C4 Compiler Analysis Report

Introduction

The C4 compiler is a small C-based compiler that translates source code into instructions, which are then executed by its custom virtual machine (VM). This report explains the key concepts and algorithms used in lexical analysis, parsing, VM execution, and memory management.

Lexical Analysis (Tokenization)

The lexical analyzer (tokenizer) in C4 converts characters from the source code into tokens—small units representing keywords, operators, and identifiers.

The next() function reads characters from the source code (p), skipping whitespace, newlines, and comments. It detects keywords and assigns token types, such as numbers (0-9), operators (+ - * / ** || == !=), and identifiers (a-z, A-Z, _).

Parsing

Unlike traditional C compilers that build an Abstract Syntax Tree (AST), C4 parses and generates code at the same time.

The expr() function handles expressions like a + b * 2, following operator precedence. The stmt() function parses statements like if, while, return, and block statements {...} using recursive descent parsing.

VM Execution

The C4 compiler has a custom virtual machine that executes the instructions generated by the parser.

The VM operates in a fetch-decode-execute loop. It manages execution using registers (pc, sp, bp, a), where:

- **pc (program counter) tracks the instruction being executed.**
- **sp (stack pointer) and bp (base pointer) handle function calls and local variables.**
- **a (accumulator) holds intermediate values.**

The stack is used for function calls and local variables, while the data section stores global variables.

Memory Management

C4 handles memory using both stack and heap allocation.

- **Stack memory (sp, bp): Used for local variables and function calls. The stack expands and shrinks with ENT (enter function) and LEV (leave function).**

- **Heap memory (malloc, free): Dynamically allocated memory using MALC (malloc) and freed using FREE. The data section is used for global variables.**

Conclusion

The C4 compiler is a compact, C-like compiler that integrates lexical analysis, parsing, code generation, and execution. Instead of producing assembly or machine code, it generates bytecode, which runs on its custom virtual machine. Memory is managed through stack and heap allocation, making C4 simple but effective.

optimization passes and generates bytecode rather than native machine code, making it slower than mainstream compilers. Despite these trade-offs, C4's self-hosting capability highlights its effectiveness as a learning tool for understanding compiler construction.

Conclusion

C4 provides an insightful look into the inner workings of a compiler, showcasing fundamental concepts such as tokenization, symbol resolution, and self-hosting. Through its next() function, it efficiently processes source code into structured tokens, enabling smooth parsing and code generation. Its symbol resolution mechanism, while simplistic, effectively handles variables and functions using a straightforward symbol table. Although C4 lacks many features of a modern C compiler, its minimalism serves as an advantage in educational contexts, allowing students and developers to grasp compiler principles without unnecessary complexity. Moreover, its self-hosting capability underscores the power of bootstrapped compilation, demonstrating how a compiler can be designed to compile itself. While it may not rival mainstream compilers in functionality or performance, C4 remains an invaluable tool for exploring the core principles of compilation.

Done by: Khalifa Aljasmi

ID:100061653