C4 Compiler Rust Implementation Report

Project overview

This project is a Rust implementation of the C4 compiler, originally written in C by Robert Swierczek. The C4 compiler is a minimalist, self-hosting C compiler designed for educational purposes. It compiles a subset of the C language into simple virtual machine code which can be interpreted.

Our Rust implementation preserves the core functionality of the original C4 compiler while leveraging Rust's memory safety features and modern programming paradigms.

Architecture Overview

The C4 Rust compiler follows the traditional compiler pipeline:

1. Lexical Analysis - Tokenizes the source code
2. Parsing & Semantic Analysis - Builds an abstract representation while checking types
3. Code Generation - Emits bytecode for the virtual machine
4. Interpretation - Executes the generated bytecode

All of these stages are implemented in a single Rust file (`main.rs`), maintaining the compact and educational nature of the original.

Core Components

1. Lexer Implementation

The lexer (`next()` function) converts the source code into a stream of tokens. Key components include:

- Token recognition for C language constructs
- Symbol table management
- Identifier and keyword differentiation
- Literal handling (numbers, strings, characters)

2. Parser and Code Generator

The parser is implemented using a recursive descent approach with precedence climbing for expressions (`expr()` function). It directly emits bytecode during parsing, a technique known as "syntax-directed translation."

Key parsing components include:

- Expression parsing (`expr()`)
- Statement parsing (`compile_statement()`)
- Control flow constructs (`compile_if_statement()`, `compile_while_statement()`)
- Function definitions (`compile_function_definition()`)

3. Symbol Table Management

The symbol table stores identifiers and their associated metadata:

- Name and hash value
- Storage class (global, local, etc.)
- Data type
- Runtime value

The Rust implementation uses a `Vec<Symbol>` to manage the symbol table, with functions for symbol lookup and insertion.

4. Virtual Machine

The virtual machine is a simple stack-based interpreter that executes the generated bytecode. It includes operations for:

- Arithmetic and logical operations
- Memory access
- Control flow
- Function calls
- System calls

Key Implementation Details

1. Type System

The compiler supports a simplified type system with:

- Basic types: CHAR, INT
- Pointer types: PTR

2. Memory Management

The memory model includes:

- Text segment (for code)
- Data segment (for global variables and literals)
- Stack (for local variables and function calls)

3. Runtime Support

The implementation includes runtime support for:

- Standard I/O operations (PRTF)
- Memory allocation (MALC, FREE)
- File operations (OPEN, READ, CLOS)

Rust Specific Improvements

Our Rust implementation includes several improvements over the C original:

1. Memory Safety- Leverages Rust's ownership system to prevent memory-related bugs
2. Error Handling - Uses Rust's Result type for proper error propagation
3. Modularity - Encapsulates functionality in a struct with impl methods
4. **String Handling** - Utilizes Rust's String type for safer string manipulation
5. Explicit Types - Uses enums for token types, opcodes, and other constants

Conclusion

This Rust implementation of the C4 compiler provides a more memory-safe and maintainable version while preserving the educational value of the original. It demonstrates how Rust's safety features can be applied to systems programming tasks traditionally done in C.

The implementation successfully compiles and runs the target subset of C, including the ability to compile itself when given the original C4 source code, thus preserving the self-hosting capability of the original.