CMPE 230

İrem Nur Yıldırım

Başak Tepe

Advanced Calculator Interpreter

Systems Programming Course Project

Submission: 01.04.2023

1. Introduction

In this project, we were expected to implement an Interpreter for an advanced calculator that does basic arithmetic and bitwise operations using C language. Our initial approach to this problem was following the structure of an interpreter: first deciding on a grammar, and then coding the lexer and the parser parts. For the lexer part, we followed a simple outline. For the parser part, we had multiple implementation choices: firstly, we tried out a recursive solution and then, we switched to implementing one that includes transforming from infix to postfix expressions.

2. Program Interface

For the activation of the program, the code can be compiled with the “make” command and executed afterwards. For termination of the program, CTRL^D keys should be pressed.

3. Program Execution

The program can be run by using the make command on the makefile within the Unix environment. Afterwards, an executable advcalc will be created. The program will accept assignments and expressions in the form of:

>variable = expression
>expression

The program can be terminated by pressing CTRL^D.

4. Input and Output

The program takes inputs from the terminal and displays the output in the terminal screen as well. Input is in the format of mathematical expressions or/and assignments. Output is in the format of integers or error messages for mathematical expressions. No display is made after an assignment expression. Until CTRL^D has been pressed to terminate the program, the user is prompted again for inputs after the display of each result.

5. Program Structure

The structure followed in this project can be summarized as the diagram in Picture 1. It simply imitates the body of an interpreter with the necessary subparts: the lexer and the interpreter.

5.1 Lexer / Tokenizer

The lexer is responsible for cutting up meaningful pieces of information from the user's input. Later, these pieces of information, or terminals, are converted into a structure called “tokens”. A token has a type and a value: **`NONE, NUM, OP,IDENT, FUNC_CALL,LPAR,RPAR,COMMA,ASSIGN,AND,OR`** make up the types of a token. On the other hand, values of a token are taken directly from the input string itself.

Token Examples

| Type | Value |
|---|---|
| NUM | 5 |
| LPAR | ( |
| COMMA | , |

5.2.Parser

Parser takes as an input the tokenized array, where every meaningful token is stored except whitespaces, and turns them into a postfix denotation. For achieving this it follows these steps:

First, it creates a stack where we will store operators or left parentheses, later on we will check the top element of the stack to decide whether append it to the postfix array

or not. After creating the stack, function starts to iterate through the token array, it directly adds each variable or number-operand- to the postfix array just because the calculation is from left to right and there is no precedence between them. If the current token is an operator or function call we check if this operator is precedent then the one we stored in the stack previously -if there is any-, if the stack has a more precedent operator waiting in the top then we add this operator to postfix. As an example for the input 3+4*5 function first adds 3 then adds + to the stack, since there is not any other one waiting and having more precedence than +. Then sees 4 and directly adds it, then sees * which also does not have a more precedent rival waiting in the stack (+ is waiting but is not more precedent) so it is also added to stack. Then the function sees 5 and directly adds it to postfix and leaves the for loop, so as we see in this example after exiting the for loop we have still some operators, in fact all the operators in this example, waiting in the stack, so we check if there is any operator remained and add all of them to postfix. Eventually, what makes this last appending from stack to postfix array operation ordered is that we add them to the stack such that they have decreasing level of precedence from top to bottom. In left and right parentheses handling this function applies two if blocks to catch left and right parentheses separately, whenever it sees a left parentheses it adds it to the stack but it is to be removed later on, when a right parentheses is encountered it adds all the operators or functions waiting in the stack - until a left parentheses is detected -to the postfix array because for loop came to the end of a term in parentheses and all the operations inside these parentheses must be added to postfix array before getting out of it.

5.4 Evaluating

For evaluating the postfix array we again create a stack, but this time it consists not of tokens but integers only, we iterate through the postfix array and if we see an integer directly append it to the stack. Otherwise if the current element is an operator, then we extract the last two integers of the stack and do the current operation on them, using a postfix array to make the evaluation function easy and basic as much as possible .

### 5.5 Hash Table

To cover the need for memory after each assignment statement, we needed a data structure that could hold tuples of information: the variable and its value. Our search for such a type reached a dead end when we realized standard libraries of the C language had no such attributes. Therefore, we implemented a quite simple hash table that included insert and find methods. No delete method was required for this project.

## 6. Examples

Example 1 (expression):
input -> (3+4)*5
output -> 35

Evaluation process:

1) Tokenizing
   (, 3, +, 4, ), *, 5
2) Infix to postfix
   (3+4)*5 -> 3 4 + 5 *
3) Evaluation
   result is 35

Example 2 (assignment):
input 2 -> a = xor(3,4)
output ->

Evaluation process:

1) Separating the variable and the expression
   Variable a
   Expression xor(3,4)
2) Tokenizing
   xor, (, 3, 4, )
3) Infix to postfix
   xor(3,4) -> 3 4 , xor
4) Evaluation
   result is 7
5) Assignment
   key "a" is hashed into the table with value 7.

7. Improvements and Extensions

One possible improvement we discussed was to come up with a better way to handle erroneous inputs. Our initial idea was to detect possible senseless inputs and overcome them case by case. This solution led to concerns about our code being redundant and the implementation being inefficient. Thus we opted for a more methodical way of handling errors by checking postfix stacks for operations and operands; however this solution could also be leaving out a few edge cases. Overall we believe the way we handle errors could be improved.

8. Difficulties Encountered

The major difficulty in this project could be addressed as the fact that both of the project partners never coded in C language before. In addition, our first approach to the problem was to code a recursive parser, which later led to some other difficulties regarding the use of parentheses in inputs. About one week into the project, we decided to abandon our first draft and use infix to postfix transformation in the parser instead.

9. Conclusion

Overall this was a challenging yet enjoyable project where we had the chance to research and discover a lot of ideas. We ended up having an Advanced Calculator Interpreter that could address some syntax errors and evaluate simple expressions as given in the project description examples. Towards the submission due date, we discovered a new concept: the Shunting Yard Algorithm. To our surprise, so far, we were implementing the Shunting Yard algorithm without explicit knowledge. We are very much pleased with the outcomes: learning about this new algorithm, the chance to apply what we learned in CMPE260 on grammars, experimenting in C, and making a Systems Programming course project, not to mention writing a documentation for the first time.

# 10. Appendices

## 10.1 Appendix A

The BNF Structure
Least precedence to most precedence is followed

```
//bitwise or
<program> ::= <expr> ('|' <expr>)*

//bitwise and
<expr> ::= <term> ('&' <term>)*

//addition and subtraction
<term> ::= <factor> ('+' <factor> | '-' <factor>)*

//multiplication
<factor> ::= <multiplicand> ('*' <multiplicand>)*

//function calls ls rs lr rr not xor
<multiplicand> ::= <xor> | <not> | <shift_rotation> | <number> | <variable>
<xor> ::= 'xor' '(' <prop> ',' <prop> ')'
<not> ::= 'not' '(' <prop> ')'
<shift_rotation> ::= <shift_left_expr> | <shift_right_expr> | <rotate_left_expr> |
<rotate_right_expr>
<shift_left_expr> ::= 'ls' '(' <prop> ',' <number> ')'
<shift_right_expr> ::= 'rs' '(' <prop> ',' <number> ')'
<rotate_left_expr> ::= 'lr' '(' <prop> ',' <number> ')'
<rotate_right_expr> ::= 'rr' '(' <prop> ',' <number> ')'


<prop> ::= <variable> | <number>
<number> ::= [0-9]+
<variable> ::= [a-zA-Z][a-zA-Z]*
```

## 10.2 Appendix B

Our source code and makefile can be found within the same zip file as this documentation.