COSC 320 - c4_report

Lina Benna 100063450

March 2025

1 Lexical Analysis

The code performs lexical analysis by reading the source code character by character and identifying tokens (such as keywords, identifiers, numbers, operators, etc.) based on specific patterns.

The function first scans character by character and processes it to determine the corresponding token. Below is a short summary on how the compiler tokenizes parts of the source code.

Character/Pattern	Token Type	Action
`\n`	Newline (Line Increment)	Increments line number
`#` or `//`	Comment	Skips until end of line
`[a-zA-Z_]`	Identifier	Stores in symbol table
`[0-9]`	Number (Decimal, Hexadecimal, or Octal)	Parses and stores the number
`'` or `"`	String Literal	Handles escape sequences and stores the string

Table 1: Token Identification in Lexical Analysis

For certain characters like {, }, ;, (,), etc., the function identifies them as specific tokens and returns immediately without further processing. The main goal of this process is to convert the raw source code into manageable tokens that can be used for parsing and further compilation steps.

2 Parsing Process

The C4 compiler directly handles the language grammar without creating an Abstract Syntax Tree. It writes instructions that can be assembled while evaluating the expression. So, it doesn't use the usual recursion method to break down the program's source code.

The parsing process begins by retrieving simple data types like int and char, and then moves on to function and variable definitions. The expr() method parses statements, handling operations like arithmetic, function calls, and memory access. It follows operator precedence, with stronger operators like multiplication and addition being evaluated last. Control structures such as if, while loops, and return statements are parsed by the stmt() function, which generates jump and branching instructions. The parser ensures correct expression evaluation, variable assignments, and proper handling of control flow.

3 Virtual Machine Implementation

The virtual machine (VM) executes compiled instructions through a main loop that continuously fetches and processes instructions. Each instruction is stored as an integer, and its operation is determined based on the instruction's code (denoted by the variable I).

The loop begins by fetching an instruction from the pc (program counter) register. Once the instruction is fetched, the VM executes it according to its opcode value. Each opcode corresponds to a specific operation, such as arithmetic, memory operations, or control flow.

For example, the LEA instruction loads a local address into the variable a, while the IMM instruction loads either an immediate value or a global address.

4 Memory Management

The code handles memory allocation and deallocation through several dynamic memory allocations. Memory for the symbol table, code section, data section, and stack is allocated using malloc. The size of each area is set to 256 KB (the value of poolsz). Specifically, memory is allocated for the symbol table (sym), code area (e), data area (data), and stack area (sp).

Once the program completes or exits, memory is deallocated using the free function. This deallocation occurs when the memory is no longer needed, such as after the program exits or when the stack is unwound following function returns. The code does not explicitly handle memory management errors, such as memory leaks, and relies on malloc to allocate memory as required. Deallocation is performed manually at the end of the program's execution using free, ensuring that the memory used is properly released and to avoid memory leaks.