**Expression Evaluation**

* An algebraic expression is a legal combination of operands and operators.
* Operand is a quantity (data) on which an operation is performed.
* Operand may be a variable like a, b, c or constants like 1, 2, 3….
* Operator is a symbol.
* Example of an expression x+y\*z

**Notations:** We can represent an expression in different notations.

**Infix:** Operator surrounded by operands.

X+Y

**Prefix:** Operator proceeded by operands.

+XY

**Postfix:** Operator followed by operands.

XY+

**Notes:**

* Infix notation is seems to be simple but difficult to evaluate.
* We need to consider the priority of operators and associativity of operators.
* Expression 2+3\*5 result depends on priority.
* No need to consider these things in the evaluation of Prefix or Postfix expressions.

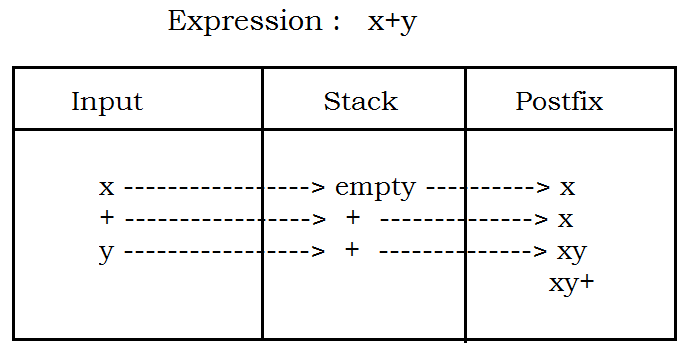
**As there are 3 notations, we have total 6 conversions.**

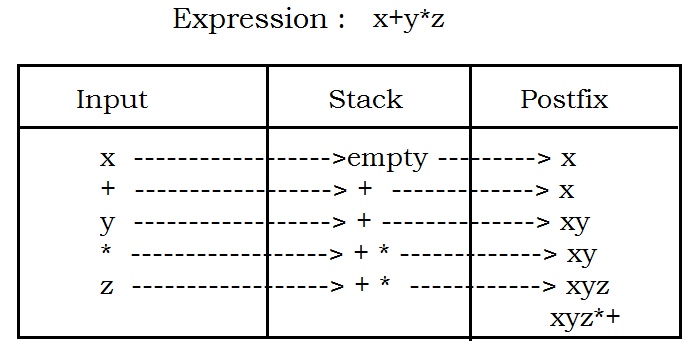
1. Infix -> Postfix
2. Infix -> Prefix
3. Prefix -> Infix
4. Prefix -> Postfix
5. Postfix -> Infix
6. Postfix -> Prefix

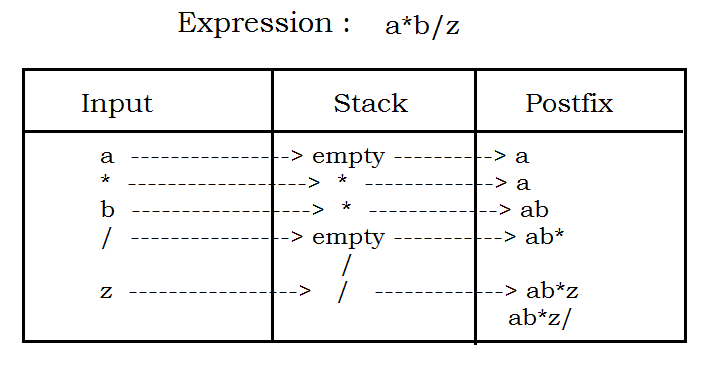
* First 2 conversions can be performed using STACK. Hence we called these conversions are applications of STACK.
* Remaining conversions use BST and STACK.

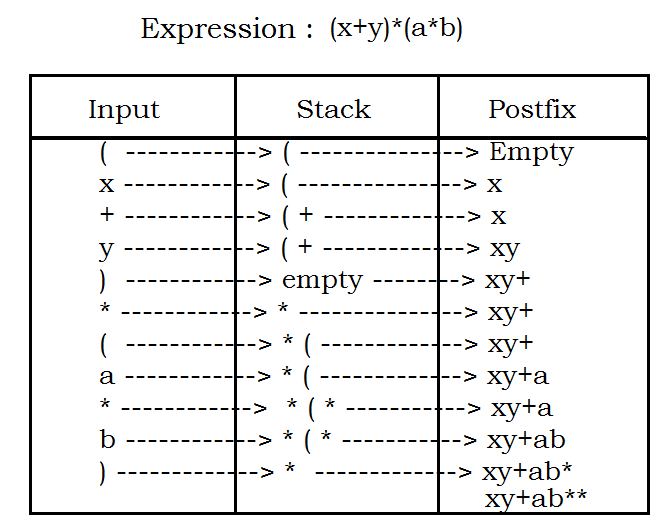
**Algorithm to convert Infix to Postfix using STACK:**

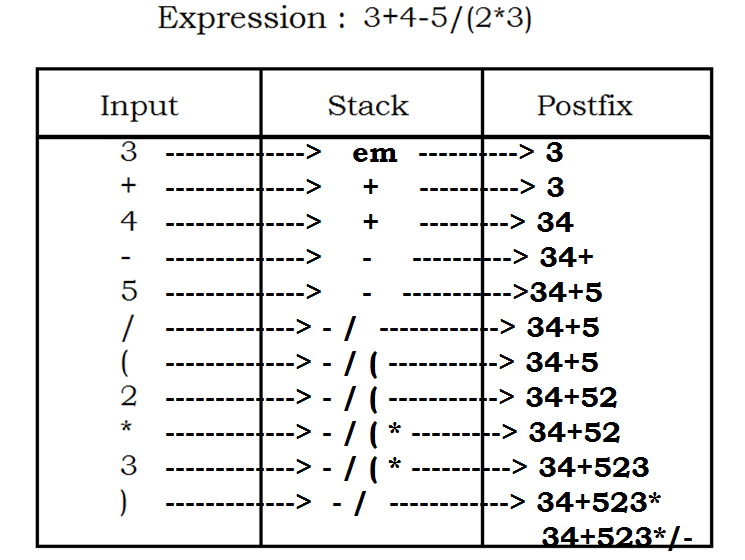
1. Read next element in the input expression.
2. If it is an operand, output it.
3. If it is an opening parenthesis, push on to the Stack.
4. If it is an operator, then
   1. If the Stack is empty, push operator on Stack
   2. If the Top of stack is Opening parenthesis, push operator on to the Stack.
   3. If the operator has higher priority than top of the Stack, push operator on to the Stack else Pop operator from the Stack and output it, repeat Step 4.
5. If the is a closing parenthesis, pop operators from stack and output them until opening parenthesis is encountered. POP and discard opening parenthesis.
6. If there is more input Go to step 1.
7. If no more input, unstuck all operators from stack and output.











**Expression: x\*y + (z+(l-m)\*n)**

**x ----------------> empty ------------>x**

**\* ----------------> \* ------------> x**

**y ----------------> \* ------------> xy**

**+ ----------------> + ------------> xy\***

**( ----------------> +( ------------> xy\***

**z ----------------> +( ------------> xy\*z**

**+ ----------------> +(+ ------------> xy\*z**

**( ----------------> +(+( ------------> xy\*z**

**l ----------------> +(+( ------------> xy\*zl**

**- ----------------> +(+(- ------------> xy\*zl**

**m ----------------> +(+(- ------------> xy\*zlm**

**) ----------------> +(+ ------------> xy\*zlm-**

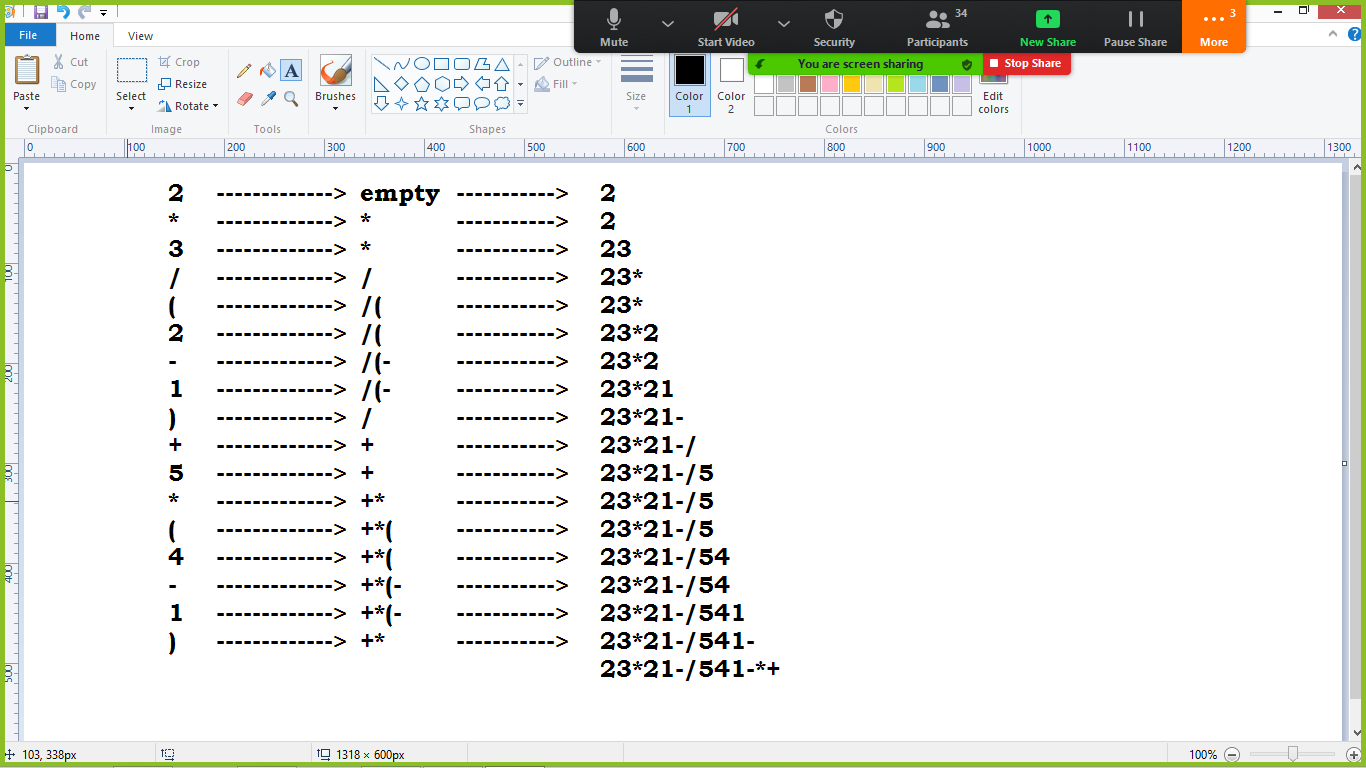
**\* ----------------> +(+\* ------------> xy\*zlm-**

**n ----------------> +(+\* ------------> xy\*zlm-n**

**) ----------------> + ------------> xy\*zlm-n\*+**

**------------> xy\*zlm-n\*++**

**Expression: 2\*3/(2-1) + 5\*(4-1)**



**Program implementation:**

#include<stdio.h>

#include<string.h>

#define LP 10

#define RP 20

#define OPERATOR 30

#define OPERAND 40

#define LPP 0

#define AP 1

#define SP 1

#define MP 2

#define DP 2

#define REMP 2

#define NONE 9

int top;

char infix[50] , stack[40] , postfix[50];

void toPostfix();

int gettype(char);

void push(char);

char pop(void);

int getPrecedence(char);

int main()

{

char ch;

do

{

top=-1;

printf("\nEnter Infix expression : ");

gets(infix);

toPostfix();

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

printf("Do you want to continue(y/n) : ");

ch = getche();

}while(ch=='y');

return 0;

}

void toPostfix()

{

int i, p, l, type, prec;

char next;

i=0;

p=0;

l=strlen(infix);

while(i<l)

{

type = gettype(infix[i]);

switch(type)

{

case LP : push(infix[i]);

break ;

case RP : while((next = pop()) != '(')

{

postfix[p] = next;

++p;

}

break ;

case OPERAND : postfix[p] = infix[i];

++p;

break;

case OPERATOR : prec = getPrecedence(infix[i]);

while((top>-1)&&(prec<=getPrecedence(stack[top])))

{

postfix[p] = pop();

++p;

}

push(infix[i]);

break;

}

i++;

}

while(top>-1)

{

postfix[p++] = pop();

}

postfix[p] = '\0';

}

int gettype(char sym)

{

switch(sym)

{

case '(' : return LP ;

case ')' : return RP ;

case '+' :

case '-' :

case '\*' :

case '/' :

case '%' : return OPERATOR ;

default : return OPERAND ;

}

}

void push(char sym)

{

++top;

stack[top] = sym;

}

char pop(void)

{

char sym;

sym = stack[top];

--top;

return sym ;

}

int getPrecedence(char sym)

{

switch(sym)

{

case '(' : return LPP;

case '+' : return AP;

case '-' : return SP;

case '\*' : return MP;

case '/' : return DP;

case '%' : return REMP;

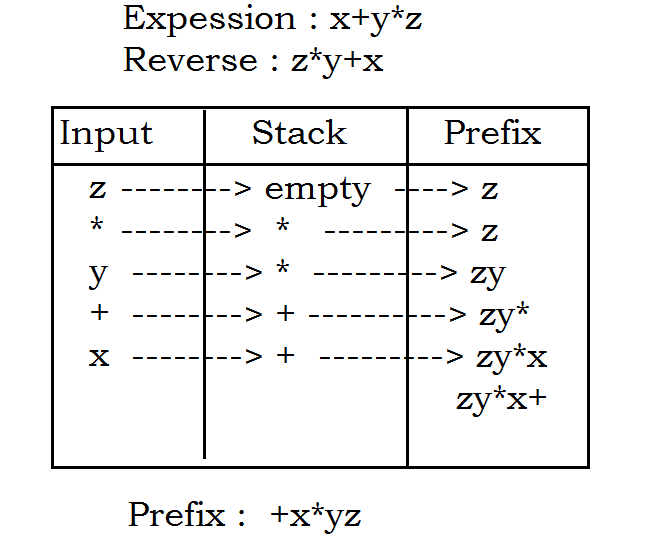
default : return NONE;

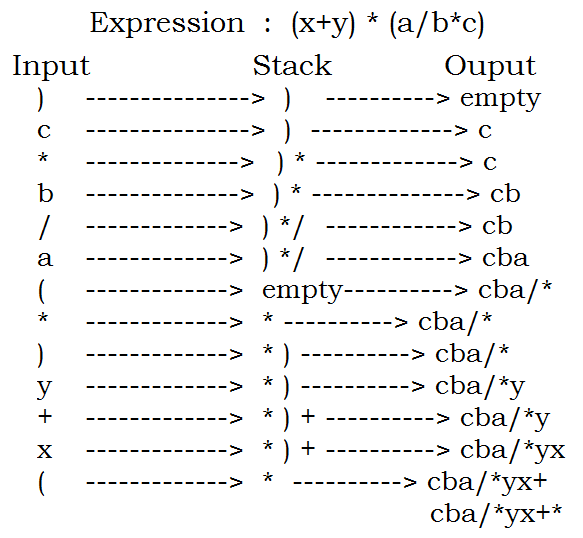
}

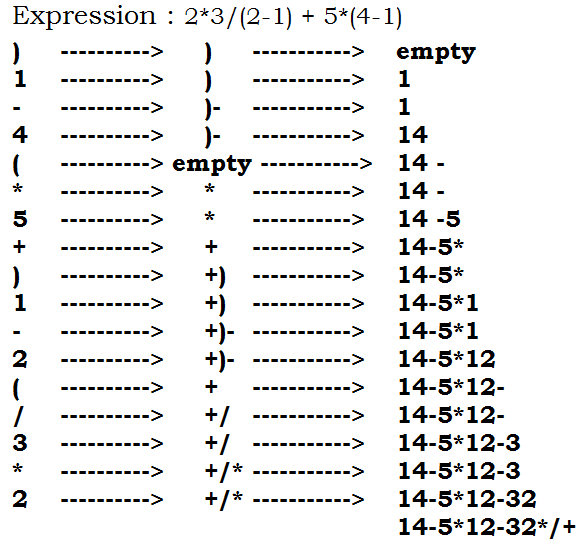
}

**Algorithm to convert Infix to Prefix using STACK:**

1. Reverse the expression string.
2. Read next element in the input expression.
3. If it is an operand, output it.
4. If it is a closing parenthesis, push on to the Stack.
5. If it is an operator, then
   1. If the Stack is empty, push operator on Stack
   2. If the Top of stack is closing parenthesis, push operator on to the Stack.
   3. If the operator has “same or higher priority” than top of the Stack, push operator on to the Stack else Pop operator from the Stack and output it, repeat Step 5.
6. If the is an Opening parenthesis, pop operators from stack and output them until closing parenthesis is encountered. POP and discard closing parenthesis.
7. If there is more input go to step 2.
8. If no more input, un stack all operators and output.
9. Reverse Output expression that is Prefix







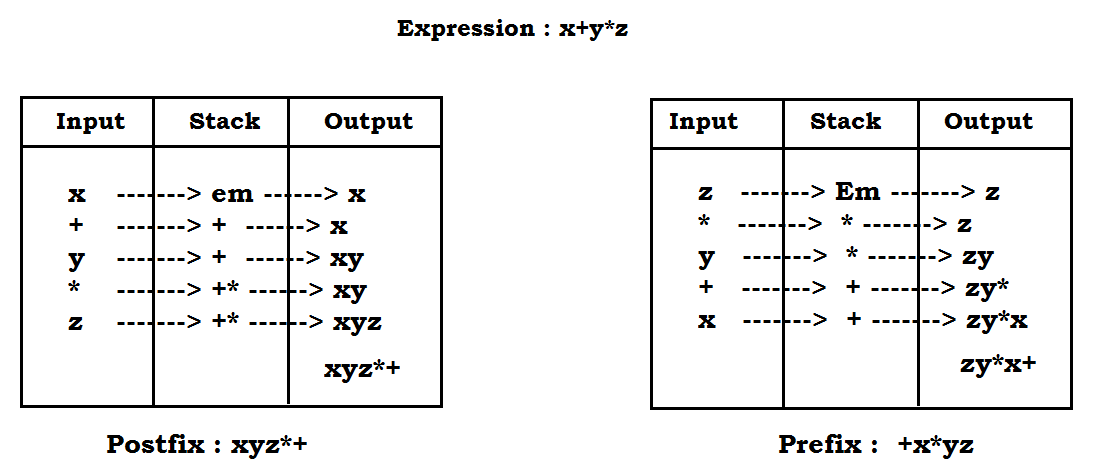
* We have converted 2 conversions using Stack.
* Remaining conversions we can perform using BST.

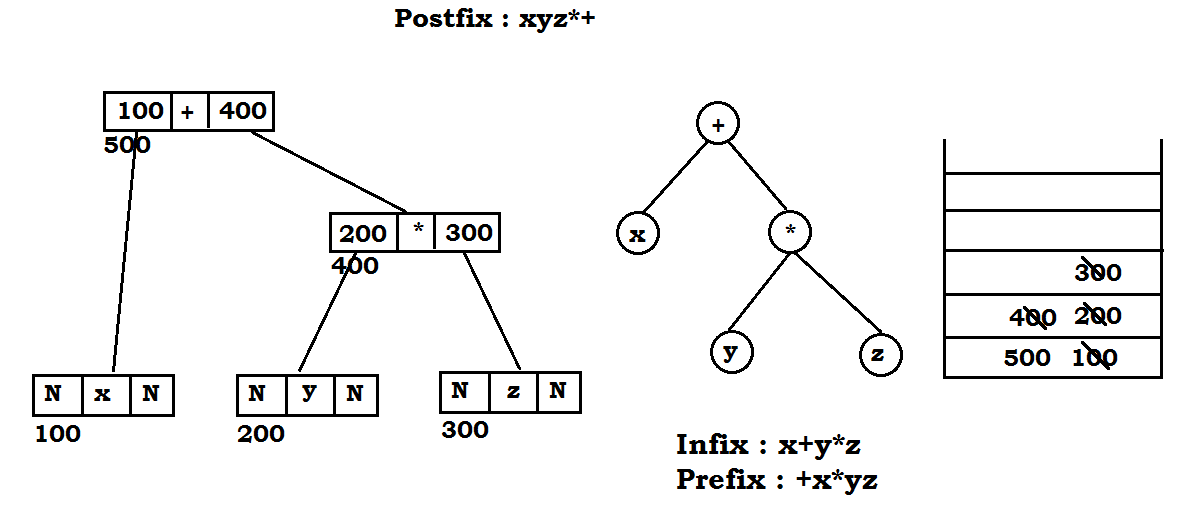
**Postfix expression conversion:**

* We need to construct the BST from Postfix expression
* In order traversal gives Infix notation
* Pre order traversal gives Prefix notation.

**Algorithm to construct Expression tree from Postfix:**

1. Examine the next element in the input
2. If it is operand then
   1. Create the leaf node(right and left child are null)
   2. Copy the operand in data part
   3. Push Node’s address on Stack.
3. If it is an operator then
   1. Create a node
   2. Copy the operator on data part
   3. POP address of node from Stack and assign to node->right
   4. POP address of node from Stack and assign to node->left
   5. PUSH node’s address on Stack.
4. If there is more input, go to step 1
5. If there is no more input, POP the address from Stack which is the address of the root node.





**Task:**

Expression: 2\*3/(2-1) + 5\*(4-1)

Construct Postfix notation

Construct Prefix notation

Construct BST tree from Postfix notation.

Pre-order traversal – Gives Prefix notation.

**Code implementation from Postfix -> Prefix:**

**Algorithm to construct Expression tree from Prefix:**

1. Reverse Prefix expression.
2. Examine the next element in the input
3. If it is operand then
   1. Create the leaf node(right and left child are null)
   2. Copy the operand in data part
   3. Push Node’s address on Stack.
4. If it is an operator then
   1. Create a node
   2. Copy the operator on data part
   3. POP address of node from Stack and assign to node->left
   4. POP address of node from Stack and assign to node->right
   5. PUSH node’s address on Stack.
5. If there is more input, go to step 2
6. If there is no more input, POP the address from Stack which is the address of the root node.

**Code implementation from Prefix -> Postfix:**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

struct node

{

char c;

struct node \*left;

struct node \*right;

};

struct node \*stk[20], \*temp, \*root;

char prefix[20],ch;

int top=-1, max=20, len;

void post(struct node\*);

void exptree();

void push(struct node\*);

struct node\* pop();

int main()

{

printf("Enter Prefix expression : ");

scanf("%s", prefix);

exptree();

printf("Postfix expression is : ");

post(root);

return 0;

}

void post(struct node \*p)

{

if(p != NULL)

{

post(p->left);

post(p->right);

printf("%c", p->c);

}

}

void exptree()

{

int i;

len = strlen(prefix);

i=len-1;

while(i>=0)

{

switch(prefix[i])

{

case '+' :

case '-' :

case '\*' :

case '/' :

case '%' : temp = (struct node\*)malloc(sizeof(struct node));

temp->c = prefix[i];

temp->left = pop();

temp->right = pop();

push(temp);

break;

default : temp = (struct node\*)malloc(sizeof(struct node));

temp->c = prefix[i];

temp->left = NULL;

temp->right = NULL;

push(temp);

break;

}

--i;

}

root = pop();

}

void push(struct node \*p)

{

if(top==max)

printf("Stack is Full \n");

else

stk[++top] = p;

}

struct node\* pop()

{

if(top==-1)

printf("Invalid expression \n");

else

return stk[top--];

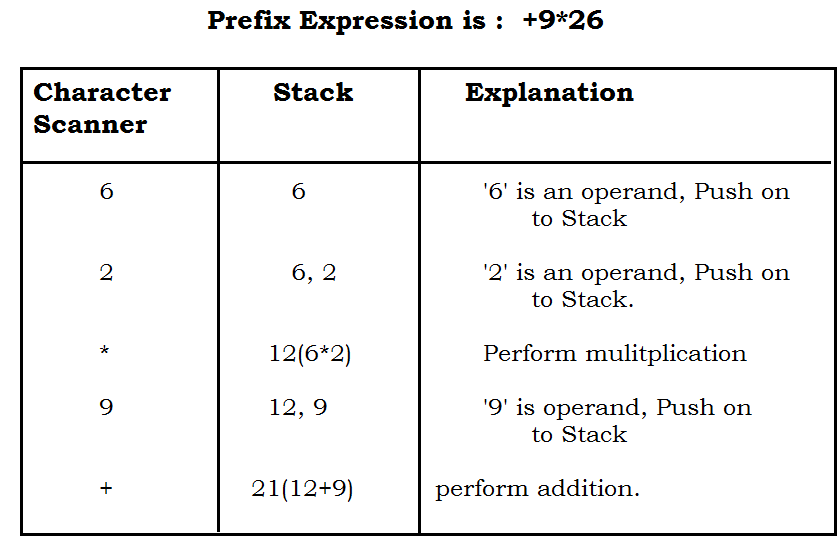
}

**Evaluation of Prefix Expression:**

* Prefix and Postfix expressions can be evaluated faster than infix expressions.
* No need to process any brackets or follow operator precedence rule.
* In Postfix or Prefix expressions, which ever the operator comes before will be evaluated first irrespective to priority.
* These expressions not having brackets to evaluate.

**Algorithm: To Evaluate Prefix(String)**

1. Put a pointer ‘p’ at the end of String(come in backward direction)
2. If character at ‘p’ is an operand, push it on to Stack.
3. If the character at ‘p’ is an operator, POP two elements from the Stack. Operate these elements according to the operator, and PUSH result back to Stack.
4. Decrement ‘p’ by 1 and go to step 2 as long as there are characters left in the expression.
5. The RESULT is stored at the top of the Stack, return it.
6. End.



#include<stdio.h>

#include<string.h>

int isOperand(char);

int evaluate(char[]);

void push(int);

int pop();

int stack[20];

int top=-1;

int main()

{

char pre[20];

int res;

printf("Enter Prefix expression : ");

gets(pre);

printf("Input expression : %s \n", pre);

res = evaluate(pre);

printf("Result is : %d \n", res);

return 0;

}

int evaluate(char exp[])

{

int j;

for(j=strlen(exp)-1 ; j>=0 ; j--)

{

if(isOperand(exp[j]))

{

push(exp[j]-'0');

}

else

{

int x,y;

x = pop();

y = pop();

switch(exp[j])

{

case '+' : push(x+y);

break;

case '-' : push(x-y);

break;

case '\*' : push(x\*y);

break;

case '/' : push(x/y);

break;

}

}

}

return pop();

}

int isOperand(char ch)

{

if(ch>=48 && ch<=57)

return 1;

else

return 0;

}

void push(int val)

{

stack[++top] = val;

}

int pop()

{

return stack[top--];

}