C4 Compiler Concepts Report

Student Name: Reham Alsereidi **Student ID: 100062722**

This report examines the foundational algorithms and concepts used in C4; a compact C compiler written in just four functions. The simplicity of C4 makes it an excellent case study for understanding compiler construction principles.

Lexical Analysis Process

The lexical analysis in C4 is handled by the next() function, which reads the source code character by character to identify tokens. The process works as follows:

Character-by-Character Scanning:

The function reads one character at a time from the input source code, using the pointer p.

Token Classification:

The code recognizes several types of tokens:

- Keywords (e.g., if, while, return)
- Identifiers (variable/function names)
- Numbers (integer literals)
- Strings and character literals
- Operators and punctuation

Token Handling:

- For identifiers, it calculates a hash value and searches the symbol table
- For numbers, it handles decimal, hexadecimal, and octal formats
- For strings, it copies the content into the data segment
- For operators, it handles multi-character operators (e.g., ==, ++, <=)

Storage:

When a token is identified, its type is stored in the global variable tk, and for certain tokens, additional information is stored in ival (for numeric values) or through the symbol table.

The lexical analyzer skips whitespace and comments, and updates the line counter when encountering newlines, which is useful for error reporting.

The Parsing Process

C4 uses a recursive descent parser with operator precedence climbing, implemented primarily in the expr() and stmt() functions. Unlike many compilers that build an explicit AST, C4 directly generates instructions while parsing:

Expression Parsing:

The expr() function handles expression parsing using two techniques:

- Initial recursive descent for primary expressions (identifiers, literals, parentheses)
- "Precedence climbing" (Top-Down Operator Precedence) for binary operators

Statement Parsing:
The stmt() function handles control structures:

- If-else statements
- While loops
- Return statements
- Compound statements (blocks)
- Expression statements

Direct Code Generation:
Instead of building an explicit AST, C4 emits code directly during parsing. Instructions are added to the e array, which serves as the output code buffer.

Symbol Table Management:
During parsing, the symbol table (the sym array) is populated with information about identifiers, including:

- Token type
- Hash value
- Name (pointer to the string)
- Class (Global, Local, Function, etc.)
- Type (INT, CHAR, PTR)
- Value (address or immediate value)

This direct code generation approach eliminates the need for a separate AST data structure and traversal, making the compiler more compact.

Virtual Machine Implementation

C4 includes a simple stack-based virtual machine to execute the compiled code. The VM operates as follows:

Instruction Set:
The VM uses a custom instruction set with operations like:

- Memory operations (LEA, IMM, LI, LC, SI, SC)
- Control flow (JMP, JSR, BZ, BNZ, ENT, ADJ, LEV)
- Arithmetic and logical operations (ADD, SUB, MUL, DIV, MOD, OR, XOR, AND)
- Comparison operations (EQ, NE, LT, GT, LE, GE)
- System calls (OPEN, READ, CLOS, PRTF, MALC, FREE, etc.)

Register Usage:

- pc: Program counter, points to the next instruction to execute
- sp: Stack pointer, points to the top of the stack
- bp: Base pointer, used for function call frame management
- a: Accumulator, stores intermediate results

Execution Loop:

- The VM runs a simple fetch-decode-execute cycle:

- Fetch the next instruction pointed to by pc
- Increment pc
- Execute the instruction, which may modify a, sp, bp, or pc

Function Calls:
Function calls are handled by:
- Saving the return address on the stack
- Saving the previous base pointer
- Allocating local variables
- Executing the function body
- Restoring the previous state on function exit

This simple VM design allows C4 to be self-hosting (able to compile itself) while remaining compact

Memory Management Approach

C4 uses a straightforward approach to memory management:

Fixed Memory Pools:
The compiler allocates four fixed-size memory pools at startup:
- Symbol table (sym): Stores information about identifiers
- Code segment (e): Stores the generated instructions
- Data segment (data): Stores global variables and string literals
- Stack (sp): Used by the VM during execution

Memory Allocation:
Memory allocation is handled in several ways:
- Fixed allocation of the pools (256KB each)
- Global variables are allocated sequentially in the data segment
- Local variables are allocated on the stack with fixed offsets
- Dynamic memory allocation is supported through system calls (MALC, FREE)

No Garbage Collection:
C4 does not implement garbage collection, following C's manual memory management model.

Stack Management:
The VM manages the stack for:
- Function call frames
- Local variables
- Temporary values during expression evaluation
- Function arguments

This simple memory management approach contributes to C4's small size while still supporting core C language features.

Conclusion

C4 demonstrates that a functional compiler and virtual machine can be implemented in a remarkably small amount of code. By combining lexical analysis, parsing, code generation, and execution into just four functions, C4 provides an accessible model for understanding how compilers work. The design choices made in C4—direct code generation instead of building an AST, a stack-based VM, and simple memory management—all contribute to its compact size while maintaining the ability to compile a significant subset of the C language.