References

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd ed.)*. Pearson Education.
2. Appel, A. W. (1998). *Modern Compiler Implementation in Java*. Cambridge University Press.
3. Crafting Interpreters by Bob Nystrom — https://craftinginterpreters.com
4. GeeksforGeeks.org – Compiler Design Tutorials
5. TutorialsPoint.com – Compiler Design Basics
6. Oracle Java Documentation – https://docs.oracle.com/en/java
7. PlantUML Official Docs – https://plantuml.com
8. Mermaid Live Editor – https://mermaid.live