Saif Adel Alafeefi
100061144
COSC320
Section 01

Key Algorithms and Concepts in C4

1. Lexical Analysis Implementation

The lexical analyzer (next() function) scans source code character-by-character to identify tokens. It uses pointer arithmetic for traversal and enum values (starting at 128 to avoid ASCII conflicts) for token classification. The analyzer recognizes:

- Keywords (if, while, return, etc.) - these get special enum values
- Identifiers (variable/function names) - anything that starts with a letter or underscore
- Constants (decimal, hex with 0x, and octal with leading 0) - all parsed differently
- String literals - with basic escape sequence handling
- Operators and delimiters - including multi-char ones like == and <=

The symbol table is implemented as a flat array with O(n) lookup - not efficient for large programs but works fine here. Each entry tracks token type, constant values, and symbol references.

2. Parsing Design

C4 uses recursive descent with operator precedence climbing instead of complex LR parsing. The expr() function handles expressions by checking token precedence levels - a clever way to implement PEMDAS without separate grammar rules for each level.

Statement processing in stmt() handles:

- if/else with conditional jumps
- while loops with backward jumps
- Code blocks with scope management
- Return statements

The parser uses backpatching for forward references - it emits jumps with placeholder addresses and fills them in later when it knows where to jump to. We did something similar in our assembly lab in CSO course but this is cleaner.

3. Virtual Machine Architecture

The VM is register-based with 4 main registers (pc, sp, bp, a) and a minimal but complete instruction set:

- Memory ops: LEA, IMM, LI/LC, SI/SC
- Control flow: JMP, JSR, BZ/BNZ
- Stack ops: PSH, ENT, LEV
- Arithmetic/Bitwise/Comparison operations
- System calls for I/O and memory management

Function calls use a standard stack frame approach with arguments pushed in reverse order, ENT creating the frame, and LEV handling return. The execution loop is a straightforward fetch-decode-execute cycle.

4. Memory Management Approach

The compiler allocates four 256KB pools at startup:

- sym: symbol table
- e: code segment
- data: global variables
- sp: runtime stack

Variable allocation follows standard C patterns with locals on the stack (bp-relative addressing) and globals in the data segment. There's no register allocation or optimization - everything uses memory access. Dynamic memory is supported through MALC/FREE instructions that wrap C library functions.

Conclusion

C4 demonstrates how to build a working compiler without unnecessary complexity. The direct code generation approach skips the AST stage, which is an interesting optimization. While not as powerful as gcc, it implements all core compiler concepts in a compact package. The most impressive feature is its self-hosting capability - it can compile itself. The precedence climbing method for expression parsing is more elegant than traditional grammar-based approaches.