COSC320 – Principles of Programming Language

Project: Rewriting the C4 Compiler in Rust

Instructor: Davor

Group Name: AlKhaznah

Student Name and ID:

Sara Alkhadhouri – 100061319

Hamda Alkindi – 100061232

Due Date: May 02, 2025, 23:59

1. Rust Safety Features and Design Impact

Rust differs from C in several areas, including memory safety, type management, and ownership restrictions, which must be taken into consideration when translating c4.c to Rust. In C, pointers and manual memory allocation are used to control the entire compiler. This is replaced with safer options in the Rust version.

• Instead of malloc pools, fixed-size arenas like Vec and Vec are utilized; nonetheless, they are bounds-checked and type-safe.
• Compared to C's integer-based token encoding, token representation use Enum tokens with variations such as Token::Num(i64), which is significantly more reliable. Switch statements and macros are replaced by pattern matching and method-based behaviour.

• In the C version, symbol tables are a flat array of integers, which is unreliable and dangerous. To ensure safety and clarity, Rusts employs a HashMap, where each identifier is a typed struct containing class, type, name, and value.

• Restructuring shared state (current token, source buffer) into a compiler struct was necessary for the first ownership model. Careful management of mutable borrows and lifetimes was required for parsing and symbol lookup.

The Rust version retains close behavioural equivalence with the original despite these limitations. Concerns about lexing, parsing, and code generation were more clearly separated when Enums, structs, and trait-based approaches were used.

2. Performance Comparison

- The absence of trash collection and bounds checking in C results in speedier startup and parsing speeds.
- The Rust version has greater cache availability thanks to typed data structures, but it may have a slight overhead because of bound checks and more structured token processing.
- Rust's usage of Vec and typed Opcode Enums may introduce significantly slower dispatch compared to C's raw integers and pointers, but otherwise, the VM execution logic (such as opcode dispatch and stack handling) is almost comparable.

  However, the Rust code eliminates undefined behavior that could cause the C version to crash and is far more resilient to edge circumstances (unexpected input or memory access).

3. Replication Challenges and Solutions

- **Manual Memory Layouts**: It was not possible to precisely replicate the C version's usage of an integer-indexed global symbol table (with offsets like Name, Type, and Class). This was reorganized into a HashMap and a proper Identifier struct in Rust.
- **Use pointers to control the flow**: It was necessary to properly duplicate the VM's jump via function pointers and addresses (pc, sp, and bp) using typed vectors and secure indexing.
- In C, **Lexical Parsing**- tokenizing combines hash-based symbol interning with manual character checking. This needed to be redone in Rust utilizing next_char, peek_char, and Unicode-safe literal and identifier handling.
- **The Stack and Heap Simulation approach**: It was necessary to replace malloc for data, text, and stack pools with typed loads/stores in Rust that matched byte-level memory access (&[u8]) while maintaining alignment and size accuracy.

Feature	C Version	Rust Version
Memory safety	Unsafe (manual malloc)	Safe (typed Vec, Box)
Parsing	Pointer arithmetic	Structured Enums, and pattern matching
Error handling	exit() calls	Result<T, String> with proper messages
Symbol table	Raw array of int	HashMap<String, Identifier>
Code readability	Compact but brittle	More verbose but structured
Runtime safety	Low	High