C4_Report

The lexical analysis process: How does the code identify and tokenize input?

The next() method handles the lexical analysis process by tokenizing and identifying the input source code. The code uses a pointer p to iterate through the source character by character in order to identify tokens. Tokens are then divided into operators, strings, characters, integers, keywords, and identifiers. It searches for patterns, identifiers and keywords. And then stores them in the symbol table using a hash function. Strings and characters are wrapped in quotes and handle escape sequences like \n, whereas numbers are parsed as decimal, octal, or hexadecimal literals. Operators are given precedence over single character ones, even multi-character ones like ++ and!=. Comments that begin with // or # are overlooked. Identifiers are managed using the symbol table (sym). This method guarantees that the source code is broken down into meaningful tokens for additional parsing.

The parsing process: How does the code construct an abstract syntax tree (AST) or equivalent representation?

The code builds instructions for the virtual machine (VM) directly without creating a separate tree structure (AST). The expr() function parses expressions, handling things like math operations (+, *) and generating VM commands like (IMM for numbers). The stmt() function handles statements like if, while, and return, creating instructions for jumps and loops. It handles nested code through using recursive calls to stmt() and expr(), and syntax correctness is enforced with recursion that looks for mistakes like missing parenthesis. This approach skips the step of building a tree and goes straight to generating code for the VM, making it simple and efficient.

The virtual machine implementation: How does the code execute the compiled instructions?

A basic stack-based virtual machine is used to execute the code's compiled instructions. The VM manipulates a stack for computations and memory access while processing opcodes iteratively in a loop. Registers such as bp which is a base pointerthat handle function call frames, sp which is a stack pointer that controls data storage, and pc which is a program counter keeps track of the following instruction. Loading values (IMM), jumping (JMP), branching (BZ "branch if zero"), and calling functions (JSR) are all important operations. Arithmetic and logic operations (e.g., ADD, EQ) pop values from the stack, compute results, and push them. System calls like OPEN and PRTF communicate with the operating system.

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

Fixed-size pre-allocated pools that are separated into sections for symbols, code, data, and the stack are used by the code to manage memory. Identifiers like variables and functions are stored in the sym array, which uses linear probing to resolve collisions. Global variables and string literals are stored in data, whereas created VM instructions are stored in the e array. Function calls and local variables are handled by the stack, which is controlled by sp and grows downward. MALC and FREE opcodes wrap system calls that imitate dynamic memory, such as malloc and free. Memory must be manually released because there is no garbage collection.