جامعة خليفة
Khalifa University

Homework 1 Report:
Deep Dive into C4 - Understanding a Self-Interpreting C Compiler

Course:	COSC320 Principles of Programming Languages
Semester:	Spring 2025
Submission Date:	03/03/2025
Student Name:	Ahmed Alkhoori
Student ID:	100061091
Instructor:	Davor Svetinovic
Teaching Assistant:	Rui Hu

Lexical Analysis Process

The next () function in c4 performs the lexical analysis, which sequentially reads the source code character by character and converts it to tokens. It finds keywords, identifiers, numbers, and operators and ignores whitespace and comments. It hashes the identifiers and stores the hashed identifiers in a symbol table (sym). It can distinguish numbers (integer and hexadecimal), character literals, and strings. Character comparisons identify operators, including multi-character operators such as ==, !=, <=, and >=. The data section stores string literals and numeric values, and each token is given a unique integer value to facilitate efficient parsing. The lexical analysis performed is minimal but effective, allowing the c4 compiler to process C-like syntax with a small memory footprint.

Parsing Process

c4 follows the recursive descent parsing approach without constructing an abstract syntax tree. It directly translates expressions and statements to machine instructions, with the latter using an implicit stack-based approach. The expr(int lev) function uses an operator precedence parsing method, “precedence climbing,” for different levels. stmt() function is responsible for processing statements and recognizing and translating constructs such as if, while return, and block structures like {} into instructions. The simplified symbol table processes variable declarations and functions where identifiers are mapped to memory locations or function entry points. Rather than an explicit AST, c4 produces executable bytecode immediately, keeping an easy-to-parse representation.

Virtual Machine Implementation

A minimalistic stack-based virtual machine (VM) executes compiled code in c4. It processes bytecode instructions sequentially on the program counter (pc), stack pointer (sp), and base pointer (bp). The main() function initializes the VM state, and the instruction fetch-execute loop continues. The operand stack is used to execute instructions like IMM (load immediate), JMP (jump), LEA (load effective address), and arithmetic operations (ADD, SUB, MUL). The JSR (jump to subroutine) and LEV (leave subroutine) are used to handle function calls and maintain local scope through the stack. This lightweight VM ensures that c4 can run the compiled programs from c4 itself, thus enabling self-hosting.

Memory Management Approach

c4 memory has three distinct sections: code (e), data (data), and stack (sp). Compiled bytecode is stored in the code section, global variables and string literals in the data section, and the stack is used for function calls and local variables. System calls like malloc() and free() provide dynamic memory allocation, which are mapped to MALC and FREE instructions. Unlike modern garbage-collected runtimes, c4 requires explicit memory management by the programmer. It uses simple pointer arithmetic for variable and stack management, enabling efficient memory use in a relatively small environment. The implementation is kept lean with this manual memory management while still supporting essential features.