Ghaya Almehairbi – 100062990
Farah Turkmani – 100061661

Github Link: https://github.com/Farah61661/c4_rust_Al_Heel.git

Rust vs. C4: A Short Comparison Report

This report compares the Rust-based re-implementation of the C4 compiler with its original C version by Robert Swierczek. We focused on how Rust's language features influenced the design and on observed performance differences when compiling and executing the same C4-supported C code.

1. How Rust's Safety Features Affected the Design

When we started rewriting C4 in Rust, one of the biggest changes was dealing with how Rust handles memory. In C, you can do almost anything with pointers which include risky or unsafe operations. Rust prevents such operations through strict rules about ownership, borrowing, and lifetimes, which made us rethink how things were structured.

For example, the original C4 implementation relied heavily on global variables to share state, such as the current token or the symbol table. In Rust, this approach wasn't practical. We had to group related data into structs and carefully manage how functions access or update them while respecting Rust's borrowing rules.

In addition, memory management also changed. Instead of manually allocating and freeing memory like in C, we used safe abstractions like Vec<T> to handle the stack, data segment, and memory simulation. This eliminated the risk of buffer overflows or dangling pointers. Since Rust enforces ownership and lifetime rules, most memory safety issues were caught at compile time. We also benefited from Rust's use of owned String types instead of raw pointers, which made managing text data more predictable.

Another major difference was error handling. In C, errors are often managed with return codes or simply ignored. Rust encourages using Result and Option types for safe and explicit error handling. While we mostly used panic!() and unwrap() in this project for simplicity, Rust's strict error expectations still helped us catch problems early and structure control flow more clearly.

We also took advantage of Rust's enums and pattern matching. Instead of defining token types with integers like in C, we used enums like Token::Number(i64) or Token::Identifier(String), which made the lexer and parser easier to read and harder to break.

So while Rust introduced some added complexity, it also gave us better safety and confidence in the correctness of our implementation. We were able to catch errors and issues early and the compiler was easier to maintain and extend.

2. Performance: C vs. Rust

From our testing, here's what we noticed:

1. **Speed**: The Rust version was slightly slower during compilation, mostly due to additional safety checks and stricter rules. But once it was compiled, the runtime was just as fast as C.
2. **Memory**: Rust used a bit more memory at runtime due to the internal safety mechanisms like bounds checking and ownership tracking. But, the impact was minor and didn't affect the overall responsiveness.
3. **Binary Size**: The Rust binary was noticeably larger and had a slightly longer startup time. This is likely due to static linking safety features. Rust's compiled binary was generally larger than the C version, but this didn't impact execution speed.
4. **Execution**: Execution of arithmetic-heavy code (such as return 8 + 9) and control structures showed no noticeable delay in either version.

In short, C was faster in some areas, but the difference wasn't huge. For small programs, both were very responsive.

3. Challenges We Faced (and How We Solved Them)

Pointers to Ownership
C4 uses raw pointers and manual memory management, which gave us a lot of freedom—but also a lot of risk. Rust doesn't allow that kind of flexibility, so we had to switch to using Vec, Box, and move semantics. It took some trial and error to get used to, but once we understood how ownership worked, it actually made things feel safer and more manageable.

Macros to Enums
In the original C4 code, tokens were just macros and integer values. We replaced those with Rust enums and used pattern matching to handle them. At first, it felt more complicated and verbose, but it made the code easier to read and harder to mess up.

Expression Parsing
One of the hardest parts was rewriting the expression parser (`expr(int lev)`). We had to carefully manage token lookahead without breaking Rust's borrow checker. Using iterators and recursive functions helped.

Debugging and Learning Curve
Rust's error messages were helpful but sometimes overwhelming. It took time to fix borrow checker issues, but once we got the hang of it, things were more predictable. Writing unit tests for each part (lexer, parser, codegen) really helped.

Final Thoughts:

Rewriting C4 in Rust was challenging, but we learned a lot. Rust forced us to write safer, cleaner code. It wasn't always easy—especially compared to how flexible (and risky) C can be—but the end result felt more solid.

In the end, Rust gave us a more modern and secure version of C4, and we're happy with how it turned out.

Output of the test1.c

```
Last login: Thu May  8 19:01:08 on ttys002
(base) farahtourkmani@farahs-MBP ~ % cd /Users/farahtourkmani/Desktop/c4_project


(base) farahtourkmani@farahs-MBP c4_project % cargo run -- src/test1.c
warning: unused variable: `arg_count`
   --> src/main.rs:650:25
    |
650 |                     let arg_count = self.stack.pop().unwrap() as usize;
    |                         ^^^^^^^^^ help: if this is intentional, prefix it
with an underscore: `_arg_count`
    |
    = note: `#[warn(unused_variables)]` on by default

warning: `c4_project` (bin "c4_project") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/c4_project src/test1.c`
exit(50)
(base) farahtourkmani@farahs-MBP c4_project %
```

Output of the testing using cargo run

```
running 17 tests
test tests::test_empty_string_tokenization ... ok
test tests::test_if_else_parsing ... ok
test tests::test_basic_function_tokenization ... ok
test tests::test_logical_expression_parsing ... ok
test tests::test_function_call_codegen ... ok
test tests::test_string_literal_tokenization ... ok
test tests::test_nested_multiplication_parsing ... ok
test tests::test_sum_expression_parsing ... ok
test tests::test_syscall_behavior ... ok
test tests::test_variable_assignment_parsing ... ok
test tests::test_variable_return_function ... ok
test tests::test_vm_addition ... ok
test tests::test_vm_branch_nonzero ... ok
test tests::test_vm_function_call ... ok
test tests::test_vm_branch_zero ... ok
test tests::test_vm_memory_access ... ok
test tests::test_while_loop_parsing ... ok

test result: ok. 17 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```