Key Algorithms in C4: A Detailed Analysis

1. **Lexical Analysis Process**

   The lexical analysis phase in the C4 interpreter is responsible for breaking the input source code into manageable tokens. This process is implemented in the next() function, which scans the input character by character, recognizing and categorizing tokens based on predefined rules.

   The interpreter employs a hashing technique, where each character's ASCII value is combined to generate a unique numerical hash. This hash is then checked against known keywords stored in the symbol table. If a match is found, the corresponding token type is assigned; otherwise, it is treated as an identifier.

   When a sequence of numeric characters is encountered, the function reads the digits and converts them into an integer value using standard base conversion techniques, supporting decimal, hexadecimal, and octal parsing. Special characters such as +, -, =, and * are recognized as operators, with multi-character operators like ==, !=, and <= also handled.

   The lexer ignores whitespace and newline characters to maintain efficient parsing. It also recognizes comments (// for single-line comments) and skips them. String and character literals are processed separately, ensuring escape sequences such as \n for newline are correctly interpreted. This lexical analysis phase is crucial for simplifying the parsing process, as it transforms raw source code into structured tokens.

2. **Parsing Process**

   Parsing in C4 involves analysing tokens produced by the lexer and generating an intermediate representation that the virtual machine can execute. Unlike traditional compilers that construct an Abstract Syntax Tree (AST), C4 directly generates intermediate code using operator-precedence parsing.

   The expr() function handles mathematical and logical expressions while enforcing operator precedence. The interpreter applies recursive descent parsing, allowing nested expressions to be processed correctly. The stmt() function is responsible for processing control structures such as if-statements, while-loops, and function calls. Statements are processed sequentially, and branching logic is implemented using jumps.

   When a function is encountered, it is added to the symbol table. Function calls result in jump-to-subroutine (JSR) instructions, enabling dynamic execution. The parser follows a precedence climbing method, ensuring operations like multiplication (*) and division (/) are executed before addition (+) and subtraction (-).

   If syntax inconsistencies are detected, such as missing parentheses or unexpected tokens, error messages are generated, and execution is halted. The direct code generation approach reduces memory overhead but makes debugging complex since no explicit AST is stored.

3. **Virtual Machine Implementation**

   C4 incorporates a simple yet efficient stack-based virtual machine that executes intermediate code. This execution engine is implemented within the main() function, which interprets instructions sequentially.

   The VM supports various operations, including arithmetic operations (ADD, SUB, MUL, DIV), control flow (JMP, BZ, BNZ), function calls (JSR, LEV), memory operations (LI, LC,

SI, SC), and system calls (PRTF, MALC, FREE, EXIT). Instructions are processed in a loop, where each opcode corresponds to a specific operation on the stack or memory.

Function calls, variable storage, and temporary values are managed using a stack, allowing efficient execution without requiring complex register management. When a function is called, arguments are pushed onto the stack, the function executes, and upon completion, the previous state is restored.

If debugging is enabled, the VM prints each executed instruction, helping to trace program execution and understand its behaviour during runtime.

4. **Memory Management Approach**

   C4 uses a simple yet effective memory management strategy that minimizes dynamic allocations and relies on pre-allocated memory segments.

   Upon initialization, the program allocates a fixed amount of memory for different segments, including the symbol table (which stores variable and function identifiers), the code segment (which holds compiled bytecode), the data segment (which stores global variables and string literals), and the stack segment (which is used for function calls and temporary storage).

   The VM supports heap allocation via MALC (malloc) and FREE (free), which are directly mapped to system calls. Memory access is managed by frequently adjusting pointers to efficiently access variables, mimicking stack-frame allocation found in traditional compilers.

   The implementation does not include garbage collection, making it the programmer's responsibility to manage memory correctly. However, upon program termination, the interpreter ensures all allocated memory is freed, preventing leaks and ensuring efficient resource utilization.

C4 is an elegantly designed interpreter that integrates lexical analysis, parsing, code generation, and execution within a compact system. By directly generating intermediate code and executing it via a stack-based virtual machine, it simplifies the compilation pipeline while maintaining efficient execution. Although its memory management is basic, its approach provides a clear understanding of low-level memory operations. These features make C4 an excellent tool for learning the fundamental principles of compilers and interpreters.