Understanding Key Algorithms in C4

C4 is a small but functional C compiler that processes code through lexical analysis, parsing, code generation, and execution within a virtual machine. Despite its small size, it implements these core compiler functions efficiently. This report explains how C4 handles tokenizing input, parsing and generating code, executing instructions, and managing memory.

Lexical Analysis

The next() function is responsible for tokenizing source code, converting raw text into recognizable tokens. It scans the input character by character, skipping spaces, comments, and newlines before identifying keywords, identifiers, numbers, and symbols. For example, the statement int x = 42; is broken down into tokens like INT, ID("x"), ASSIGN, NUM(42), and SEMICOLON. Unlike more complex compilers, C4 processes tokens on the fly rather than storing them in a structured list, keeping the compilation process lightweight.

Parsing and Code Generation

C4 does not explicitly build an Abstract Syntax Tree (AST). Instead, it parses expressions and statements recursively using functions like expr() and stmt(), directly generating machine-like instructions in the process. expr() evaluates arithmetic expressions while considering operator precedence, ensuring that multiplication occurs before addition. stmt() handles statements like if, while, and return, converting them into jump instructions for control flow. This approach eliminates the need for an intermediate representation, making C4's compilation process more direct.

Virtual Machine Execution

Once the code is parsed and converted into a set of low-level instructions, C4 executes it using a stack-based virtual machine. The VM processes instructions for loading values, performing arithmetic, and managing function calls. For example, return a + b; is compiled into instructions that push a and b onto the stack, execute an addition, and return the result. The VM maintains a structured execution model, where function calls push new stack frames, and results are passed back upon return.

Memory Management

C4 manually manages memory by dividing it into different regions: a stack for local variables and function calls, a heap for dynamically allocated data, and a symbol table for storing identifiers. The stack handles temporary storage during function execution, while the heap is used for persistent allocations via malloc(). Unlike modern compilers, C4 does not include garbage collection or automatic memory cleanup, meaning memory must be explicitly managed to avoid leaks.

Conclusion

C4 focuses on simplicity and efficiency, implementing only the essential features of a C compiler. Its lexer processes tokens in real-time, its parser translates statements directly into executable instructions, and its virtual machine executes compiled code using a stack-based model. Memory is managed manually through stack and heap allocations. These design choices make C4 self-hosting, meaning it can compile itself, a rare capability for such a small compiler. While it lacks support for advanced C features like structs and floating-point arithmetic, its minimalistic approach provides a clear view of how a compiler works.