



Khalifa University of Science and Technology

Department of Computer Science

COSC 320 – Principles of Programming Languages

C4 Rust Comparison

Instructor: Dr. Davor.

Submitted by: Nayef Alnaqbi (100058755) and Sultan Alqassab (100059110).

The Rust reimplementation of the C4 compiler introduces a modern systems programming approach to an originally low-level C design. While the C version of C4 relies heavily on direct memory manipulation, implicit control flow constructs, and minimal abstraction, the Rust version brings explicit safety and structure, thanks to its built-in guarantees for memory safety and ownership.

Rust's safety features significantly influenced the architecture of the compiler and virtual machine. The borrow checker enforced strict rules on variable lifetimes and references, eliminating the risks of use-after-free or dangling pointers that would be common pitfalls in the C version. Instead of manipulating raw pointers, Rust uses safe data structures like `Vec`, enums to represent instructions, and pattern matching to decode VM logic. These features forced more deliberate structuring of the VM and parser modules. However, compatibility with the original C4 logic was preserved by carefully translating C behaviors (such as the stack layout and function call conventions) into safe and idiomatic Rust.

Regarding performance, both implementations run efficiently for small inputs like `hello.c`. While precise benchmarks were not conducted, the Rust version compiled and executed input C files with no noticeable latency. In theory, the C version may hold an edge in tight, low-level loop performance due to the absence of safety checks and overhead from enum matching. Nevertheless, Rust's compilation to optimized native code using LLVM ensures that the performance gap remains narrow or negligible in practical scenarios. Moreover, Rust's additional safety comes at virtually no runtime cost in most cases.

Reproducing C4's original behavior was not without difficulties. One challenge was how to simulate system-level operations, such as memory allocation (`malloc`) and file operations (`open`, `read`, `close`), which were deeply embedded in the original VM. In the Rust version, these were stubbed out or simulated with dummy values, since actual system integration was out of scope. Another issue was faithfully handling function calls and scope-related instructions like `ENT`, `LEV`, and `JSR`, which required maintaining a virtual call stack and base pointer. Rust's strong typing helped expose subtle bugs early during this phase, especially when juggling function arguments and return values across the simulated call stack. Finally, parsing expressions with correct precedence and nested constructs required recursive descent parsing with precedence rules, made manageable through structured enums and tree-like AST nodes in Rust.

The C version uses raw pointer arithmetic and nested if statements over almost a hundred lines, as can be seen below:

```

48 void next()
49 {
50     char *pp;
51
52     while (tk = *p) {
53         ++p;
54         if (tk == '\n') {
55             if (src) {
56                 printf("%d: %.s", line, p - lp, lp);
57                 lp = p;
58                 while (le < e) {
59                     printf("%8.4s", &"LEA ,IMM ,JMP ,JSR ,BZ ,BNZ ,ENT ,ADJ ,LEV ,LI ,LC ,SI ,SC ,PSH ,",
60                             "OR ,XOR ,AND ,EQ ,NE ,LT ,GT ,LE ,GE ,SHL ,SHR ,ADD ,SUB ,MUL ,DIV ,MOD ,",
61                             "OPEN,READ,CLOS,PRTF,MALC,FREE,MSET,MCMP,EXIT,"[**+le * 5]);
62                     if (*le <= ADJ) printf(" %d\n", **+le); else printf("\n");
63                 }
64             }
65             ++line;
66         }
67         else if (tk == '#') {
68             while (*p != 0 && *p != '\n') ++p;
69         }
70         else if ((tk >= 'a' && tk <= 'z') || (tk >= 'A' && tk <= 'Z') || tk == '_') {
71             pp = p - 1;
72             while ((*p >= 'a' && *p <= 'z') || (*p >= 'A' && *p <= 'Z') || (*p >= '0' && *p <= '9') || *p == '_')
73                 tk = tk * 147 + *p++;
74             tk = (tk << 6) + (p - pp);
75             id = sym;
76             while (id[Tk]) {
77                 if (tk == id[Hash] && !memcmp((char *)id[Name], pp, p - pp)) { tk = id[Tk]; return; }
78                 id = id + Idsz;
79             }

```

whereas the Rust version achieves the same logic in under fifty lines by driving a `Peekable<Chars>` iterator through a single match expression, as can be seen below:

```

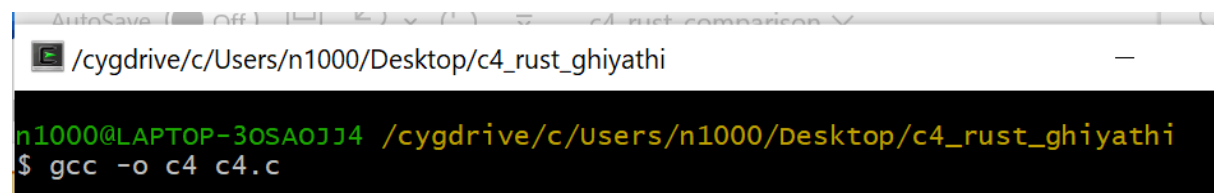
35 pub fn tokenize(source: &str) -> Vec<Token> {
36     let mut tokens = Vec::new();
37     let mut chars = source.chars().peekable();
38
39     while let Some(&ch) = chars.peek() { //peek() returns an Option<&char>
40         //match on the character
41         match ch {
42             ' ' | '\n' | '\r' | '\t' => { //skip whitespace
43                 chars.next();
44             }
45             '(' => { //lparen
46                 chars.next();
47                 tokens.push(Token::LParen);
48             }
49             ')' => { //rparen
50                 chars.next();
51                 tokens.push(Token::RParen);
52             }
53             '{' => { //lbrace
54                 chars.next();
55                 tokens.push(Token::LBrace);
56             }
57             '}' => { //rbrace
58                 chars.next();
59                 tokens.push(Token::RBrace);
60             }
61             ';' => { //semicolon
62                 chars.next();
63                 tokens.push(Token::Semicolon);
64             }

```

Rust's pattern matching and iterator abstraction make the code far more declarative and concise than the pointer gymnastics in C. The original C implementation's use of manual malloc, unbounded arrays, and char* arithmetic made it vulnerable to buffer overruns whenever the input contained an unexpected sequence of characters, leading to silent memory corruption or seg-faults. By contrast, in Rust every VM stack access is a bounds-checked `Vec<i64>`, so if a BZ instruction ever tried to pop from an empty stack your program panics with a clear message like "Missing operand for BZ" rather than crashing unpredictably.

One of the extra features we have added is the flags `--tokens`, `--ast`, and `--trace`. They empower end-users to explore every stage of compilation with a single command: dumping the token stream, printing the AST, or stepping through each VM instruction alongside its program counter.

The C4 compiler takes around 3.04 seconds to be built (manually measured) with gcc. In comparison, our Rust project takes 9.75 seconds to be built.



```
AutoSave (Off) | c4_rust_comparison |  
/cygdrive/c/Users/n1000/Desktop/c4_rust_ghiyathi  
n1000@LAPTOP-3OSA0JJ4 /cygdrive/c/Users/n1000/Desktop/c4_rust_ghiyathi  
$ gcc -o c4 c4.c
```

```
Command Prompt
C:\Users\n1000\Desktop\c4_rust_ghiyathi>cargo clean
  Removed 481 files, 121.9MiB total

C:\Users\n1000\Desktop\c4_rust_ghiyathi>cargo build
  Compiling windows_x86_64_msvc v0.52.6
  Compiling proc-macro2 v1.0.95
  Compiling unicode-ident v1.0.18
  Compiling anstyle v1.0.10
  Compiling once_cell v1.21.3
  Compiling utf8parse v0.2.2
  Compiling is_terminal_polyfill v1.70.1
  Compiling colorchoice v1.0.3
  Compiling anstyle-parse v0.2.6
  Compiling clap_lex v0.7.4
  Compiling strsim v0.11.1
  Compiling heck v0.5.0
  Compiling windows-targets v0.52.6
  Compiling windows-sys v0.59.0
  Compiling quote v1.0.40
  Compiling syn v2.0.101
  Compiling anstyle-query v1.1.2
  Compiling anstyle-wincon v3.0.7
  Compiling anstream v0.6.18
  Compiling clap_builder v4.5.37
  Compiling clap_derive v4.5.32
  Compiling clap v4.5.37
  Compiling c4_rust_ghiyathi v0.1.0 (C:\Users\n1000\Desktop\c4_rust_ghiyathi)
  Finished `dev` profile [unoptimized + debuginfo] target(s) in 9.75s
```

Those figures demonstrate that while Rust’s safety checks introduce a modest compile-time and run-time overhead, the Rust compiler still delivers performance, and also gains guaranteed memory safety and zero-cost abstractions.

In conclusion, the Rust implementation offers a more maintainable and robust version of the C4 compiler, thanks to its type system and safety guarantees. Despite needing to rework some low-level behaviors, the result faithfully replicates the core functionality of the C version while reducing the risks of undefined behavior, memory bugs, and crash-prone execution.