A Report

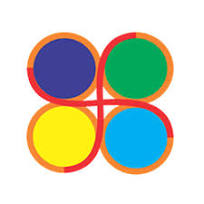
on

Micro Project

**Subject: Data Structures and Algorithms**

**Subject Code: 4330704**

**Branch: Computer Engineering**



**Government Polytechnic, Ahmedabad**

**Project Team:**

**• Khush Panchal (226170307075)**

**• Kavan Bhavsar (226170307015)**

**• Bhavya Bagadia (226170307004)**

**Project Title: Expression Converter**

Description: The Expression Converter project is designed to offer a versatile solution for converting infix expressions to postfix and prefix notations, as well as evaluating postfix or prefix expressions. This report provides an overview of the project's components, functionality, and user interface.

Project Components:

1. Stack Implementation:

• The project utilizes a stack data structure encapsulated within a structure named "Stack."

• Essential stack operations such as initialize, isEmpty, push, pop, and peek are implemented.

• The stack is a fundamental component for managing infix to postfix conversion and expression evaluation.

2. Infix to Postfix Conversion:

• The infix too postfix function transforms an infix expression into a postfix expression.

• It adheres to the shunting-yard algorithm, considering operator precedence and associativity.

• The stack plays a critical role in handling operators and parentheses during the conversion process.

3. Infix to Prefix Conversion:

• The project features a function for converting infix expressions to prefix notation.

• This process involves the reversal of the infix expression, subsequent infix to postfix conversion, and final reversal of the postfix expression to obtain the desired prefix expression.

4. Evaluation of Postfix or Prefix Expressions:

• The project supports the evaluation of both postfix and prefix expressions.

• A dedicated stack (evalst) manages operands and executes arithmetic operations based on encountered operators.

• The evaluation process incorporates error handling for scenarios such as division by zero and insufficient operands.

5. User Interface:

• The main function serves as the user interface, offering options for various expression conversions and evaluations.

• Options include converting infix to postfix, infix to prefix, evaluating postfix or prefix expressions, and exiting the program.

• The console-based interface is designed for simplicity and user-friendliness.

6. User Feedback and Error Handling:

• The project ensures user feedback and informative error messages in cases of input errors, mismatched parentheses, or other issues.

• Detailed error messages are displayed on the console, aiding users in identifying and rectifying expression-related concerns.

7. Visual Appeal:

• The project incorporates a banner function that showcases an ASCII art banner, enhancing the visual appeal of the console interface and providing a distinctive project identity.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include <windows.h>

#define MAX\_SIZE 50

struct Stack {

int top;

float items[MAX\_SIZE];

};

void initialize(struct Stack \*s) {

s->top = -1;

}

int isEmpty(struct Stack \*s) {

return s->top == -1;

}

void push(struct Stack \*s, float value) {

if (s->top == MAX\_SIZE - 1) {

printf("Internal error ocuured\n");

exit(EXIT\_FAILURE);

}

s->items[++s->top] = value;

}

float pop(struct Stack \*s) {

if (s->top==-1) {

printf("Internal error ocuured\n");

exit(EXIT\_FAILURE);

}

return s->items[s->top--];

}

float peek(struct Stack \*s) {

if (isEmpty(s)) {

printf("Internal error ocuured\n");

exit(EXIT\_FAILURE);

}

return s->items[s->top];

}

int isOperand(char ch) {

return isdigit(ch) || ch == '.';

}

int isOperator(char ch) {

return ch == '+' || ch == '-' || ch == '\*' || ch == '/';

}

int precedence(char op) {

if (op == '+' || op == '-')

return 1;

else if (op == '\*' || op == '/')

return 2;

return 0;

}

void infixToPostfix(char infix[], char postfix[]) {

struct Stack stack;

initialize(&stack);

int i = 0, j = 0;

while (infix[i] != '\0') {

if (isdigit(infix[i]) || infix[i] == '.') {

postfix[j++] = infix[i++];

} else if (infix[i] == '(') {

push(&stack, infix[i++]);

} else if (infix[i] == ')') {

while (!isEmpty(&stack) && peek(&stack) != '(') {

postfix[j++] = pop(&stack);

}

if (isEmpty(&stack)) {

fprintf(stderr, "Error: Mismatched parentheses\n");

return;

}

pop(&stack);

i++;

} else if (isOperator(infix[i])) {

while (!isEmpty(&stack) && precedence(peek(&stack)) >= precedence(infix[i])) {

postfix[j++] = pop(&stack);

}

push(&stack, infix[i++]);

} else {

fprintf(stderr, "Error: Invalid character '%c' in the infix expression\n", infix[i]);

return;

}

}

while (!isEmpty(&stack)) {

postfix[j++] = pop(&stack);

}

postfix[j] = '\0';

while (!isEmpty(&stack)) {

if (peek(&stack) == '(' || peek(&stack) == ')') {

printf(stderr, "Error: Mismatched parentheses\n");

return;

}

pop(&stack);

}

}

void banner() {

printf(" \_\_\_ \_\_ \_\_ \_\_\_ \_\_ \_\_ \_\_ \n");

printf("|\_\_ \\\_/ |\_\_) |\_\_) |\_\_ /\_\_` /\_\_` | / \\ |\\ |\n");

printf("|\_\_\_ / \\ | | \\ |\_\_\_ .\_\_/ .\_\_/ | \\\_\_/ | \\|\n");

printf(" \_\_ \_\_ \_\_\_ \_\_ \_\_\_ \_\_\_ \_\_ \n");

printf(" / ` / \\ |\\ | \\ / |\_\_ |\_\_) | |\_\_ |\_\_) \n");

printf(" \\\_\_, \\\_\_/ | \\| \\/ |\_\_\_ | \\ | |\_\_\_ | \\ \n\n");

}

void reverseString(char str[]) {

int length = strlen(str);

int start = 0;

int end = length - 1;

while (start < end) {

char temp = str[start];

str[start] = str[end];

str[end] = temp;

start++;

end--;

}

}

void revstr(char \*str1) {

int i, len, temp;

len = strlen(str1);

for (i=0;i<len/2;i++) {

temp = str1[i];

str1[i] = str1[len - i - 1];

str1[len - i - 1] = temp;

}

for (i = 0; i < len; i++) {

if (str1[i] == '(') {

str1[i] = ')';

} else if (str1[i] == ')') {

str1[i] = '(';

} else {

continue;

}

}

}

struct evalst {

int top;

unsigned capacity;

float\* array;

};

void initEval(struct evalst\* st, unsigned capacity) {

st->top = -1;

st->capacity = capacity;

st->array = (float\*)malloc(st->capacity \* sizeof(float));

}

int isEvalOperand(char ch) {

return isdigit(ch) || ch == '.';

}

int isEvalOperator(char ch) {

return ch == '+' || ch == '-' || ch == '\*' || ch == '/';

}

float evalPop(struct evalst\* st) {

if (st->top == -1) {

printf("Error: st underflow\n");

exit(EXIT\_FAILURE);

}

return st->array[st->top--];

}

void evalPush(struct evalst\* st, float item) {

if (st->top == st->capacity - 1) {

printf("Error: st overflow\n");

exit(EXIT\_FAILURE);

}

st->array[++st->top] = item;

}

float evaluatePostfix(char expr[]) {

struct evalst st;

initEval(&st, strlen(expr));

int i = 0;

while (expr[i] != '\0') {

if (isEvalOperand(expr[i])) {

float operand = 0;

while (isEvalOperand(expr[i])) {

operand = operand \* 10 + (expr[i] - '0');

i++;

}

if (expr[i] == '.') {

i++; // Skip the dot

float decimalPlace = 0.1;

while (isdigit(expr[i])) {

operand += (expr[i] - '0') \* decimalPlace;

decimalPlace \*= 0.1;

i++;

}

}

evalPush(&st, operand);

} else if (isEvalOperator(expr[i])) {

if (st.top < 1) {

printf("Error: Insufficient operands for operator %c\n", expr[i]);

exit(EXIT\_FAILURE);

}

float operand2 = evalPop(&st);

float operand1 = evalPop(&st);

switch (expr[i]) {

case '+':

evalPush(&st, operand1 + operand2);

break;

case '-':

evalPush(&st, operand1 - operand2);

break;

case '\*':

evalPush(&st, operand1 \* operand2);

break;

case '/':

if (operand2 == 0) {

printf("Error: Division by zero\n");

exit(EXIT\_FAILURE);

}

evalPush(&st, operand1 / operand2);

break;

}

i++;

} else {

i++;

}

}

if (st.top != 0) {

printf("Error: Invalid postfix expression\n");

exit(EXIT\_FAILURE);

}

return evalPop(&st);

}

int main() {

char infix[MAX\_SIZE], postfix[MAX\_SIZE], expression[MAX\_SIZE];

char ch;

int len;

do {

system("cls");

banner();

printf("1) Infix to Postfix\n");

printf("2) Infix to Prefix\n");

printf("3) Evaluate Prefix or Postfix expression\n");

printf("4) Exit\n");

printf("Enter your choice: ");

scanf(" %c", &ch);

getchar();

switch (ch) {

case '1':

len = 0;

printf("\n\nEnter an infix expression: ");

fgets(infix, MAX\_SIZE, stdin);

len = strlen(infix);

if (infix[len - 1] == '\n') {

infix[len - 1] = '\0';

}

infixToPostfix(infix, postfix);

printf("\n\nPostfix expression: %s\n\n\n\n", postfix);

system("pause");

break;

case '2':

len = 0;

printf("\n\nEnter an infix expression: ");

fgets(infix, MAX\_SIZE, stdin);

len = strlen(infix);

if (infix[len - 1] == '\n') {

infix[len - 1] = '\0';

}

revstr(infix);

infixToPostfix(infix, postfix);

revstr(postfix);

printf("\n\nPrefix expression: %s\n\n\n\n", postfix);

system("pause");

break;

case '4':

printf("\n\nExiting...\n\n\n\n");

break;

default:

printf("\n\nInvalid Choice chose only from 1-4...\n\n\n\n");

system("pause");

break;

case '3':

printf("\n\nEnter postfix or prefix expression: ");

fgets(expression, MAX\_SIZE, stdin);

if (expression[0]=='+' || expression[0]=='-' || expression[0]=='\*' || expression[0]=='/'){

revstr(expression);

}

float result = evaluatePostfix(expression);

printf("\n\nResult: %.2f\n\n\n\n", result);

system("pause");

break;

}

} while (ch != '4');

return 0;

}

**Output:**

****

**Conclusion**:

The Expression Converter project effectively implements core functionality for expression conversion and evaluation. It adheres to established algorithms and best practices for error handling, ensuring a robust and user-friendly experience.