The lexical analysis process: How does the code identify and tokenize input?
The lexical analysis process in C4 is the first phase of compilation, responsible for breaking down the source code into tokens. The next() function handles this process by reading characters from the source code and categorizing them into tokens based on the type and context.

- **Character Processing:** This function reads characters from the source code one by one. It skips over whitespace, newlines, and comments, so only relevant code is processed.
- **Identifiers and Keywords:** The function reads alphabetic characters or underscores to form an identifier. It then checks if the identifier matches any keywords or functions. If a match is found, the corresponding token is returned if not then it's treated as a user-defined identifier.
- **Numbers:** The function handles decimal, hexadecimal, and octal numbers. It reads the digits and converts them into their integer value, storing the result in ival.
- **Strings and Characters:** String literals and character literals are processed. And escape sequences like \n are handled, and the string data is stored in the data section.
- **Operators and Punctuation:** The function recognizes operators like +, -, *, /, and punctuation like ;, {, }. It also handles operators like ==, !=, ++, --.

The parsing process: How does the code construct an abstract syntax tree (AST) or equivalent representation?
In C4, the parsing process is creating an intermediate representation of the source code, which is then used for code generation. The functions expr() and stmt() handle parsing expressions and statements.

- **Expression Parsing:** The expr() function processes expressions, which consist of variables, numbers, and operators. For instance, in the expression x + 5 * y, the parser determines the correct order of operations so multiplication before addition and constructs a structure to represent the calculation. It manages function calls, variable references, and mathematical operations, as well as more intricate expressions such as x++ (increment) or *ptr (dereferencing a pointer).
- **Statement Parsing:** The stmt() function processes instructions such as if conditions, while loops, and return statements. When encountering an if statement, it parses the condition within the parentheses and the corresponding code block to execute if the condition is true. Additionally, it manages while loops, function returns, and code blocks in {}.

The virtual machine implementation: How does the code execute the compiled instructions?
The virtual machine (VM) in C4 functions as a little computer that executes the compiled program by processing byte code instructions generated by the parser inside the main() function.

- **Instruction Execution:** The VM operates on a set of instructions such as IMM (load a value), ADD (add two numbers), or JMP (jump to a different part of the code), each represented by a num value. For instance, IMM 10 loads the value 10 into a register,

while ADD retrieves two numbers from the stack, adds them, and pushes the result back.

- **Stack-Based Operations:** Data management in the VM is handled using stack mechanism. When performing operations like addition, the numbers are pushed onto the stack, processed by the ADD instruction, and then the result will be pushed back in the stack.
- **Control Flow:** The VM manages control flow through jumps and conditional branches. In scenarios like an if statement, the VM evaluates the condition and directs the program flow based on the condition's outcome.

The memory management approach: How does the code handle memory allocation and deallocation?

C4 has a simple memory management approach without a garbage collector or complex memory allocation. It uses memory pools for different purposes such as the symbol table, code area, data area, and stack. Memory allocation is done manually using malloc() and deallocation is done using free().

Global variables are allocated space in the data area upon declaration. C4 does not have automatic garbage collection, so memory leaks can occur if unused memory is not free. Despite this, for small programs, memory management in C4 is typically not a problem because they usually have a minimal design.

Conclusion

The code shows the implementation of a compiler and virtual machine. Its lexical analyzer tokenizes source code, the parser generates byte code, and the virtual machine executes byte code using stacks. The manual memory management system is simple but may not be very suitable for larger programs instead its better for small programs. So in summary, it shows the fundamental concepts of compilation and execution.