Design and Optimization of a Mini Compiler for Subset of C++

Yasir Fareed, Umer Qazi, Huma Ibrar

February 15, 2026

Abstract

This paper presents the design and implementation of a Mini Optimizing Compiler (MOC) for a subset of C++ language. The MOC is capable of performing lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization (constant folding and dead code elimination), and code generation. The paper emphasizes the key optimization techniques and illustrates the modular design of the compiler. The effectiveness of the compiler is demonstrated through error reporting and the generation of assembly like instructions from the input source code.

1 Introduction

Compilers are an essential tool for translating high-level programming languages into machine-readable code. This paper discusses the development of a Mini Optimizing Compiler (MOC), focusing on core compilation phases such as lexical analysis, syntax validation, semantic checking, intermediate code generation, and optimization techniques. The project implements a subset of C++ and demonstrates optimization methods like constant folding and dead code elimination.

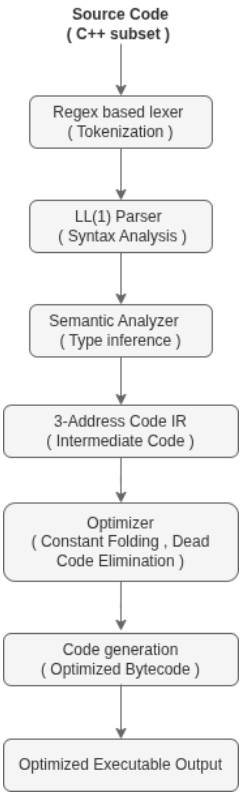

Figure 1: End-to-End Pipeline Diagram

2 Objectives

The primary objectives of this research are:

- To build a lexer using regular expressions that tokenizes the source code into meaningful tokens (keywords, identifiers, literals, and operators).
- To implement an LL(1) parser that ensures syntactical correctness through recursive descent parsing and validates grammar compliance.
- To perform semantic analysis, ensuring type correctness, symbol validation, and type inference for variables declared with `auto`.
- To generate intermediate code in the form of Three Address Code (TAC) from the abstract syntax tree (AST) representation.
- To optimize the TAC using constant folding (evaluating constant expressions at compile time) and dead code elimination (removing unused computations).
- To translate the optimized TAC into assembly like code and execute it using a register based virtual machine interpreter.

3 Methodology

The methodology behind the MOC follows a traditional multi phase approach, which includes:

- **Lexical Analysis:** Tokenizes the source code using regular expressions, transforming the input into manageable components (keywords, operators, literals, etc.). This phase is implemented through the `Lexer` class.
- **Syntax Analysis:** Uses an LL(1) parser, where recursive descent parsing handles expressions, variable declarations, and control flow statements such as `if`, `while`, and `cout`. This phase is implemented through the `Parser` class.
- **Semantic Analysis:** Ensures type safety, symbol validation, and type inference for variables declared with `auto`. It checks compatibility for expressions and assigns types based on their initializations. This phase is implemented using the `SemanticAnalyzer` class.
- **Intermediate Code Generation (TAC):** Converts the Abstract Syntax Tree (AST) into Three Address Code (TAC), which is an intermediate representation that simplifies optimization and code generation. The `IRGenerator` class is responsible for TAC generation.
- **Optimization:** Implements constant folding to evaluate constant expressions during compilation and dead code elimination to remove unused variables and computations.
- **Code Generation:** Converts optimized TAC into assembly like code, ready for execution on a virtual machine or further processing. The assembly code is generated using a simplified register based model, and instructions follow a load compute store pattern.

4 Project Design and Code

The following sections highlight the key implementation steps:

4.1 Lexer Implementation

- **Tokenization:** The `Lexer` class defines a set of regular expressions for different token types (keywords, operators, literals, etc.). The `token_specification` list specifies regex patterns for these tokens, and the `_tokenize` method processes the source code into a list of tokens.
- **Error Handling:** Any invalid characters encountered during lexing are reported as lexical errors.

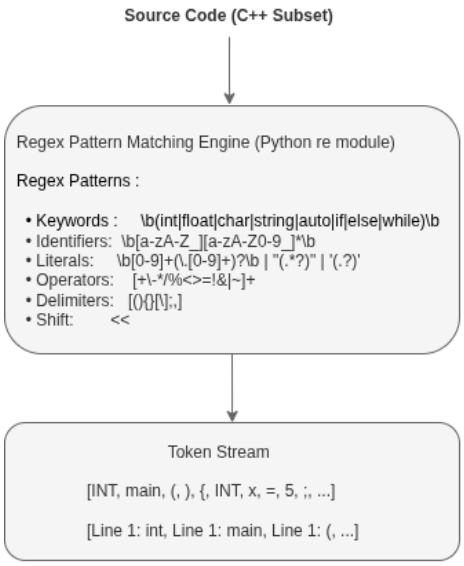

Figure 2: Lexer implementation flow: Source code processed through regex pattern matching into token stream

4.2 Parser Implementation (LL(1))

- **Recursive Descent Parsing:** The parser functions (`statement`, `var_decl_or_function`, `assignment`, etc.) correspond to grammar rules and parse the tokens into an Abstract Syntax Tree (AST).
- **Error Reporting:** The parser uses `expect` and `advance` methods to validate tokens and ensure grammatical correctness. Errors are reported for unexpected tokens.

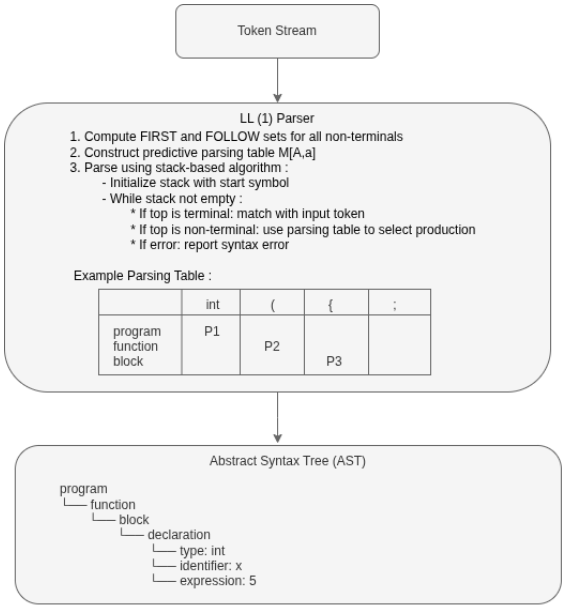

Figure 3: LL(1) Parser implementation flow

4.3 Semantic Analysis

- **Symbol Table:** The `SemanticAnalyzer` class maintains a symbol table mapping variable names to their types. It checks for undeclared variables, duplicate declarations, and type mismatches.
- **Type Inference:** The analyzer handles `auto` type declarations by inferring types from initializers. It also performs type compatibility checks for operations.

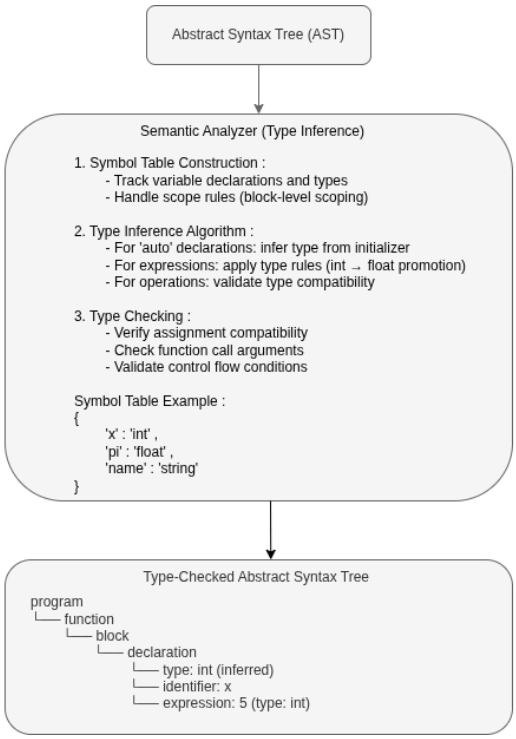

Figure 4: Semantic Analyzer with Type Inference implementation flow

4.4 Intermediate Code Generation

- **TAC Generation:** The `IRGenerator` class converts the AST into TAC instructions. Each operation (assignment, arithmetic, relational, print) generates a corresponding TAC instruction (e.g., `t1 = a + b`).
- **Temporary Variables:** Intermediate results are stored in temporary variables (`t1`, `t2`, etc.), and labels are used for control flow operations.

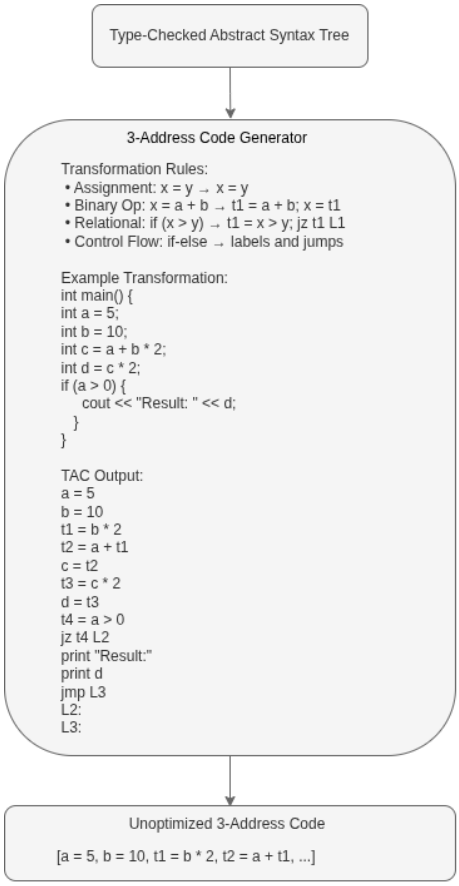

Figure 5: Intermediate code generation process: Transformation from type-checked abstract syntax tree to unoptimized Three-Address Code (TAC)

4.5 Optimization Techniques

- **Constant Folding:** Constant expressions (e.g., `5 + 10`) are evaluated during compilation, and redundant operations are eliminated.
- **Dead Code Elimination:** Code that computes values not used later is removed, improving performance.

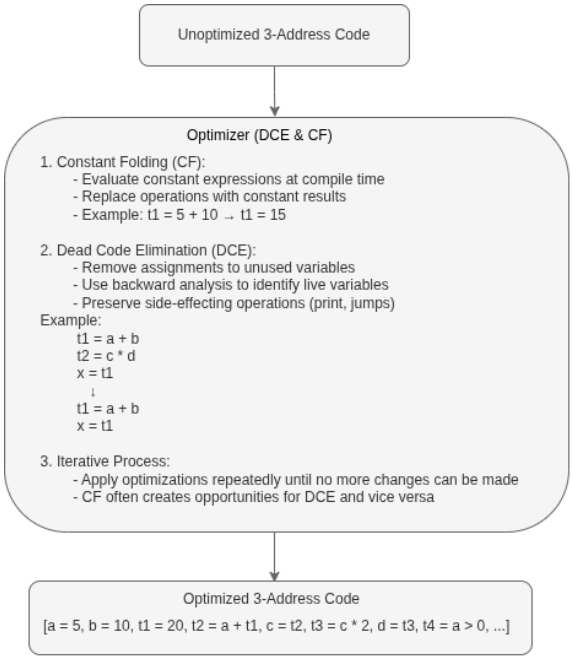

Figure 6: Optimization workflow: Application of constant folding (CF) and dead code elimination (DCE) to transform unoptimized Three-Address Code into optimized form

4.6 Code Generation

- **Assembly-like Code:** The optimized TAC is translated into assembly-like instructions, which can be executed or further processed. Instructions are mapped to simple operations such as `MOV`, `ADD`, `JMP`, etc.

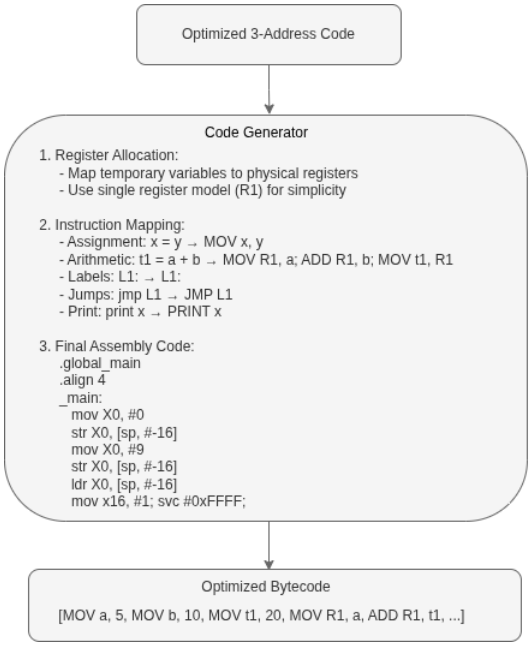

Figure 7: Code generation process: Transformation of optimized Three-Address Code (TAC) into assembly-like instructions and optimized bytecode

5 Results

- **Compilation Phases:** The paper demonstrates the MOC's ability to handle multiple C++ constructs, such as arithmetic and relational expressions.
- **Output Example:** A sample input program (e.g., calculating the sum of `a + b * c`) is provided, showing the tokenized output, AST, TAC, optimized TAC, assembly instructions, and final output.
- **Optimization Results:** Constant folding and dead code elimination improve efficiency by reducing the number of operations and memory usage.

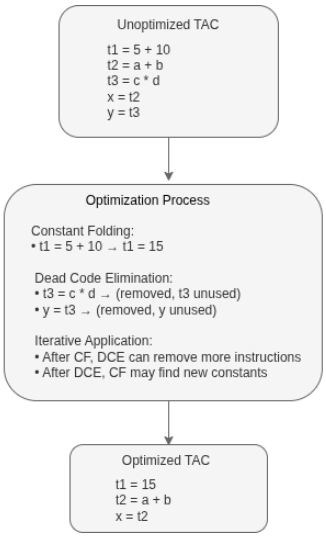

Figure 8: Optimization process demonstration: Transformation of unoptimized Three-Address Code through constant folding and dead code elimination to produce optimized intermediate representation

6 Error Handling

The compiler includes comprehensive error reporting across all phases:

- **Lexical Errors:** Invalid identifiers or characters.
- **Syntax Errors:** Missing semicolons or mismatched parentheses.
- **Semantic Errors:** Type mismatches or undeclared variables.

7 Conclusion

The Mini Optimizing Compiler successfully demonstrates the core phases of a compiler for a practical subset of C++. The implementation of optimization techniques like constant folding and dead code elimination improves runtime efficiency and reduces instruction count without altering the program's behavior. The compiler is modular, allowing for easy extensions and improvements in the future.

References

[1] Ghuloum, A. (2017). *New trends and challenges in source code optimization.* ResearchGate. https://www.researchgate.net/publication/336771654_New_trends_and_Challenges_in_Source_Code_Optimization

[2] Nwanze, A., & Daniel, N. N. (2022). *Compiler construction detail design.* SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5086763

[3] Dutta, D., Sonowal, N., & Hazarika, I. (2020). *Developing a modular compiler for a subset of a C-like language.* ResearchGate. https://www.researchgate.net/publication/364344665_Compiler_Construction_Detail_Design

February 15, 2026