**Design and implement an algorithm for conversion of an expression from one form to another. Demonstrate its working with suitable inputs.**

**INFIX TO PREFIX CONVERSION**

**THEORY**

When we write mathematical expression in a program, we use infix expression. These expressions will be converted into equivalent machine instructions by compiler using stacks. Using stacks we can efficiently convert the expressions from infix to postfix, infix to prefix, postfix to prefix and postfix to infix.

**Representations of expressions:**

* Infix expression
* Postfix expression
* Prefix expression

**Infix expression**: In an expression if an operator is in between two operands, the expression is called an infix expression. Ex: a+ b

**Postfix expression:** In an expression, if an operator follows the two operands (i.e. operator comes after the two operands), the expression is called postfix expression. Ex: ab+

**Prefix expression:** In an expression, if an operator precedes two operands ( i.e. operator comes before the two operands), the expression is called prefix expression. Ex: +ab

The expression is evaluated according to the precedence of the operators.

Example: 6\*(2+3)-5 = 25

Brackets have the highest precedence and are evaluated first.

Multiplication and division has higher precedence over addition and subtraction.

**CODE WITH COMMENTS**

#include <stdio.h>

#include<stdio.h>

#include<string.h>

#include<limits.h>

#define max 100

int top=-1;

char stack[max];//universal variable declaration

int isFull(){//function defination

return top==max-1;

}

int isEmpty(){//function defination

return top==-1;

}

void push(char item){//function defination

if(isFull())//checks condition

return;

top++;

stack[top]=item;//item pushed

}

int pop(){//function defination

if(isEmpty())//checks condition

return INT\_MIN;

return stack[top--];//item popped

}

int peek(){//function defination

if(isEmpty())//checks condition

return INT\_MIN;

return stack[top];

}

int checkIfOperator(char ch){//function defination

return(ch>='a'&&ch<='z')||(ch>='A'&&ch<='Z');

}

int precedence(char ch){//function defination

switch(ch){//checks condition

case '+':

case '-': return 1;

case '\*':

case '/': return 2;

case '^': return 3;

}

return -1;

}

int getPostfix(char\* expression){//function defination

int i, j;//variable declaration

for(i=0,j=-1;expression[i];j++)

{//elseif ladder

if(checkIfOperand(expression[i]))

expression[++j]=expression[i];//checks condition

else if(expression[i]=='(')

push(expression[i]);

else if(expression[i]==')')

{

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

expression[++j]=pop(stack);//checks condition

if(!isEmpty(stack)&&peek(stack)!='(')

return -1;

else pop(stack);

}

else

{

while(!isEmpty(stack)&&precedence(expression[i])<=precedence(peek(stack)))//checks condition

expression[++j]=pop(stack);

push(expression[i]);//item pushed

}

}

while(!isEmpty(stack))//checks condition

expression[++j]=pop(stack);

expression[++j]='\0';

}

void reverse(char \*exp){//function defination

int size=strlen(exp), j=size, i=0;

char temp[size];//variable declaration

temp[j--]='\0';

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

temp[j]=exp[i];

j--;

i++;

}

strcpy(exp,temp);

}

void brackets(char\* exp){//function defination

int i=0;//variable declaration

while(exp[i]!='\0'){//checks condition

if(exp[i]=='(')

exp[i]=')';

else if(exp[i]==')')

exp[i]='(';

i++;

}

}

void InfixtoPrefix(char \*exp){//function defination

int size=strlen(exp);

reverse(exp);

brackets(exp);

getPostfix(exp);

reverse(exp);//function call

}

int main() {//main function

printf("the infix is: ");

char expression[]="(A+B\*C)";//infix expression

printf("%s\n",expression);

InfixtoPrefix(expression);//function call

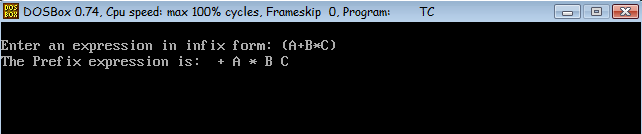
printf("the prefix is: ");

printf("%s\n",expression);

return 0;

}

**OUTPUT**



**ALGORITHM**

Step 1. Reverse the infix expression.  
Step 2. Make Every '(' as ')' and every ')' as '('  
Step 3. Convert expression to postfix form.

Step 4. Reverse the expression.  
Step 5. Exit

**INFIX TO POSTFIX CONVERSION**

**THEORY**

When we write mathematical expression in a program, we use infix expression. These expressions will be converted into equivalent machine instructions by compiler using stacks. Using stacks we can efficiently convert the expressions from infix to postfix, infix to prefix, postfix to prefix and postfix to infix.

**Representations of expressions:**

* Infix expression
* Postfix expression
* Prefix expression

**Infix expression**: In an expression if an operator is in between two operands, the expression is called an infix expression. Ex: a+ b

**Postfix expression:** In an expression, if an operator follows the two operands (i.e. operator comes after the two operands), the expression is called postfix expression. Ex: ab+

**Prefix expression:** In an expression, if an operator precedes two operands ( i.e. operator comes before the two operands), the expression is called prefix expression. Ex: +ab

The expression is evaluated according to the precedence of the operators.

Example: 6\*(2+3)-5 = 25

Brackets have the highest precedence and are evaluated first.

Multiplication and division has higher precedence over addition and subtraction.

**CODE WITH COMMENTS**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAX 100//header file

char stack[100];

char infix[MAX],postfix[MAX];

int top=-1;//universal declaration

void push(char);

int precedence(char);

char pop();

void inTopost();

int space(char);

void print();

int isEmpty();//function declaration

int main()//main function

{

printf("Enter the infix expression:");//infix exp entered

scanf("%s",infix);//infix exp scanned

inTopost();//function call

print();//function call

return 0;

}

void inTopost()

{

int i,j=0;

char next;

char symbol;//variable declaration

for(i=0;i<strlen(infix);i++)//checks condition

{

symbol=infix[i];

switch(symbol)//checks cases

{

case '(':

push(symbol);//function call

break;

case ')':

while((next=pop())!='(')

postfix[j++]=next;

break;

case '+':

case '-':

case '\*':

case '/':

case '^':

while(!isEmpty()&&precedence(symbol)<=precedence(stack[top]))//checks condition

postfix[j++]=pop();//function call

push(symbol);//function call

break;

default:

postfix[j++]=symbol;

}

}

while(!isEmpty())//checks condition

postfix[j++]=pop();//function call

postfix[j];

}

int precedence(char symbol)

{

switch(symbol)//checks cases

{

case '^':return 3;

case '/':

case '\*': return 2;

case '+':

case '-':return 1;

default:return 0;

}

}

void print()

{

int i=0;//vriable declaration

printf("The postfix expression is:\n");

while(postfix[i])//checks condition

{

printf("%c",postfix[i++]);

}

printf("\n");

}

void push(char c)

{

int Max;//variable declaration

if(top==Max-1)//checks condition

printf("stack overflow\n");

top++;//element pushed

stack[top]=c;

}

char pop()

{

char c;//variable declaration

if(top==-1)//checks condition

{

printf("stack underflow\n");

exit(1);

}

c=stack[top];

top=top-1;//element popped

return c;

}

int isEmpty()

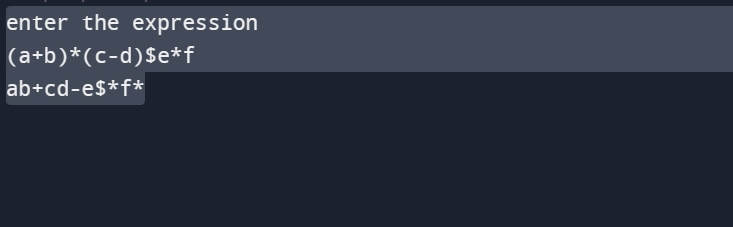
{

if(top==-1) return 1;//checks condition

else return 0;

}

**OUTPUT**



**ALGORITHM**

1. If the input character is an operand, print it.
2. If the input character is an operator-

If stack is empty push it to the stack.

If its precedence value is greater than the precedence value of the character on top, push.

If its precedence value is lower or equal then pop from stack and print while precedence of top char is more than the precedence value of the input character.

1. If the input character is ‘)’, then pop and print until top is ‘(‘. (Pop ‘(‘ but don’t print it.)
2. If stack becomes empty before encountering ‘(‘, then it’s a invalid expression.
3. Repeat steps 1-4 until input expression is completely read.
4. Pop the remaining elements from stack and print them.

The above method handles right associativity of exponentiation operator (here, ^) by assigning it higher precedence value outside stack and lower precedence value inside stack whereas it’s opposite for left associative operators.

**prefix TO POSTFIX CONVERSION**

**THEORY**

When we write mathematical expression in a program, we use infix expression. These expressions will be converted into equivalent machine instructions by compiler using stacks. Using stacks we can efficiently convert the expressions from infix to postfix, infix to prefix, postfix to prefix and postfix to infix.

**Representations of expressions:**

* Infix expression
* Postfix expression
* Prefix expression

**Infix expression**: In an expression if an operator is in between two operands, the expression is called an infix expression. Ex: a+ b

**Postfix expression:** In an expression, if an operator follows the two operands (i.e. operator comes after the two operands), the expression is called postfix expression. Ex: ab+

**Prefix expression:** In an expression, if an operator precedes two operands ( i.e. operator comes before the two operands), the expression is called prefix expression. Ex: +ab

The expression is evaluated according to the precedence of the operators.

**CODE WITH COMMENTS**

**#include<stdio.h>**

**#include<stdlib.h>**

**#include<string.h>**

**#include<math.h>**

**#define blank ' '**

**#define tab '\t'**

**#define max 100;//header file declaration**

**char \*pop();**

**char stack[max][max], prefix[max];//universal variable declaration**

**void push(char \*str);**

**int isEmpty();**

**int white\_space(char symbol);**

**void pre\_post();**

**int top;//function declaration**

**int main()//main function**

**{**

**int top=-1;//variable declaration**

**printf("enter prefix expression\n");**

**scanf("%s",prefix);**

**pre\_post();//function call**

**}**

**void pre\_post()**

**{**

**int i;//variable declaration**

**char operand1[max], operand2[max], symbol, temp[2], strin[max];**

**for(i=strlen(prefix)-1;i>=0;i--)//checks condition**

**{**

**symbol=prefix[i];**

**temp[0]=symbol;**

**temp[1]='\0';**

**if(!white\_space(symbol))//checks condition**

**{**

**switch(symbol)**

**{**

**case '+':**

**case '-':**

**case '\*':**

**case '/':**

**case '^':**

**case '%': strcpy(operand1,pop());**

**strcpy(operand2,pop());**

**strcpy(strin,operand1);**

**strcat(strin,operand2);**

**strcat(strin,temp);**

**push(strin);**

**break;**

**default: push(temp);//function call**

**}**

**}**

**}**

**printf("\npostfix expression is \n");**

**printf("%s",stack[100]);//prints exp**

**}**

**void push(char \*str)//function defination**

**{**

**if(top>max)//checks condition**

**{**

**printf("stack overflow\n");**

**exit(1);**

**}**

**else**

**{**

**top++;//increment**

**strcpy(stack[top],str);**

**}**

**}**

**char \*pop()//function defination**

**{**

**if(top==-1)//checks condition**

**{**

**printf("\nstack underflow\n");**

**exit(2);**

**}**

**else return stack[top--];**

**}**

**int isempty()//function defination**

**{**

**if(top==-1) return 1;//checks condition**

**else return 0;**

**}**

**int white\_space(char symbol)//function defination**

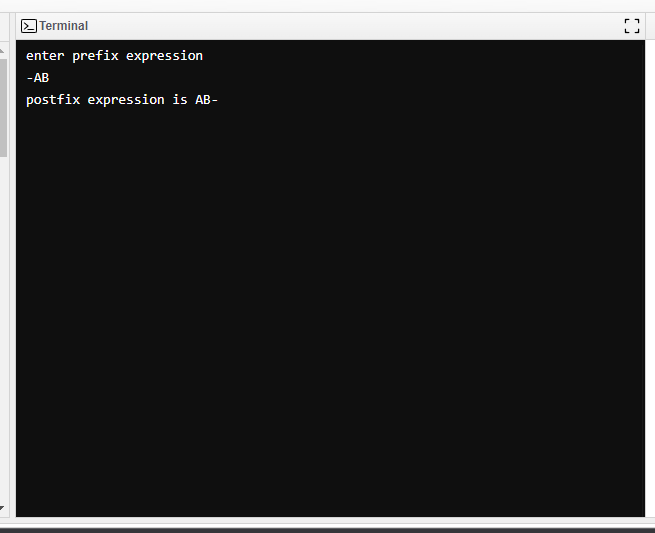
**{ //checks condition**

**if(symbol==blank||symbol==tab||symbol=='\0') return 1;**

**else return 0;**

**}**

**OUTPUT**

****

**ALGORITHM**

Read the Prefix expression in reverse order (from right to left)

If the symbol is an operand, then push it onto the Stack

If the symbol is an operator, then pop two operands from the Stack   
Create a string by concatenating the two operands and the operator after them.   
**string = operand1 + operand2 + operator**   
And push the resultant string back to Stack

Repeat the above steps until end of Prefix expression.

thesymbol *is an operand, then push it onto the Stack*

* *If the symbol is an operator, then pop two operands from the Stack   
  Create a string by concatenating the two operands and the operator before them.****string = operator + operand2 + operand1****And push the resultant string back to Stack*
* *Repeat the above steps until end of Postfix expression.*
* *Read the Postfix expression from left to right*
* *If the symbol is an operand, then push it onto the Stack*
* *If the symbol is an operator, then pop two operands from the Stack   
  Create a string by concatenating the two operands and the operator before them.****string = operator + operand2 + operand1****And push the resultant string back to Stack*
* *Repeat the above steps until end of Postfix e*