CMPE 230: SYSTEMS PROGRAMMING

In this project, the primary objective is developing an interpreter for an advanced calculator, using C programming language. The calculator has basic arithmetic operations, binary operations and functions. The interpreter is responsible for checking the input for possible syntax errors and generating the correct result unless the input is invalid.

Execution of program is done by using the terminal, so it can be said that the program interface is the terminal of OS. The program doesn’t need any parameters, I nserting the file path of the executable file of program to the terminal is sufficient for execution of the program. Termination of the program is done by the termination command of terminal (e.g. Ctrl + C for Windows).

As it was written earlier, the main functionality of the program is calculating the result of given input by a user. The input is directly taken from the terminal, since the program doesn’t take any parameters, all the user has to do is inserting the operation to the terminal and pressing the Enter button. Unless the input has an error, given operation is done by the program. In case of an invalid input, “Error!” message will be printed to the terminal.

The operations can be categorized into two groups: equations (a = 2 \* 5) and non-equations (2 \* 5). For equations, left hand side of an operation must be a single variable. Right hand side of an operation is considered as the value of the variable, which is stored in the program for the upcoming operations. After the assignment, nothing will be printed to the terminal. Whereas for non-equations, the result is printed to the terminal.

The program is consisted of only one file. This file has the main method and all other methods which constructs the whole algorithm. In the beginning of the file, there are type definitions and structs, which are essential for the program. Method declarations, global variables and main method are right after these type definitions. Main method doesn’t take any parameters and return anything, it takes input and calls the necessary methods in order. When the method is done, frees the memory.

Before going further into the algorithm, it is useful to understand the structs which were defined in the beginning of file. The first one of them is Token, represents the smallest unit of the given inputs, can also be considered lexeme. Token has 4 members and the first one of these members is TokenType. It is the terminals (in BNF notation) in the scientific calculator. It is very helpful to classify and specialize the Token, whether it is an operator, function call, variable etc. There are 18 different TokenType’s, their functionality can be easily understood by its name. (e.g. ADDITION for “+”, COMMA for “,“ ).

Tokens have 3 other members: Id, name and number. Id is the string form of TokenType. It makes debugging very easy, since C doesn’t give structs a default “toString” method. Name and number isn’t significant for all Tokens, however they are very useful for specific types of Tokens and other methods. Name is very functional for VARIABLE TokenType and another struct whose has the same name, Variable (will be explained explicitly). Number is also very useful for the CONST TokenType, it sstores the corresponding integer value of the integer part of the string in the given input.

Another important struct which is frequently used in the code is Node. It is used in creating parsing trees which represents the divided form of given input and they are constructed by Nodes. As traditional binary trees, Node keeps track of its right and left children, also has a name and carries data. Except leaf nodes (they represent CONST and VAR Tokens) and the nodes representing NOT function (it has one child), all nodes have two children.

The last struct in the code is Variable. It has a very specific use, only in hash table, which is an array of Variable pointers and enables us to reach the value of variables. Variables has a key, which is the same string with the name member of VARIABLE Tokens, and the integer data. If any data hasn’t provided to the program yet and the user wants to use it, 0 is assigned as the default value of Variables. When user assigns a value to a variable, hashFunction() generates the index of position where the variable will be inserted, and insert() function inserts the variable pointer to that index.

Now we can discuss the algorithm. It has three main parts: Lexical analysis, parsing and evaluating. First part of the algorithm is lexical analysis, which is done by createToken() method. It takes 2 parameters, the first one is the input string and the second one is the address of the number of tokens, which was assigned 0 at the beginning in the main method. Number of tokens is crucial in terms of error checking (will be explained later). The method returns the list of tokens, tokenized form of given input. In the upcoming part of algorithm, the parsing part, this list of tokens will be used.

After the lexical analysis, the parsing part is done by multiple recursive methods. Parsing part is the most crucial part of the algorithm. The main idea behind the algorithm is dividing the current expression into terms and factors recursively, considering the operational precedence (from least to most). Primary parsing function is parse() function, which takes 2 parameter: The token list and position. Token list is tokenized form of input, the result of the lexical analysis, and the position is the index, indicates which token is currently processing. All parsing functions have these parameters.

All parsing functions except parseFnc() and parseF() starts with calling other methods. Each method calls the first level above in operational precedence. Unless the input has a function with two parameters(lr(), rr(), ls(), rs()), last call is to parseF() method. parseF() function is responsible for creating nodes for variables, integers and not() function. not() function is included to this method because it just takes one parameter and it has higher precedence than other function calls. Highest level which can be reached by the algorithm is the parseF() method, since we can’t go further than integers and variables. If the input has at least one of these functions with two parameters, last call is to parseFnc(). Main reason why the program was implemented in this way is that these functions directly returns integer values, so they can be considered as integers, after evaluating the expression they have as parameters. After finding the values they return, parsing continues in a recursive way.

Last part of the algorithm is the evaluating part. It is done by a single method, evaluate(). It takes just one parameter, root node of the tree. From starting the root, it goes until the leaf nodes of tree recursively. When it reaches the leaf node, since its children are NULL, it goes back to parent node and does the necessary operation depending on the token type of node. This part of algorithm is only done when the input is valid. If any kind of error detected in lexical analysis or parsing, this part of algorithm won’t be executed. This control is done by a global variable, errorFlag. Similarly, printing the value of evaluate function is controlled by another global variable, printFlag.

It wasn’t an easy task for us to implement an interpreter. Constructing the algorithm was hard and took considerable amount time. Coding the parseF() was the hardest part of the program, after finding out that part things started to get easier. It might take less time if we were coding in Java, however I enjoyed the flexibility of C. This flexibility sometimes caused problems but generally we enjoyed coding in C. But C compiler isn’t as reliable as Java compiler so debugging was a bit hard, maybe if we were more experienced at coding in C it would be easier. Detecting errors was very tricky, it would be very helpful if we had more example inputs. Trying to cover whole syntax errors was like an another project, I hope we succeed it. In addition to these; description was clear, we didn’t have difficulties on understanding the task, also Piazza questions helped us to find bugs in our program. To sum up, in the beginning the project seemed very hard and it was until reaching some point. After the understanding how should parsing be done in theory, implementing was relatively easy. At the end of the day, we think it was a good and satisfying experience.