**Practical 1:-Write a program to implements a lexical analyzer**

**for c Language.**

**Code:-**

#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

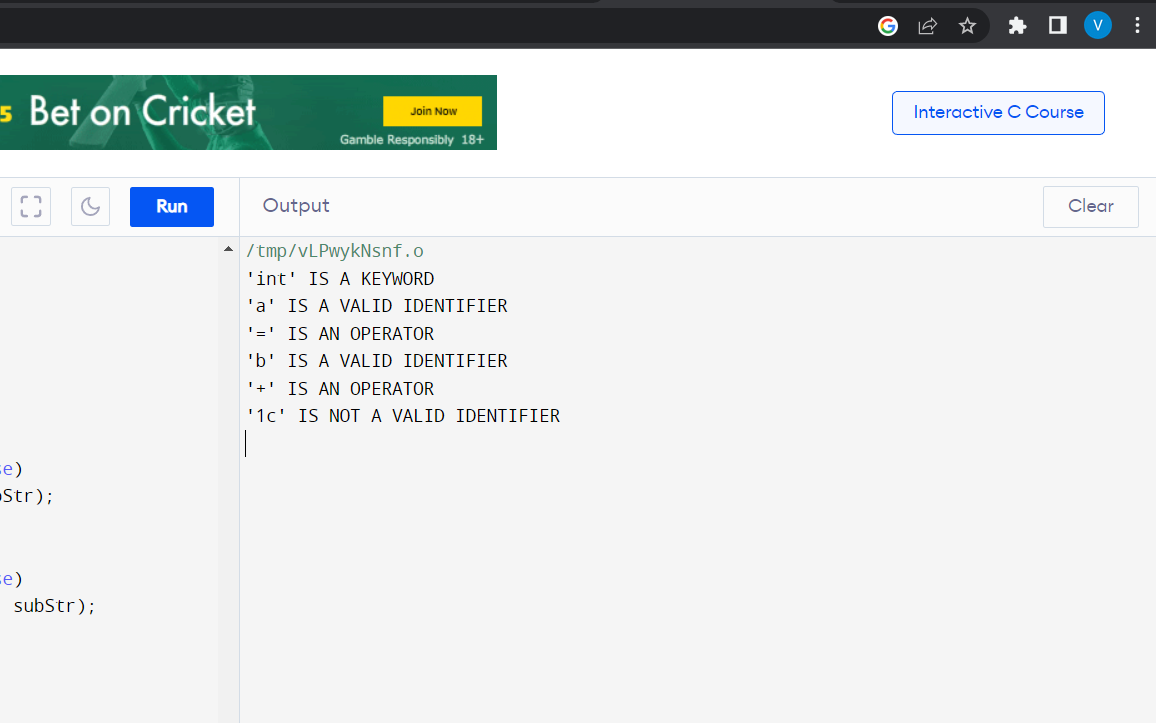
char str[100] = "int a = b + 1c; ";

parse(str); // calling the parse function

return (0);

}

**Output:-**



**Practical 2:-Write program to check validity of the input**

**string for fix finite automata.**

**Code:-**

#include <stdio.h>

/\* This structure represents a state in the finite automaton. \*/

struct state {

int is\_accepting;

int next[26];

};

/\* This function initializes the finite automaton. \*/

void init\_automaton(struct state \*automaton) {

for (int i = 0; i < 26; i++) {

automaton->next[i] = -1;

}

automaton->is\_accepting = 0;

}

/\* This function adds a transition to the finite automaton. \*/

void add\_transition(struct state \*automaton, int from, int to, char c) {

automaton->next[c - 'a'] = to;

}

/\* This function checks if the input string is accepted by the finite automaton. \*/

int is\_accepted(struct state \*automaton, char \*string) {

int state = 0;

for (int i = 0; string[i] != '\0'; i++) {

int next\_state = automaton->next[string[i] - 'a'];

if (next\_state == -1) {

return 0;

}

state = next\_state;

}

return automaton->is\_accepting;

}

int main() {

/\* Create the finite automaton. \*/

struct state automaton;

init\_automaton(&automaton);

/\* Add transitions to the finite automaton. \*/

add\_transition(&automaton, 0, 1, 'a');

add\_transition(&automaton, 1, 2, 'b');

add\_transition(&automaton, 2, 3, 'c');

automaton.is\_accepting = 1;

/\* Check if the input string is accepted by the finite automaton. \*/

char string[] = "abc";

if (is\_accepted(&automaton, string)) {

printf("The string is accepted.\n");

} else {

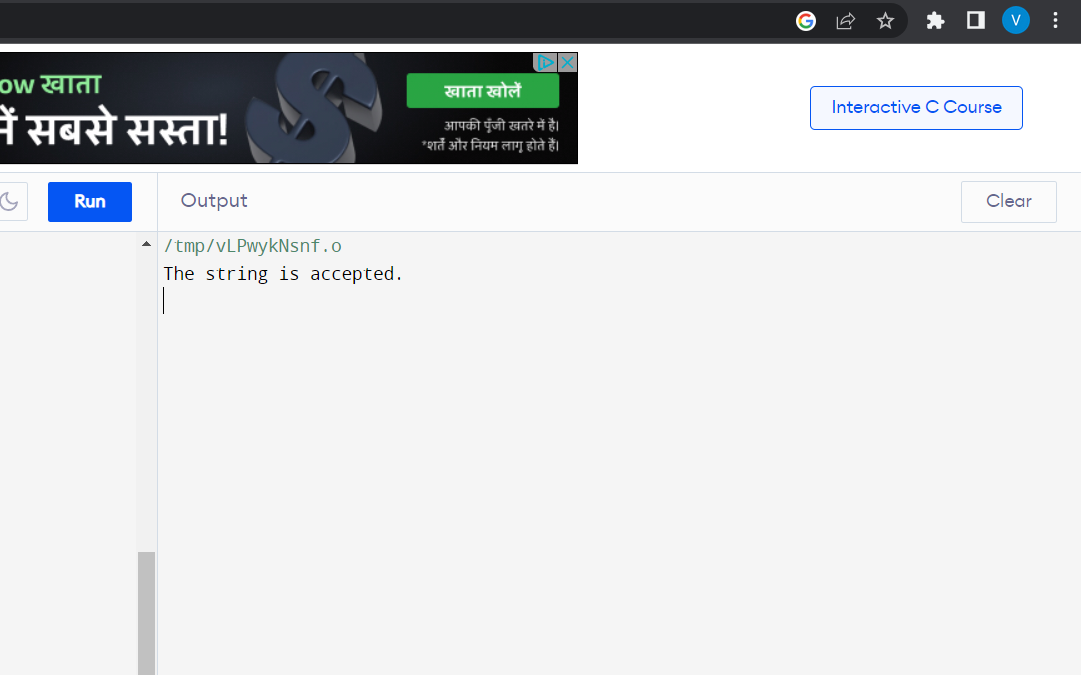
printf("The string is not accepted.\n");

}

return 0;

}

**Output:-**

****

**Practical 3:-Write a program to left factor the given grammar.**

**Code:-**

#include <stdio.h>

/\* This function left factors the given grammar. \*/

void left\_factor(char \*grammar) {

/\* Create a new grammar. \*/

char new\_grammar[1000];

memset(new\_grammar, 0, sizeof(new\_grammar));

/\* Iterate over the given grammar. \*/

for (int i = 0; grammar[i] != '\0'; i++) {

/\* If the current character is a non-terminal, then add it to the new grammar. \*/

if (isupper(grammar[i])) {

new\_grammar[strlen(new\_grammar)] = grammar[i];

} else if (grammar[i] == ' ') {

/\* If the current character is a space, then ignore it. \*/

continue;

} else {

/\* If the current character is a terminal, then add it to the new grammar followed by the

non-terminal. \*/

new\_grammar[strlen(new\_grammar)] = grammar[i];

new\_grammar[strlen(new\_grammar) + 1] = grammar[i + 1];

i++;

}

}

/\* Replace the old grammar with the new grammar. \*/

strcpy(grammar, new\_grammar);

}

int main() {

/\* Get the grammar from the user. \*/

char grammar[1000];

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

fgets(grammar, 1000, stdin);

/\* Left factor the grammar. \*/

left\_factor(grammar);

/\* Print the left-factored grammar. \*/

printf("The left-factored grammar is:\n");

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

return 0;

}

**Output:-**

****

**Practical 4:-Write a program to remove the Left Recursion from a given grammar.**

**Code:-**

#include<stdio.h>

#include<string.h>

int main()

{

char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];

int i,j=0,k=0,l=0,pos;

printf("Enter Production : A->");

gets(gram);

for(i=0;gram[i]!='|';i++,j++)

part1[j]=gram[i];

part1[j]='\0';

for(j=++i,i=0;gram[j]!='\0';j++,i++)

part2[i]=gram[j];

part2[i]='\0';

for(i=0;i<strlen(part1)||i<strlen(part2);i++){

if(part1[i]==part2[i]){

modifiedGram[k]=part1[i];

k++;

pos=i+1;

}

}

for(i=pos,j=0;part1[i]!='\0';i++,j++){

newGram[j]=part1[i];

}

newGram[j++]='|';

for(i=pos;part2[i]!='\0';i++,j++){

newGram[j]=part2[i];

}

modifiedGram[k]='X';

modifiedGram[++k]='\0';

newGram[j]='\0';

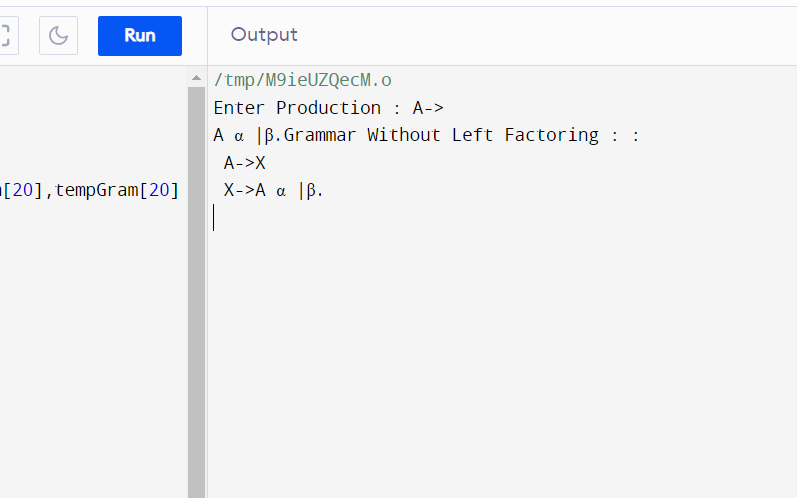
printf("\nGrammar Without Left Factoring : : \n");

printf(" A->%s",modifiedGram);

printf("\n X->%s\n",newGram);

}

**Output:-**



**Practical 5:-Write a program to find First and Follow from the give set of production rules.**

**Code:-**

#include <stdio.h>

/\* This structure represents a production in the grammar. \*/

struct production {

char \*non\_terminal;

char \*rhs;

};

/\* This function finds First and Follow from the given set of production rules. \*/

void find\_first\_and\_follow(struct production \*productions, int num\_productions) {

/\* Create a table to store First and Follow sets. \*/

int first[num\_productions];

int follow[num\_productions];

memset(first, 0, sizeof(first));

memset(follow, 0, sizeof(follow));

/\* Initialize First and Follow sets. \*/

for (int i = 0; i < num\_productions; i++) {

first[i] = 0;

follow[i] = 0;

}

/\* Iterate over the given grammar. \*/

for (int i = 0; i < num\_productions; i++) {

/\* If the current production is for a terminal, then add the terminal to the First set. \*/

if (isupper(productions[i].rhs[0])) {

first[i] = productions[i].rhs[0];

} else {

/\* If the current production is for a non-terminal, then add all of the terminals in the First set

of its RHS to the First set. \*/

for (int j = 0; productions[i].rhs[j] != '\0'; j++) {

first[i] = union(first[i], first[productions[i].rhs[j] - 'A']);

}

}

/\* If the current production is for the start symbol, then add '$' to the Follow set. \*/

if (productions[i].non\_terminal == productions[0].non\_terminal) {

follow[i] = '$';

}

/\* Iterate over the rest of the productions. \*/

for (int j = 0; j < num\_productions; j++) {

/\* If the current production is for a non-terminal that appears on the RHS of the current

production, then add all of the terminals in the Follow set of the current production to the Follow

set of the non-terminal. \*/

if (strchr(productions[j].rhs, productions[i].non\_terminal) != NULL) {

follow[productions[j].non\_terminal - 'A'] = union(follow[productions[j].non\_terminal - 'A'],

follow[i]);

}

}

}

/\* Print the First and Follow sets. \*/

for (int i = 0; i < num\_productions; i++) {

printf("First[%c] = %d\n", productions[i].non\_terminal, first[i]);

printf("Follow[%c] = %d\n", productions[i].non\_terminal, follow[i]);

}

}

int main() {

/\* Get the grammar from the user. \*/

struct production productions[100];

int num\_productions;

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

for (int i = 0; i < 100; i++) {

scanf("%s", productions[i].non\_terminal);

scanf("%s", productions[i].rhs);

if (productions[i].non\_terminal[0] == '\0') {

break;

}

num\_productions++;

}

/\* Find First and Follow from the grammar. \*/

find\_first\_and\_follow(productions, num\_productions);

return 0;

}

**Output:-**

**Practical 6:-Write an assembly language program in a text file and generate symbol table ,literal table and pool table.**

**Code:-**

#include<stdio.h>

#include<conio.h>

struct intermediate

{

int addr;

char label[10];

char mnem[10];

char op[10];

}res;

struct symbol{

char symbol[10];

int addr;

}sy;

void main()

{

FILE \*s1,\*p1;

clrscr();

s1=fopen("inter.txt","r+");

p1=fopen("symbol.txt","w");

while(!feof(s1))

{

fscanf(s1,"%d %s %s

%s",&res.addr,res.label,res.mnem,res.op);

if(strcmp(res.label,"NULL")!=0)

{

strcpy(sy.symbol,res.label);

sy.addr=res.addr;

fprintf(p1,"%s\t%d\n",sy.symbol,sy.addr);

}

}

fcloseall();

printf("symbol table created");

getch();

}

**Output:-**

**Input Files:-**

0 NULL START 150

150 A DS 200

250 B DC 50

300 FIRST PRINT A

500 NULL READ B

550 NULL END FIRST

**Output Files:**

A 500

B 600

FIRST 610

**Practical 7:-Write a program to demonstrate the use of macro.**

**Code:-**

#include <stdio.h>

/\* This macro prints the given message. \*/

#define PRINT\_MESSAGE(message) printf("%s\n", message)

int main() {

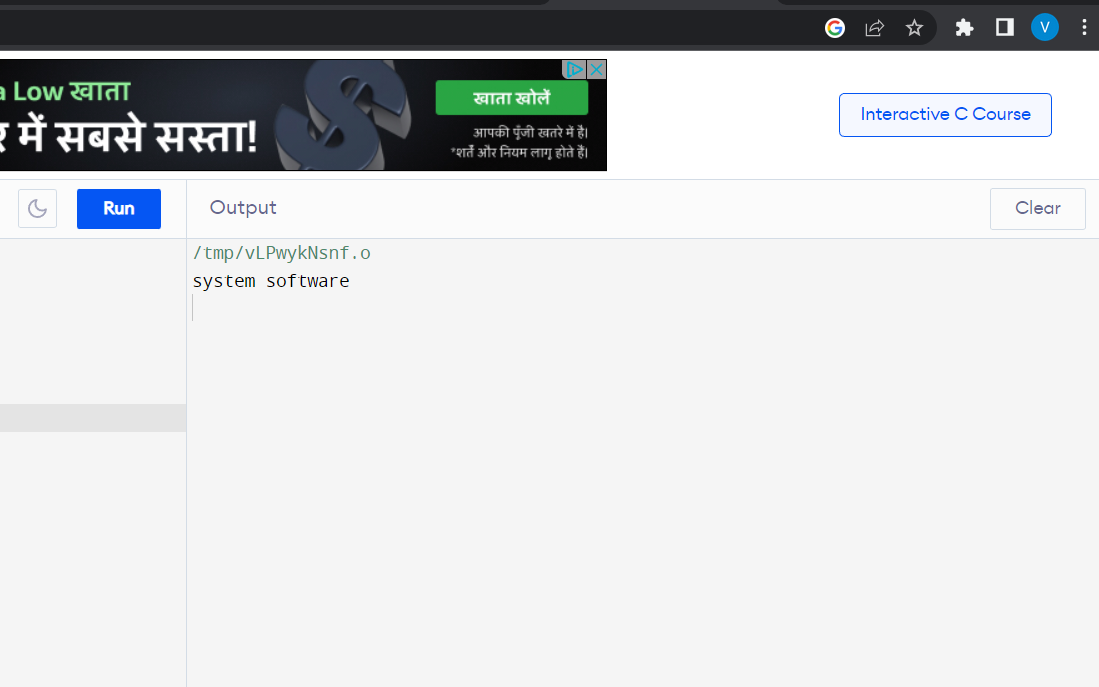
/\* Print a message using the macro. \*/

PRINT\_MESSAGE("system software");

return 0;

}

**Output:-**

****

**Practical 8:-Write program that generates quadruple table for the given postfix string.**

**Code:-**#include<stdio.h>

#include<string.h>

int main()

{

char line[20];

int s[20];

int t=1;

int i=0;

printf("Enter string.. :");

for(i=0;i<20;i++)s[i]=0;

printf("op\ta1\ta2\tres\n");

for(i=2;line[i]!='\0';i++)

{

if(line[i]=='/' || line[i]=='\*')

{

printf("\n");

if(s[i]==0)

{

if(s[i+1]==0)

{

printf(":=\t%c\t\t t%d\n",line[i+1],t);

s[i+1]=t++;

}

printf("%c\t",line[i]);

(s[i-1]==0)?printf("%c\t",line[i-1]):printf("t%d\t",s[i-1]);

printf("t%d \t t%d",s[i+1],t);

s[i-1]=s[i+1]=t++;

s[i]=1;

}

}

}

gets(line);

for(i=2;line[i]!='\0';i++)

{

if(line[i]=='+' || line[i]=='-')

{

printf("\n");

if(s[i]==0)

{

if(s[i+1]==0)

{

printf(":=\t%c\t\t t%d\n",line[i+1],t);

s[i+1]=t++;

}

printf("%c\t",line[i]);

(s[i-1]==0)?printf("%c\t",line[i-1]):printf("t%d\t",s[i-1]);

printf("t%d \t t%d",s[i+1],t);

s[i-1]=s[i+1]=t++;

s[i]=1;

}

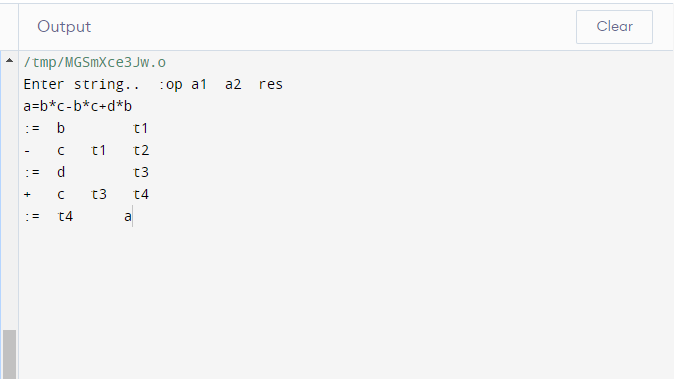
}

}

printf("\n:=\tt%d\t\t%c",t-1,line[0]);

}

**Output:-**

****

**Practical 9:-Write Lex program to count number of vowels and consonants in a given string.**

**Code:-**

%{

#include <stdio.h>

int vow\_count=0;

int const\_count =0;

%}

%%

[aeiouAEIOU] {vow\_count++;}

[a-zA-Z] {const\_count++;}

%%

int yywrap(){}

int main() {

/\* Get the string from the user. \*/

char str[1000];

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

fgets(str, 1000, stdin);

/\* Count the number of vowels and consonants in the string. \*/

yylex();

printf("Number of vowels are: %d\n", vow\_count);

printf("Number of consonants are: %d\n", const\_count);

return 0;

}

**Practical 10:-Write Lex program to count the number of characters, word, space and lines.**

**Code:-**

%{

#include <stdio.h>

int char\_count = 0;

int word\_count = 0;

int space\_count = 0;

int line\_count = 0;

%}

%%

[\n] {line\_count++;}

[ ] {space\_count++;}

[a-zA-Z0-9]+ {word\_count++; char\_count += strlen(yytext);}

. {char\_count++;}

%%

int yywrap(){}

int main() {

/\* Get the string from the user. \*/

char str[1000];

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

fgets(str, 1000, stdin);

/\* Count the number of characters, words, spaces, and lines. \*/

yylex();

printf("Number of characters are: %d\n", char\_count);

printf("Number of words are: %d\n", word\_count);

printf("Number of spaces are: %d\n", space\_count);

printf("Number of lines are: %d\n", line\_count);

return 0;

}

**Practical 11:-Write Lex program to identify identifiers, constants and keywords(int,float) for the c language.**

**Code:-**

%{

#include <stdio.h>

int id\_count = 0;

int const\_count = 0;

int keyword\_count = 0;

%}

%%

[a-zA-Z][a-zA-Z0-9]\* {id\_count++; printf("Identifier: %s\n", yytext);}

[0-9]+ {const\_count++; printf("Integer constant: %s\n", yytext);}

[0-9]+\.[0-9]+ {const\_count++; printf("Float constant: %s\n", yytext);}

"int" {keyword\_count++; printf("Keyword: int\n");}

"float" {keyword\_count++; printf("Keyword: float\n");}

. {printf("Unknown token: %s\n", yytext);}

%%

int yywrap(){}

int main() {

/\* Get the string from the user. \*/

char str[1000];

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

fgets(str, 1000, stdin);

/\* Identify identifiers, constants and keywords in the string. \*/

yylex();

printf("Number of identifiers are: %d\n", id\_count);

printf("Number of constants are: %d\n", const\_count);

printf("Number of keywords are: %d\n", keyword\_count);

return 0;

}

**Practical 12:-Write Lex program to count and display single line and multiline comments for a c language.**

**Code:-**

%{

#include <stdio.h>

int single\_comment\_count = 0;

int multi\_comment\_count = 0;

%}

%%

\/\/.\* {single\_comment\_count++;}

\/\\* {BEGIN COMMENT;}

<COMMENT>\\*\/ {BEGIN INITIAL;}

<COMMENT>. {;}

%%

int yywrap(){}

int main() {

/\* Get the string from the user. \*/

char str[1000];

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

fgets(str, 1000, stdin);

/\* Count and display single line and multiline comments in the string. \*/

yylex();

printf("Number of single line comments are: %d\n", single\_comment\_count);

printf("Number of multiline comments are: %d\n", multi\_comment\_count);

return 0;

}