Data Structures
Lab Manual 7

Topic: Evaluation of Expressions

Session: Spring 2023

Faculty of Information Technology
UCP Lahore Pakistan

Objectives:

The evaluation of expressions is important in many areas, including computer programming, scientific calculations, and engineering applications. It allows us to perform complex calculations quickly and accurately, and is used in a wide range of applications, from calculating the trajectory of a spacecraft to analyzing financial data.

Other objectives of evaluating expressions may include:

1. Ensuring that the expression is well-formed: Before evaluating an expression, it is important to check that it is well-formed, i.e., it has proper syntax and is free of errors. This includes checking for balanced parentheses, correct order of operators, and valid operands.
2. Optimizing the evaluation process: Evaluating expressions can be computationally expensive, especially when dealing with large expressions. One objective of evaluating expressions is to optimize the process by minimizing the number of computations required or by using efficient data structures and algorithms.
3. Handling expressions with variables: Expressions can include variables, which can have different values at different times. Evaluating expressions with variables involves substituting the values of the variables in the expression and then evaluating the resulting arithmetic expression.

Overall, the objective of evaluating expressions is to accurately determine the value of an expression while ensuring that it is well-formed, efficient, and handles variables appropriately.

Expressions Evaluation:

The evaluation of expressions typically follows the order of operations, which dictates the order in which arithmetic operations should be performed. The order of operations is as follows:

1. Parentheses: Evaluate expressions within parentheses first.
2. Exponents: Evaluate exponents (i.e., powers and roots) next.
3. Multiplication and division: Perform multiplication and division operations in order from left to right.
4. Addition and subtraction: Perform addition and subtraction operations in order from left to right.

For example, consider the expression 3 + 4 * 2. According to the order of operations, we must perform the multiplication operation first, followed by the addition operation. Therefore, the expression evaluates as follows:

3 + 4 * 2 = 3 + 8 = 11

Another important aspect of expression evaluation is ensuring that the expression is well-formed. This involves checking for balanced parentheses, correct order of operators, and valid operands. For example, the expression 2 * (3 + 4)) is not well-formed because it has an extra closing parenthesis.

In addition, expression evaluation may involve handling variables, which can have

different values at different times. Evaluating expressions with variables involves substituting the values of the variables in the expression and then evaluating the resulting arithmetic expression.
Arithmetic Expressions can be written in one of three forms:

- Infix Notation: Operators are written between the operands they operate on, e.g. 3 + 4.
- Prefix Notation: Operators are written before the operands, e.g + 3 4
- Postfix Notation: Operators are written after operands.

Implementation of Expression Evaluation:
Here's an implementation of the stack-based algorithm for expression evaluation. This algorithm uses a stack data structure to store operands and operators as they are encountered in the expression, and to perform the necessary calculations when operators are encountered.
The algorithm works as follows:
1. Start by initializing an empty stack.
2. Iterate through the expression from left to right.
3. If the current token is an operand (a number or variable), push it onto the stack.
4. If the current token is an operator (+, -, *, /, ^), pop the top two operands from the
   stack and apply the operator to them, then push the result back onto the stack.
5. When the iteration is complete, the final result will be the top element of the stack.

For example, consider the expression "3 + 4 * 2". Here is how the stack-based algorithm would evaluate this expression:
1. Push 3 onto the stack. (Stack: [3])
2. Encounter"+", ignore it.
3. Push 4 onto the stack. (Stack: [3, 4])
4. Encounter"*", pop 4 and 3, multiply them, and push the result (8) onto the stack.
   (Stack: [8])
5. Encounter end of expression, result is the top element of the stack (8).

This algorithm can be modified to handle other types of expressions, such as those containing parentheses or functions, by modifying the way that operators and operands are pushed and popped from the stack.
Overall, implementing expression evaluation in DSA requires a solid understanding of stacks and the order of operations, and can be a useful exercise in algorithm design and implementation.

Arithmetic Expression Evaluation:

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as Infix notation, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. Polish notation (prefix notation)

It refers to the notation in which the operator is placed before its two operands. Here no parentheses are required, i.e.

+AB

2. Reverse Polish notation (postfix notation)

It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses are required in Reverse Polish notation, i.e.,

AB+

Stack-organized computers are better suited for post-fix notation than the traditional infix notation. Thus, the infix notation must be converted to the postfix notation. The conversion from infix notation to postfix notation must take into consideration the operational hierarchy.

There are 3 levels of precedence for 5 binary operators as given below:
Highest: Exponentiation (^)
Next highest: Multiplication (*) and division (/)
Lowest: Addition(+) and Subtraction(-)

For example

Infix notation: (A-B)*[C/(D+E)+F]
Post-fix notation: AB- CDE +/F +*

Here, we first perform the arithmetic inside the parentheses (A-B) and (D+E). The division of C/(D+E) must be done prior to the addition with F. After that multiply the two terms inside the parentheses and bracket.
Now we need to calculate the value of these arithmetic operations by using a stack.
The procedure for getting the result is:

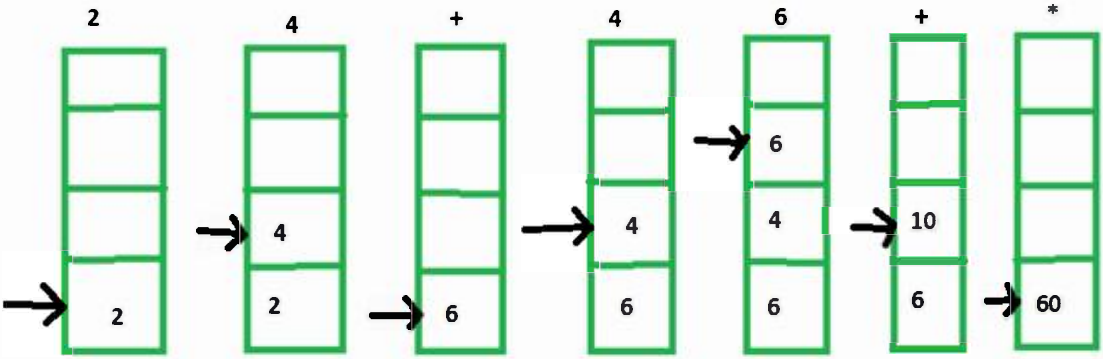
1. Convert the expression in Reverse Polish notation(post-fix notation).
2. Push the operands into the stack in the order they appear.
3. When any operator encounters then pop two topmost operands for

executing
the operation.

4. After execution push the result obtained into the stack.
5. After the complete execution of expression, the final result remains on the top of the stack.

For example
Infix notation: (2+4) * (4+6)
Post-fix notation: 2 4 + 4 6 + *
Result: 60

The stack operations for this expression evaluation is shown below:



Stack operations to evaluate (2+4)*(4+6)

Sample Task:

A program that evaluates the arithmetic infix expression from left to right, using the stack-based algorithm to handle operator precedence and parentheses and then display it on console.

Pseudocode:

1. Define a function named precedence, which takes an operator as input and returns its precedence value.
2. Define another function named applyOp, which takes two integers and an operator and returns the result of the arithmetic operation performed on them.
3. Define a function evaluate, which takes an infix expression as input and returns its result.
4. Two stacks, values and ops, are initialized to store the operands and operators respectively.

5. A loop is run over each character of the input expression.
6. If the character is a space, it is skipped.
7. If the character is a digit, it is read as an integer and pushed onto the values stack.
8. If the character is an opening bracket, it is pushed onto the ops stack.
9. If the character is a closing bracket, all the operators in ops stack are popped and the corresponding operations are performed on the top two values of the values stack until an opening bracket is encountered. The opening bracket is also popped from the ops stack.
10. If the character is an operator, all the operators in ops stack with equal or higher precedence are popped and the corresponding operations are performed on the top two values of the values stack until a lower precedence operator is encountered or ops stack becomes empty. The current operator is then pushed onto the ops stack.
11. After the loop, if there are any remaining operators in ops stack, they are popped and the corresponding operations are performed on the top two values of the values stack until ops stack becomes empty.
12. The result is the top element of the values stack, which is returned.
13. In the main function, the user is prompted to enter an infix expression, which is read using getline.
14. The evaluate function is called with the input expression as argument, and the result is stored in result.
15. The result is printed on the console using cout.
16. End of program

Implementation:

```
#include <iostream>
#include <stack>
#include <string>
#include <sstream>

using namespace std;

int precedence(char op) {
   if (op=='+' || op=='-') return 1;
   if (op == '*' || op == '/') return 2;
   return 0;
}

int applyOp(int a, int b, char op) {
   switch (op) {
   case '+': return a + b;
   case '-': return a - b;
   case '*': return a * b;
   case '/': return a / b;
   }
   return 0;
}

int evaluate(string expression) {
   stack<int> values;
   stack<char> ops;
```

```
for (int i= 0; i < expression.length(); i++) {
    if (expression[i] == ' ')
        continue;

    if (isdigit(expression[i])) {
        int val= 0;
        while (i < expression.length() && isdigit(expression[i])) {
            val = (val * 10) + (expression[i] - '0');
            i++;
        }
        i--;

        values.push(val);
    }
    else if (expression[i] == '(') {
        ops.push(expression[i]);
    }
    else if (expression[i] == ')') {
        while (!ops.empty() && ops.top() != '(') {
            int val2 = values.top();
            values.pop();

            int val1 = values.top();
            values.pop();

            char op = ops.top();
            ops.pop();

            values.push(applyOp(val1, val2, op));
        }

        if (!ops.empty())
            ops.pop();
    }
    else {
        while (!ops.empty() && precedence(ops.top()) >= precedence(expression[i])) {
            int val2 = values.top();
            values.pop();

            int val1 = values.top();
            values.pop();

            char op = ops.top();
            ops.pop();

            values.push(applyOp(val1, val2, op));
        }

        ops.push(expression[i]);
    }
}

while (!ops.empty()) {
    int val2 = values.top();
    values.pop();

    int val1 = values.top();
    values.pop();

    char op = ops.top();
```

```
            ops.pop();

            values.push(applyOp(val1, val2, op));
        }

        return values.top();
}

int main() {
    string expression;
    cout << "Enter an infix expression: ";
    getline(cin, expression);

    int result= evaluate(expression);
    cout << "The result is: " << result<< endl;

    return 0;
}
```

Lab Task:

1. Convert the following infix expression into a postfix expression using your own stack template.
   a + b - (c + d * e / (f - g)) * (h + i)

   Now, evaluate the postfix expression by providing a value for each variable.

2. Convert the following infix expression into a prefix expression

   a - b + c * (d / e - (f + g))

3. Check if an expression is balanced or not

   Example
   {[{}{}]}[()], {{}{}}, []{}() are balanced expressions.

   {()}[), {(}) are not balanced.