Introduction

C4 is a minimalist, self-hosting C compiler that features a straightforward implementation of a tokenizer, parser, virtual machine (VM), and memory management. This report provides an overview of these key components, illustrating how C4 handles the lexical analysis, parsing, execution, and memory management of C code.

Lexical Analysis Process

The lexical analysis in C4 is managed by the next() function, which serves as the tokenizer. This function scans the source code character by character to classify tokens into several categories including identifiers, keywords, numbers, strings, and operators. It distinguishes tokens by looking for patterns that match different elements of the C language:

- **Identifiers and Keywords:** It identifies alphabetic characters that might form keywords (like if, while) or user-defined identifiers (variable/function names). The function generates a hash for each identifier to speed up the symbol table lookup.
- **Numeric Literals:** C4 recognizes decimal, hexadecimal, and octal numbers. It converts these textual representations into their numeric equivalents directly during tokenization, which simplifies the parsing stage.
- **Strings and Characters:** For string and character literals enclosed in quotes, C4 stores the sequence of characters and handles escape sequences like \n.
- **Operators and Punctuation:** The tokenizer recognizes C operators (+, -, *, /, etc.) and punctuation (;, {, }, etc.), crucial for parsing expressions and statements.

Each token identified by next() facilitates the syntactic parsing by providing a stream of meaningful elements rather than raw text, laying the groundwork for the construction of an internal representation of the code.

Parsing Process

C4's parser does not construct a traditional Abstract Syntax Tree (AST); instead, it generates bytecode directly as it parses the code through functions like expr() for expressions and stmt() for statements. The parsing process in C4 involves:

- **Expressions:** The expr() function uses a recursive approach to handle precedence and associativity of operators. This function handles everything from arithmetic calculations to function calls and assignments. It evaluates the expressions and directly emits bytecode that represents these operations.
- **Statements:** The stmt() function manages control flow constructs like if-conditions, loops, and function declarations. It reads the tokens generated by next() and decides the course of action for each statement type, emitting bytecode for jumps, loops, and branches as needed.

This approach of directly emitting bytecode instead of building an AST simplifies the compiler but limits the potential for complex optimizations that could be achieved with a more traditional multi-pass compiler.

Virtual Machine Implementation

The virtual machine (VM) in C4 is a simple stack-based executor that runs the bytecode generated by the parser. It features:

- **Registers and Stack Management:** The VM uses a combination of program counters, stack pointers, and base pointers to manage execution flow and local/global variable access.
- **Instruction Set:** The VM implements a basic set of instructions like IMM (load immediate value), JMP (jump), JSR (jump to subroutine), and arithmetic operations (ADD, SUB, etc.). Each instruction is designed to manipulate the stack or control flow directly.
- **System Calls Integration:** For operations like I/O, C4 integrates system calls within its VM, allowing the execution of external functions like printf and malloc.

The VM reads and executes instructions one at a time, adjusting the stack and program counters as it proceeds. This execution model, while not optimized for performance, is sufficient for demonstrating the principles of computation and compiler design.

Memory Management Approach

C4 handles memory statically, allocating fixed-size blocks for the symbol table, code, data, and stack at the start of the execution. The allocations are performed once, and the sizes are predetermined:

- **Static Allocation:** All main memory segments (symbol table, code, data, stack) are allocated using malloc at the initialization phase of the compiler. Each segment receives a large block of memory that should suffice for the compilation and execution of small to medium-sized programs.
- **No Dynamic Reallocation:** C4 does not support resizing these memory segments during runtime. If the memory allocated initially is exhausted, the compiler or the running program will fail.
- **Manual Memory Management:** For dynamic memory needs within the compiled programs (like those requested via malloc), C4 provides built-in system call support that interfaces with the underlying operating system's memory management.

Conclusion

C4's design as a self-hosting compiler demonstrates a balance between simplicity and functionality. The straightforward implementations of lexical analysis, parsing, execution via a virtual machine, and static memory management provide a foundation for understanding how compilers work. Although lacking the features and optimizations of more sophisticated compilers, C4 serves as an excellent educational tool for those interested in compiler construction and low-level programming concepts.