Khalifa University of Science and Technology

Department of Computer Science

COSC 320 – Principles of Programming Languages

Assignment 1

Instructor: Dr. Davor.

Submitted by: Nayef Alnaqbi (100058755).

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

Lexical analysis in C4 involves reading the source code and splitting it into meaningful tokens, a process supervised mainly by the next() function. This function scans through the source code letter by letter while looking for keywords, identifiers, numbers, and operators. It strips away whitespace and comments while recognizing special characters like parentheses, brackets, and semicolons. The next() function makes a hash of an identifier, then compares the hash to a symbol table to see if it was defined as a variable or keyword. If it is a keyword like if, while, or return, it is given an appropriate token type. Numeric constants are parsed according to their style and conditions for sets in decimal, hexadecimal, and octal format. Literal strings are positioned in memory with the first address being their token value. Operators such as ==, !=, <=, and >= are immediately recognized upon the next character after noticing the = or < signs. By classifying tokens, recognizing numbers, and managing the symbol table, the next() function token stream ready for the parsing phase of the process. It makes sure the token stream is prepared by handling these issues dynamically.

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

Instead of the C4 compiler using a standard method of recursion for extracting the base structure of the program source code, it tackles the language grammar without generating an Abstract Syntax Tree. Instead, instructions that can be assembled are written while the expression is being evaluated. The first step to Perform Parsing is to retrieve the most elementary data type which could be an int, char and then parse the rest of the function definitions and variable definitions. The parsing of statements is done through the expr() method, where the parsing for arithmetic operations, function calls, and memory fetches is done. This function also follows the binding strength of operators in a multi-level nested manner where the strong operators

such as asterisk and plus are evaluated last. The more delicate operators which group the code like if, while loop and return statements are dealt with by the stmt() function where the branching and jump instructions are programmed. The parser guarantees that the expressions are evaluated in a correct manner, variables are given values, and the control loop structures are performed in the right way. Because in this case we talk about a “collapsed” representation of the tree parsers, there is no clear hierarchy between instructions and translation units of code. Because the AST is explicit, parsing and code generation are decoupled which makes the compiler easier to modify and expand, at the cost of additional weight.

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

The virtual machine for C4 code uses a basic stack machine to run the compiled code. The virtual machine runs a simplified instruction set crafted solely for the compiler. It supports various functions such as using and storing numbers, branching commands, calling functions, and even system functions. The process starts with loading the compiled bytecode, as well as the setting of the stack. The last step is jumping to the entry of the main() function. The virtual machine keeps track of state and function call frames using registers like pc (program counter), sp (stack pointer), and bp (base pointer). Commands like ADD (addition) and SUB (subtraction) manipulate the stack and allow for calculations to be made. Branching instructions such as BZ (branch if zero) and BNP (branch if not zero), allow for condition execution while JMP allows for changing instructions. Function calls are done using the JSR (jump to subroutine) command, the opposing command LEV (leave subroutine) state the instructions last and goes back to the previous step. The virtual machine also includes system calls that handle file and memory allocation, and standard output functions like printf.

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

The C4 compiler's memory management is simple, yet efficient. A memory pool, which stores symbols, compiled code, and runtime data, is pre-allocated. This pool is divided into different regions: one for the symbol table, another for the compiled bytecode, and a third for the stack. Stack allocations occur dynamically when functions are called, which assign local variables offsets to the base pointer. Global variables are allocated fixed memory addresses within the data segment. Memory allocation from the heap is enabled through malloc(), and deallocation is done using free(). If the dynamically allocated memory is not explicitly freed, it may cause memory leaks since the compiler does not implement garbage collection. The stack is managed using pointer arithmetic and local variables are tracked through explicitly pushing and popping function call frames. The approach takes advantage of efficient execution, but the burden of managing memory is placed on the programmer, which can be a tedious task.