Name – Prerna Sunil Jadhav **SAP ID - 60004220127**

Experiment No – 01

AIM: : Implementation of Infix to postfix expression-Transformation and its evaluation program.

Transformation

Theory:

Infix expressions are readable and solvable by humans. The computer cannot differentiate the operators and parenthesis easily, that's why postfix conversion is needed.

To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

Symbol	Postfix String	Stack
A	A	
+	A	+
B	AB	+
*	AB	+*
C	ABC	+*
/	ABC*	+/
D	ABC*D	+/
-	ABC*D/+	-
E	ABC*D/+E	-
	ABC*D/+E-	

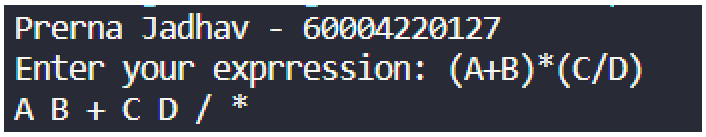
Algorithm:

```
1.  Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.
2.  Push "("onto Stack, and add ")" to the end of X.
3.  Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
4.  If an operand is encountered, add it to Y.
5.  If a left parenthesis is encountered, push it onto Stack.
6.  If an operator is encountered ,then:
7.  Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
8.  Add operator to Stack.
[End of If]
9.  If a right parenthesis is encountered ,then:
10. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
11. Remove the left Parenthesis.
[End of If]
[End of If]
12. END.
```

PROGRAM:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
// program to convert infix to postfix
int top = -1;
char stack[100];

void push(char a)
{
    top++;
    stack[top] = a;
}

char pop()
{
    if (top == -1)
        return -1;
    else
        return stack[top--];
}

int prior(char a)
{
    if (a == '(')
        return 0;
    if (a == '+' || a == '-')
        return 1;
    if (a == '*' || a == '/')
        return 2;
    if (a == '^')
        return 3;
    return 0;
}

int main()
{
    printf("Prerna Jadhav - 60004220127\n");
    char expr[100];
    char x;
    int i;
    printf("Enter your exprression: ");
    scanf("%s", expr);
    for (i = 0; i < strlen(expr); i++)
    {
        if (isalnum(expr[i]))
        {
            printf("%c ", expr[i]);
        }
        else
        {
            if (expr[i] == '(')
                push(expr[i]);
            else if (expr[i] == ')')
            {
                while ((x = pop()) != '(')
                {
                    printf("%c ", x);
                }
            }
            else
            {
                while (prior(expr[i]) <= prior(stack[top]))
                    printf("%c ", pop());
                push(expr[i]);
            }
        }
    }
    while (top != -1)
    {
        printf("%c ", pop());
    }
    return 0;
}
```

OUTPUT:



```
Prerna Jadhav - 60004220127
Enter your exprression: (A+B)*(C/D)
A B + C D / *
```

Evaluation

Theory:

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis is not required in postfix.

Serial No.	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4, 5	
3.	6	Push	4, 5, 6	
4.	*	Pop (2 elements) & Evaluate	4	
5.		Push result (30)	4, 30	5 * 6 = 30
6.	+	Pop (2 elements) & Evaluate	Empty	
7.		Push result (34)	34	4 + 30 = 34
8.		No more elements (pop)	Empty	**34 (Result)**

Follow the steps mentioned below to evaluate postfix expression using stack:

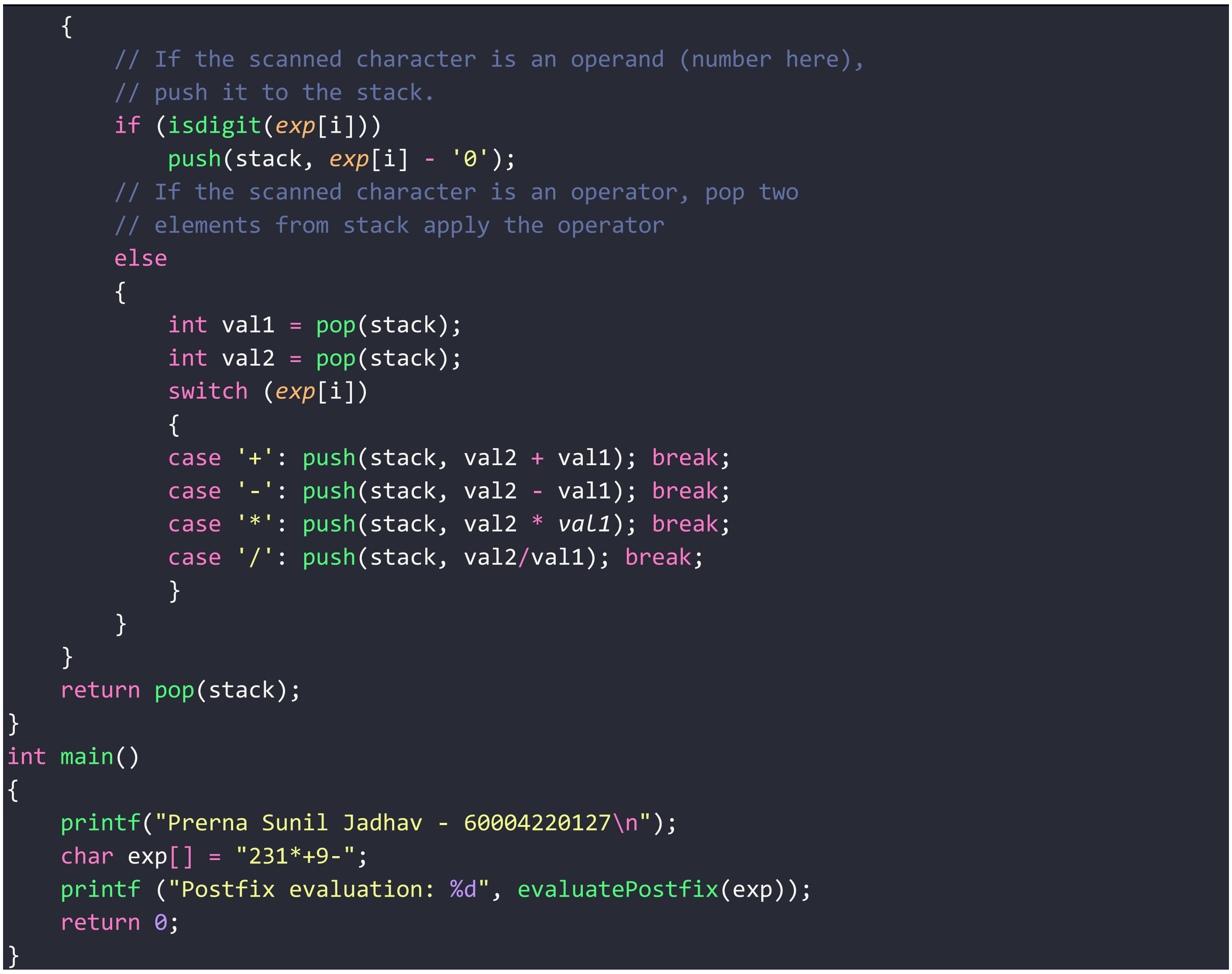
- Create a stack to store operands (or values).
- Scan the given expression from left to right and do the following for every scanned element.
- If the element is a number, push it into the stack
- If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack
- When the expression is ended, the number in the stack is the final answer

Algorithm:

```
1.  Create a stack that holds integer type data to store the operands of the given postfix
expression. Let it be st.
2.  Iterate over the string from left to right and do the following -
    •   If the current element is an operand, push it into the stack.
    •   Otherwise, if the current element is an operator (say /)do the following -
    •   Pop an element from st, let it be op2.
    •   Pop another element from st, let it be op1.
    •   Computer the result of op1 / op2, and push it into the stack. Note the order i.e.
op1 / op2 should not be changed otherwise it will affect the final result in some cases.
3.  At last, st will consist of a single element i.e. the result after evaluating the
postfix expression.
```

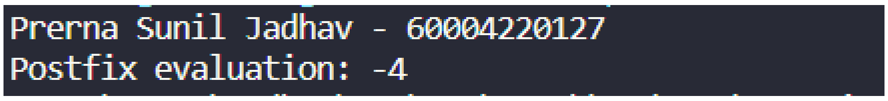
Program:

```c
// C program to evaluate value of a postfix expression
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};
// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}
int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}
char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}
char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}
void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}
// The main function that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;
    // See if stack was created successfully
    if (!stack) return -1;
    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
```



```
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');
        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
            case '+': push(stack, val2 + val1); break;
            case '-': push(stack, val2 - val1); break;
            case '*': push(stack, val2 * val1); break;
            case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}
int main()
{
    printf("Prerna Sunil Jadhav - 60004220127\n");
    char exp[] = "231*+9-";
    printf ("Postfix evaluation: %d", evaluatePostfix(exp));
    return 0;
}
```

Output:



```
Prerna Sunil Jadhav - 60004220127
Postfix evaluation: -4
```

Conclusion:

Infix notation is the notation in which operators come between the required operands.

Postfix notation is the type of notation in which operator comes after the operand.

Infix expression can be converted to postfix expression using stack.