Course: CMPE 230 SYSTEMS PROGRAMMING

Students names: GÜLŞEN SABAK - MAHMUT BUĞRA MERT

Submitted Person: MAHMUT BUĞRA MERT

Project Name: AdvCALC

PROGRAMMING PROJECT

Submission Date: 01.04.2023

Phase 1: INTRODUCTION:

In the AdvCalc project, we implemented an interpreter for an advanced calculator using the C programming language. The advanced calculator (AdvCalc) will accept expressions and assignment statements. Our algorithm is based on the lexical analysis and Shunting Yard algorithm which helps us to transform infix expressions and assignments to postfix notation. While implementing the AdvCalc, we used different data structures. These data structures are: Linked List based Stack and Queue to implement Shunting Yard algorithm and Dictionary to calculate assignments such as "a = 5 +3" etc. Also, we determine precedence orders among operators and functions for the calculation part to calculate the values written in the interpreter correctly. At the end we create some functions which are useful to print errors. Lastly, we print the result if there is no error.

Phase 2: PROGRAM INTERFACE:

To run the program, first you need to open the terminal. Then, you should go to the root folder where "main.c" and "Makefile" are located via using terminal. After that, you can run the program by typing "make" and pressing enter. After you finish with the program, you can terminate the program by pressing Ctrl + D simultaneously.

Phase 3: PROGRAM EXECUTION:

For the input part, we expect users to put expressions and assignments to our advanced calculator. However, the numbers have to be in "integer" format. Also, we expect advcalc to calculate "summation (a + b), multiplication (a * b), subtraction (a - b), bitwise and (a & b), bitwise or (a | b), shifting right (rs(a,i)) and left (ls(a,i)) , rotating right(rr(a,i)) and left (lr(a,i)), bitwise xor(xor(a,b)), and bitwise complement (not(a))." In addition, advcalc helps users to assign a variable to a number, and use, update this variable later. Also, advcalc is case-sensitive for variable names. For the output part, we have 2 options which are printing result and printing error message.

Case for printing result: If all operations are valid and all numbers are integer, then advcalc calculate the result and print it.

There are plenty of edge cases to print error message but I will give below, some cases for printing Error:

If parentheses are in the wrong order or missing, then we print the error message.

If a user puts an "n" parameter inside the function which expects "m" parameters ($n \neq m$) , then it prints an error message.

At the end, if user wants to leave the advcalc, then just push "<Ctrl-D>."

Phase 4: INPUT AND OUTPUT:

The advcalc program takes input from the terminal. User can write the input, when he/she sees the "> ". Output is written to the next line of the input. However, if the user writes input as an assignment, then output will not print anything.
For example in "a)" part output will be printed.
a)
> 9 * 6 - 4
50
However, in "b)" part output will not be printed.
b)
>x = 54
> y = x+12 - 4* 8

Phase 5: PROGRAM STRUCTURE:
Data Types:
Token: Token is a single unit we get, after parsing the line. It has 3 data fields.
Data Fields:
`TokenType type` → Stores the type of the token.
For example: TOKEN_TYPE_NUMBER
`char val[257]` → Token value. For example: "xor", "24", "a", "(" .
`char identifierValue[257]` → Identifiers' integer value. For example: "10", "0", "24".

TokenType: It is an enumerator. It holds Token types.

Node: Node has 2 different properties. One of them is Token and the other is Node pointer which points to the next Node. It is used for implementing Queue, Stack, and Dictionary.
Data Fields:
`Token data` → Stores Tokens.
`Node* nextNode` → Points to the next Node.

Queue Implementation: Queue is used to implement Shunting-yard Algorithm. It has 4 functions:
`void enqueue(Node* *firstNode, Token data):` It creates a Node, stores a Token(data) in it, and bonds the Node with the other Nodes. "firstNode" is a pointer to the pointer which points to the first Node of the Queue.
`void dequeue(Node* *firstNode):` It deletes an element from the end of the Queue. "firstNode" is a pointer to the pointer which points to the first Node of the Queue.
`void printQueue(Node* currentNode):` We use it to debug the code, but it is not used intrinsically. If you want to understand the algorithm correctly, you can call it in the main

part. It prints the elements of the Queue. “currentNode” is a pointer to the first Node of the Queue.
`bool hasOneElt(Node* currentNode):` It returns true if the Queue has one element. “currentNode” is a pointer to the first Node of the Queue.

Stack Implementation: Stack is used to implement Shunting-yard Algorithm. It has 5 functions:
`void push(Node* *firstNode, Token data):` It creates a Node, stores a Token(data) in it, and bonds the Node with the other Nodes. “firstNode” is a pointer to the pointer which points to the first Node of the Stack.
`Token peek(Node* *firstNode):` It returns the element at the top of the Stack. “firstNode” is a pointer to the pointer which points to the first Node of the Stack.
`void pop(Node* *firstNode):` It deletes the element at the top of the Stack. “firstNode” is a pointer to the pointer which points to the first Node of the Stack.
`bool isEmpty(Node* firstNode):` It returns true if the Stack is empty. “firstNode” is a pointer to the first Node of the Stack.
`void printStack(Node* currentNode):` It prints the elements of the Stack. “currentNode” is a pointer to the first Node of the Stack.

Dictionary Implementation: Dictionary is used to store initialized variables. It has 5 functions:
`char* get(Node* startNode ,char* identifier):` It returns the identifierValue of the given identifier. “startNode” is a pointer to the first Node of the Dictionary.
`void add(Node* *startNode, Token newToken):` It adds a new identifier to the Dictionary. “startNode” is a pointer to the pointer of the first Node of the Dictionary.
`void printDict(Node* startNode):` It prints the elements of the Dictionary. “startNode” is a pointer to the first Node of the Dictionary.
`void updateValue(Node* startNode, char newValue[257], char identifier[257] ):` It updates the value of an identifier in the Dictionary. “startNode” is a pointer to the first Node of the Dictionary.
`bool doesExist(Node* startNode ,char* identifier):` It returns true if the given identifier exists in the Dictionary. “startNode” is a pointer to the first Node of the Dictionary.

We can summarize our code in 3 main steps:

- First, make lexical analysis of the input line with parse(),
- Then, transform the analyzed line from infix to postfix notation.
- Then, using the postfix notation, calculate the expression and print it or assign it to the given variable.

Phase 6: EXAMPLES:

One example for parentheses errors:

```
> ((8 - 9)*7 +(12 -(8) *3) -9 *(12 +4) - (123*789)*(159)
Error!
```

Example in the below, shows the characteristic of advcalc, when it sees the assignment. If you didn't initialize the variable, advcalc gives 0 as its value. Also, it can calculate expressions with initialized or uninitialized variables.

```
> a = 78
> b =                a+4 -                c*                8 + e-2
> b
80
> x = 1
> y = x+ 5*9-8
> z = x -y*2+2
> z
-73
> y
38
> d
0
> y+d
38
```

Also there are some examples in below to show how functions are calculated by advcalc:

```
> xor(               14, (        rs (2 , 15)   ))
14
> rs (2 , 15)
0
> xor(14, 0)
14
```

Also advcalc return Error in invalid operations:

```
> 4 / 9 - 6*                              7
Error!
```

Phase 7: IMPROVEMENTS AND EXTENSIONS:

Actually, there are some weak points about the advcalc. For example, we can make a lot of calculations with advcalc but some basic calculations still don't exist such as division operator etc. Apart from this, the advcalc can just calculate, if the inputs are integer. We can optimize it to calculate float numbers too.

In our opinion, one of the strong points about advcalc is, it can calculate some different expressions which normal calculators cannot do. For example it can calculate left, right rotations and left, right shifts .

Phase 8: DIFFICULTIES ENCOUNTERED:

Actually, we haven't faced much difficulty when we are implementing the project. However, sometimes we get "Segmentation fault - core dumped error." After that we recognize the reasons for the error. First reason is writing the string in this format: `char * str;` instead of `char str[].` Second reason is about the queue. When we dequeue the queue, if there is no element in it, we get this error. We solved it via firstly pushing the stack after that dequeue the queue. Facing the second error, helps us to understand the data structures clearly. Also, with the help of the first error, we learn C better. In addition, we learn how to debug a C code when we are trying to find the reasons for errors.

Phase 9: CONCLUSION:

At the end of this project, we have learnt how to use struct in C and create our data types, using pointers effectively, using the past knowledge for creating Queue, Stack, Dictionary, working as a team and general structures of C. In our perspective, this project is a little bit hard to implement but it is very appropriate to understand how systems work in a computer. In a nutshell, implementing an interpreter is hard, but joyful and teachful.