COSC 320 Principles of Programming Languages
c4 Compiler Rust Implementation

Lina Benna 100063450
Farah Hussain Hassan 100061366

0.1 Overview

Translating the C4 compiler from C to Rust involved systematically mapping low-level C constructs and memory operations into Rust's safer and more structured paradigm. The main concern revolved around the use of pointers, which in C offered unchecked direct memory manipulation, while in Rust, the same functionality was simulated using Vec¡T¿ for dynamic arrays and indexed access, and structs to encapsulate and manage related data safely. Table 1 summarizes a few differences seen between the two implementations.

C Feature	Rust Equivalent
Pointers	`Vec<T>` with indexing; raw pointers (`*mut`, `*const`) in `unsafe` blocks only if necessary
Arrays	`Vec<T>` and &str slices
Structs	`struct` with `impl` blocks to simulate object-oriented design
Enums for Tokens/AST Nodes	`enum` with pattern matching
Function pointers / Jump tables	Closures, trait objects, or `match` expressions
Manual memory allocation	Smart pointers; `unsafe` used sparingly when needed

Table 1: Mapping C features to their Rust equivalents

1 Lexer

Translating C4's next() function into Rust replaced manual pointer arithmetic with safe indexing and owned data structures like Vec and String, leveraging Rust's Option, match, and borrowing system to eliminate undefined behavior and make the lexer's control flow and memory management explicitly safe and verifiable.

In C4, lexing was typically performed using direct pointer manipulation, for example incrementing a `char *src` pointer to read characters from the input. This can easily lead to memory errors like buffer overflows or dereferencing invalid memory. In contrast, our Rust lexer uses a `&str` slice for the source code and a `usize position` index to track reading progress. Moreover, the `advance()` method updates the `current_char` safely using bounds-checked indexing and stores the character in an `Option<char>`, making the end-of-file condition explicit and eliminating undefined behavior. This design decision is inherently safer than manipulating raw pointers.

Furthermore, the `Lexer` struct methods also use `&mut self` to mutate internal state in a controlled, exclusive way. This prevents bugs from shared mutable access, which are common in C programs when using global variables or passing around pointers. For example, the `peek()` method safely inspects the next character without advancing the state as seen in the code snippet below, preserving lexer integrity, unlike C4's lookahead logic which might rely on side-effect-laden pointer operations.

```
fn peek(&self) -> Option<char> { // sampe implementation as advance() method
        if self.position < self.source.len() {
            Some(self.source.as_bytes()[self.position] as char)
        } else {
            None
        }
    }
```

The Rust lexer also cleanly handles lexical constructs through `match` expressions. In C, this would require verbose `if-else` chains or fragile `switch` statements. Rust's `match` ensured all branches (e.g., for =, ==, !=, <=, etc.) are exhaustively checked at

compile time, reducing the chance of forgotten edge cases. Moreover, string and character literals were handled using owned `String` and UTF-8 compliant `char` types, which are safer and more expressive than C's `char*` or ASCII-based character literals.

2 Parser

The Rust implementation introduced modern error handling with Result and Option, eliminating many memory management issues through ownership rules, enhancing readability and safety with pattern matching, and providing more robust type safety.

Memory management in Rust was also handled automatically using ownership and borrowing. This is evident in how the parser's data is managed, with ownership being tracked and enforced by the Rust compiler. For example, String in Rust has ownership semantics, and it ensures that there are no double frees or memory leaks in most cases. The String type in Rust was used to hold input data, and it was automatically deallocated when the variable went out of scope. This differs significantly from C, where the programmer is responsible for allocating and freeing memory, which could lead to memory leaks if not managed properly.

Lastly, the recursive descent parsing in Rust was implemented using Rust's pattern matching, which simplified the recursive functions and made the control flow more readable. The use of match allowed for better handling of different cases, such as the result of parsing a specific token or expression. In c4, recursive descent parsing was done using simple if/else or switch statements, and it's more cumbersome to manage multiple token types or recursive cases. Below is a sample example of the readability behind Rust's match statement relative to C:

```
fn expr(&mut self, lev: i32) {
    let token = self.tk.clone();

    match token {
        Token::None => {
            eprintln!("{}: unexpected eof in expression", self.line);
            std::process::exit(-1);
        }
        ...
```

3 Virtual Machine

In c4, memory management is done through dynamic allocation (malloc) and pointer arithmetic. The VM sections like the stack, symbol table, text, and data areas are dynamically allocated based on predefined sizes. Pointers were used for manipulating memory locations. Rust relies on its ownership system to manage memory. The memory areas for the symbol table, text, and data were allocated using Vec, which automatically managed memory and ensured safety without needing explicit allocation and de-allocation.

Moreover, c4 uses argc and argv to handle command-line arguments. It checks for flags like -s (for source output) and -d (for debug) to control behavior. In our Rust implementation, the command-line arguments are collected into a Vec¡String¿, providing more flexibility and ease of use with Rust's iterator and collection features. See the code snippet below for reference.

```
let args: Vec<String> = std::env::args().collect();

let mut argc = args.len() - 1; // Exclude the program name
let mut argv = &args[1..];
```

Moreover, c4 exits the program when the EXIT instruction is encountered, using manual memory cleanup. In the Rust translation, we use return to exit from the main function when the EXIT instruction is encountered. The use of Vec ensures that memory management is automatically handled when the function ends, eliminating the need for explicit cleanup.

Overall, the Rust VM implementation for the C4 compiler follows the same core structure as the original implementation but benefits from Rust's memory safety features and modern language constructs. The use of Vec, pattern matching, and error handling blocks help manage memory safely while maintaining control over low-level operations like pointer manipulation, which would typically be handled manually in C.