PROJECT BASED LEARNING REPORT
ON
EXPRESSION EVALUATOR USING STACK

Submitted by:
Team Members :
1.VANSHIKA DHUL - 1/24/SET/BCS/293
2.SIMRA FATIMA - 1/24/SET/BCS/307

Under the guidance of
Faculty Mentor: Ms.Rashi Sahay (Ass. Professor)

Department of Computer Science and Engineering
Manav Rachna International Institute of Research and Studies

Academic Year: 2025–2026

Abstract

This project implements an Expression Evaluator in C which parses and evaluates mathematical expressions consisting of operators and operands. The program supports infix to postfix conversion and evaluates the postfix expressions using stack data structures. The evaluator validates input expressions and computes results for arithmetic calculations, thereby demonstrating the practical use of stacks and expression parsing techniques in computer science.

Introduction

An expression evaluator is a computer program used to calculate the value of arithmetic expressions. Expressions written in infix notation, such as “3 + 4 * 2”, need to be converted and evaluated systematically to provide correct results. This project implements an expression evaluator in C using stack data structures for operators and operands. The program serves as a fundamental example of parsing, operator precedence, and stack implementation, often used in compiler design and interpreters.

Objectives

- Understand and implement stacks for managing operators and operands.
- Develop an algorithm to convert infix expressions to postfix expressions.
- Implement postfix expression evaluation to produce accurate results.
- Validate input expressions for correctness.
- Gain practical experience in C programming and data structure usage.

System Analysis

The system accepts arithmetic expressions from the user as input strings. It identifies operators and operands, converts infix expressions to postfix notation considering operator precedence, and evaluates the postfix expression using stacks. The system provides error messages for invalid input expressions and supports basic arithmetic operations: addition, subtraction, multiplication, division, and exponentiation.

Implementation

Data Structures Used :

- Stacks for integers (operands) and for operator characters and strings.
- Arrays to store temporary expressions during conversion and evaluation.

Main Functionalities :

1. Initializing stacks to hold operands and operators.
2. Checking precedence of operators to guide infix to postfix conversion.
3. Parsing the input expression character-by-character.

4.Converting infix expression to postfix using operator stack.
5.Evaluating postfix expression by performing operations on operand stack.

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <stdbool.h>

#define MAXSIZE 100

// Stack for Integers
typedef struct {
    int data[MAXSIZE];
    int top;
} Stack;
void initStack(Stack *s) {
    s->top = -1;
}
bool isEmpty(Stack *s) {
    return s->top == -1;
}
bool isFull(Stack *s) {
    return s->top == MAXSIZE - 1;
}
```

```
void push(Stack *s, int val) {
    if (isFull(s)) {
        printf("Stack Overflow\n");
        return;
    }
    s->data[++(s->top)] = val;
}
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack Underflow\n");
        exit(EXIT_FAILURE);
    }
    return s->data[(s->top)--];
}
int peek(Stack *s) {
    if (isEmpty(s))
        return -1;
    return s->data[s->top];
}

// Stack for Operators (strings for infix)
#define MAXSIZEMAXSIZE MAXSIZE
typedef struct {
    char ops[MAXSIZE][MAXSIZEMAXSIZE];
    int top;
} OpStack;
void initOpStack(OpStack *s) {
    s->top = -1;
```

```
}
bool isEmptyOp(OpStack *s) {
   return s->top == -1;
}
void pushOp(OpStack *s, const char *op) {
   if (s->top == MAXSIZE - 1) {
      printf("Op Stack Overflow\n");
      return;
   }
   strcpy(s->ops[++(s->top)], op);
}
char *popOp(OpStack *s) {
   if (isEmptyOp(s)) {
      printf("Op Stack Underflow\n");
      exit(EXIT_FAILURE);
   }
   return s->ops[(s->top)--];
}

char *peekOp(OpStack *s) {
   if (isEmptyOp(s))
      return NULL;
   return s->ops[s->top];
}
```

```
// Operator Precedence
int precedence(const char op) {
    if (!op) return 0;
    switch(op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
    }
    return 0;
}
bool isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}

// Convert Infix to Postfix
void infixToPostfix(const char *infix, char *postfix) {
    OpStack stack;
    initOpStack(&stack);
    int i = 0, k = 0;
    while (infix[i]) {
        if (isdigit(infix[i])) {
            // Append operands (numbers) directly to postfix
            while (isdigit(infix[i])) {
```

```
            postfix[k++] = infix[i++];
        }
        postfix[k++] = ' ';
        continue;
    }
    else if (infix[i] == '(') {
        char op[2] = "(";
        pushOp(&stack, op);
    }
    else if (infix[i] == ')') {
        while (!isEmptyOp(&stack) && strcmp(peekOp(&stack), "(") != 0) {
            char *op = popOp(&stack);
            postfix[k++] = op[0];
            postfix[k++] = ' ';
        }
        if (!isEmptyOp(&stack) && strcmp(peekOp(&stack), "(") == 0)
            popOp(&stack);
        else {
            printf("Invalid Expression\n");
            exit(EXIT_FAILURE);
        }
    }
    else if (isOperator(infix[i])) {
        while (!isEmptyOp(&stack) && precedence(peekOp(&stack)[0]) >=
precedence(infix[i])) {
            char *op = popOp(&stack);
            postfix[k++] = op[0];
            postfix[k++] = ' ';
        }
```

```
            char op[2] = {infix[i], '\0'};
            pushOp(&stack, op);
        }
        else if (infix[i] != ' ') {
            printf("Invalid Character %c\n", infix[i]);
            exit(EXIT_FAILURE);
        }
        i++;
    }
    while (!isEmptyOp(&stack)) {
        if(strcmp(peekOp(&stack), "(") == 0) {
            printf("Invalid Expression\n");
            exit(EXIT_FAILURE);
        }
        char *op = popOp(&stack);
        postfix[k++] = op[0];
        postfix[k++] = ' ';
    }
    postfix[k] = '\0';
}

// Evaluate Postfix Expression
int evaluatePostfix(const char *postfix) {
    Stack stack;
    initStack(&stack);
    int i = 0;
    while (postfix[i]) {
        if (isdigit(postfix[i])) {
```

```
    int num = 0;
    while (isdigit(postfix[i])) {
        num = num * 10 + (postfix[i] - '0');
        i++;
    }
    push(&stack, num);
}
else if (postfix[i] == ' ') {
    i++;
    continue;
}
else if (isOperator(postfix[i])) {
    int val2 = pop(&stack);
    int val1 = pop(&stack);
    switch (postfix[i]) {
        case '+': push(&stack, val1 + val2); break;
        case '-': push(&stack, val1 - val2); break;
        case '*': push(&stack, val1 * val2); break;
        case '/':
            if (val2 == 0) {
                printf("Division by Zero Error\n");
                exit(EXIT_FAILURE);
            }
            push(&stack, val1 / val2); break;
        case '^': push(&stack, (int)pow(val1, val2)); break;
    }
    i++;
}
```

```
        else {
            printf("Invalid Character %c in postfix\n", postfix[i]);
            exit(EXIT_FAILURE);
        }
    }
    return pop(&stack);
}

// Main Program
int main() {
    char infix[MAXSIZE], postfix[MAXSIZE];
    printf("Enter an arithmetic expression: ");
    fgets(infix, MAXSIZE, stdin);
    infixToPostfix(infix, postfix);
    printf("Postfix Expression: %s\n", postfix);
    int result = evaluatePostfix(postfix);
    printf("Result: %d\n", result);
    return 0;
}
```

Results and Discussion

The program reads a valid arithmetic expression from the user, converts it into postfix notation, and evaluates it. The stack-based approach ensures the correct order of operations according to operator precedence. Sample test expressions include complex calculations with multiple operator precedence levels, parentheses, and power operations. Invalid expressions or division by zero cases are handled with clear error messages.

Conclusion and Future Work

This project successfully demonstrates the implementation of expression evaluation using stacks in C. It highlights key programming concepts such as parsing, data structures, and algorithm design for arithmetic evaluation. Future enhancements could include support for floating-point numbers, unary operators, functions (sin, cos), and enhanced error recovery mechanisms.

References

1.Data Structures and Algorithms in C, Adam Drozdek

2.The C Programming Language, Kernighan & Ritchie

3.Online tutorials on stack implementation and expression evaluation