## Lab program-2

WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operands + (plus) - (minus) * (multiply) and / (divide)

Write the pseudocode/program in your observation and get it corrected from your batch faculty. Then you start executing the code.

| Function required |
| --- |
| push() |
| pop() |
| peek() |
| precedance() |
| associativity() |
| infix to postfix (infix[] postfix[]) |

### Pseudocode

### Algorithm infix to postfix

1. Input an expression from user.
2. check whether it is a valid expression or not using balanced paranthesis.
3. Declare two arrays to store the entered expression or not using postfix expression.
4. Run a loop to check the elements of entered expression
5. If operands arrive, just print (store it in postfix array)
   - If stack is empty or contains open(left) paranthesis on top, push the incoming operator onto the stack.
   - If incoming symbol is '(', push it onto the stack.
   - If incoming symbol is ')', pop the stack and store the operators into postfix until left paranthesis is found.
   - If incoming ~~symbol~~ symbol has high precedance than top of the stack, push it on to the stack.
   - If incoming symbol has lower precedance than top of the stack, pop and print (store) the top. Then rest the incoming operator against the new top at the stack.
   - If incoming symbol has equal precedence with top of the stack. Use associativity rule.
   - At the end of the expr, pop & print (store) all the operators of stack.

Code

INFIX TO POSTFIX CONVERSION

```
#include<stdio.h>
#include< ctype.h>
#include <string.h>
#define MAX 100
char stack[MAX];
int top= -1;
// push function
void push (char c){
    if (top==MAX-1){
        Printf("stack overflow\n");
        return;
    }
    stack[++top] = c;
}
// pop function
char pop(){
    if (top==-1){
        printf ("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}
// peek function
char peek(){
    if (top==-1) return -1;
    return stack[top];
}
// function to return precedence of operators int
  precedance (char op){
    switch (op){
        case '+':
        case '-':
                return 1;
```

```
    case '*':
    case '/':
        return 2;
    case '^':
        return 3;  // Highest precedence.
    case '(':
        return 0;
  }
  return -1;
}

// function to return associativity
// 0 = Left-to-Right, 1 = Right-to-Left
int associativity (char op) {
    if (op == '^')
      return 1; // Right-to-left
    return 0; // +, -, */ -> Left to Right
}

// function to convert infix to postfix
void InfixTopostfix (char infix[], char postfix[]){
    int i, k=0;
    char c;
    for (i=0; infix[i] != '\0'; i++){
      c = infix[i];
      if (isalnum(c)) {
        // operand -> directly to postfix
        postfix [k++] = c;
      }
    else if (c == '(')){
      push (c);
    }
  else if (c==')') {
    while (peek() != '('){
        postfix [k++] = pop();
    }
```

```
        pop();  // discard '('
    }
    else {
    // operator
    while (top != -1 &&
        ((precedence(peek()) > precedence(c)) ||
        (precedence(peek()) == precedence(c) &&
        associativity(c) == 0))) { // L- to -R
        postfix[k++] = pop();
        }
        push(c);
    }
}
// pop remaining operators
while (top != -1){
    postfix[k++] = pop();
} postfix[k] = '\0';
}
int main(){
    char infix[MAX], postfix[MAX];
    printf("Enter a valid paranthesised infix expression: ");
    scanf("%s", infix);
    infix to postfix(infix, postfix);
    printf("Postfix Expression: %s\n", postfix);
    return 0;
}
```

o/p

Enter a valid paranthesized infix expression (A+B)*C

postfix ~~expres~~ Expression: AB+C*

o/p Enter a valid paranthesized infix expression;

: A*B+C*D-E

postfix Expression: AB*CD*+E-

Enter a valid paranthesized infix expression:

(A+(B*C-(D/E^F)*G)*H)

postfix Expression: ABC*DEF^/G*-H*+

MG
6/10/25.