Project Reflection

Mini-C Compiler

CS4031 - Compiler Construction | Fall 2025

Team Members

Subhan (22K-4316)

Sadiq (22K-4303)

Muhammad Ahmed Haque (22K-4232)

Muhammad Irtiza (22K-4638)

Umer (22K-4160)

Junaid (22K-4369)

December 2025

What We Learned

Building this Mini-C compiler provided deep, hands-on understanding of how programming languages are translated into executable code. The project reinforced that compilers are not monolithic black boxes but carefully orchestrated pipelines where each phase transforms data in well-defined ways.

Lexical Analysis

We learned how regular expressions and finite automata convert raw text into structured tokens. Understanding token precedence - matching keywords before identifiers - and how DFAs can be systematically constructed from regex patterns was crucial.

Syntax Analysis

Recursive descent parsing revealed the elegance of context-free grammars. We gained practical experience eliminating left recursion, handling operator precedence (especially for ternary and assignment operators), and constructing abstract syntax trees that preserve program structure.

Semantic Analysis

This phase taught us the complexity of type systems. Building symbol tables with proper scope chains, implementing type checking, and handling implicit conversions (integer promotions, pointer arithmetic, array decay) required careful attention to language semantics.

Intermediate Representation

Using TACKY three-address code showed us how high-level constructs decompose into simple operations. This abstraction layer cleanly separates frontend concerns (parsing, type checking) from backend concerns (register allocation, instruction selection).

Optimization

Even basic techniques like constant folding and dead store elimination can meaningfully reduce code size. We now appreciate why production compilers invest heavily in optimization passes.

Code Generation

Generating x86-64 assembly taught us low-level details: the System V calling convention, register constraints, stack frame layout, and instruction fixing for two-operand instructions. Translating IR to working machine code was deeply satisfying.

Additional Skills

Beyond compiler theory, we improved our Python programming skills (classes, recursion, pattern matching), applied data structure knowledge practically (trees, hash maps, linked structures), and developed systematic testing methodologies.

What We Would Improve

Error Reporting

Our compiler identifies errors but lacks precise line numbers and source context. Better diagnostics would make debugging user programs much easier.

Register Allocation

Currently, we spill most values to the stack. Implementing graph coloring or linear scan allocation would produce more efficient code.

Optimization Depth

We implemented basic optimizations but omitted common subexpression elimination, loop-invariant code motion, and strength reduction - techniques that would significantly improve generated code quality.

Language Coverage

Adding switch statements, enum types, typedef, and preprocessor directives would make Mini-C more practical for real programs.

Multiple Targets

The compiler only targets x86-64. Adding ARM64 or RISC-V backends would demonstrate true portability of our IR design.

Conclusion

This project transformed abstract compiler theory into concrete understanding. We now see compilers not as magic but as engineering - careful design, modular phases, and systematic testing. These skills extend beyond compilers to language tooling, static analysis, and software architecture generally.

Team Contributions

Member	Roll Number	Contributions
Subhan	22K-4316	Documentation, Implementation
Sadiq	22K-4303	Documentation, Implementation
Muhammad Ahmed Haque	22K-4232	Documentation, Implementation
Muhammad Irtiza	22K-4638	Documentation, Implementation
Umer	22K-4160	Documentation, Implementation
Junaid	22K-4369	Documentation, Implementation