Evaluation of Expressions Using Stack

Mr. Akash Yadav
Assistant Professor
Artificial Intelligence & Data Science

Lesson Plan

Subject/Course	**Competitive Coding**
Lesson Title	**Evaluation of Expressions Using Stack**

Lesson Objectives
Learn how operators and operands are handled using LIFO principle.
Implement a program to evaluate a postfix expression using a stack.
Understand how this concept applies in compilers and calculators.

Problem Statement:

Write a program to deal with real-world situations where Stack data structure is widely used Evaluation of expression: Stacks are used to evaluate expressions, especially in languages that use postfix or prefix notation. Operators and operands are pushed onto the stack, and operations are performed based on the LIFO principle.

Objective :

To understand how the Stack data structure is used in evaluating mathematical expressions like postfix (Reverse Polish Notation) or prefix forms.

Concept

Expressions can be written in three common notations:

- Infix: (A + B) * C
- Prefix: * + A B C
- Postfix: A B + C *

In postfix (RPN), operators come after operands.

✅ No need for parentheses

✅ Easy to evaluate using stacks

Why Stack?

Because operations are performed in reverse order of insertion — ideal for **LIFO** behavior.

Steps to Evaluate a Postfix Expression:

1. Initialize an empty stack.
2. Scan the expression from left to right.
3. If operand: Push it onto the stack.
4. If operator:
    - Pop two operands from the stack.
    - Perform the operation.
    - Push the result back onto the stack.
5. After scanning the entire expression,
6. the final result will be on the top of the stack.

Example:

Expression → 5 6 2 + * 12 4 / -

Example:

Let's evaluate → *5 6 2 + * 12 4 / -*

Step	Symbol	Stack Content	Operation / Action
1	5	[5]	Push operand
2	6	[5, 6]	Push operand
3	2	[5, 6, 2]	Push operand
4	+	[5, 8]	6 + 2 = 8
5	*	[40]	5 * 8 = 40
6	12	[40, 12]	Push operand
7	4	[40, 12, 4]	Push operand
8	/	[40, 3]	12 / 4 = 3
9	-	[37]	40 - 3 = 37

Final Result : 37

Postfix Evaluation — Step-by-Step

1. Read the postfix expression left → right (tokens separated by spaces).
2. If token is an operand, push it onto the stack.
3. If token is an operator (+ - * /) :
    - Pop the top value → b.
    - Pop the next top → a.
    - Compute result = a (op) b.
    - Push result back onto the stack.
4. Repeat until all tokens processed.
5. Final answer = single value remaining on stack (stack top).
6. Handle errors: if you try to pop when stack has <2 values → invalid expression; after finish, if stack size ≠ 1 → invalid expression.

(Use integer division only if expression expects integers; otherwise handle floats.)

Code Implementation:

```java
import java.util.Stack;

public class Practical2_PostfixEvaluation {

    public static int evaluate(String exp) {
        Stack<Integer> stack = new Stack<>();

        for (char ch : exp.toCharArray()) {
            if (Character.isDigit(ch)) {
                stack.push(ch - '0');
            } else {
                int b = stack.pop();
                int a = stack.pop();
                switch (ch) {
                    case '+': stack.push(a + b); break;
                    case '-': stack.push(a - b); break;
                    case '*': stack.push(a * b); break;
                    case '/': stack.push(a / b); break;
                }
            }
        }
        return stack.pop();
    }
```

```
24
25      public static void main(String[] args) {
26          String exp = "231*+9-";
27          System.out.println("Postfix Expression: " + exp);
28          System.out.println("Result = " + evaluate(exp));
29      }
30  }
```

Output :

```
Postfix Expression: 231*+9-

Result = -4
```

Time & Space Complexity

- Time Complexity: O(n)
- Space Complexity: O(n)

Explanation:

- Each element (operand or operator) is processed once.
- Stack operations (push/pop) are constant time → overall linear complexity.

Summary

✅ These problems test real-world **stack behavior in expression parsing and runtime tracking**.

✅ Focus areas:

- Operand/operator management
- Parentheses handling
- Operator precedence
- Expression parsing logic

Real-World Applications:

- Expression Evaluation in Compilers — parsing & executing code efficiently.
- Scientific Calculators — internally use postfix evaluation.
- Expression Converters — converting infix → postfix → evaluation.
- Parsing and Syntax Checking — validating expressions in interpreters.

Practice Questions:

1. LeetCode 150 — Evaluate Reverse Polish Notation

🔗 *https://leetcode.com/problems/evaluate-reverse-polish-notation/*

Description: Given an array of strings tokens representing an expression in Reverse Polish Notation, evaluate the expression.

Valid operators: +, -, *, /.

Example Input: ["2","1","+","3","*"]

Output: 9

(Explanation: (2 + 1) * 3 = 9)

2. LeetCode 224 — Basic Calculator

🔗 *https://leetcode.com/problems/basic-calculator/*

Description: Implement a basic calculator to evaluate a simple expression string containing +, -, (,), and spaces.
Use stack to handle parentheses and operator precedence.

Example Input: "(1+(4+5+2)-3)+(6+8)"
Output: 23

3. LeetCode 227 — Basic Calculator II

🔗 *https://leetcode.com/problems/basic-calculator-ii/*

Description: Evaluate a string expression containing non-negative integers and operators +, -, *, /.
No parentheses — but handle operator precedence correctly.

Example Input: "3+2*2"
Output: 7

4. LeetCode 227 — Basic Calculator III

🔗 *https://leetcode.com/problems/basic-calculator-iii/*

Description: Like Calculator II, but supports parentheses (,) and must respect nested evaluation.

Use stack for both operands and operators.

Example Input: "2*(5+5*2)/3+(6/2+8)"

Output: 21

5. LeetCode 636 — Exclusive Time of Functions

🔗 *https://leetcode.com/problems/exclusive-time-of-functions/*

Description: You are given logs of function start and end times.
Use a stack to compute each function's exclusive execution time.

Example Input:

```
n = 2
logs = ["0:start:0","1:start:2","1:end:5","0:end:6"]
```

Output: [3,4]

THANKS!