




PROJECT AND TEAM INFORMATION

Project Title

Project C: Developing a Compiler for the C Programming Language

Student / Team Information

CD-VI-T017	
Team member 1 (Team Lead) Syed Mohd Mehdi Zaidi youshazaidi104@gmail.com 2021493 Section - A	
Team member 2 Piyush Pal piyushpalgg26@gmail.com 2021349 Section - B	
Team member 3 Kirri Tharun tharunprajitha2017@gmail.com 2021279 Section - E	

Team member 4

Tushar Dwivedi

tushar582002@gmail.com

2021763

Section – A-RQ



PROPOSAL DESCRIPTION

Motivation

The motivation for this project stems from an interest in the complexity of programming language translation and the mechanisms that drive software engineering today. Through the implementation of a C compiler from scratch, I will have direct exposure to the most significant features of a compiler, such as lexical analysis, parsing, semantic analysis, generation of intermediate code, and code optimization. This project bridges the gap between the abstract material learned through courses in compiler design and its concrete applications, giving a thorough overview of how source code is translated into executable code. Construction of a compiler also helps promote a greater understanding of language structure and the sophisticated decisions that accompany processing syntactic and semantic rules.

State of the Art / Current solution

C is a low-level programming language in systems programming, and compilers like GCC and Clang offer powerful, production-quality utilities that convert C source code to highly optimized machine code. These compilers go through complex phases that include lexical and syntactic analysis, optimization phases, and platform-dependent code generation. Even with the sophisticated capabilities and richness of these tools, their complexity poses daunting challenges to comprehensive study. Instructional efforts like MiniC, C-, and SmallC provide simplified models of C compilers, allowing learning-oriented study of compiler construction techniques. This project extends these teaching scaffolds, with emphasis on the implementation of basic C language features and main compilation phases using a modular architectural framework.

Project Goals and Milestones

The primary goal is to develop an instructional C compiler that facilitates easier transformation of C source code into intermediate assembly code or virtual machine representation.

Milestones:

1. Lexical Analysis

- Design token specifications using regular expressions.
- Implement a lexical analyzer to tokenize input source code.

2. Syntax Analysis

- Define the grammar for a subset of the C language.
- Build a parser (using recursive descent or a parser generator like YACC).
- Construct an Abstract Syntax Tree (AST).

3. Semantic Analysis

- Implement symbol table construction.
- Type checking and scope resolution.
- Handle semantic rules (e.g., variable declarations, type compatibility).

4. Intermediate Code Generation

- Generate three-address code (TAC) or other intermediate representations (IR).
- Translate AST into IR.

5. Code Optimization

- Implement basic optimization techniques like constant folding and dead code elimination.

6. Code Generation

- Translate IR into target code (assembly-like or VM code).
- Create a basic virtual machine or integrate with an assembler.

Project Approach

The compiler will be coded in C using a modular design that divides each of the compilation phases into distinct modules. The implementation will emphasize simplicity and readability over performance or completeness and thus is suitable for educational purposes.

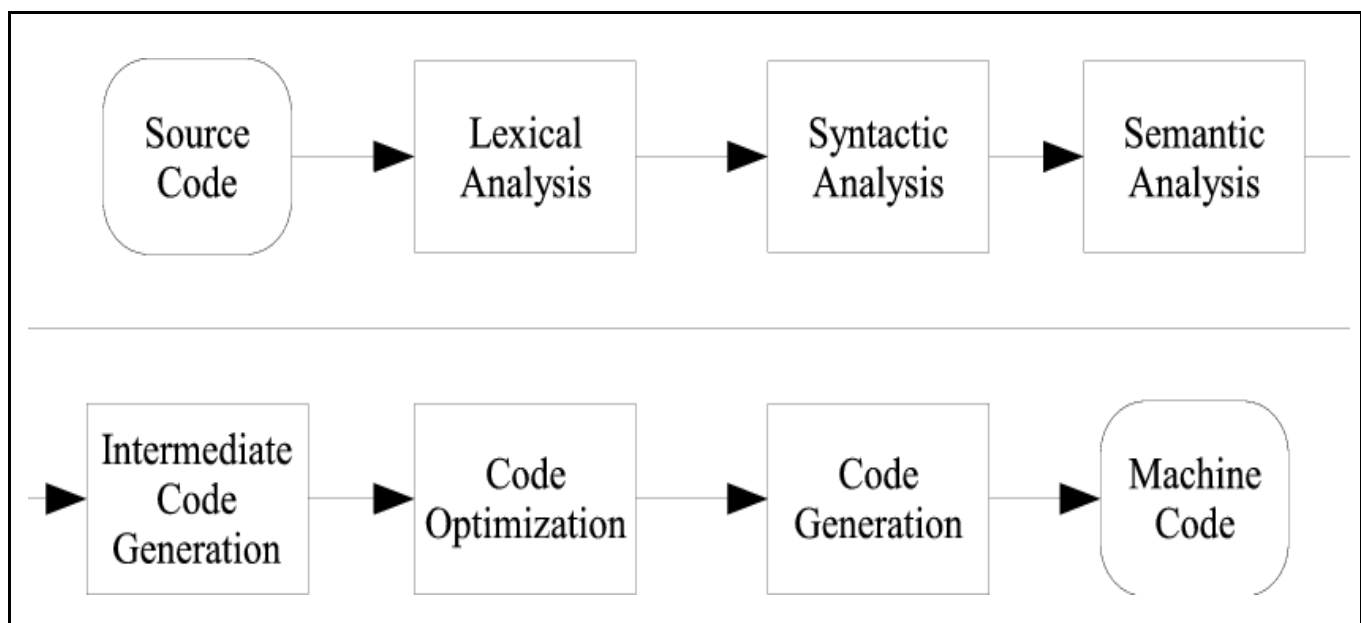
Core technologies and libraries:

- C Programming Language: For its low level functionality and educational worth.
- Custom Implementation of Lexical Analyser: To automatically generate lexical analyzers.
- Yacc: For syntax analysis.
- Standard C Libraries: stdio.h, stdlib.h, string.h, etc.

Compiler Architecture:

- Lexer: Translates input source code into a stream of tokens.://.
- Parser: Constructs the AST according to C grammar rules.
- Semantic Analyzer: Checks for correctness of types and builds symbol table.
- IR Generator: Produces intermediate code from the AST.
- Code Generator: Converts IR to low-level target code.

System Architecture (High Level Diagram)



Project Outcome / Deliverables

Functional Deliverables:

- Lexical analyzer for tokenizing C code.
- Parser and AST builder for a subset of C.
- Semantic analyzer with built-in symbol table management.
- Intermediate code generator.
- Code generator for a minimalist VM or a simplified assembly.

Final Deliverables:

- Complete C compiler (with make support).
- Sample test cases and corresponding C programs.
- Well-documented source code.
- Project report including architectural diagrams and descriptions.
- GitHub repository containing all the code, documentation, and usage instructions.

Assumptions

The project uses a Unix/Linux environment and a C compiler (e.g., GCC) to compile the compiler itself. It compiles to a simplified form of C, with no fancy features like pointers, dynamic memory allocation, and preprocessing directives. The output will be an intermediate representation or pseudo-assembly rather than actual machine code. The compiler will be aimed at educational simplicity rather than optimization or performance.

References

References:

- Compilers: Principles, Techniques, and Tools by Aho, Lam, Sethi, and Ullman (Dragon Book)
- Engineering a Compiler by Cooper and Torczon
- Flex and Bison documentation
- GitHub: tommyettinger/minic
- GitHub: rui314/9cc – A Small C Compiler
- Stanford CS143: Compilers Course Materials
- MIT OpenCourseWare: 6.035 Computer Language Engineering
- TutorialsPoint and GeeksForGeeks: Compiler Design Tutorials