**Practical No. 6**

**Aim:** Write a program to convert an infix expression to a postfix expression using stacks.

**Theory**

Infix expressions place operators between operands (e.g., A + B). Postfix (Reverse Polish Notation) places operators after operands (e.g., A B +). Converting to postfix is useful because postfix can be evaluated easily using a stack without considering precedence or parentheses during evaluation.

The conversion uses a stack to temporarily hold operators and ensures output respects operator **precedence** and **associativity**. Typical precedence: ^ > \* / > + -. Note: ^ (exponent) is usually **right-associative**, while the others are **left-associative**.

This practical assumes operands are single characters (letters or digits) and input has no spaces. It can be extended later to handle multi-character operands or spaced tokens.

**Algorithm (step-by-step)**

1. Initialize an empty stack for operators and an empty output string for postfix.
2. Scan the infix expression left to right:
   * If the symbol is an operand (letter or digit), append it to the postfix output.
   * If it is '(', push it onto the stack.
   * If it is ')', pop operators from the stack to output until '(' is popped.
   * If it is an operator op:
     + While the top of the stack contains an operator with **higher precedence**, or with **equal precedence and left associativity**, pop it to output.
     + Push op onto the stack.
3. After scanning, pop any remaining operators from the stack to the output.
4. The output string is the postfix expression.

**Program (C)**

#include <stdio.h> // printf, scanf

#include <string.h> // strlen, strcat

#include <ctype.h> // isalpha, isdigit

#define MAX 100 // Maximum size for expression and stack

// Function to get precedence of operators

int precedence(char op) { // returns an integer: higher => higher precedence

if (op == '^') return 3; // '^' has highest precedence here

if (op == '\*' || op == '/') return 2; // '\*' and '/' next

if (op == '+' || op == '-') return 1; // '+' and '-' lowest among these

return 0; // non-operators give 0

}

// Function to check right associativity (true for '^')

int isRightAssociative(char op) { // returns 1 if operator is right-associative

if (op == '^') return 1; // '^' is right-associative

return 0; // others considered left-associative

}

// Function to check if character is a supported operator

int isOperator(char ch) { // returns 1 if ch is one of supported operators

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '^');

}

int main() {

char infix[MAX]; // buffer to store input infix expression

char postfix[MAX] = ""; // buffer to build postfix expression (start empty)

char stack[MAX]; // stack to hold operators and parentheses

int top = -1; // top = -1 means stack is empty

int i, k = 0; // i: loop index for infix, k: index for postfix

// Prompt and read infix expression (no spaces)

printf("Enter infix expression (no spaces, operands single-char): ");

scanf("%s", infix); // read expression as string

// Process each character of infix

for (i = 0; i < (int)strlen(infix); i++) {

char ch = infix[i]; // current character

if (isalpha(ch) || isdigit(ch)) { // if operand (letter or digit)

postfix[k++] = ch; // append operand to postfix

}

else if (ch == '(') { // if opening parenthesis

stack[++top] = ch; // push '(' onto stack

}

else if (ch == ')') { // if closing parenthesis

// pop until matching '('

while (top != -1 && stack[top] != '(') {

postfix[k++] = stack[top--]; // pop operator to postfix

}

if (top != -1 && stack[top] == '(') // if '(' found

top--; // pop '(' and discard it

else { // mismatched parenthesis

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

return 1; // exit with error

}

}

else if (isOperator(ch)) { // if current char is an operator

// While there's an operator on stack with higher precedence

// or same precedence and left-associative, pop it to output

while (top != -1 && isOperator(stack[top]) &&

( (isRightAssociative(ch) == 0 && precedence(ch) <= precedence(stack[top])) ||

(isRightAssociative(ch) == 1 && precedence(ch) < precedence(stack[top])) )) {

postfix[k++] = stack[top--]; // pop to postfix

}

stack[++top] = ch; // push current operator on stack

}

else { // invalid character encountered

printf("Error: Invalid character '%c' in expression\n", ch);

return 1; // exit with error

}

}

// Pop any remaining operators from stack to postfix

while (top != -1) {

if (stack[top] == '(' || stack[top] == ')') { // if parentheses remain, mismatch

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

return 1;

}

postfix[k++] = stack[top--]; // pop operator to postfix

}

postfix[k] = '\0'; // null-terminate postfix string

// Print result

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

return 0; // successful termination

}

**Output**

**Example 1 - operator precedence**

Enter infix expression (no spaces, operands single-char): A+B\*C

Postfix expression: ABC\*+

**Example 2 - parentheses**

Enter infix expression (no spaces, operands single-char): (A+B)\*C

Postfix expression: AB+C\*

**Example 3 - right associativity of exponent**

Enter infix expression (no spaces, operands single-char): A^B^C

Postfix expression: ABC^^

**Example 4 - digits as operands**

Enter infix expression (no spaces, operands single-char): (1+23)\*4

Error: Invalid character '2' in expression

Note: this program assumes single-character operands. To handle 23 or multi-digit numbers, use a tokenizer and space-separated input.

**Conclusion**

* Implemented a stack-based converter from infix to postfix that respects operator precedence, associativity, and parentheses.
* The program is intentionally simple (single-character operands, no spaces) so the core logic is clear - it can be extended later to handle multi-digit operands or whitespace-separated tokens.