Key Algorithms and Concepts in the C4 Compiler

The C4 compiler, a minimal self-hosting C compiler written by Robert Swierczek, integrates lexical analysis, parsing, code generation, and execution into four core functions: next(), expr(), stmt(), and main(). Its simplicity and efficiency stem from carefully designed algorithms and concepts tailored to a compact implementation. This report explores four key aspects: lexical analysis, parsing, virtual machine (VM) implementation, and memory management.

1. Lexical Analysis Process: Token Identification and Tokenization

The lexical analysis in C4 is handled by the next() function, which serves as the lexer. It reads the source code character by character from a global pointer char *p, identifying tokens and updating global variables tk (token type) and ival (token value). The process begins by checking the current character *p. If it's a newline (\n), the line number (line) increments, and optional source printing occurs. For identifiers (letters or _), next() computes a hash using a polynomial formula (tk = tk * 147 + *p) and searches the symbol table (*sym) for matches, adding new entries if needed. Numbers are parsed as decimal, hexadecimal (e.g., 0x1A), or octal (e.g., 077), with ival storing the result and tk set to Num. Operators and punctuation (e.g., +, ==, {) are identified by pattern matching, often checking the next character for multi-character tokens (e.g., +=). Strings and characters (e.g., "hello", 'a') are processed by copying characters to the data segment (*data) and setting ival accordingly.

This single-pass, hand-written lexer avoids the complexity of a separate tokenization phase or regular expression engine, aligning with C4's minimal design. The hash-based identifier lookup ensures quick symbol resolution, while direct token generation feeds seamlessly into the parser.

2. Parsing Process: Constructing a Representation

C4's parsing is split between expr() for expressions and stmt() for statements, using a hybrid approach that doesn't explicitly build an Abstract Syntax Tree (AST). Instead, it generates VM instructions directly into an array (int *e) as it parses, effectively creating an implicit representation. expr() employs a precedence climbing algorithm, a variant of recursive descent, to handle operator precedence. It takes a precedence level (lev) and parses expressions bottom-up, emitting instructions for operands (e.g., IMM for numbers) and operators (e.g., ADD, MUL) as it encounters them. For example, parsing a + b * c generates instructions to load a, push it, load b, multiply by c, then add the result. stmt() uses straightforward recursive descent to handle control structures like if, while, and return, emitting jump instructions (e.g., BZ, JMP) to manage flow. The symbol table (*sym) tracks variable types and scopes, guiding code generation without a separate AST.

By skipping an explicit AST, C4 reduces memory usage and complexity, directly translating syntax into executable instructions. The precedence climbing in expr() efficiently handles operator hierarchies, while stmt()'s simplicity suits the limited statement types supported, making it ideal for a minimal compiler.

3. Virtual Machine Implementation: Executing Instructions

The VM, embedded in main(), executes the instructions stored in *e using a stack-based architecture. It maintains a program counter (*pc), stack pointer (*sp), base pointer (*bp), and accumulator (a). Execution loops indefinitely, fetching each opcode from *pc and incrementing it. Opcodes are categorized into:

- **Control**: JMP jumps to an address, JSR calls subroutines (pushing return addresses), BZ/BNZ branch based on a.
- **Memory**: LEA computes local addresses, LI/LC load integers/chars, SI/SC store them.
- **Arithmetic/Logic**: ADD, MUL, AND, etc., pop operands from the stack, compute results, and store them in a.
- **System Calls**: PRTF (printf), MALC (malloc), etc., interface with the host system. The stack (*sp) grows downward, supporting function calls (ENT saves bp, LEV restores it) and temporaries (PSH pushes values). For example, executing IMM 5; PSH; IMM 3; ADD loads 5, pushes it, loads 3, and adds them, leaving 8 in a.

The stack-based VM simplifies code generation and execution, eliminating the need for register allocation. Its direct interpretation of opcodes aligns with C4's goal of immediate execution post-compilation, while supporting a minimal yet functional instruction set.

4. Memory Management Approach: Allocation and Deallocation

C4 manages memory using a combination of static allocation and runtime system calls. At startup, main() allocates fixed-size pools (256 KB each) for the symbol table (*sym), code (*e), data segment (*data), and stack (*sp) via malloc(). These pools are initialized with memset() and never resized, relying on the assumption that the input fits within these bounds. The data segment (*data) grows as strings and globals are allocated, with next() advancing *data for strings and main() for globals. Local variables use stack offsets (loc) relative to *bp, managed by ENT/LEV opcodes. Runtime allocation (e.g., malloc()) is exposed via the MALC opcode, with FREE for deallocation, passing control to the host system's C library. No garbage collection or automatic deallocation occurs beyond free() calls.

Fixed pools minimize overhead and ensure predictable memory usage, critical for a minimal compiler. Delegating dynamic allocation to the host system avoids implementing a custom allocator, keeping C4 lightweight. The stack-based locals integrate seamlessly with the VM, balancing simplicity and functionality.

Conclusion

C4's algorithms reflect a deliberate trade-off between functionality and simplicity. The lexer's single-pass tokenization, parser's direct code emission, stack-based VM, and static memory pools enable a compact, self-hosting compiler. While limited in scope (e.g., no structs, floats), these choices make C4 an elegant demonstration of compiler essentials, blending parsing and execution into a cohesive system.