C4 Rust Comparison

Date of Submission: May 4, 2025

COSC 320 – Principles of Programming Languages

Spring 2025

Instructors: Dr. Davor Svetinovic

Group Members:

Abdullah Alfalasi 100053699

Khalifa Aljasmi 100061653

Introduction

This report compares the Rust implementation of the C4 compiler with the original C version, focusing on Rust's safety features, performance differences, and challenges in replicating C4's behavior. The goal was to maintain compatibility with C4's functionality while leveraging Rust's modern features.

Rust's Safety Features and Design Impact

Rust's safety guarantees, such as memory safety and lifetimes, significantly influenced the design of the C4 compiler implementation while ensuring compatibility with the original C version. In the C version, memory management is manual, relying on 'malloc' and 'free', which can lead to issues like dangling pointers or memory leaks. Rust eliminates these risks through its ownership model and borrowing rules. For instance, in the Rust implementation, data structures like the token vector in the lexer are managed using Rust's 'Vec', which automatically handles memory allocation and deallocation, preventing memory leaks without explicit intervention. Lifetimes in Rust ensured that references to tokens or AST nodes remained valid throughout their use. In the parser, when passing references to tokens, Rust's borrow checker enforced that these references did not outlive the underlying data, avoiding dangling pointer issues common in C. This required careful structuring of the code, such as ensuring the token vector lived as long as the parser instance, but it guaranteed safety without sacrificing compatibility with C4's tokenization and parsing logic. Rust's strict type system also impacted error handling. The C version uses return codes and global state (e.g., 'exit' on errors), which can obscure error origins. In contrast, Rust's 'Result' and 'Option' types allowed for explicit error handling in the lexer and parser.

Performance Differences

Qualitatively, the Rust implementation exhibits performance characteristics comparable to the C version when compiling the same C code, though some differences arise due to Rust's safety overhead and optimizations. The C version benefits from minimal runtime checks, as it directly manipulates memory and lacks bounds checking. For a simple program like 'int main() return 0; ', the C version compiles slightly faster due to its lightweight nature—typically completing in under 0.1 seconds on a standard machine.

The Rust implementation, compiled in ‘dev‘ profile (unoptimized with debug info), takes slightly longer—around 1–2 seconds for compilation and execution of the same program, as seen in build logs. This is due to Rust’s bounds checking on vectors (e.g., in the lexer’s token array) and ownership checks at runtime. However, when compiled in ‘release‘ profile with optimizations, Rust’s performance approaches that of C, often completing in under 0.5 seconds. Rust’s zero-cost abstractions, such as its iterator API, allowed the lexer to process characters efficiently, matching C’s ‘for‘ loop performance while maintaining safety.

Challenges in Replicating C4’s Behavior

Replicating C4’s behavior in Rust presented several challenges, primarily due to differences in language paradigms. One significant challenge was handling C4’s permissive syntax, such as not requiring semicolons after certain statements (e.g., function definitions). In the Rust implementation, the parser initially enforced semicolons too aggressively, leading to errors like “Missing semicolon at position 8 with token Operator(’’)” for ‘int main() return 0; ‘. This was resolved by introducing an ‘inblock‘flag in the parser to differentiate between level statements and block–level statements, ensuring function definitions were parsed without expecting a trailing semicolon, thus maintaining compatibility with C4’s syntax. Another challenge was replicating C4’s error recovery. The C version often terminates on errors, whereas Rust’s ownership rules made it harder to “bail out” mid-parsing without unwinding the stack. To address this, the Rust implementation used ‘Result‘ to propagate errors up the call stack, allowing the parser to report issues (e.g., missing semicolons) and halt gracefully, mirroring C4’s behavior while improving error clarity. Finally, C4’s reliance on global state for symbol tables was incompatible with Rust’s safety model. The Rust implementation encapsulated state within structs like ‘Parser‘ and ‘Lexer‘, using Rust’s ownership to manage access. This ensured thread safety and prevented data races, aligning with Rust’s principles while preserving C4’s functionality.

Conclusion

The Rust implementation of C4 successfully maintains compatibility with the original C version while leveraging Rust's safety features to enhance reliability. Memory safety and lifetimes eliminated common C pitfalls, though they introduced slight performance overhead in unoptimized builds. Challenges in syntax parsing and error handling were addressed through careful design, ensuring the Rust version mirrors C4's behavior with improved safety and maintainability.