Comparison Report: Rust vs. C Implementation of the C4 Compiler

By: Salem Al Haddad (100059213) and Ahmed Al Nuaimi (100058862)

This was a fun project!

Overview

The C4 compiler is a minimalist, self-hosting C compiler written in C by Robert Swierczek. Its Rust implementation was developed to maintain compatibility while leveraging Rust's safety guarantees, modern tooling, and modular design. This report compares the design decisions, safety improvements, architecture, feature additions, and limitations observed during the reimplementation. Although most unit tests passed, a few critical tests failed, preventing full runtime correctness. These are detailed in the testing section below.

---

1. Safety Features

Memory Safety

- **C Version**: Relies on manual memory management with raw pointers (`char *p`, `int *e`, etc.), and operations such as `malloc`, `memset`, and unchecked pointer arithmetic.
- **Rust Version**: Uses `Vec`, `Option`, and Rust's ownership model to enforce memory safety. Raw pointers and unsafe blocks are minimized or eliminated.
- **Impact**: Eliminates common C issues like use-after-free, buffer overflows, and memory leaks.

Type Safety

- **C Version**: Uses C-style enums and integer values for tokens, types, and opcodes.
- **Rust Version**: Introduces strongly typed enums like `Opcode`, `Type`, and `Token`, enabling exhaustive pattern matching and compile-time type checking.

- **Impact**: Reduces runtime errors due to type mismatches and makes the code easier to maintain.

Lifetimes and Borrowing

- **Rust's Model**: Forced explicit structuring into encapsulated components (e.g., `Lexer`, `Parser`, `VM`) with controlled data flow.
- **Result**: Improved modularity and testability at the cost of added complexity and learning curve.

---

2. Code Organization and Maintainability

Modular Design

- **C Version**: Single monolithic file with tightly coupled code.
- **Rust Version**: Modular organization with the following structure:
  - `symbol_table.rs`: Symbol table management
  - `types.rs`: Type system
  - `declaration.rs`: Declaration parsing
  - `expression.rs`: Expression parsing
  - `statement.rs`: Statement parsing
- **Result**: Greater clarity, reusability, and testability.

Error Handling

- **C Version**: Uses `printf` and manual error codes.

- **Rust Version**: Provides richer error messages, debug logs (e.g., `println!("DEBUG: ...")`), and better edge case handling.
- **Result**: Significantly improved debugging and maintainability.

---

3. Performance Considerations

- **Startup & Execution**: C executes slightly faster due to its minimal abstraction and lack of runtime checks.
- **Rust Overhead**: Introduces minor overhead from safety checks but offers stable, predictable behavior.
- **LLVM Backend**: Rust compiles via LLVM, producing highly optimized machine code—runtime performance remains competitive with C in most scenarios.

---

4. Challenges and Compatibility Solutions

Challenge	Original C (C4)	Rust Implementation
Pointer-heavy Design	Raw global pointers	Replaced with `Vec`, slices, and references
Global Symbol Table	Array-based flat layout (`id[Tk]`)	Encapsulated `HashMap<String, Symbol>` with structured access
Manual Memory Management	Uses `malloc` and pointer math	Replaced by safe allocation (`Box`, `Vec`)
Expression Parsing	Recursive descent with unsafe recursion	Used enums and `match` expressions for structured control flow
VM Stack & Execution	Raw memory pointer arithmetic	Safe stack abstraction using `Vec<i64>` with bounds checks

---

5. Feature Additions

Enhanced Type System

- Support for both `int` and `char`
- Pointer types via `Type::Ptr`
- Type size calculations (1 byte for `char`, 4 bytes for `int`/pointers)
- Type checking and conversion

Improved Symbol Table Management

- Hierarchical scope system using a scope stack
- Accurate symbol resolution with `lookup` and `lookup_current_scope`
- Symbol classification: `Global`, `Local`, `Function`, `Sys`
- Memory management for local variables

Enhanced Expression Parsing

- Full operator precedence support (14 levels)
- Support for postfix operators: `++`, `--`, `[]`
- Structured handling of function calls and arguments
- Improved debugging and error handling

Better Code Organization

- Modular parser structure
  - `symbol_table.rs`, `types.rs`, `declaration.rs`, `expression.rs`, `statement.rs`

Improved Error Handling

- More expressive error messages
- Debug outputs for internal states
- Handles syntax edge cases more robustly

Enhanced Memory Management

- Accurate tracking of local variable offsets
- Scoped memory control with `enter_scope` and `exit_scope`
- Improved handling of function parameters

Improved System Function Support

- Built-in support for:
    - `open`, `read`, `close`
    - `printf`
    - `malloc`, `free`
    - `memset`, `memcmp`
    - `exit`

Better Code Generation

- More efficient output
- Proper argument passing
- Improved memory management inside the VM

---

6. Unit Testing Results

Failed Tests

- **Hello World**
  - **Status**: ❌ Failed
  - **Error**: Expected ',' or ')' in function call, found: `Some(Str("Hello, World!\n"))`
  - **Cause**: Issue with `printf` syntax parsing
- **Factorial**
  - **Status**: ❌ Failed
  - **Error**: Expected ';' after return statement, found: `Some(Num(1))`
  - **Cause**: Syntax misinterpretation in `return` parsing

Successful Tests (10 Passed)

- Lexer tests
- Basic parser tests
- VM instruction tests
- Type system tests
- Symbol table tests

Interpretation: The core components (lexer, parser, VM) are functional, but end-to-end compilation is still failing due to edge-case syntax errors.

---

7. Known Limitations

- The Rust implementation does **not** yet generate correct runtime behavior for all compiled programs.
- While individual units function well, full programs (like `printf("Hello")`) still have parsing and VM execution issues.
- Debugging and refinement are needed for correct code generation and virtual machine stack behavior.

Conclusion

Translating C4 to Rust led to a safer, better-structured, and more maintainable compiler. Despite some runtime execution bugs, the Rust version excels in memory safety, modularity, and expressiveness. The effort highlights the tradeoff between low-level control in C and compile-time guarantees in Rust.

With continued development, this Rust-based C4 implementation could evolve into a powerful educational tool, maintaining compatibility with the original while avoiding its common pitfalls.