Lexical Analysis: How Does the Code Identify and Tokenize Input?

The next() function is responsible for lexical analysis, playing a key role in breaking down the source code into tokens. It reads the code character by character using a pointer p, while lp marks the start of the current line. The function skips over spaces and newlines, and whenever it encounters a newline (\n), it increases the line counter. If debugging is enabled, it prints the current line and any intermediate instructions, helping track how the source code is transformed into low-level operations. The function also ignores comments and preprocessor directives, such as lines that start with #. Identifiers (like variable names) are recognized when they start with a letter or an underscore, and the lexer reads characters until it finds a non-alphanumeric one. A hash is calculated for each identifier, which helps locate it in the symbol table; if it exists, it is reused, otherwise, a new entry is made. Numeric values are converted into integers, and string literals (inside quotes) are stored in a special memory section, handling escape characters like \n. Operators and punctuation are identified using conditions, ensuring that = and == are correctly distinguished. Each token is assigned a specific identifier, like Num for numbers or Id for identifiers, which simplifies the next phase of compilation. This structured approach allows for efficient lexical analysis, ensuring that the source code is broken down in a way that makes parsing and execution smoother.

Parsing: How Does the Code Construct an Abstract Syntax Tree (AST) or Equivalent Representation?

Instead of creating a full AST, the parsing process in this compiler uses a recursive descent approach to directly convert source code into intermediate instructions stored in an array e. The expr() function plays a central role in handling expressions. If it detects a number (Num), it generates an IMM (Immediate) instruction with the value. String literals are handled similarly, but their memory address is stored instead. When an identifier (Id) appears, the parser checks if it's a function call by looking for parentheses. If so, it processes arguments and generates the appropriate call instructions (JSR for user-defined functions and system call instructions for built-in functions). Operator precedence is handled using a loop (while (tk >= lev)) to make sure expressions like addition and multiplication are evaluated correctly. Statements like if and while are parsed using the stmt() function. For if statements, the parser evaluates the condition and generates a BZ (Branch Zero) instruction to control execution flow. Loops (while) are handled by keeping track of instruction locations and using jump (JMP) instructions to repeat execution. Functions are defined using ENT (to set up the stack frame) and LEV (to return from the function). By integrating parsing and code generation in one step, this approach allows for a streamlined compilation process, avoiding the overhead of building a separate AST while still maintaining structure and clarity in the compiled code.

Virtual Machine: How Does the Code Execute Compiled Instructions?

The virtual machine in the code executes the compiled instructions through a well-structured fetch-decode-execute loop that mimics a basic CPU's operation. Initially, the compilation phase generates intermediate code stored in a dedicated memory area, after which the main function configures the VM environment by initializing essential registers including the program counter, stack pointer, and base pointer. The VM then enters an infinite loop in which it fetches the next instruction using the program counter, increments a cycle counter, and decodes the instruction through a series of conditional statements before executing it. For instance, when an immediate value opcode (IMM) is encountered, the value is loaded into an accumulator variable and the program counter is advanced. Control flow instructions, such as JMP and JSR, alter the execution sequence by updating the program counter, with JSR also saving the return address onto the stack to facilitate subroutine calls. Additionally, opcodes are mapped to a concatenated mnemonic string of 4-character identifiers, and in debug mode, these mnemonics and their operands are printed for clarity. Arithmetic and logical operations are handled in a stack-based manner using operations like ADD, SUB, MUL, DIV, and MOD. Furthermore, specialized opcodes manage stack frames (ENT and LEV), ensuring proper handling of function calls and local variables. Overall, it effectively bridges abstraction levels.

Memory Management: How Does the Code Handle Memory Allocation and Deallocation?

The memory management strategy in the code effectively combines static memory pools with dynamic allocation to handle various runtime needs. At startup, the program allocates large, fixed-size memory blocks using malloc for different areas: the symbol table (sym) holds identifiers and metadata, the code/text area (e) stores the compiled instructions, the data area (data) reserves space for global variables and string literals, and the stack area (sp) supports function call management and local variable storage. These memory pools are all initialized to zero using memset, ensuring a clean state before the compilation begins. The design leverages static allocation to reduce overhead and fragmentation by dedicating ample space for compilation artifacts, while dynamic memory allocation is reserved for runtime needs. For instance, when the VM encounters a MALC opcode, it calls the standard library malloc to allocate memory on the fly, and later frees it with the FREE opcode, thereby preventing memory leaks. Furthermore, operations like memset and memcmp are used to manage memory content during execution, supporting operations such as initializing buffers and comparing data blocks. The stack is managed by explicitly adjusting pointers for each function call. The ENT opcode saves the current base pointer and allocates space for new local variables, while the LEV opcode restores the previous execution context when a function returns. This dual approach of combining pre-allocated pools for predictable, high-volume data with dynamic allocation for flexible, runtime requirements results in a robust and efficient memory management system, ensuring that both static and dynamic data are handled optimally throughout the lifecycle of the program.