CMPE 230

Students:

Hanaa Zaqout 2020400381

Deniz Bilge Akkoç 2020400135

Project:

Advanced Calculator

C Programming Language

Submission date:

1. April 2023

− Introduction:

In this project we implemented an advanced calculator on c language that aimed to calculate basic “ +,-,* ” and bitwise operations “xor, and, or, ...”. This calculator enables users to assign variables and calculate expressions. We divided the project into little functions like input cleaner, infix to postfix transformer, evaluator etc. and worked on them individually.

− Program Interface & Execution:

On the terminal, go to the file, write the **make** command then run the program with "./advcalc". Now the code is ready to get input. There are two types of inputs: assignments and expressions. Users can assign a value to a variable or calculate an expression. Assigned variables can be used in other expressions and assignments. Calculator will stay open and keep getting inputs until the user uses the Ctrl+d command.

− Input and Output:

Input can be read from the terminal. Input can be either an assignment (e.g. x=1+2*3) or a query (e.g. x+2). Our program can calculate the value of any expression including the following operations: + - * xor …. Also, you can comment on any line using %. If the input format is not valid, the program prints "Error!".

− Program Structure

Applying the following functions to every line read by fgets():

- **Cleaning input:** convert the given line to an array of tokens(strings) skipping any spaces or tabs.
- **Converting to Postfix:** apply the BNF grammar (mentioned below) on the array of tokens producing a postfix string considering the priority of operations. This part catches most of the errors in the input's syntax.
- **Evaluating the Postfix:** by using stack, pushing operands and whenever come by an operation pop number1 and pop number2. Push the value of "number1 operation number2" back to the stack.
- **Adding to Hashmap:** implementation of our hashmap has 3 arrays.
  Array1: stores 1 if the index is busy otherwise 0.
  Array2: stores variables' names
  Array3: stores values
  How do we hash the variables? We sum up the alphabetic index of each letter in the variable name. Apply mode operation on the sum if it exceeds the length of the hashmap array. If the calculated index is busy, we do linear increment.

BNF Grammar:

The BNF grammar used in building the program:

<expr> := <option> <more options>
<more options> := "|" <option> <more options> | E
<option>:= <prop> <more props>
<more props>:= & <prop> <more props> | E
<prop>:= <term> <more terms>
<more terms> := + <term> <more terms>
| - <term><more terms>
| E
<term>:= <factor> <more factors>
<more factors> := * <factor><more factors>
| E
<factor>:= (<expr>) | <var> | <num> | <logic>
<logic> := xor(<expr>, <expr>) | lr(<expr>, <expr>) | … | not (<expr>)
<var>:= [a-zA-Z]

Remark:

1. “|” will show up as a terminal in the string.
2. := used this symbol to define

All non terminals (e.g. <expr>, <option>, <term>,...)have been implemented as functions to call over and over again. Terminal values (e.g. x, +, xor, (,), +, …) presented as variables that we read from input then we combine them over and over till we build our postfix.

− Examples

x=3+4*7
Derivation of the parse tree:
<var>= <expr>
x=<expr>
x=<option> <more options>
x=<prop> <more props> <more options>
x= <term> <more terms> <more props> <more options>
x= <factor> <more factors> <more terms> <more props> <more options>
x= <num> <more factors> <more terms> <more props> <more options>
x= 3 <more factors> <more terms> <more props> <more options>
x= 3 E <more terms> <more props> <more options>
x= 3 + <term> <more terms> <more props> <more options>
x= 3 + <factor> <more factors> <more terms> <more props> <more options>
x= 3 + <num> <more factors> <more terms> <more props> <more options>
x= 3 + 4 <more factors> <more terms> <more props> <more options>
x= 3 + 4 * <factor><more factors><more terms> <more props> <more options>
x= 3 + 4 * <num><more factors><more terms> <more props> <more options>
x= 3 + 4 * 7<more factors><more terms> <more props> <more options>
x= 3 + 4 * 7 E <more terms> <more props> <more options>
x= 3 + 4 * 7 E <more props> <more options>
x= 3 + 4 * 7 E <more options>
x= 3 + 4 * 7 E
x=3+4*7

While going through the parse tree, our code creates that postfix string that will be evaluated using the stack later.

− Improvements and Extensions

There are around 800 lines of code in our program and because we are not comfortable with using C language and its libraries, we wrote many functions -like space trimmer- that probably already exist in some libraries. Moreover because we work separately, it is possible that there exist functions that could work more efficiently if we combine them together.

− Difficulties Encountered

This was probably our first time working on a project with somebody, it was hard to debug the code and understand each other's code. We tried to solve this by defining functions input and outputs but it was still hard. Learning C and using it in this project was harder because this project requires lots of string operations and in C this means a lot of pointers. We had some pointer crises that took a lot of time to understand why those errors happened.

− Conclusion

We learned pointers are awesome and garbage collector’s creator is my true love. Just kidding, C is more coder dependent than we are used to in java or python, so it feels like we have more knowledge of how memory works after this homework. We learned some pointer tricks and it felt good too.