Comparison of Rust Implementation to Original C4 Compiler

Meera Alawadhi

Hamda

May 1, 2025

Introduction

The C4 compiler, first built in C, is a compact tool designed to compile a simplified version of C code. Our project, c4_rust_team_alfaqa, reimagines this compiler in Rust, focusing on the lexing and parsing stages to create an Abstract Syntax Tree (AST). This report dives into how the Rust version stacks up against the original C implementation, exploring Rust's safety advantages, performance trade-offs, and the hurdles we faced in keeping the two versions aligned.

Memory Safety

The C version relied on manual memory handling with malloc and free, which could lead to issues like memory leaks or dangling pointers, especially when juggling strings or tokens. Rust handles memory automatically through its ownership system. In our lexer (c4.rs), we used String and Vec<char> to process input, letting Rust clean up memory when it was no longer needed—no free calls required. The parser's AST, built with enums like Expr and Stmt, used Box<Expr> for nested expressions, avoiding the null pointer risks common in C's raw pointers. Despite these changes, the Rust version produced the same AST as C4 for code like if (x) { return 5; } else { return 0; }.

Lifetimes

Rust's lifetime rules ensured we didn't accidentally access invalid memory. In the lexer, we converted the input string into a Vec<char> owned by the Lexer struct, sidestepping complex borrowing issues that C's char* pointers demanded. The parser's current_token: Result<Token, String> let us safely borrow tokens during parsing, preventing errors like data races. These features kept the compiler's single-pass parsing behavior intact while making it more robust.

Performance Differences

We didn't run detailed performance tests since our focus was on getting the parsing right, but we can make some educated guesses about how Rust compares to C when handling code like { x = 5; printf("%s", p); return 0; }.

Rust's abstractions, like String and Vec, are designed to be nearly as fast as C's raw arrays and pointers. The lexer's Vec<char> might add a small allocation cost compared to C's direct char* access, but Rust's optimizations keep this minimal. The parser, which uses a single-pass, recursive-descent approach in both versions, should perform similarly. Rust's bounds checking adds a slight overhead absent in C, but for C4's typical inputs, this barely matters. With Rust's optimized builds (cargo build –release), we expect performance to be within 10–20% of C's, close enough for C4's purposes. Future work, like adding a virtual machine, would need proper benchmarking to confirm this

Challenges and Solutions

Replicating C4's behavior in Rust presented several challenges, primarily due to differences in language paradigms and safety requirements.

Challenge: Replicating C's Flexible Parsing

C4's C version was forgiving, often overlooking minor syntax errors like missing semicolons by skipping tokens. Rust's strict type system pushed us to be more explicit. Our parser (parse_stmt in c4.rs) enforces rules like semicolon termination, returning errors for invalid syntax. To stay compatible, we ensured the AST matched C4's for valid code, using Result<Token, String> to handle errors cleanly while keeping error messages similar to the original.

Challenge: Test Failures and Debugging

Early tests, like test_parse_block and test_parse_if_else_stmt, failed due to issues with semicolons and else branches. We simplified parse_stmt to parse expressions first, then check for = or ;, and fixed parse_block to handle braces correctly. Rust's ownership rules caused an E0507 error in parse_block, which we resolved by borrowing &self.current_token, aligning with Rust's constraints while keeping C4's logic intact.

Conclusion

Our Rust version of C4 faithfully recreates the original's lexing and parsing while tapping into Rust's safety features to eliminate C's common pitfalls, like memory errors. Performance is likely very close to C's, with Rust's small overhead balanced by greater reliability. We navigated challenges like C's lenient parsing and unsafe strings through careful design, delivering a solid foundation for future additions, like a virtual machine. This project shows Rust's power for building reliable, high-performance systems like compilers.