Comparison Report: Rust Implementation vs. Original C4 Compiler

• Introduction

This report outlines the design, performance, and development challenges encountered while rewriting the C4 compiler in Rust. The original C4 compiler, written in C, is a minimal compiler that demonstrates basic tokenization, parsing, and virtual machine execution for a small subset of the C language. The Rust version was developed with the goals of preserving functional equivalence, enabling self-hosting, and leveraging Rust's modern language features to improve code safety and maintainability.

• Design and Safety

Rust's Safety and Abstraction

The Rust implementation uses its powerful type system and ownership model to eliminate common errors such as memory leaks, buffer overflows, and data races issues inherent in the original C implementation. Instead of relying on manual memory management (e.g., `malloc`, `free`), the Rust version uses safe abstractions like `Vec` and `Option`. This prevents common pitfalls and contributes to a more robust design.

Modular Architecture

Both implementations follow a modular design separating lexing, parsing, and execution. In the Rust version, the code is divided into modules (`lexer.rs`, `parser.rs`, and `vm.rs`). Rust's pattern matching and enum types were used to represent tokens and opcodes, replacing integer constants in C. This change improves readability and minimizes errors in handling different language constructs.

Error Handling

Unlike the original C code, which aborts on error (using `exit(-1)`), the Rust version employs the `Result` type for error propagation. This allows the compiler to handle errors gracefully and provide meaningful diagnostic messages to the developer, enhancing maintainability and ease of debugging.

• Performance Comparison

Execution Speed

Preliminary benchmarks indicate that the Rust implementation is competitive with the original C version. In some cases, Rust's zero-cost abstractions and aggressive compile-time optimizations allow it to match or slightly exceed the performance of the C implementation. Although the VM loop in both versions exhibits similar behavior, Rust's safety checks (which are largely optimized away in release builds) do not introduce a significant overhead.

Memory Usage

The Rust version's use of `Vec` for dynamic arrays and its stack-based VM contribute to efficient memory usage. The absence of manual pointer arithmetic and explicit memory management improves reliability without substantially increasing memory consumption compared to the original C implementation.

- # Challenges and Solutions

Ownership Model and Borrowing

One of the main challenges was adapting the C code's pointer arithmetic and manual memory management into Rust's ownership and borrowing model. To address this, dynamic arrays (e.g., `Vec`) replaced raw memory buffers, and functions were refactored to pass slices and references. This required a careful redesign of the VM's stack management, ensuring that all operations were memory-safe.

Parsing and Code Generation

Translating the recursive descent parser involved rethinking the design to utilize Rust's pattern matching. While the logic closely mirrors that of the C version, the refactoring necessitated a more functional style. The resulting parser uses explicit error handling (via `Result`) and recursive functions that provide clearer control flow and better maintainability.

Integration of Unit Testing and Documentation

A significant advantage of the Rust ecosystem is its built-in testing framework. Implementing unit tests for the lexer, parser, and VM not only ensured correctness during incremental development but also enhanced the overall code quality. Comprehensive documentation using Rust doc comments helped clarify design decisions and usage, ensuring that future maintainers can understand the system without needing to decipher legacy C code.

- # Conclusion

The Rust reimplementation of the C4 compiler successfully preserves the functionality of the original while significantly enhancing safety, maintainability, and clarity. Although challenges arose particularly in adapting C's memory management paradigms the use of Rust's type system, error handling, and testing infrastructure resulted in a robust, efficient, and modern compiler design. The performance remains competitive, and the improved error reporting and documentation pave the way for future extensions, including self-hosting and support for additional C constructs.