1. Lexical Analysis Process:

The lexical analyzer (lexer) in the C4 compiler is responsible for converting the source code into tokens. It uses the ‘next()’ function to read characters and identify keywords, identifiers, numbers, and operators. For example, when it encounters a number, it converts it into a ‘Num’ token and stores its value in ‘ival’. The lexer is designed to be simple and efficient, avoiding complex regular expressions or external tools.

2. Parsing Process:

The parser in the C4 compiler converts tokens into an intermediate representation (IR) in the form of virtual machine instructions. The ‘expr()’ function handles expression parsing using the precedence climbing method. It generates instructions like ‘IMM’, ‘PSH’, and ‘ADD’ and stores them in the ‘e’ array. This approach avoids the need for a full abstract syntax tree (AST) and directly emits instructions, making the compiler fast and efficient.

3. Virtual Machine Implementation:

The virtual machine (VM) in the C4 compiler executes the compiled instructions. It uses a stack-based architecture with registers like ‘pc’ (program counter), ‘sp’ (stack pointer), and ‘bp’ (base pointer). For example, the ‘ADD’ instruction pops two values from the stack, adds them, and pushes the result back. The stack-based design is simple to implement and easy to debug, closely mirroring the structure of the emitted code.

4. Memory Management Approach:

The C4 compiler manages memory for global variables, local variables, and dynamic allocations. Global variables are stored in the ‘data’ section, while local variables are managed on the stack using ‘sp’ and ‘bp’. Dynamic memory is allocated using `malloc` and freed using ‘free’. The stack is used for local variables because it’s fast and automatically cleaned up when functions return. Heap allocation is used sparingly to avoid fragmentation and leaks.