COSC320 - Principles of Programming Languages
Shawkah Group

1. Rust's Safety Features and Design Impact

Translating the original c4.c to Rust introduced several design changes primarily to uphold Rust's memory safety and ownership guarantees. Notable impacts include:

- **Ownership & Lifetimes**: The original C implementation uses raw pointers (char *p, int *e, etc.) to manage memory and state transitions. In Rust, this is translated using Vec, slices, and Box smart pointers, all of which require explicit ownership and borrowing rules. The Rust version avoids undefined behavior by making memory allocations explicit and safe.

- **Explicit Memory Management**: C4's memory areas (text, data, stack, symbol) are dynamically allocated via malloc in C. In Rust, these are replaced with vectors (Vec<T>) or Box<[T]>, ensuring bounds-checked access and preventing overflows.

- **Pattern Matching over Manual Control**: Where C uses large switch or if-else chains with token integers and opcodes, the Rust version often replaces these with enum types and match expressions, improving readability and compile-time exhaustiveness checks.

- **Type Safety and Enums**: The Rust implementation leverages enum types (e.g., Token, Opcode) instead of plain int, reducing logic errors tied to integer misinterpretation.

2. Performance Considerations

- **Qualitative Comparison**: C remains slightly faster in raw performance due to its low-level optimizations and lack of bounds checks. However, Rust's performance is very close and far more reliable in preventing memory errors or segmentation faults.

- **Memory Overhead**: Rust's additional safety comes with some overhead, particularly from bounds checking, safe memory wrappers, and panic handling. However, the zero-cost abstractions and inlining by LLVM often eliminate most of this overhead at compile time. Rust version performs within 10–20% of the C version's execution time. This trade-off is often acceptable given Rust's increased safety and maintainability.

3. Challenges and Solutions in Replicating Behavior

- **Pointer Arithmetic**: Translating raw pointer manipulation (e++, sp--, etc.) was non-trivial. The Rust version often simulates this with Vec and index manipulation using indices to navigate arrays instead of raw pointers safely. In some cases, unsafe blocks

were used sparingly and carefully.

- **Symbol Table Representation**: The C version uses raw arrays and manual indexing (e.g., id[Val], id[Class]). Rust replaces this with struct types, hashmaps, or enum variants, mapping indices to named fields, which improves readability but required rewriting many sections.

- **Virtual Machine Execution**: The stack-based virtual machine logic from C (while(1) { switch(*pc++) }) was replicated with a loop and match statement on opcodes. Managing the program counter (pc) and stack (sp) with safe references rather than raw addresses posed a challenge but was resolved by careful tracking of index values or wrapping pointers in Box, etc..

- **Debugging and Tracing**: The C code's debug printing (printf) was ported to Rust's println!, but handling formatting and state tracing required rethinking how to expose or log internal VM state.