**Practical-10**

**Aim:** Finding “Follow” set Input: The string consists of grammar symbols. Output: The Follow set for a given string. Explanation: The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the Follow set of the given string.

**Code:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_PRODUCTIONS 10

#define MAX\_LENGTH 10

int n, m = 0;

char productions[MAX\_PRODUCTIONS][MAX\_LENGTH];

char followSet[MAX\_LENGTH];

void follow(char c);

void first(char c);

int main() {

int z;

char c, ch;

printf("Enter the no. of productions: ");

scanf("%d", &n);

printf("Enter the productions (epsilon=$):\n");

for (int i = 0; i < n; i++) {

scanf("%s%c", productions[i], &ch);

}

do {

m = 0; // Reset follow set size

printf("Enter the element whose FOLLOW is to be found: ");

scanf(" %c", &c); // Notice the space before %c to consume any newline

follow(c);

printf("FOLLOW(%c) = { ", c);

for (int i = 0; i < m; i++) {

printf("%c ", followSet[i]);

}

printf("}\n");

printf("Do you want to continue (0/1)? ");

scanf("%d%c", &z, &ch);

} while (z == 1);

return 0;

}

void follow(char c) {

if (productions[0][0] == c) {

followSet[m++] = '$';

}

for (int i = 0; i < n; i++) {

for (int j = 2; j < strlen(productions[i]); j++) {

if (productions[i][j] == c) {

// Check the next character

if (productions[i][j + 1] != '\0') {

first(productions[i][j + 1]);

}

// If there's no next character, find the follow of the left-hand side

if (productions[i][j + 1] == '\0' && c != productions[i][0]) {

follow(productions[i][0]);

}

}

}

}

}

void first(char c) {

if (!isupper(c)) {

followSet[m++] = c; // Add terminal to follow set

}

for (int k = 0; k < n; k++) {

if (productions[k][0] == c) {

if (productions[k][2] == '$') {

follow(productions[k][0]); // Follow the left-hand side

} else if (islower(productions[k][2])) {

followSet[m++] = productions[k][2]; // Add terminal to follow set

} else {

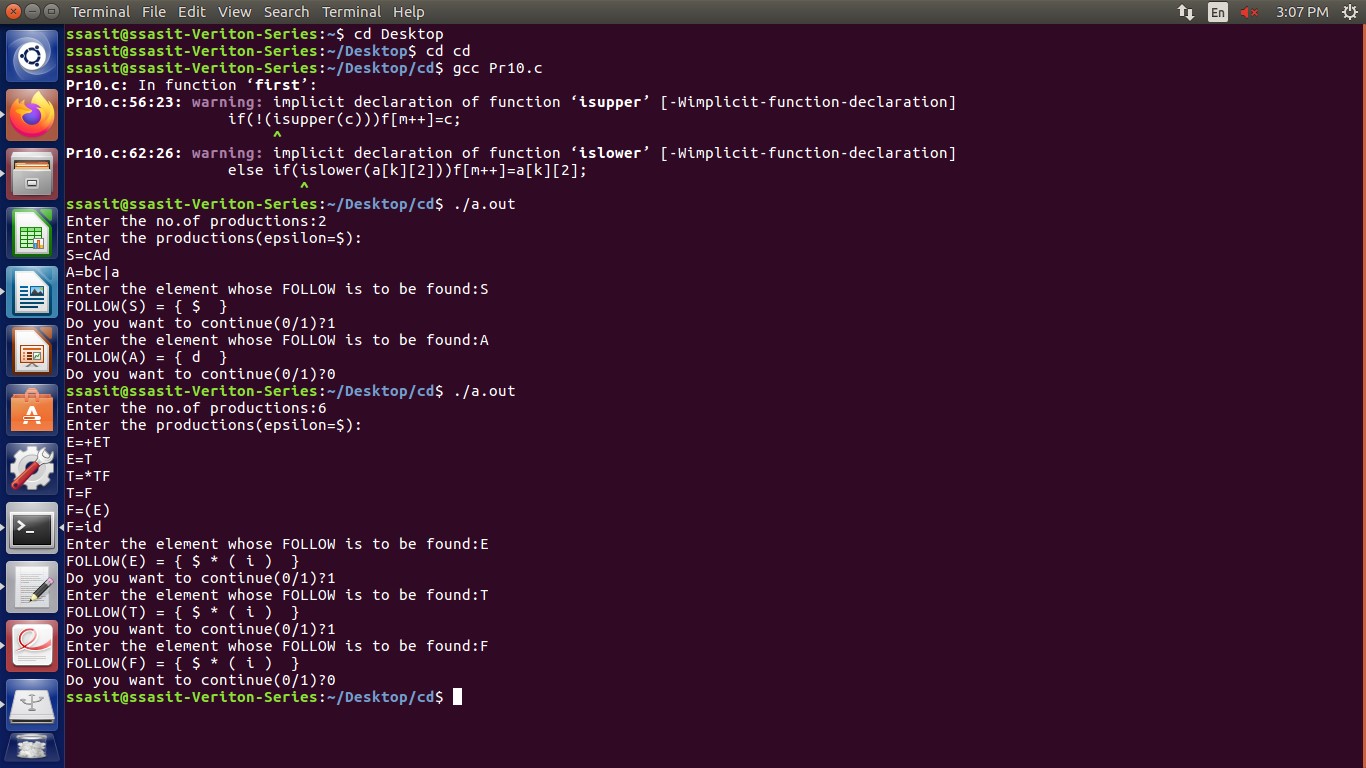
first(productions[k][2]); // Recursive call for non-terminal

}

}

}

}

**Output:**

**Practical-11**

**Aim:** Implement a C program for constructing LL (1) parsing.

**Code:**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

#include<stdlib.h>

void followfirst(char , int , int);

void findfirst(char , int , int);

void follow(char c);

int count,n=0;

char calc\_first[10][100];

char calc\_follow[10][100];

int m=0;

char production[10][10], first[10];

char f[10];

int k;

char ck;

int e;

int main(int argc,char \*\*argv)

{

int jm=0;

int km=0;

int i,choice;

char c,ch;

printf("How many productions ? :");

scanf("%d",&count);

printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n\n",count);

for(i=0;i<count;i++)

{

scanf("%s%c",production[i],&ch);

}

int kay;

char done[count];

int ptr = -1;

for(k=0;k<count;k++){

for(kay=0;kay<100;kay++){

calc\_first[k][kay] = '!';

}

}

int point1 = 0,point2,xxx;

for(k=0;k<count;k++)

{

c=production[k][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

findfirst(c,0,0);

ptr+=1;

done[ptr] = c;

printf("\n First(%c)= { ",c);

calc\_first[point1][point2++] = c;

for(i=0+jm;i<n;i++){

int lark = 0,chk = 0;

for(lark=0;lark<point2;lark++){

if (first[i] == calc\_first[point1][lark]){

chk = 1;

break;

}

}

if(chk == 0){

printf("%c, ",first[i]);

calc\_first[point1][point2++] = first[i];

}

}

printf("}\n");

jm=n;

point1++;

}

printf("\n");

printf("-----------------------------------------------\n\n");

char donee[count];

ptr = -1;

for(k=0;k<count;k++){

for(kay=0;kay<100;kay++){

calc\_follow[k][kay] = '!';

}

}

point1 = 0;

int land = 0;

for(e=0;e<count;e++)

{

ck=production[e][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(ck == donee[kay])

xxx = 1;

if (xxx == 1)

continue;

land += 1;

follow(ck);

ptr+=1;

donee[ptr] = ck;

printf(" Follow(%c) = { ",ck);

calc\_follow[point1][point2++] = ck;

for(i=0+km;i<m;i++){

int lark = 0,chk = 0;

for(lark=0;lark<point2;lark++){

if (f[i] == calc\_follow[point1][lark]){

chk = 1;

break;

}

}

if(chk == 0){

printf("%c, ",f[i]);

calc\_follow[point1][point2++] = f[i];

}

}

printf(" }\n\n");

km=m;

point1++;

}

char ter[10];

for(k=0;k<10;k++){

ter[k] = '!';

}

int ap,vp,sid = 0;

for(k=0;k<count;k++){

for(kay=0;kay<count;kay++){

if(!isupper(production[k][kay]) && production[k][kay]!= '#' && production[k][kay] != '=' && production[k][kay] != '\0'){

vp = 0;

for(ap = 0;ap < sid; ap++){

if(production[k][kay] == ter[ap]){

vp = 1;

break;

}

}

if(vp == 0){

ter[sid] = production[k][kay];

sid ++;

}

}

}

}

ter[sid] = '$';

sid++;

printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammer :-");

printf("\n\t\t\t\t\t\t\t^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");

printf("\n\t\t\t=====================================================================================================================\n");

printf("\t\t\t\t|\t");

for(ap = 0;ap < sid; ap++){

printf("%c\t\t",ter[ap]);

}

printf("\n\t\t\t=====================================================================================================================\n");

char first\_prod[count][sid];

for(ap=0;ap<count;ap++){

int destiny = 0;

k = 2;

int ct = 0;

char tem[100];

while(production[ap][k] != '\0'){

if(!isupper(production[ap][k])){

tem[ct++] = production[ap][k];

tem[ct++] = '\_';

tem[ct++] = '\0';

k++;

break;

}

else{

int zap=0;

int tuna = 0;

for(zap=0;zap<count;zap++){

if(calc\_first[zap][0] == production[ap][k]){

for(tuna=1;tuna<100;tuna++){

if(calc\_first[zap][tuna] != '!'){

tem[ct++] = calc\_first[zap][tuna];

}

else

break;

}

break;

}

}

tem[ct++] = '\_';

}

k++;

}

int zap = 0,tuna;

for(tuna = 0;tuna<ct;tuna++){

if(tem[tuna] == '#'){

zap = 1;

}

else if(tem[tuna] == '\_'){

if(zap == 1){

zap = 0;

}

else

break;

}

else{

first\_prod[ap][destiny++] = tem[tuna];

}

}

}

char table[land][sid+1];

ptr = -1;

for(ap = 0; ap < land ; ap++){

for(kay = 0; kay < (sid + 1) ; kay++){

table[ap][kay] = '!';

}

}

for(ap = 0; ap < count ; ap++){

ck = production[ap][0];

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(ck == table[kay][0])

xxx = 1;

if (xxx == 1)

continue;

else{

ptr = ptr + 1;

table[ptr][0] = ck;

}

}

for(ap = 0; ap < count ; ap++){

int tuna = 0;

while(first\_prod[ap][tuna] != '\0'){

int to,ni=0;

for(to=0;to<sid;to++){

if(first\_prod[ap][tuna] == ter[to]){

ni = 1;

}

}

if(ni == 1){

char xz = production[ap][0];

int cz=0;

while(table[cz][0] != xz){

cz = cz + 1;

}

int vz=0;

while(ter[vz] != first\_prod[ap][tuna]){

vz = vz + 1;

}

table[cz][vz+1] = (char)(ap + 65);

}

tuna++;

}

}

for(k=0;k<sid;k++){

for(kay=0;kay<100;kay++){

if(calc\_first[k][kay] == '!'){

break;

}

else if(calc\_first[k][kay] == '#'){

int fz = 1;

while(calc\_follow[k][fz] != '!'){

char xz = production[k][0];

int cz=0;

while(table[cz][0] != xz){

cz = cz + 1;

}

int vz=0;

while(ter[vz] != calc\_follow[k][fz]){

vz = vz + 1;

}

table[k][vz+1] = '#';

fz++;

}

break;

}

}

}

for(ap = 0; ap < land ; ap++){

printf("\t\t\t %c\t|\t",table[ap][0]);

for(kay = 1; kay < (sid + 1) ; kay++){

if(table[ap][kay] == '!')

printf("\t\t");

else if(table[ap][kay] == '#')

printf("%c=#\t\t",table[ap][0]);

else{

int mum = (int)(table[ap][kay]);

mum -= 65;

printf("%s\t\t",production[mum]);

}

}

printf("\n");

printf("\t\t\t---------------------------------------------------------------------------------------------------------------------");

printf("\n");

}

int j;

printf("\n\nPlease enter the desired INPUT STRING = ");

char input[100];

scanf("%s%c",input,&ch);

printf("\n\t\t\t\t\t===========================================================================\n");

printf("\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction");

printf("\n\t\t\t\t\t===========================================================================\n");

int i\_ptr = 0,s\_ptr = 1;

char stack[100];

stack[0] = '$';

stack[1] = table[0][0];

while(s\_ptr != -1){

printf("\t\t\t\t\t\t");

int vamp = 0;

for(vamp=0;vamp<=s\_ptr;vamp++){

printf("%c",stack[vamp]);

}

printf("\t\t\t");

vamp = i\_ptr;

while(input[vamp] != '\0'){

printf("%c",input[vamp]);

vamp++;

}

printf("\t\t\t");

char her = input[i\_ptr];

char him = stack[s\_ptr];

s\_ptr--;

if(!isupper(him)){

if(her == him){

i\_ptr++;

printf("POP ACTION\n");

}

else{

printf("\nString Not Accepted by LL(1) Parser !!\n");

exit(0);

}

}

else{

for(i=0;i<sid;i++){

if(ter[i] == her)

break;

}

char produ[100];

for(j=0;j<land;j++){

if(him == table[j][0]){

if (table[j][i+1] == '#'){

printf("%c=#\n",table[j][0]);

produ[0] = '#';

produ[1] = '\0';

}

else if(table[j][i+1] != '!'){

int mum = (int)(table[j][i+1]);

mum -= 65;

strcpy(produ,production[mum]);

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

}

else{

printf("\nString Not Accepted by LL(1) Parser !!\n");

exit(0);

}

}

}

int le = strlen(produ);

le = le - 1;

if(le == 0){

continue;

}

for(j=le;j>=2;j--){

s\_ptr++;

stack[s\_ptr] = produ[j];

}

}

}

printf("\n\t\t\t=======================================================================================================================\n");

if (input[i\_ptr] == '\0'){

printf("\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");

}

else

printf("\n\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");

printf("\t\t\t=======================================================================================================================\n");

}

void follow(char c)

{

int i ,j;

if(production[0][0]==c){

f[m++]='$';

}

for(i=0;i<10;i++)

{

for(j=2;j<10;j++)

{

if(production[i][j]==c)

{

if(production[i][j+1]!='\0'){

followfirst(production[i][j+1],i,(j+2));

}

if(production[i][j+1]=='\0'&&c!=production[i][0]){

follow(production[i][0]);

}

}

}

}

}

void findfirst(char c ,int q1 , int q2)

{

int j;

if(!(isupper(c))){

first[n++]=c;

}

for(j=0;j<count;j++)

{

if(production[j][0]==c)

{

if(production[j][2]=='#'){

if(production[q1][q2] == '\0')

first[n++]='#';

else if(production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))

{

findfirst(production[q1][q2], q1, (q2+1));

}

else

first[n++]='#';

}

else if(!isupper(production[j][2])){

first[n++]=production[j][2];

}

else {

findfirst(production[j][2], j, 3);

}

}

}

}

void followfirst(char c, int c1 , int c2)

{

int k;

if(!(isupper(c)))

f[m++]=c;

else{

int i=0,j=1;

for(i=0;i<count;i++)

{

if(calc\_first[i][0] == c)

break;

}

while(calc\_first[i][j] != '!')

{

if(calc\_first[i][j] != '#'){

f[m++] = calc\_first[i][j];

}

else{

if(production[c1][c2] == '\0'){

follow(production[c1][0]);

}

else{

followfirst(production[c1][c2],c1,c2+1);

}

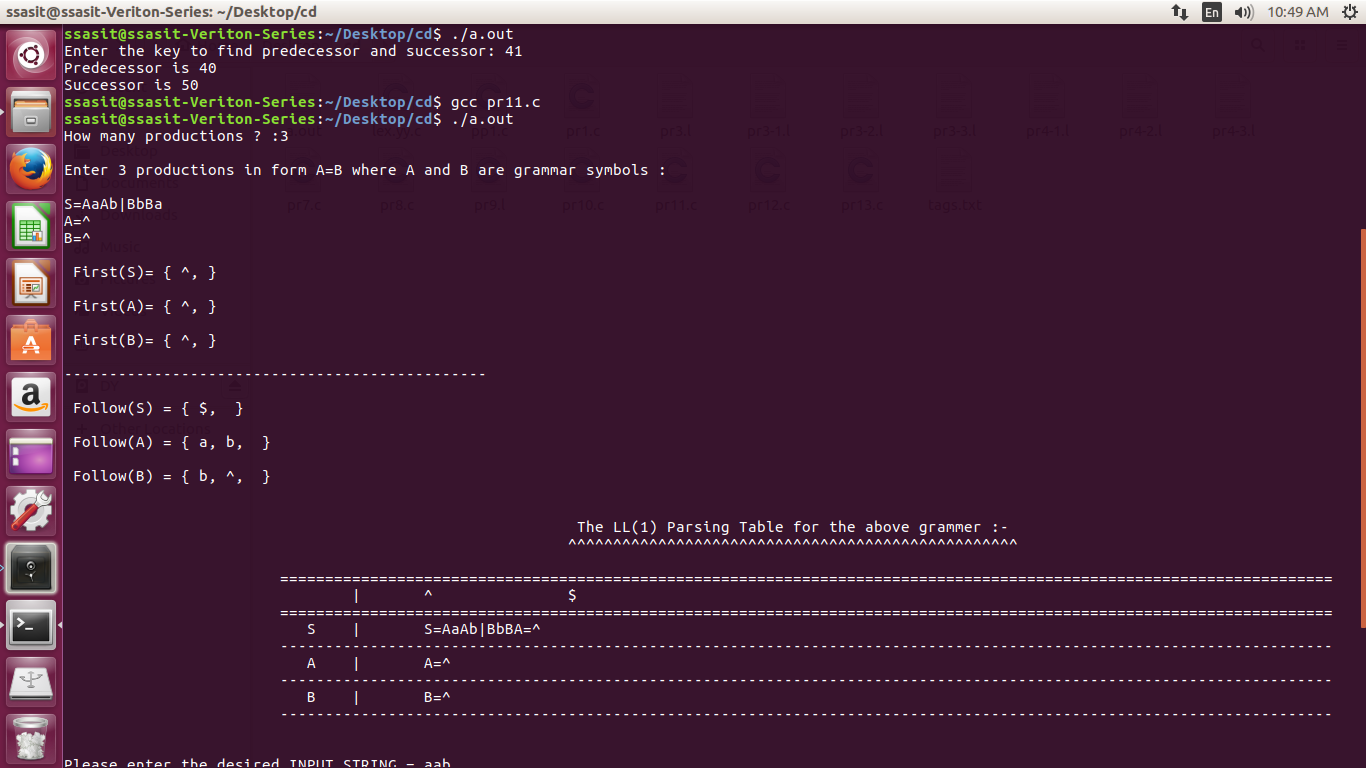
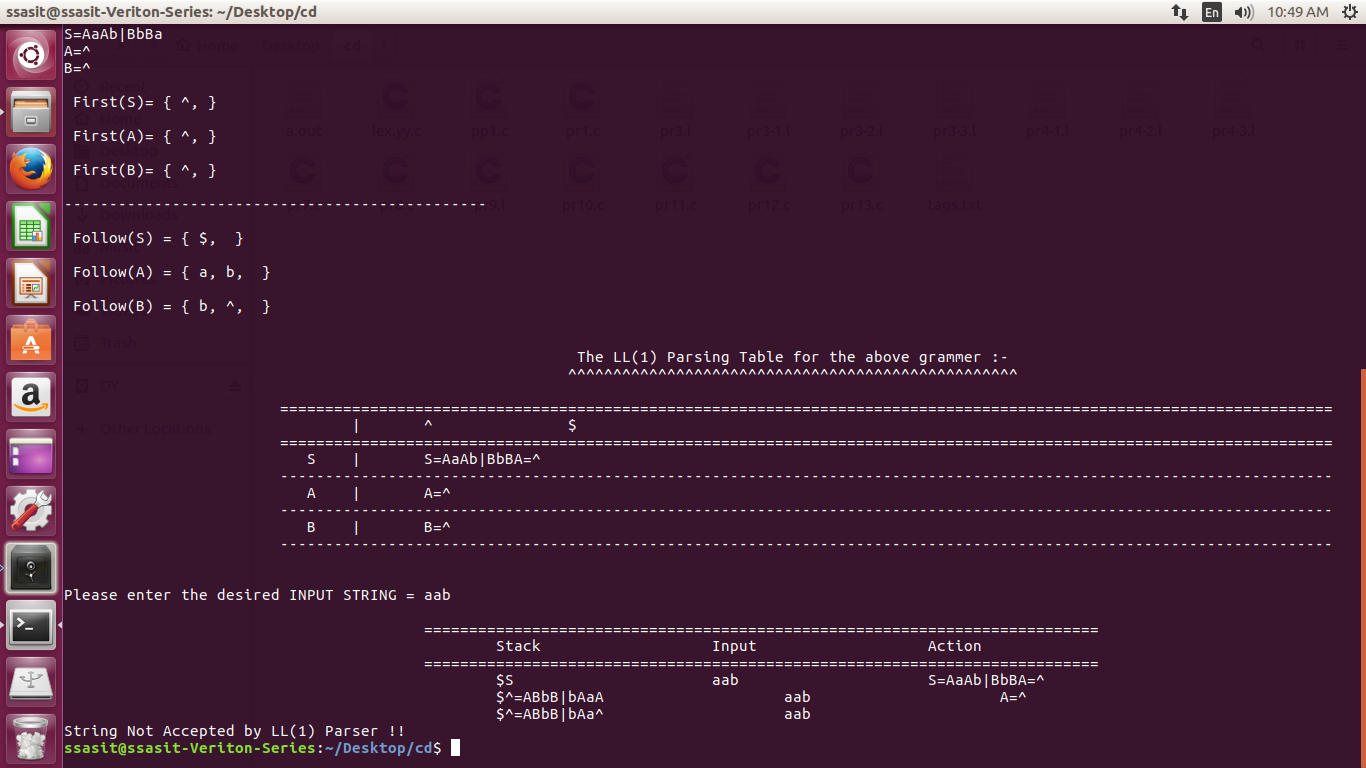
}

j++;

}

}

}

**Output:**

**Practical-7**

**Aim:** Generate 3-tuple intermediate code for given infix expression.

**Code:**

#include <stdio.h>

#include <string.h>

void pm();

void plus();

void div();

int i, j, l;

char ex[20], expr[20], expr1[20], id1[5], op[5], id2[5];

void reverse\_string(char \*str);

int main() {

printf("Enter the expression with an arithmetic operator: ");

scanf("%19s", ex); // Limit input size to avoid buffer overflow

strcpy(expr, ex); // Copy the expression into expr

l = strlen(expr); // Get the length of the expression

expr1[0] = '\0'; // Initialize expr1 as an empty string

// Parse the input expression

for (i = 0; i < l; i++) {

if (expr[i] == '+' || expr[i] == '-') {

// Check if the next operator has higher precedence

if (expr[i + 2] == '/' || expr[i + 2] == '\*') {

pm(); // Handle precedence case

break;

} else {

plus(); // Handle + or - operation

break;

}

} else if (expr[i] == '/' || expr[i] == '\*') {

div(); // Handle division or multiplication

break;

}

}

return 0;

}

// Function to handle precedence (pm)

void pm() {

reverse\_string(expr); // Reverse the expression

j = l - i - 1; // Calculate the position from where to cut the expression

strncpy(expr1, expr, j); // Copy the relevant part

expr1[j] = '\0'; // Null-terminate the string

reverse\_string(expr1); // Reverse back the expression

printf("Three address code:\n");

printf("temp = %s\n", expr1);

printf("temp1 = %c %c temp\n", expr[j + 1], expr[j]);

}

// Function to handle division or multiplication (div)

void div() {

strncpy(expr1, expr, i + 2); // Copy the part of the expression up to the operator

expr1[i + 2] = '\0'; // Null-terminate the string

printf("Three address code:\n");

printf("temp = %s\n", expr1);

printf("temp1 = temp %c %c\n", expr[i + 2], expr[i + 3]);

}

// Function to handle addition or subtraction (plus)

void plus() {

strncpy(expr1, expr, i + 2); // Copy the part of the expression up to the operator

expr1[i + 2] = '\0'; // Null-terminate the string

printf("Three address code:\n");

printf("temp = %s\n", expr1);

printf("temp1 = temp %c %c\n", expr[i + 2], expr[i + 3]);

}

// Helper function to reverse a string

void reverse\_string(char \*str) {

int len = strlen(str);

for (int i = 0; i < len / 2; i++) {

char temp = str[i];

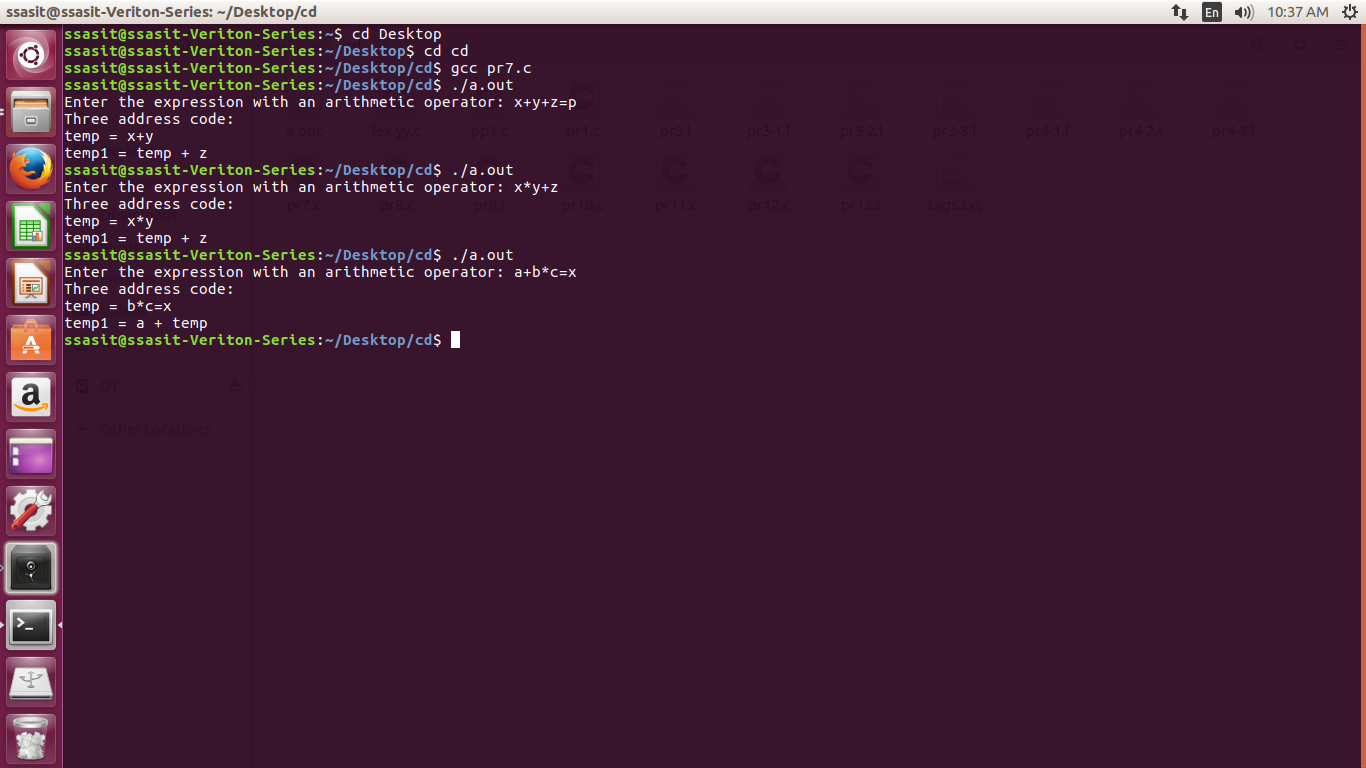
str[i] = str[len - i - 1];

str[len - i - 1] = temp;

}

}

**Output:**

****

**Practical-8**

**Aim:** Extract Predecessor and Successor from given Control Flow Graph

**Code:**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int key;

struct Node\* left;

struct Node\* right;

};

struct Node\* pre = NULL;

struct Node\* suc = NULL;

void findPreSuc(struct Node\* root, int key) {

if (root == NULL)

return;

if (root->key == key) {

if (root->left != NULL) {

struct Node\* tmp = root->left;

while (tmp->right != NULL)

tmp = tmp->right;

pre = tmp;

}

if (root->right != NULL) {

struct Node\* tmp = root->right;

while (tmp->left != NULL)

tmp = tmp->left;

suc = tmp;

}

return;

}

if (root->key > key) {

suc = root;

findPreSuc(root->left, key);

} else {

pre = root;

findPreSuc(root->right, key);

}

}

struct Node\* insert(struct Node\* node, int key) {

if (node == NULL) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->key = key;

newNode->left = newNode->right = NULL;

return newNode;

}

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

return node;

}

int main() {

int key;

printf("Enter the key to find predecessor and successor: ");

scanf("%d", &key);

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

findPreSuc(root, key);

if (pre != NULL)

printf("Predecessor is %d\n", pre->key);

else

printf("No Predecessor\n");

if (suc != NULL)

printf("Successor is %d\n", suc->key);

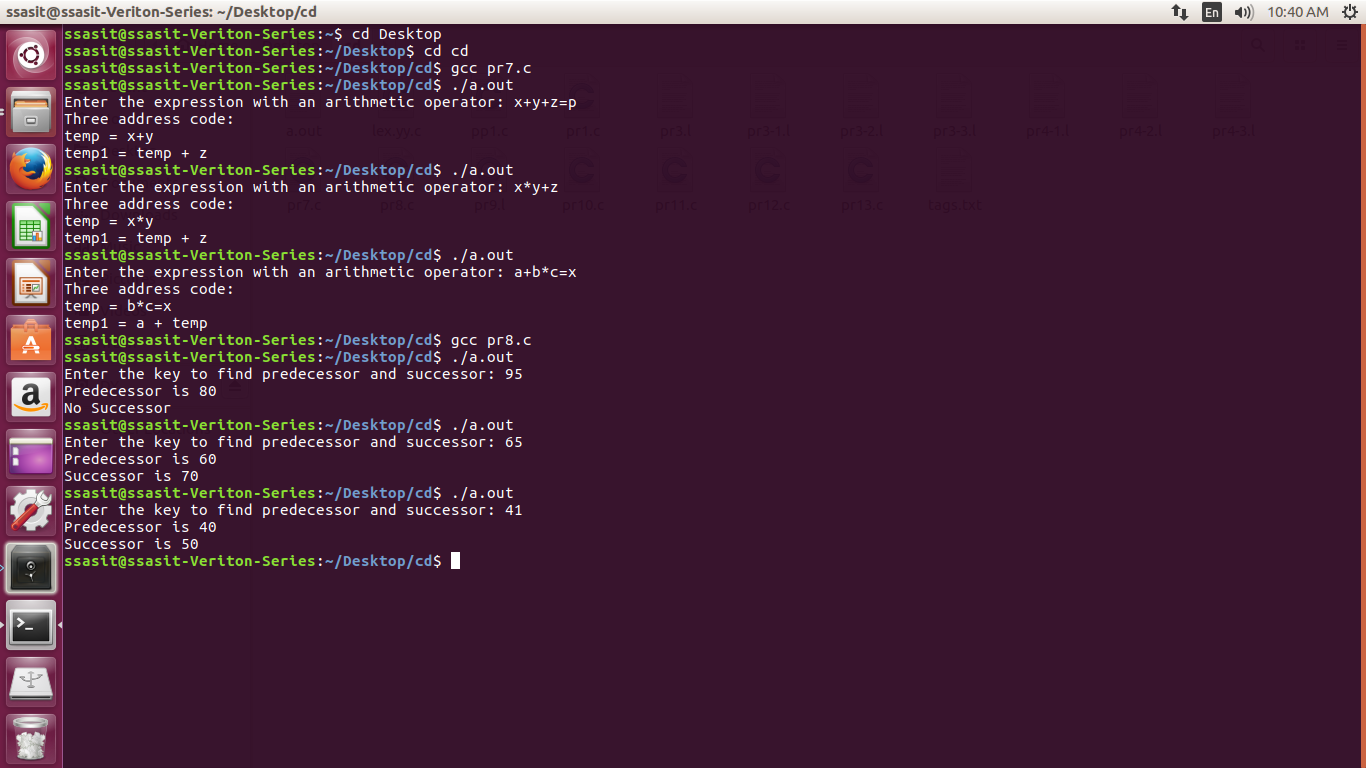
else

printf("No Successor\n");

return 0;

}

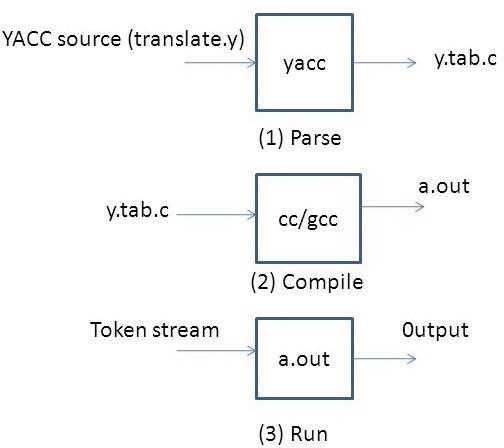
**Output:**

****

**Practical-9**

**Aim:** Introduction to YACC and generate Calculator Program.

* YACC (Yet Another Compiler Compiler) is a tool used to generate a parser. This document is a tutorial for the use of YACC to generate a parser for ExpL. YACC translates a given Context Free Grammar (CFG) specifications (input in input\_file.y) into a C implementation (y.tab.c) of a corresponding push down automaton (i.e., a finite state machine with a stack). This C program when compiled, yields an executable parser.



### **How yacc works?**

* The input to **yacc** describes the rules of a grammar. **yacc** uses these rules to produce the source code for a program that parses the grammar. You can then compile this source code to obtain a program that reads input, parses it according to the grammar, and takes action based on the result.
* The source code produced by **yacc** is written in the C programming language. It consists of a number of data tables that represent the grammar, plus a C function named **yyparse()**. By default, **yacc** symbol names used begin with **yy**. This is an historical convention, dating back to **yacc**'s predecessor, UNIX **yacc**. You can avoid conflicts with **yacc** names by avoiding symbols that start with **yy**.

**The structure of YACC programs:**

* A YACC program consists of three sections: Declarations, Rules and Auxiliary functions. (Note the similarity with the structure of LEX programs).

DECLARATIONS

%% RULES

%%

AUXILIARY FUNCTIONS

**Code:**

#include<stdio.h>

#include "y.tab.h"

#include<ctype.h>

extern int yylval;

%%

[0-9]+ {

yylval=atoi(yytext);

return NUMBER;

}

[\t] ;

[\n] return 0;

. return yytext[0];

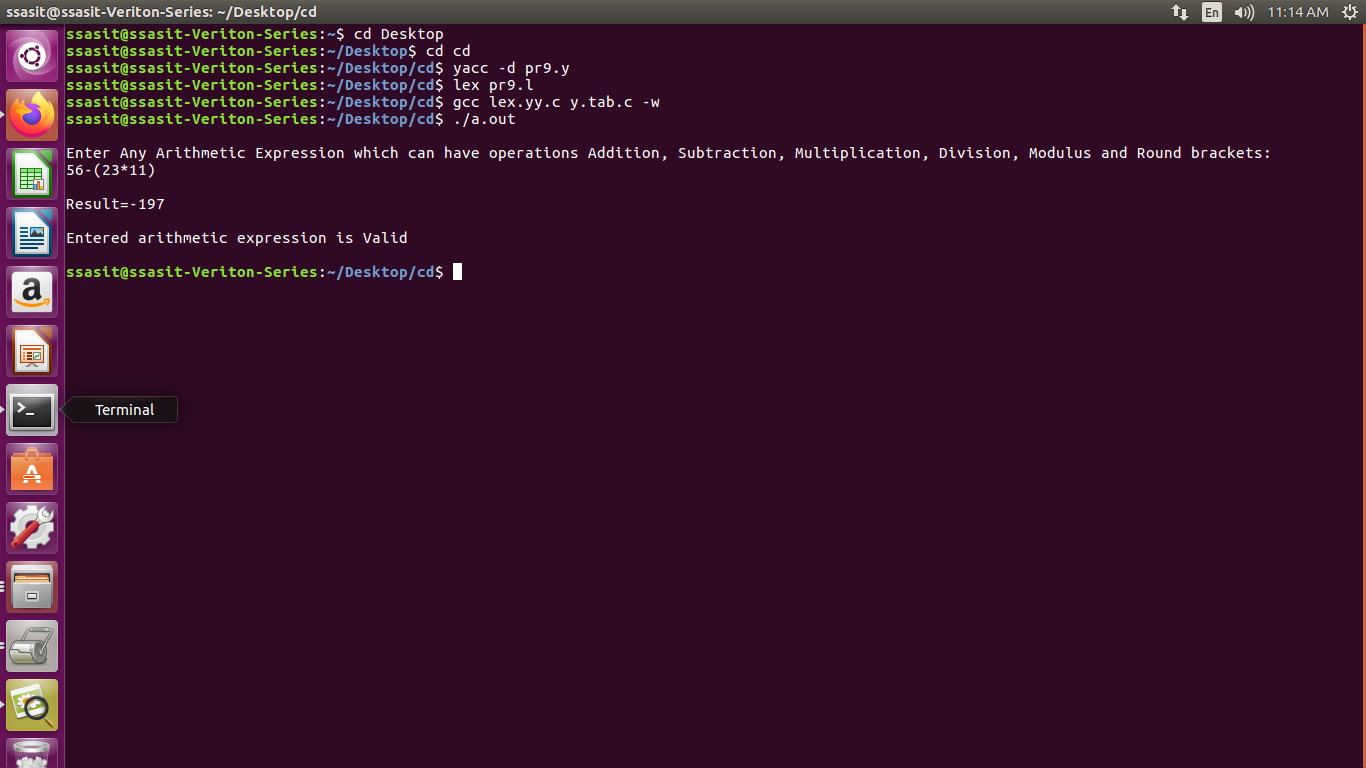
%%

int yywrap()

{

return 1;

}

**Output:**

**Practical-12**

**Aim:** Implement a C program to implement LALR parsing.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function prototypes

void push(char \*, int \*, char);

char stacktop(char \*);

void isproduct(char, char);

int ister(char);

int isnter(char);

int isstate(char);

void error();

void isreduce(char, char);

char pop(char \*, int \*);

void printt(char \*, int \*, char[], int);

void rep(char[], int);

// Structures for action and goto tables

struct action {

char row[6][5];

};

struct gotol {

char r[3][4];

};

// Action and Goto tables

const struct action A[12] = {

{"sf", "emp", "emp", "se", "emp", "emp"},

{"emp", "sg", "emp", "emp", "emp", "acc"},

{"emp", "rc", "sh", "emp", "rc", "rc"},

{"emp", "re", "re", "emp", "re", "re"},

{"sf", "emp", "emp", "se", "emp", "emp"},

{"emp", "rg", "rg", "emp", "rg", "rg"},

{"sf", "emp", "emp", "se", "emp", "emp"},

{"sf", "emp", "emp", "se", "emp", "emp"},

{"emp", "sg", "emp", "emp", "sl", "emp"},

{"emp", "rb", "sh", "emp", "rb", "rb"},

{"emp", "rb", "rd", "emp", "rd", "rd"},

{"emp", "rf", "rf", "emp", "rf", "rf"}

};

const struct gotol G[12] = {

{"b", "c", "d"},

{"emp", "emp", "emp"},

{"emp", "emp", "emp"},

{"emp", "emp", "emp"},

{"i", "c", "d"},

{"emp", "emp", "emp"},

{"emp", "j", "d"},

{"emp", "emp", "k"},

{"emp", "emp", "emp"},

{"emp", "emp", "emp"}

};

// Terminal and non-terminal symbols

char ter[6] = {'i', '+', '\*', ')', '(', '$'};

char nter[3] = {'E', 'T', 'F'};

char states[12] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'm', 'j', 'k', 'l'};

char stack[100];

int top = -1;

char temp[10];

// Grammar rules

struct grammar {

char left;

char right[5];

};

const struct grammar rl[6] = {

{'E', "e+T"},

{'E', "T"},

{'T', "T\*F"},

{'T', "F"},

{'F', "(E)"},

{'F', "i"}

};

// Push function to add an item to the stack

void push(char \*s, int \*sp, char item) {

if (\*sp == 100) {

printf("Stack is full\n");

} else {

\*sp = \*sp + 1;

s[\*sp] = item;

}

}

// Get the top item of the stack

char stacktop(char \*s) {

return s[top];

}

// Determine the product action based on input and stack top

void isproduct(char x, char p) {

int k = ister(x);

int l = isstate(p);

strcpy(temp, A[l - 1].row[k - 1]);

}

// Check if the character is a terminal

int ister(char x) {

for (int i = 0; i < 6; i++) {

if (x == ter[i]) return i + 1;

}

return 0;

}

// Check if the character is a non-terminal

int isnter(char x) {

for (int i = 0; i < 3; i++) {

if (x == nter[i]) return i + 1;

}

return 0;

}

// Check if the character is a state

int isstate(char p) {

for (int i = 0; i < 12; i++) {

if (p == states[i]) return i + 1;

}

return 0;

}

// Error handling function

void error() {

printf("Error in the input\n");

exit(0);

}

// Perform reduction based on the state and non-terminal

void isreduce(char x, char p) {

int k = isstate(x);

int l = isnter(p);

strcpy(temp, G[k - 1].r[l - 1]);

}

// Pop function to remove and return the top item from the stack

char pop(char \*s, int \*sp) {

if (\*sp == -1) {

printf("Stack is empty\n");

return '\0';

}

return s[(\*sp)--];

}

// Print the current state of the stack and input

void printt(char \*t, int \*p, char inp[], int i) {

printf("\n");

for (int r = 0; r <= \*p; r++) rep(t, r);

printf("\t\t\t");

for (int r = i; inp[r] != '\0'; r++) printf("%c", inp[r]);

}

// Helper function to represent states

void rep(char t[], int r) {

char c = t[r];

switch (c) {

case 'a': printf("0"); break;

case 'b': printf("1"); break;

case 'c': printf("2"); break;

case 'd': printf("3"); break;

case 'e': printf("4"); break;

case 'f': printf("5"); break;

case 'g': printf("6"); break;

case 'h': printf("7"); break;

case 'm': printf("8"); break;

case 'j': printf("9"); break;

case 'k': printf("10"); break;

case 'l': printf("11"); break;

default: printf("%c", t[r]); break;

}

}

// Main function

int main() {

char inp[80], x, p, dl[80], y, bl = 'a';

int i = 0, j, k, l, n, m;

printf("Enter the input: ");

if (scanf("%79s", inp) != 1) {

printf("Error reading input\n");

return 1;

}

// Append termination symbol

int len = strlen(inp);

inp[len] = '$';

inp[len + 1] = '\0';

// Initialize the stack

push(stack, &top, bl);

printf("\nStack \t\t\t Input");

printt(stack, &top, inp, i);

do {

x = inp[i];

p = stacktop(stack);

isproduct(x, p);

if (strcmp(temp, "emp") == 0) {

error();

}

if (strcmp(temp, "acc") == 0) {

break;

}

// Shift action

if (temp[0] == 's') {

push(stack, &top, inp[i]);

push(stack, &top, temp[1]);

i++;

}

// Reduce action

else if (temp[0] == 'r') {

j = isstate(temp[1]);

strcpy(temp, rl[j - 2].right);

dl[0] = rl[j - 2].left;

dl[1] = '\0';

n = strlen(temp);

for (k = 0; k < 2 \* n; k++) pop(stack, &top);

for (m = 0; dl[m] != '\0'; m++) push(stack, &top, dl[m]);

l = top;

y = stack[l - 1];

isreduce(y, dl[0]);

for (m = 0; temp[m] != '\0'; m++) push(stack, &top, temp[m]);

}

printt(stack, &top, inp, i);

} while (inp[i] != '\0');

// Final acceptance check

if (strcmp(temp, "acc") == 0) {

printf("\nAccept the input\n");

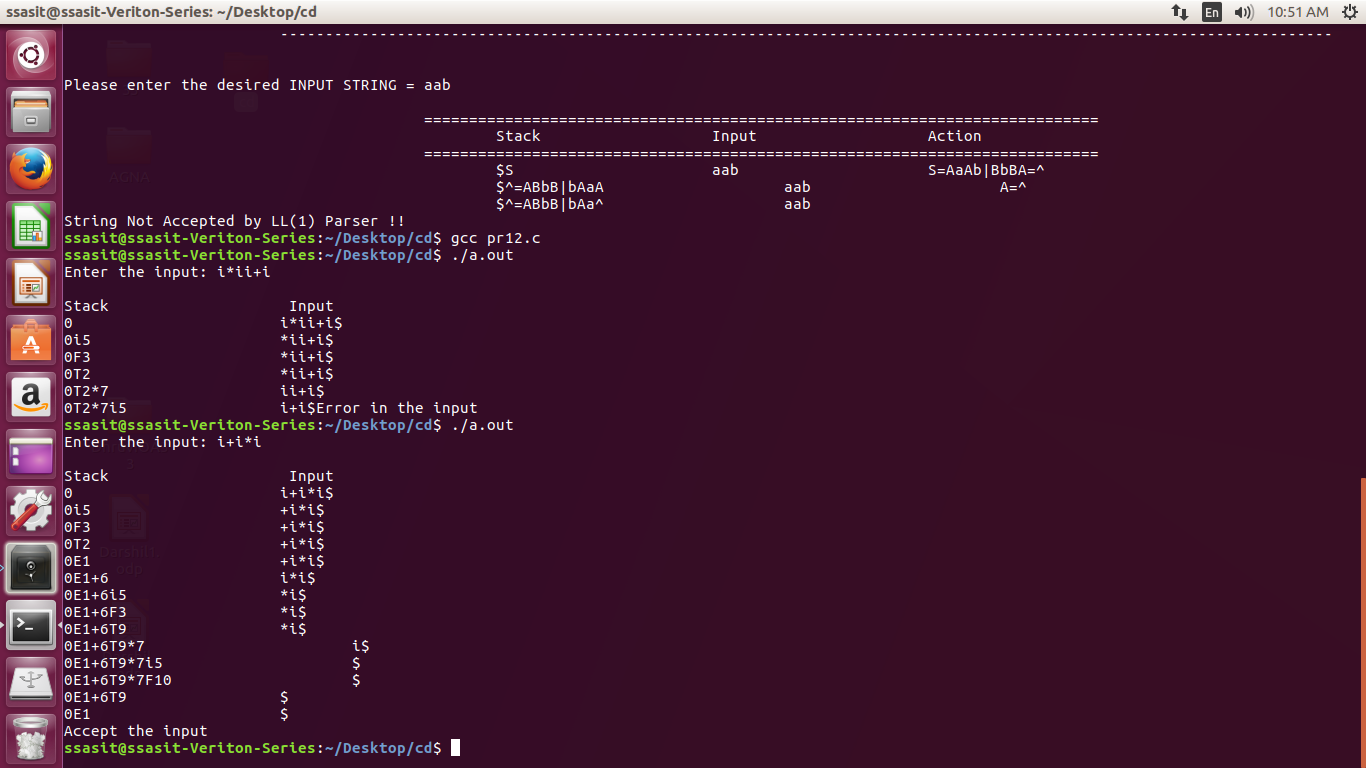
} else {

printf("\nDo not accept the input\n");

}

return 0;

}

**Output:**

**Practical-13**

**Aim:** Implement a C program to implement operator precedence parsing.

**Code:**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

char \*input; int

i=0;

char lasthandle[6],stack[50],handles[][5]={")E(","E\*E","E+E","i","E^E"}; //(E)

becomes )E( when pushed to stack

int top=0,l;

char prec[9][9]={

/\*stack + - \* / ^ i ( ) $ \*/

/\* + \*/ '>', '>','<','<','<','<','<','>','>',

/\* - \*/ '>', '>','<','<','<','<','<','>','>',

/\* \* \*/ '>', '>','>','>','<','<','<','>','>',

/\* / \*/ '>', '>','>','>','<','<','<','>','>',

/\* ^ \*/ '>', '>','>','>','<','<','<','>','>',

/\* i \*/ '>', '>','>','>','>','e','e','>','>',

/\* ( \*/ '<', '<','<','<','<','<','<','>','e',

/\* ) \*/ '>', '>','>','>','>','e','e','>','>',

/\* $ \*/ '<', '<','<','<','<','<','<','<','>',

};

int getindex(char c)

{ switch(c)

{

case '+':return 0;

case '-':return 1;

case '\*':return 2;

case '/':return 3; case

'^':return 4; case

'i':return 5; case

'(':return 6; case

')':return 7; case

'$':return 8;

}

}

int shift()

{

stack[++top]=\*(input+i++);

stack[top+1]='\0';

}

int reduce()

{

int i,len,found,t;

for(i=0;i<5;i++)//selecting handles

{

len=strlen(handles[i]);

if(stack[top]==handles[i][0]&&top+1>=len)

{

found=1;

for(t=0;t<len;t++){

if(stack[top-t]!=handles[i][t]) {

found=0;

break;

}

}

if(found==1){

stack[top-t+1]='E';

top=top-t+1;

strcpy(lasthandle,handles[i]);

stack[top+1]='\0';

return 1;//successful reduction

}

}

}

return 0;

}

void dispstack(){

int j;

for(j=0;j<=top;j++)

printf("%c",stack[j]);

}

void dispinput(){

int j;

for(j=i;j<l;j++)

printf("%c",\*(input+j)); }

void main(){

int j;

input=(char\*)malloc(50\*sizeof(char));

printf("\nEnter the string\n");

scanf("%s",input);

input=strcat(input,"$");

l=strlen(input);

strcpy(stack,"$");

printf("\nSTACK\tINPUT\tACTION");

while(i<=l){

shift();

printf("\n");

dispstack();

printf("\t");

dispinput();

printf("\tShift");

if(prec[getindex(stack[top])][getindex(input[i])]=='>'){

while(reduce()){

printf("\n");

dispstack();

printf("\t");

dispinput();

printf("\tReduced: E->%s",lasthandle);

}

}

}

if(strcmp(stack,"$E$")==0)

printf("\nAccepted;"); else

printf("\nNot Accepted;");

}

**Output:**

