CMPE 230: SYSTEMS PROGRAMMING

01.04.2023

PROJECT 1

PREPARED BY:

- ERAY EROĞLU 2020400096
- BERKE KARTAL 2020400198

In this project, the primary objective is to develop 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 the program is done by using the terminal, so it can be said that the program interface is the terminal of the OS. The program doesn't need any parameters, inserting the file path of the executable file of the program into the terminal is sufficient for the execution. Termination of the program is done by the termination command of the terminal (e.g. Ctrl + D for Linux).

As mentioned 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 to insert the operation into the terminal and press the Enter button. Unless the input has an error, the given operation is done by the program. In case of invalid input, an "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, the left-hand side of an operation must be a single variable. The right-hand side of an operation is considered 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 consists of only one file. This file has the main method and all other methods which construct the whole algorithm. At the beginning of the file, there are type definitions and structs essential for the program. Method declarations, global variables, and the main method are right after these type definitions. The main method doesn't take any parameters and return anything; it takes input and calls the necessary methods in order. Once the method is done, memory is freed.

Before going further into the algorithm, it is useful to understand the structs that were defined at the beginning of the file. The first one of them is a Token, which represents the smallest unit of the given inputs and can also be considered a lexeme. The token has 4 members and the first one of which 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 or a variable, etc. There are 18 different TokenTypes, their functionality can be easily understood by their names. (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 aren'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 that has the same name, Variable (will be explained explicitly). The number is also very useful for the CONST TokenType, it stores the corresponding integer value of the integer part of the string in the given input.

Another important struct that is frequently used in the code is Node. It is used for creating parsing trees which represent 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 for 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 that enables us to reach the value of variables, yet it is only used in a hash table, which is an array of Variable pointers. Variables have a key, which is the same string with the name member of VARIABLE Tokens, and the integer data. If any data hasn't been provided to the program yet and the user wants to use it, 0 is assigned as the default value of Variables. When the user assigns a value to a variable, the hashFunction() generates the index of position where the variable will be inserted, and the 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. The first part of the algorithm, lexical analysis, is done by the 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. The number of tokens is crucial in terms of error checking. The method returns the list of tokens (the tokenized form of the given input), which will be used in the upcoming part of the algorithm, the parsing part.

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

All parsing functions except parseFnc() and parseF() start by calling other methods. Each method calls the first level above in the operational precedence. Unless the input has a function with two parameters(lr(), rr(), ls(), rs()), the last call is to parseF() method. parseF() function is responsible for creating variable nodes, integer nodes, and not() function nodes. The not() function is included in this method as it just takes one parameter and it has higher precedence than other function calls. Since we can't go further than integers and variables, the highest level that can be reached by the algorithm is the parseF() method. If the input has at least one of these functions with two parameters, the last call is to parseFnc(). The program is implemented so, as these functions directly return integer values and can be considered as integers after evaluating the expression they have as parameters.

The last part of the algorithm is the evaluation part, which is done by a single method, evaluate(). It takes just one parameter: the root node of the tree. Starting from the root, it goes until the leaf nodes of the tree recursively. When it reaches the leaf node, since its children are NULL, it goes back to the parent node and does the necessary operation depending on the token type of the node. This part of the algorithm is done only in cases of valid input. , which means that if any kind of error is detected in lexical analysis or parsing, this part of the algorithm won't be executed. This control is done by a global variable, errorFlag. Similarly, printing the value of evaluate() method 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 a considerable amount of time. Coding the parseF() was the hardest part of the program. It might have taken less time if we had been coding in Java, however, I enjoyed the flexibility of C. This flexibility sometimes caused problems but generally, we enjoyed coding in C. But as the C compiler isn’t as reliable as the Java compiler, debugging was a bit hard, maybe if we had been more experienced at coding in C it would have been easier. Detecting errors was very tricky, it would have been very helpful if we’ve had more example inputs. Trying to cover whole syntax errors was like another project, and we hope we successfully covered them all. In addition to these points; the description was clear, and we didn’t have difficulties understanding the task, also Piazza questions helped us to find bugs in our program. To sum up, at the beginning, the project seemed very hard and it was so up until reaching some point. After understanding how parsing should be done in theory, implementing was relatively easy. At the end of the day, we think it was a good and satisfying experience.

Input and Output Examples:

First example:

```
> x = 4

> x * 2

8

> y = 3 * x

> y

12

> x – y

-4

> <Ctrl -D>
```

Second example:

```
> x = 15 15

Error!

> LEDZEPPELIN

0

> a             =                        87

> a

87

> <Ctrl -D>
```

Third example:

```
> z = 77 – 8

> x = not(not (2) * z

Error!

> x = not(not (2)) * z

> x

138

> z = 2

> y = z + (-5)

Error!

> y

0

> <Ctrl -D>
```

Fourth example:

```
> Eray = ls (4,2)

> Berke = 15

> f = Eray – berke

>f

16

> k = Eray – Berke

> k

1

> Eray –

Error!

> <Ctrl -D>
```