**Compiler Design LAB**

**DOPPALAPUDI KALYAN KUMAR**

**AP20110010313**

**WEEK 1**

**1. Write a program in C that recognizes the following languages.**

**a. Set of all strings over binary alphabet containing even number of 0’s and even**

number of 1’s.

#include<stdio.h>

int main()

{

int state=0,i=0;

char token,input[20];

printf("Enter input string \t :");

scanf("%s",input);

while((token=input[i++])!='\0')

{

switch(state)

{

case 0: if(token=='0')

state=1;

else if(token=='1')

state=2;

else

{

printf("Invalid token");

}

break;

case 1: if(token=='0')

state=0;

else if(token=='1')

state=3;

else

{

printf("Invalid token");

}

break;

case 2: if(token=='0')

state=3;

else if(token=='1')

state=0;

else

{

printf("Invalid token");

}

break;

case 3: if(token=='0')

state=2;

else if(token=='1')

state=1;

else

{

printf("Invalid token");

}

break;

}

}

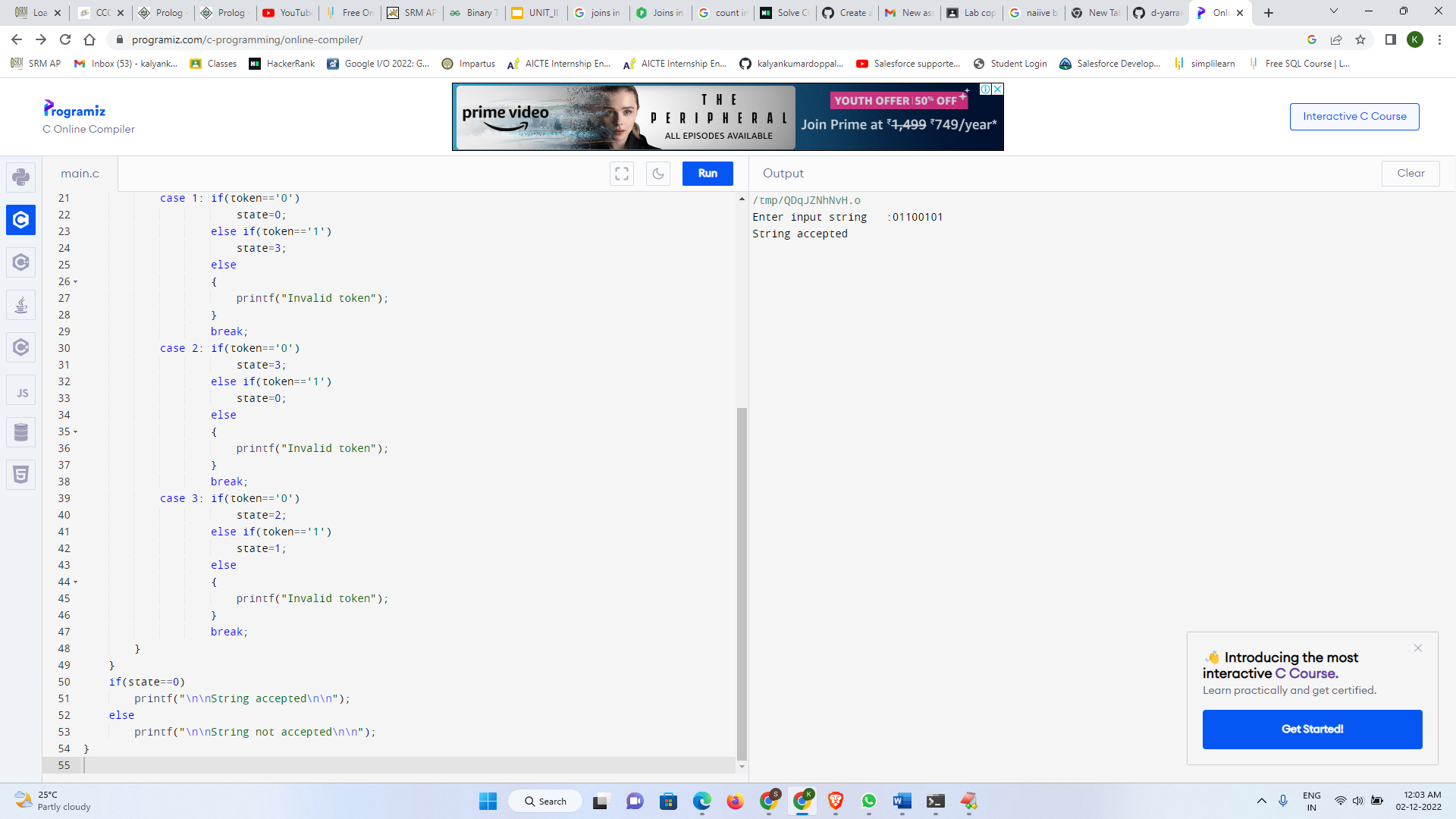
if(state==0)

printf("\n\nString accepted\n\n");

else

printf("\n\nString not accepted\n\n");

}



**b. Lab Assignment: Set of all strings ending with two symbols of same type.**

**CODE**

#include<stdio.h>

int main()

{

char state='a';

int length, i;

char n[20];

printf("Enter the String: ");

scanf("%s",n);

for(i=0;n[i]!='\0';i++)

{

switch(state)

{

case 'a':

if(n[i]=='0'||n[i]=='1'||n[i]=='2'||n[i]=='3'||n[i]=='4'||n[i]=='5'||n[i]=='6'||n[i]=='7'||n[i]=='8'||n[i]=='9')

state='a';

else if(n[i]=='.')

state='b';

else

state='d';

break;

case 'b':

if(n[i]=='0'|n[i]=='1'||n[i]=='2'||n[i]=='3'||n[i]=='4'||n[i]=='5'||n[i]=='6'||n[i]=='7'||n[i]=='8'||n[i]=='9')

state='c';

else

state='d';

break;

case 'c':

if(n[i]=='0'||n[i]=='1'||n[i]=='2'||n[i]=='3'||n[i]=='4'||n[i]=='5'||n[i]=='6'||n[i]=='7'||n[i]=='8'||n[i]=='9')

state='c';

else

state='d';

break;

case 'd':

state='d';

break; }

}

if(state=='c'||state=='a')

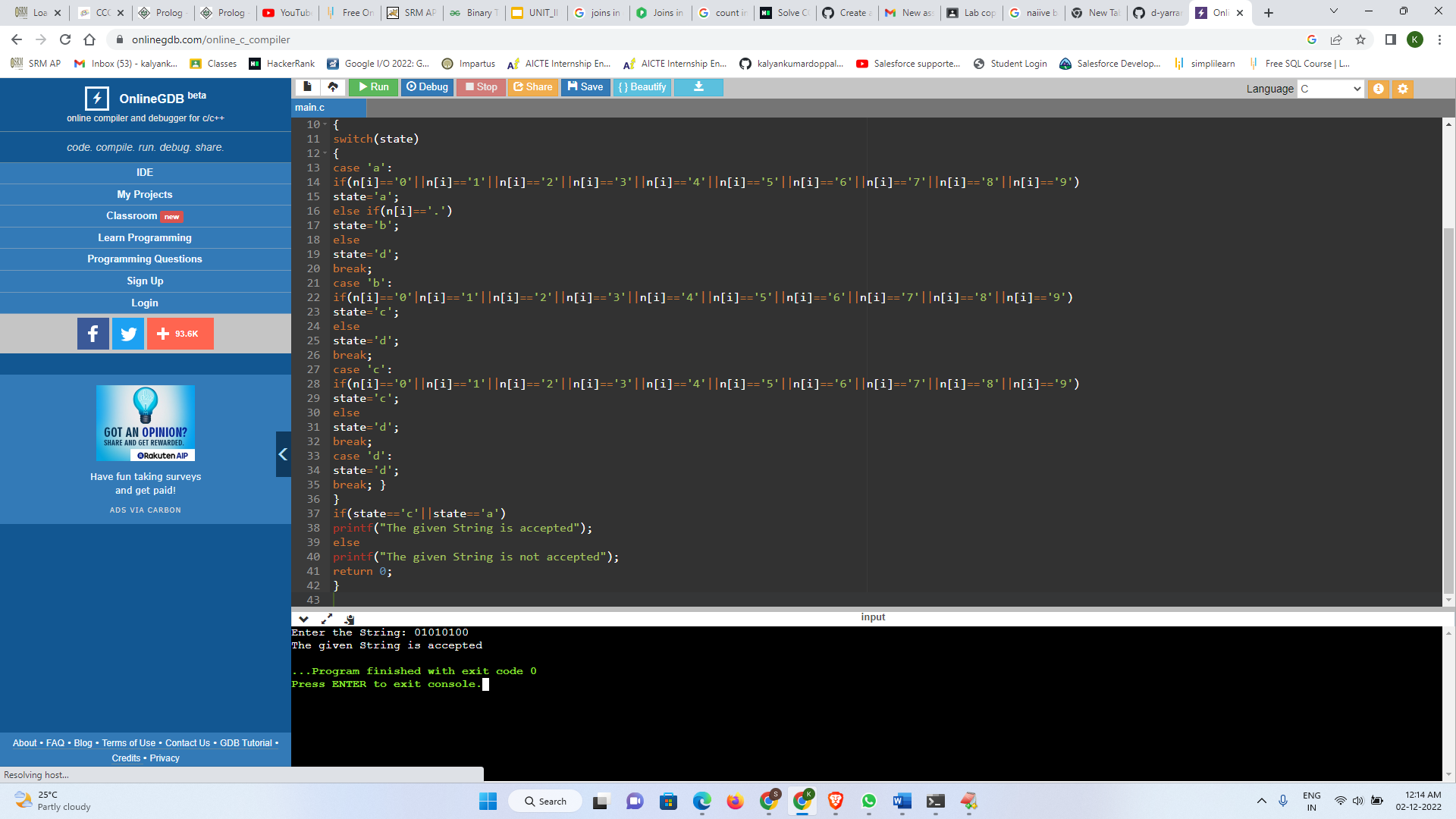
printf("The given String is accepted");

else

printf("The given String is not accepted");

return 0;

}



**WEEK 2**

**2. Implement lexical analyzer using C for recognizing the following tokens:**

**A minimum of 10 keywords of your choice**

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER.

bool isDelimiter(char ch)

{

if (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

ch == '[' || ch == ']' || ch == '{' || ch == '}')

return (true);

return (false);

}

// Returns 'true' if the character is an OPERATOR.

bool isOperator(char ch)

{

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '>' || ch == '<' ||

ch == '=')

return (true);

return (false);

}

// Returns 'true' if the string is a VALID IDENTIFIER.

bool validIdentifier(char\* str)

{

if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

str[0] == '3' || str[0] == '4' || str[0] == '5' ||

str[0] == '6' || str[0] == '7' || str[0] == '8' ||

str[0] == '9' || isDelimiter(str[0]) == true)

return (false);

return (true);

}

// Returns 'true' if the string is a KEYWORD.

bool isKeyword(char\* str)

{

if (!strcmp(str, "if") || !strcmp(str, "else") ||

!strcmp(str, "while") || !strcmp(str, "do") ||

!strcmp(str, "break") ||

!strcmp(str, "continue") || !strcmp(str, "int")

|| !strcmp(str, "double") || !strcmp(str, "float")

|| !strcmp(str, "return") || !strcmp(str, "char")

|| !strcmp(str, "case") || !strcmp(str, "char")

|| !strcmp(str, "sizeof") || !strcmp(str, "long")

|| !strcmp(str, "short") || !strcmp(str, "typedef")

|| !strcmp(str, "switch") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "static")

|| !strcmp(str, "struct") || !strcmp(str, "goto"))

return (true);

return (false);

}

// Returns 'true' if the string is an INTEGER.

bool isInteger(char\* str)

{

int i, len = strlen(str);

if (len == 0)

return (false);

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

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' || (str[i] == '-' && i > 0))

return (false);

}

return (true);

}

// Returns 'true' if the string is a REAL NUMBER.

bool isRealNumber(char\* str)

{

int i, len = strlen(str);

bool hasDecimal = false;

if (len == 0)

return (false);

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

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' && str[i] != '.' ||

(str[i] == '-' && i > 0))

return (false);

if (str[i] == '.')

hasDecimal = true;

}

return (hasDecimal);

}

// Extracts the SUBSTRING.

char\* subString(char\* str, int left, int right)

{

int i;

char\* subStr = (char\*)malloc(

sizeof(char) \* (right - left + 2));

for (i = left; i <= right; i++)

subStr[i - left] = str[i];

subStr[right - left + 1] = '\0';

return (subStr);

}

// Parsing the input STRING.

void parse(char\* str)

{

int left = 0, right = 0;

int len = strlen(str);

while (right <= len && left <= right) {

if (isDelimiter(str[right]) == false)

right++;

if (isDelimiter(str[right]) == true && left == right) {

if (isOperator(str[right]) == true)

printf("'%c' IS AN OPERATOR\n", str[right]);

right++;

left = right;

} else if (isDelimiter(str[right]) == true && left != right

|| (right == len && left != right)) {

char\* subStr = subString(str, left, right - 1);

if (isKeyword(subStr) == true)

printf("'%s' IS A KEYWORD\n", subStr);

else if (isInteger(subStr) == true)

printf("'%s' IS AN INTEGER\n", subStr);

else if (isRealNumber(subStr) == true)

printf("'%s' IS A REAL NUMBER\n", subStr);

else if (validIdentifier(subStr) == true

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS A VALID IDENTIFIER\n", subStr);

else if (validIdentifier(subStr) == false

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

left = right;

}

}

return;

}

// DRIVER FUNCTION

int main()

{

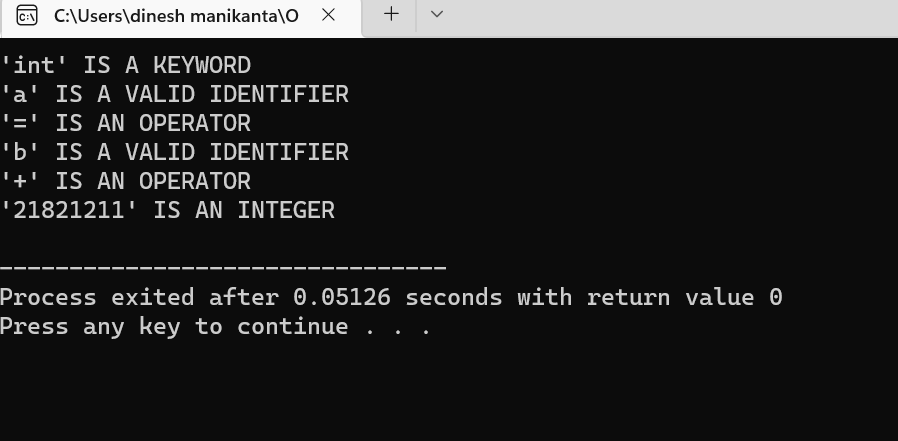
// maximum length of string is 100 here

char str[100] = "int a = b + 21821211; ";

parse(str); // calling the parse function

return (0);

}



**Identifiers with the regular expression : letter(letter | digit)\***

**Integers with the regular expression: digit+**

**Relational operators: &lt;, &gt;, &lt;=, &gt;=, ==, !=**

%{

#include<stdio.h>

%}

%%

["<" | "<=" | ">" | ">=" | "==" | "!="] {printf("%s is a relational operator\n",yytext);}

%%

int yywrap()

{

return 1;

}

int main()

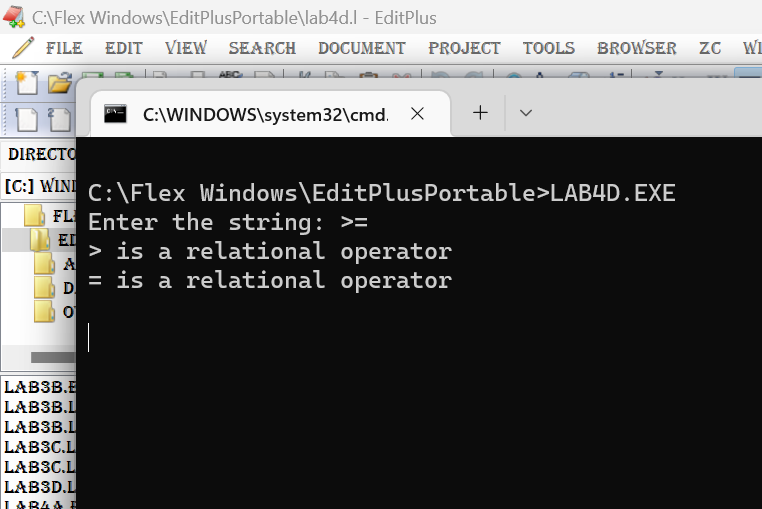
{

printf("Enter the string: ");

yylex();

return 0;

}



**Storing identifiers in symbol table.**

**Using files for input and output.**

**Week-3**

1. **Identification of Vowels and Consonants**

%option noyywrap

%{

#include<stdio.h>

%}

%%

[aeiouAEIOU]+ {printf("vowel\n");}

[a-zA-Z] {printf("consonant\n");}

%%

int main()

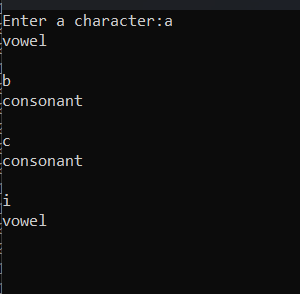
{

printf("Enter a character:");

yylex();

return 0;

}



**b. count number of vowels and consonants**

**code:**

%option noyywrap

%{

#include<stdio.h>

int v\_c = 0;

int c\_c = 0;

%}

%%

[aeiouAEIOU] {v\_c++;}

[a-zA-Z] {c\_c++;}

%%

int main()

{

printf("Enter string of vowels and consonants:");

yylex();

printf("Number of vowels: %d\n",v\_c);

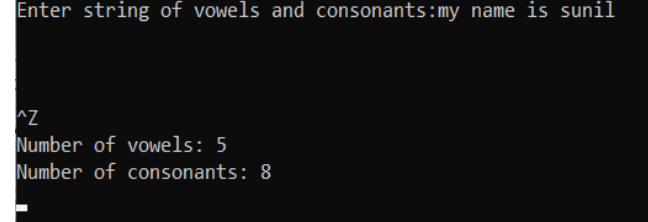
printf("Number of consonants: %d\n",c\_c);

getch();

return 0;

}

**Output**

****

**c. Count the number of Lines in given input**

**code:**

%option noyywrap

%{

#include<stdio.h>

int c=0;

%}

%%

\n c++;

.

%%

int main()

{

printf("Enter Input:");

yylex();

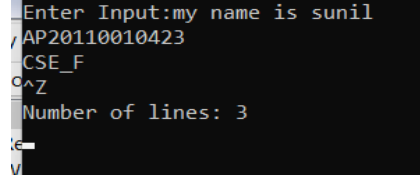
printf("Number of lines: %d\n",c);

getch();

return 0;

}

**Output:**

****

**d. Recognize strings ending with 00**

**code:**

%option noyywrap

%{

#include<stdio.h>

%}

%%

[0-1]\*00 {printf("Given string ends with 00");}

[0-1]\* {printf("Given string does not ends with 00");}

%%

int main()

{

printf("Enter the string:");

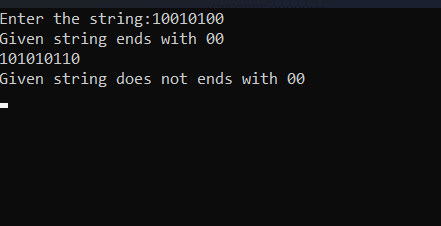
yylex();

getch();

return 0;

}

**Output:**



**e. Recognize a string with three consecutive 0’s**

**code:**

%option noyywrap

%{

#include<stdio.h>

%}

%%

[0-1]\*000[0-1]\* {printf("Given string has three

consecutive zeros");}

[0-1]\* {printf("Given string does not have three

consecutive zeros");}

%%

int main()

{

printf("Enter the string:");

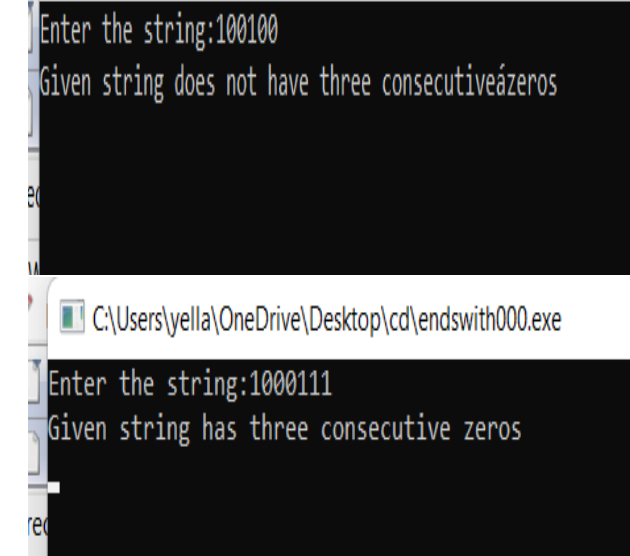
yylex();

getch();

return 0;

}

**Output**

****

**WEEK 4**

**4. Implement lexical analyzer using LEX for recognizing the following tokens:**

**A minimum of 10 keywords of your choice**

%{

#include<stdio.h>

%}

%%

auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern|return|union|continue|for|signed|void|do|if|static|while|default|goto|sizeof|volatile|const|float|short {printf("%s is a Keyword",yytext);}

[a-zA-Z][a-z A-Z 0-9]\* {printf("%s is an identifier\n",yytext);}

[0-9]+ {printf("%s is a number\n",yytext);}

["<" | "<=" | ">" | ">=" | "==" | "!="] {printf("%s is a relational operator\n",yytext);}

%%

int yywrap()

{

return 1;

}

int main()

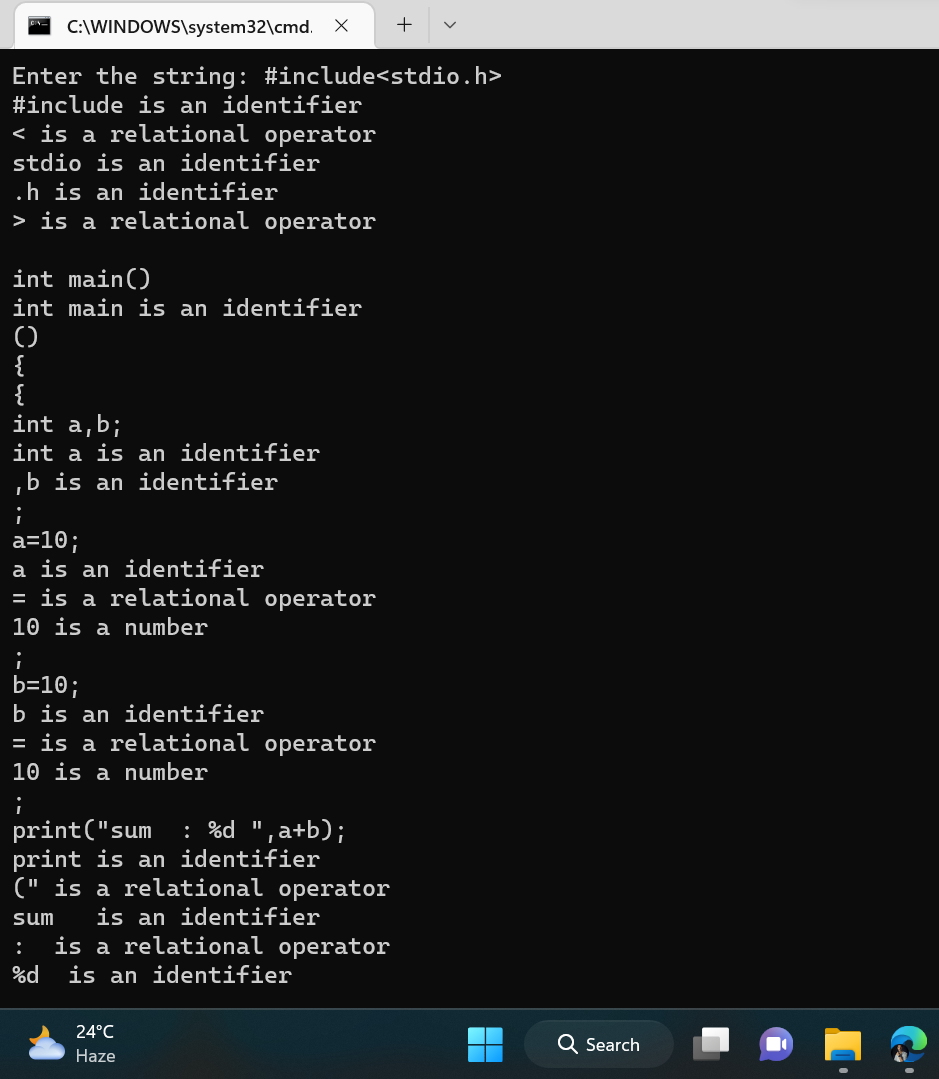
{

printf("Enter the string: ");

yylex();

return 0;

}



Identifiers with the regular expression : letter(letter | digit)\*

%{

#include<stdio.h>

%}

%%

[a-zA-Z][a-z A-Z 0-9]\* {printf("%s is an Letter\n",yytext);}

%%

int yywrap()

{

return 1;

}

int main()

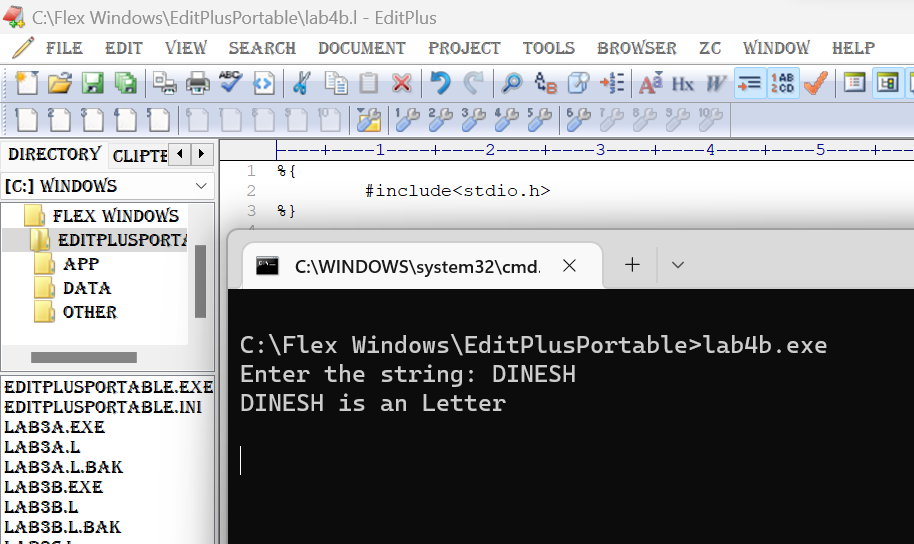
{

printf("Enter the string: ");

yylex();

return 0;

}



**Integers with the regular expression: digit+**

%{

#include<stdio.h>

%}

%%

[0-9]+ {printf("%s is a number\n",yytext);}

%%

int yywrap()

{

return 1;

}

int main()

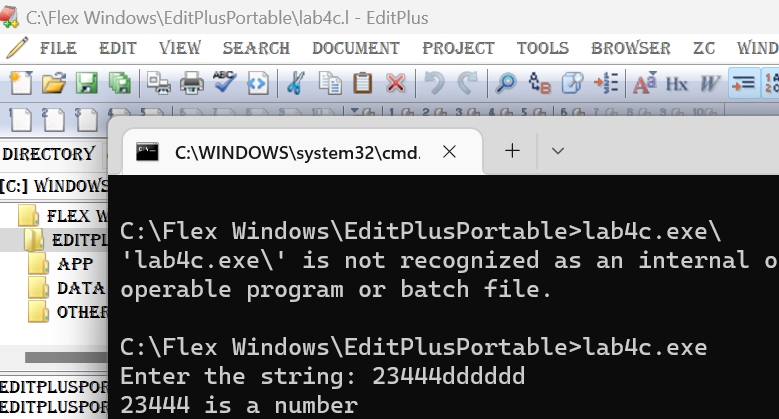
{

printf("Enter the string: ");

yylex();

return 0;

}



**Relational operators: &lt;, &gt;, &lt;=, &gt;=, ==, !=**

%{

#include<stdio.h>

%}

%%

["<" | "<=" | ">" | ">=" | "==" | "!="] {printf("%s is a relational operator\n",yytext);}

%%

int yywrap()

{

return 1;

}

int main()

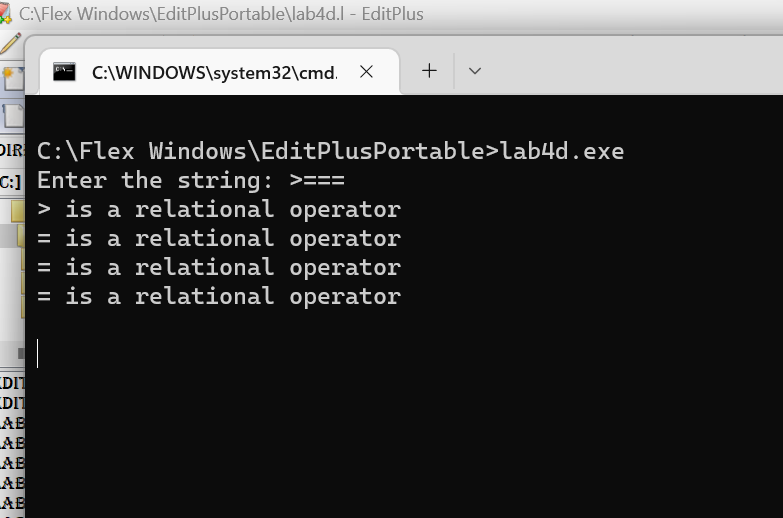
{

printf("Enter the string: ");

yylex();

return 0;

}



Ignores everything between multi line comments (/\* …. \*/)

Storing identifiers in symbol table

Using files for input and output.

**5. Lab Assignment:**

Consider the following mini Language, a simple procedural high-level language, only

operating on integer data, with a syntax looking vaguely like a simple C crossed with

Pascal. The syntax of the language is definedby the following BNF grammar:

**Week 6: Recursive Descent Parser**

**6.**

//grammar

/\*

E->TE'

E'->+TE'|NULL

T->FT'

T'->\*FT'|NULL

F->(E)|a

\*/

#include<stdio.h>

#include<string.h>

void E();

void T();

void F();

void Eds();

void Tds();

char input[1000];

int i,error;

int main()

{

printf("Enter the input:\n");

gets(input);

E();

if(i==strlen(input)&&error==0)

{

printf("String is accepted");

}

else

{

printf("string is rejected");

}

}

void E()

{

T();

Eds();

}

void T()

{

F();

Tds();

}

void Eds()

{

if(input[i]=='+')

{

i++;

T();

Eds();

}

}

void Tds()

{

if(input[i]=='\*')

{

i++;

F();

Tds();

}

}

void F()

{

if(input[i]=='a')

{

i++;

}

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

{

i++;

E();

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

{

i++;

}

else

{

error=1;

}

}

else

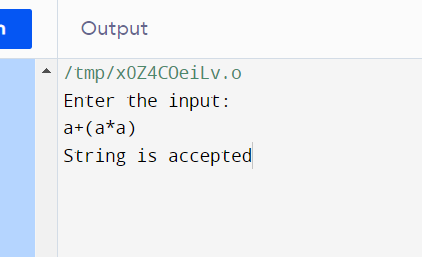
{

error=1;

}

}

OUTPUT:



**7.**

#include<stdio.h>

#include<string.h>

int S() ,L() ,Ldash();

char \*ip;

char string[50];

int main()

{

printf("Enter the string\n");

scanf("%s",string);

ip=string;

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

if(S() && \*ip=='\0'){

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

printf("\n String is accepted\n");

}

else{

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

printf("String not accepted\n");

}

}

int S()

{

if(\*ip=='(')

{

printf("%s\tS->(L) \n",ip);

ip++;

if(L())

{

if(\*ip==')')

{

ip++;

return 1;

}

else

return 0;

}

else

return 0;

}

else if(\*ip=='a')

{

ip++;

printf("%s\tS->a \n",ip);

return 1;

}

else

return 1;

}

int L(){

printf("%s\tL->SL' \n",ip);

if(S())

{

if(Ldash())

{

return 1;

}

else

return 0;

}

else

return 0;

}

int Ldash(){

if(\*ip==',')

{

printf("%s\tL'->,SL' \n",ip);

ip++;

if(S())

{

if(Ldash()){

return 1;

}

else

return 0;

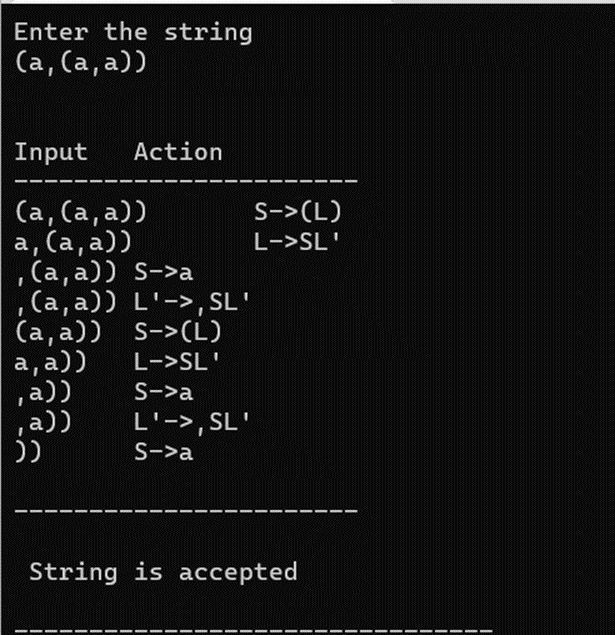
}

else

return 0;

    }

}



**7. Lab Assignment: Construct Recursive Descent Parser for the grammar**

**G = ({S, L}, {(, ), a, ,}, {S (L) | a ; L L, S | S}, S) and verify the acceptability of the**

**following strings:**

**i. (a,(a,a))**

**ii. (a,((a,a),(a,a)))**

#include <stdio.h>

#include <string.h>

#define SIZE 100

char s[SIZE];

int i=0;

void S(), L(), L\_();

int main() {

printf("Enter a string: ");

gets(s);

S();

if(i==strlen(s)) {

printf("\nString parsed.");

} else {

printf("Error in parsing string.");

}

}

void S() {

if(s[i]=='(') {

i++;

L();

if(s[i]==')') {

i++;

}

} else if(s[i]=='a') {

i++;

}

}

void L() {

S();

L\_();

}

void L\_() {

if(s[i]==',') {

i++;

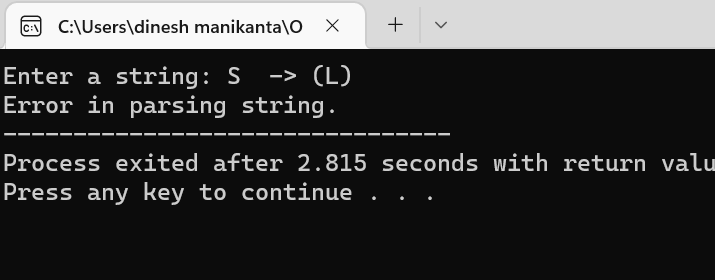
S();

L\_();

}

return;

}



**8. Write a C program for the computation of FIRST and FOLLOW for a given CFG**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

// Functions to calculate Follow

void followfirst(char, int, int);

void follow(char c);

// Function to calculate First

void findfirst(char, int, int);

int count, n = 0;

// Stores the final result

// of the First Sets

char calc\_first[10][100];

// Stores the final result

// of the Follow Sets

char calc\_follow[10][100];

int m = 0;

// Stores the production rules

char production[10][10];

char f[10], first[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;

count = 8;

// The Input grammar

strcpy(production[0], "E=TR");

strcpy(production[1], "R=+TR");

strcpy(production[2], "R=#");

strcpy(production[3], "T=FY");

strcpy(production[4], "Y=\*FY");

strcpy(production[5], "Y=#");

strcpy(production[6], "F=(E)");

strcpy(production[7], "F=i");

int kay;

char done[count];

int ptr = -1;

// Initializing the calc\_first array

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;

// Checking if First of c has

// already been calculated

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

if(c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

// Function call

findfirst(c, 0, 0);

ptr += 1;

// Adding c to the calculated list

done[ptr] = c;

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

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

// Printing the First Sets of the grammar

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;

// Initializing the calc\_follow array

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;

// Checking if Follow of ck

// has already been calculated

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

if(ck == donee[kay])

xxx = 1;

if (xxx == 1)

continue;

land += 1;

// Function call

follow(ck);

ptr += 1;

// Adding ck to the calculated list

donee[ptr] = ck;

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

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

// Printing the Follow Sets of the grammar

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++;

}

}

void follow(char c)

{

int i, j;

// Adding "$" to the follow

// set of the start symbol

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')

{

// Calculate the first of the next

// Non-Terminal in the production

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

}

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

{

// Calculate the follow of the Non-Terminal

// in the L.H.S. of the production

follow(production[i][0]);

}

}

}

}

}

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

{

int j;

// The case where we

// encounter a Terminal

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))

{

// Recursion to calculate First of New

// Non-Terminal we encounter after epsilon

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

}

else

first[n++] = '#';

}

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

{

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

}

else

{

// Recursion to calculate First of

// New Non-Terminal we encounter

// at the beginning

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

}

}

}

}

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

{

int k;

// The case where we encounter

// a Terminal

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;

}

//Including the First set of the

// Non-Terminal in the Follow of

// the original query

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

{

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

{

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

}

else

{

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

{

// Case where we reach the

// end of a production

follow(production[c1][0]);

}

else

{

// Recursion to the next symbol

// in case we encounter a "#"

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

}

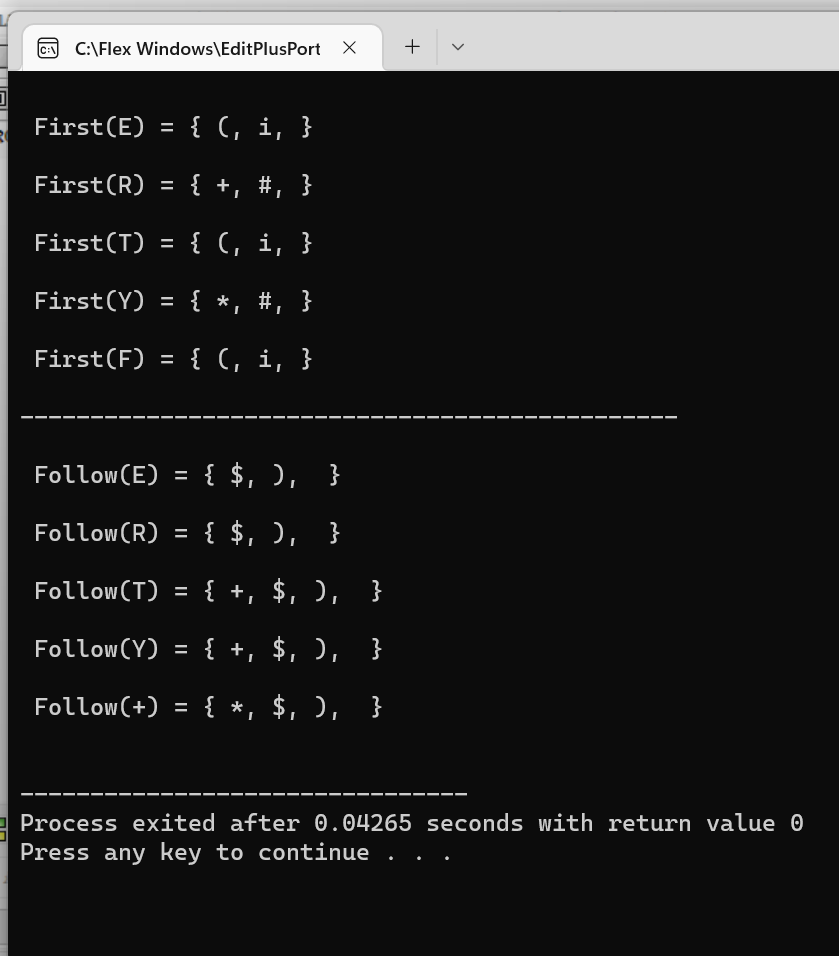
}

j++;

}

}

}



**Week 8: Predictive Parser**

**9. Implement non-recursive Predictive Parser for the grammar**

**S -&gt; aBa**

**B -&gt; bB | ε**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

int i=0,top=0;

char stack[20],ip[20];

void push(char c)

{

if (top>=20)

printf("Stack Overflow");

else

stack[top++]=c;

}

void pop(void)

{

if(top<0)

printf("Stack underflow");

else

top--;

}

void error(void)

{

printf("\n\nSyntax Error!!!! String is invalid\n");

getch();

exit(0);

}

int main()

{

int n;

printf("The given grammar is\n\n");

printf("S -> aBa\n");

printf("B -> bB | epsilon \n\n");

printf("Enter the string to be parsed:\n");

scanf("%s",ip);

n=strlen(ip);

ip[n]='$';

ip[n+1]='\0';

push('$');

push('S');

while(ip[i]!='\0')

{ if(ip[i]=='$' && stack[top-1]=='$')

{

printf("\n\n Successful parsing of string \n");

return(1);

}

else

if(ip[i]==stack[top-1])

{

printf("\nmatch of %c occured ",ip[i]);

i++;pop();

}

else

{

if(stack[top-1]=='S' && ip[i]=='a')

{

printf(" \n S ->aBa");

pop();

push('a');

push('B');

push('a');

}

else

if(stack[top-1]=='B' && ip[i]=='b')

{

printf("\n B ->bB");

pop();push('B');push('b');

}

else

if(stack[top-1]=='B' && ip[i]=='a')

{

printf("\n B -> epsilon");

pop();

}

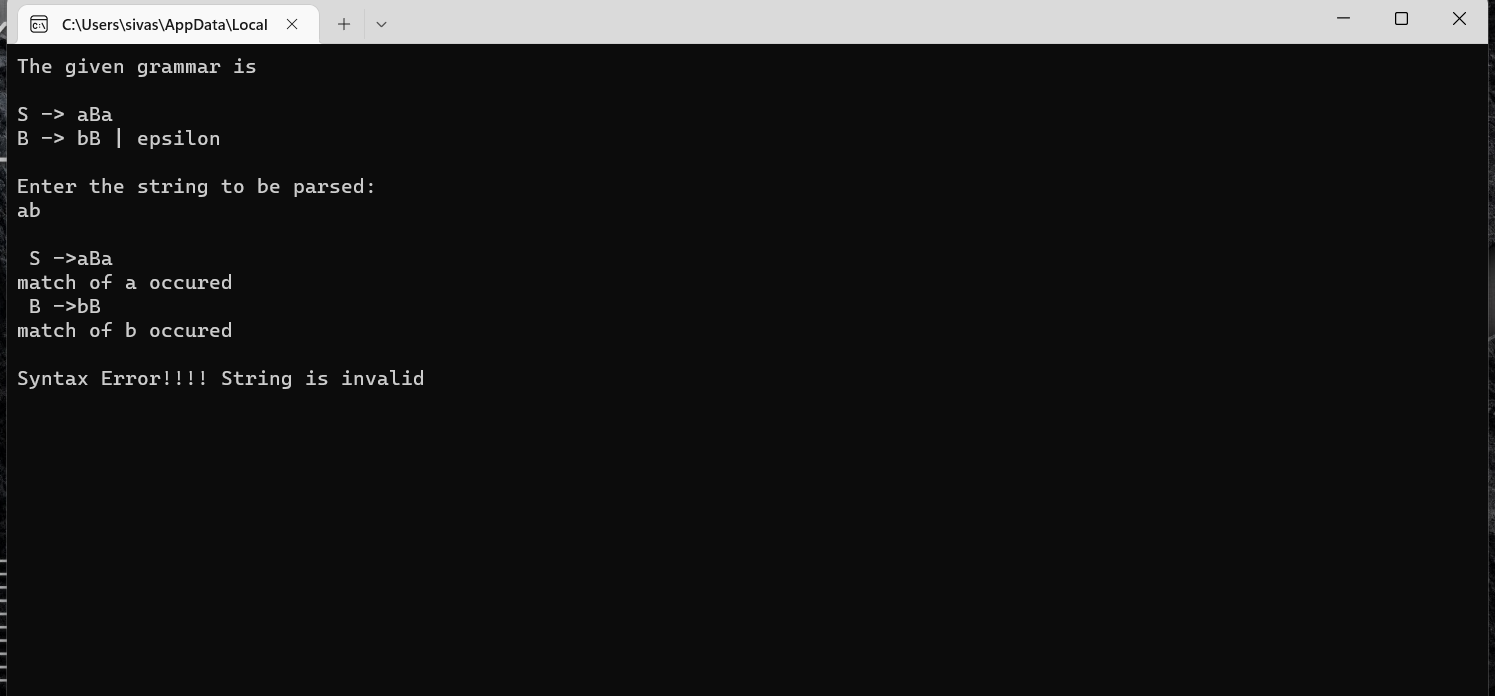
else

error();

}

}

}//end of main



**10. Lab Assignment: Implement Predictive Parser using C for the Expression Grammar**

**E TE’**

**E’ +TE’ | ε**

**T FT’**

**T’ \*FT’ | ε**

**F (E) | d**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

int i=0,top=0;

char stack[20],ip[20];

void push(char c)

{

if(top>=20)

{

printf("Stack Overflow");

}

else

{

stack[top++]=c;

}

}

void pop(void)

{

if(top<0)

{

printf("stack underflow");

}

else

{

top--;

}

}

void error(void)

{

printf("\n syntax error");

getch();

exit(0);

}

int main()

{

int n;

printf("The given grammar is\n\n");

printf("E->TE'\n");

printf("E'->+TE'|epsilon\n");

printf("T->FT'\n");

printf("T'->\*FT'|epsilon\n");

printf("F->(E)|d\n\n");

printf("Enter the string to be parsed:\n");

scanf("%s",ip);

n=strlen(ip);

ip[n]='$';

ip[n+1]='\0';

push('$');

push('E');

while(ip[i]!='\0')

{

if(ip[i]=='$'&&stack[top-1]=='$')

{

printf("\n\n successful parsing of string\n");

return 1;

}

else if(ip[i]==stack[top-1])

{

printf("\nMatch of %c",ip[i]);

i++;

pop();

}

else

{

if(stack[top-1]=='E'&&(ip[i]=='d'||ip[i]=='('))

{

printf("\n E->TE'");

pop();

push('A');

push('T');

}

else if(stack[top-1]=='A'&&ip[i]=='+')

{

printf("\nE'->+TE'");

pop();

push('A');

push('T');

push('+');

}

else if(stack[top-1]=='A'&&(ip[i]==')'||ip[i]=='$'))

{

printf("E'->epsilon");

pop();

}

else if(stack[top-1]=='T'&&(ip[i]=='d'||ip[i]=='('))

{

printf("\n T->FT'");

pop();

push('B');

push('F');

}

else if(stack[top-1]=='B'&&ip[i]=='\*')

{

printf("\n T->\*FT'");

pop();

push('B');

push('F');

push('\*');

}

else if(stack[top-1]=='B'&&(ip[i]=='+'||ip[i]==')'||ip[i]=='$'))

{

printf("\nT'->epsilon");

pop();

}

else if(stack[top-1]=='F'&&ip[i]=='d')

{

printf("\n F->d");

pop();

push('d');

}

else if(stack[top-1]=='F'&&ip[i]=='(')

{

printf("\n F->(E)");

pop();

push(')');

push('E');

push('(');

}

else

{

error();

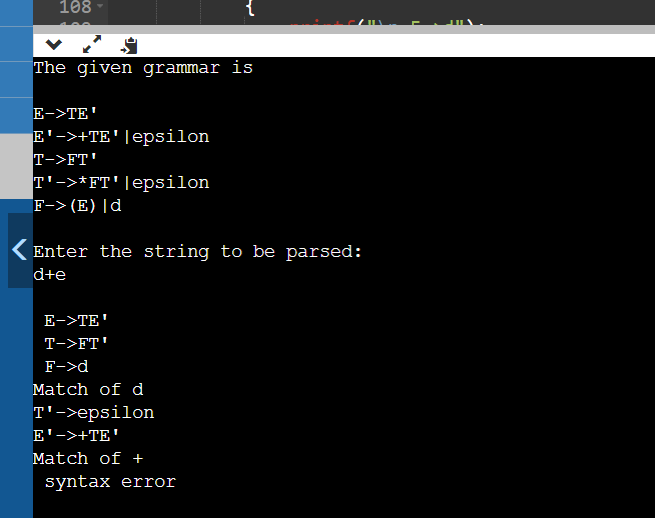
}

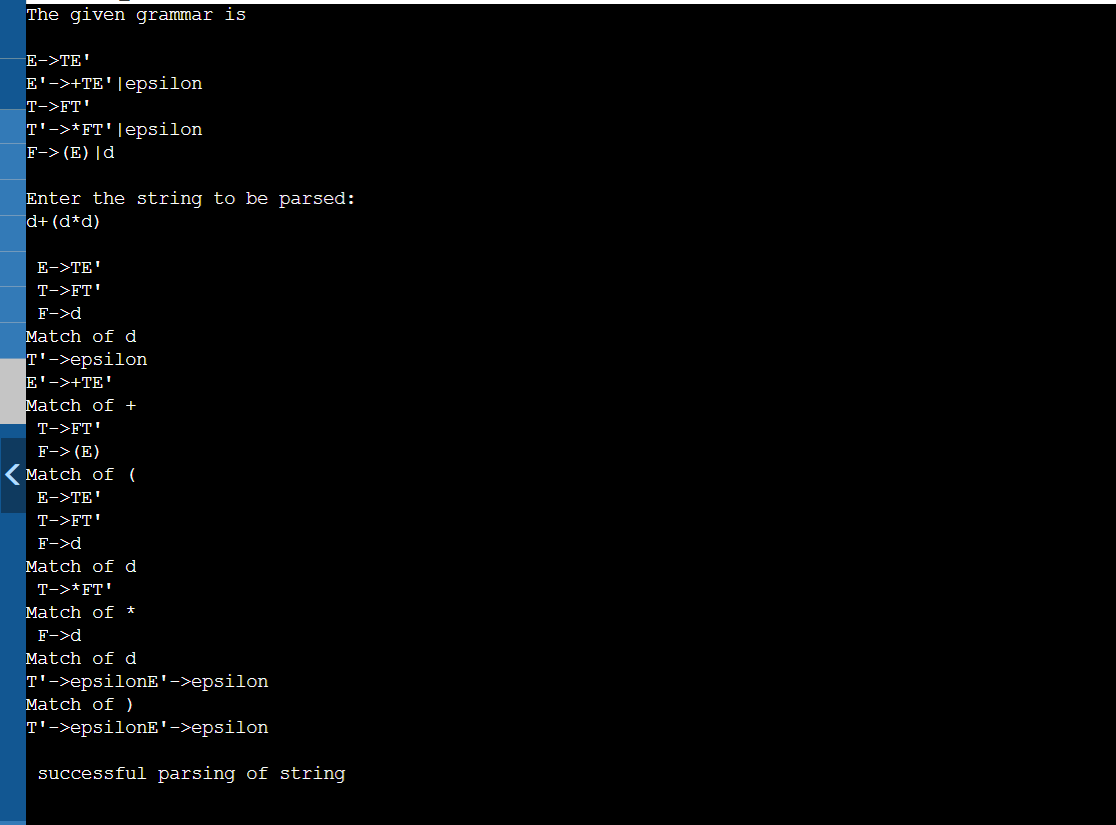
}

}

}

OUTPUT:





**Week 9: Shift Reduce Parser**

**11. Implementation of Shift Reduce parser using C for the following grammar and illustrate**

**the parser’s actions for a valid and an invalid string.**

**EE+E**

**EE\*E**

**E(E)**

**Ed**

#include<stdio.h>

#include<stdlib.h>

void pop(),push(char),display();

char stack[100]="\0";

char inputbuffer[100];

int top=-1;

char \*ip;

void main()

{

printf("E->E+E\n");

printf("E\_>E\*E\n");

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

printf("E->d\n");

printf("Enter the input string followed by $\n");

scanf("%s",inputbuffer);

ip=inputbuffer;

push('$');

printf("STACK\tBUFFER\tACTION\n");

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

display();

do

{

if((stack[top]=='E'&&stack[top-1]=='$')&&(\*(ip)=='$'))

break;

if(stack[top]=='$')

{

push(\*ip);

ip++;

printf("shift");

}

else if(stack[top]=='d')

{

display();

pop();

push('E');

printf("Reduce E->d\n");

}

else if(stack[top]=='E'&&stack[top-1]=='+'&&stack[top-2]=='E'&&ip!=' ')

{

display();

pop();

pop();

pop();

push('E');

printf("Reduce E->E+E");

}

else if(stack[top]=='E'&&stack[top-1]=='\*'&&stack[top-2]=='E')

{

display();

pop();

pop();

pop();

push('E');

printf("Reduce E->E\*E");

}

else if(stack[top]==')'&&stack[top-1]=='E'&&stack[top-2]=='(')

{

display();

pop();

pop();

pop();

push('E');

printf("Reduce E->(E)");

}

else

{

display();

push(\*ip);

ip++;

printf("shift");

}

}

while(1);

display();

printf("Accept\n\n\n");

}

void push(char c)

{

top++;

stack[top]=c;

}

void pop()

{

stack[top]='\0';

top--;

}

void display()

{

printf("\n%s\t%s\t",stack,ip);

}

OUTPUT:



**12. Lab Assignment: Implementation of Shift Reduce parser using C for the following**

#include<stdio.h>

#include<stdlib.h>

void pop(),push(char),display();

char stack[100]="\0", input[100], \*ip;

int top=-1;

void push(char c)

{

top++;

stack[top]=c;

}

void pop()

{

stack[top]='\0';

top--;

}

void display()

{

printf("\n%s\t%s\t",stack,ip);

}

void main()

{

printf("S->0S0\n");

printf("S->1S1\n");

printf("S->2\n");

printf("Enter the input string followed by $ \n");

scanf("%s",input);

ip=input;

push('$');

printf("STACK\t BUFFER \t ACTION\n");

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

display();

if(stack[top]=='$' && \*ip=='$'){

printf("Null Input");

exit(0);

}

do

{

if((stack[top]=='S' && stack[top-1]=='$') && (\*(ip)=='$'))

{

display();

printf(" Valid\n\n\n");

break;

}

if(stack[top]=='$')

{

push(\*ip);

ip++;

printf("Shift");

}

else if(stack[top]=='2')

{

display();

pop();

push('S');

printf("Reduce S->2");

}

else if(stack[top]=='0' && stack[top-1]=='S' && stack[top-2]=='0')

{

display();

pop();

pop();

pop();

push('S');

printf("Reduce S->0S0");

}

else if(stack[top]=='1' && stack[top-1]=='S' && stack[top-2]=='1')

{

display();

pop();

pop();

pop();

push('S');

printf("Reduce S->1S1");

}

else if(\*ip=='$')

{ printf(" Invalid\n\n\n");

break;

}

else

{

display();

push(\*ip);

ip++;

printf("shift");

}

}while(1);

}

**Week 10: LALR Parser**

**13. Implement LALR parser using LEX and YACC for the following Gramma**r:

%{

#include<stdio.h>

%}

%token NUMBER

%%

S: E { printf("The result is =%d\n",$1);}

;

E: E'+'T { $$ = $1 + $3; }

| T { $$ = $1;}

;

T: T'\*'F { $$ = $1 \* $3; }

| F { $$ = $1;}

;

F: '('E')' { $$ = $2;}

| NUMBER { $$ = $1;}

;

%%

int main(){

yyparse();

}

int yywrap(){

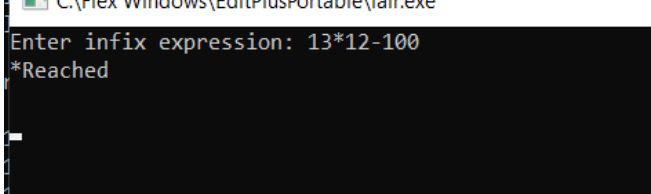
return 1;

}

void yyerror(char \*s){

printf("Error %s",s);

}



14

%{

#include<stdio.h>

%}

%token DIGIT

%left '-' '+'

%left '\*' '/'

%nonassoc UMINUS

%%

S: E { printf("The result is =%d\n",$1);}

;

E: E'+'E { $$ = $1 + $3; }

| E'-'E { $$ = $1 - $3; }

| E'\*'E { $$ = $1 \* $3; }

| E'/'E { if($3 == 0)

yyerror("Divide by zero");

else

$$ = $1 / $3; }

| '-'E %prec UMINUS { $$ = -$2; }

| '(' E ')' { $$ = $2; }

| DIGIT {$$ = $1;}

;

%%

int main()

{

yyparse();

}

int yywrap(){

return 1;

}

void yyerror(char \*s){

printf("Error %s",s);exit(0);

}