Compiler Design Lab Record

Meher Preetham Kommera

AP20110010348

CSE-E

Week 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.

a)

CODE:

```
#include<stdio.h>
#define max 100
int main() {
char s[max],f='a';
int i;
printf("eNTER tHE sTRING tO bE cHECKED: ");
scanf("%s",s);
for(i=0;s[i]!='\0';i++) {

switch(f) {
case 'a': if(s[i]=='0') f='b';
else if(s[i]=='1') f='d';
break;
case 'b': if(s[i]=='0') f='a';
else if(s[i]=='1') f='c';
break;
case 'c': if(s[i]=='0') f='d';
else if(s[i]=='1') f='b';
```

```
break;
case 'd': if(s[i]=='0') f='c';
else if(s[i]=='1') f='a';
break;
}
}
if(f=='a')
printf("\nsTRING iS aCCEPTED.");
else printf("\nsTRING iS nOT aCCEPTED.");
return 0;
}
```

Output:

```
eNTER tHE sTRING tO bE cHECKED: 001010011

sTRING iS nOT aCCEPTED.

...Program finished with exit code 0
Press ENTER to exit console.
```

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

code:

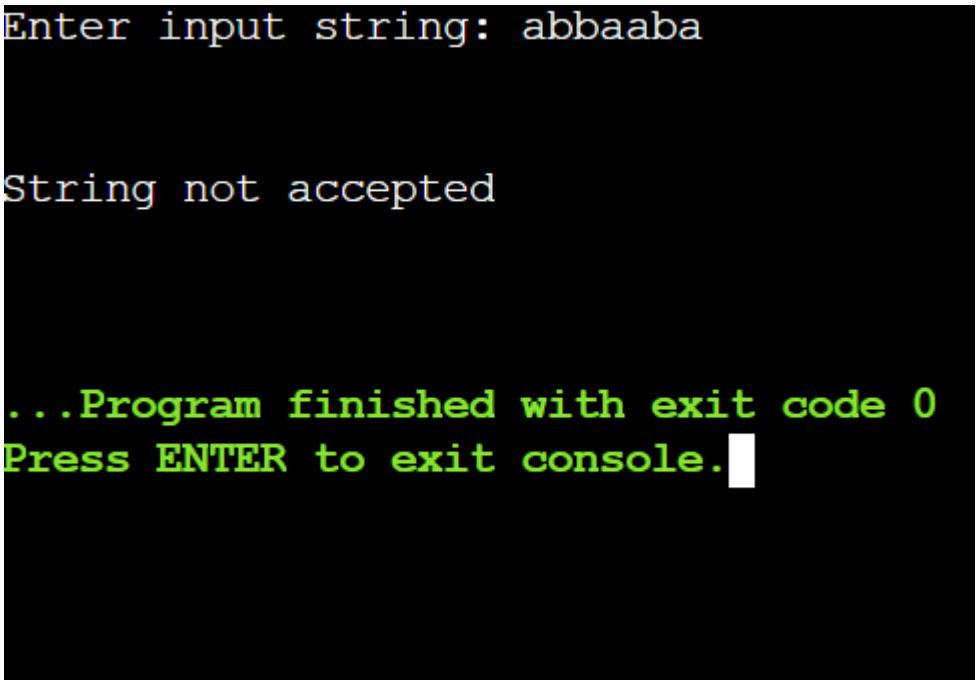
```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int state = 0, i = 0;
   char token, input[20];
   printf("Enter input string \t :");
   scanf("%s", input);
   //printf("Given string is : %s");
```

```
while ((token = input[i++]) != '\0')
{
   // printf("current token : %c \n",token);
   switch (state)
   {
   case 0:
      if (token == 'a')
         state = 1;
      else if (token == 'b')
         state = 2;
      else
      {
         printf("Invalid token");
         exit(0);
      }
      break;
   case 1:
      if (token == 'a')
         state = 3;
      else if (token == 'b')
         state = 2;
      else
      {
         printf("Invalid token");
         exit(0);
      }

      break;
   case 2:
      if (token == 'a')
```

```
        state = 1;
    else if (token == 'b')
        state = 4;
    else
    {
        printf("Invalid token");
        exit(0);
    }
    break;
case 3:
    if (token == 'a')
        state = 3;
    else if (token == 'b')
        state = 2;
    else
    {
        printf("Invalid token");
        exit(0);
    }
    break;
case 4:
    if (token == 'a')
        state = 1;
    else if (token == 'b')
        state = 4;
    else
    {
        printf("Invalid token");
        exit(0);
    }
    break;
```

```
    }
    // printf("state = %d  ",state);
  }
  if (state == 3 || state == 4)
    printf("\n\nString accepted\n\n");
  else
    printf("\n\nString not accepted\n\n");
}
```

Output:



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

A minimum of 10 keywords of your choice

Code:

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
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);
}
bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=')
        return (true);
    return (false);
}
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);
}
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")
```

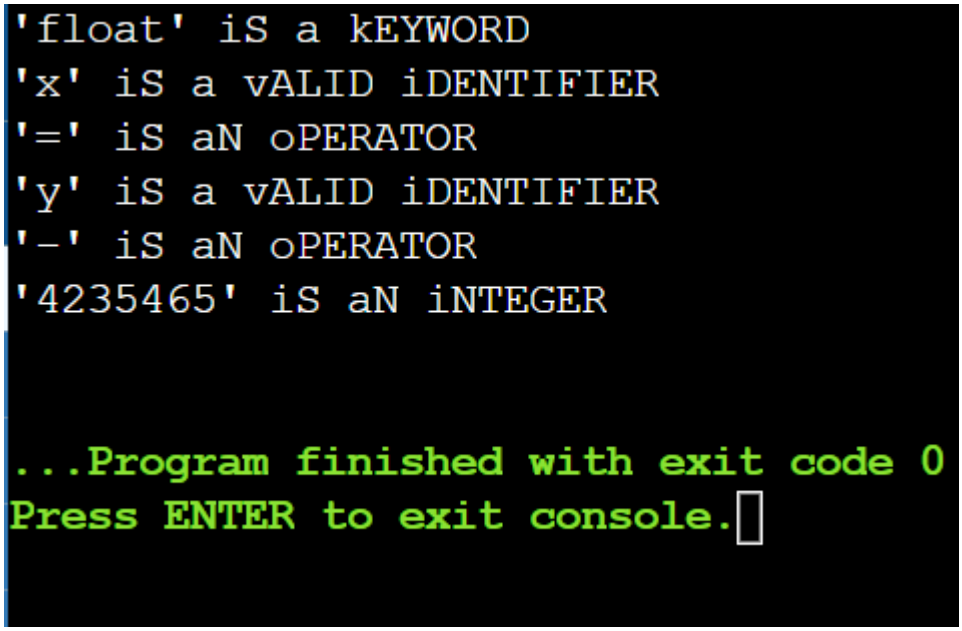
```
        || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char")
        || !strcmp(str, "sizeof") || !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef")
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (true);
    return (false);
}
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);
}
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);
}
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);
}
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;
}
int main()
{
    char str[100] = "float x  = y - 4235465";
    parse(str);
    return (0);
}
```

Output:



Week 3: Implement the following programs using Lex tool

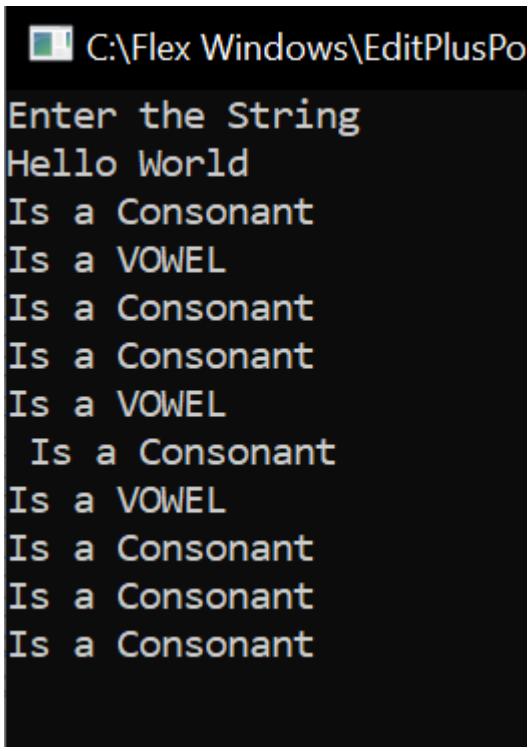
a. Identification of Vowels and Consonants

Code:

```
%{
#include<stdio.h>
int vowel=0;
int cons=0;
```

```
%}


%%
"a"|"e"|"i"|"o"|"u"|"A"|"E"|"I"|"O"|"U" {printf("iS a vOWEL.\n");vowel++;}
[a-zA-z] {printf("iS a cONSONENT.\n");cons++;}
%%
int yywrap()
{
return 1;
}
main()
{
printf("eNTER tHE sTRING: \n");
yylex();
printf("vOWEL = %d aND cONSONENT = %d", vowel, cons);
}
```

Output:

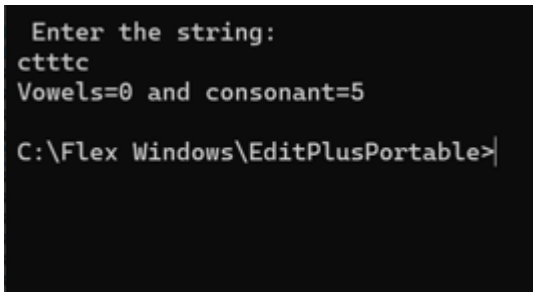


b) count number of vowels and consonants

code:

```
%{

int vc=0,cc=0;
%}
vowel [aeiou]+
consonant [^aeiou]
eol \n
%%
{eol} return 0;
[\t]+ ;
{vowel} {vc++;}
{consonant} {cc++;}

%%
int main()
{
printf("eNTER tHE sTRING: ");
yylex();
printf("\nvOWEL = %d aND cONSONENT = %d\n", vc, cc);
return 0;
}

int yywrap()

{

return 1;

}
```

Output:



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

□ A minimum of 10 keywords of your choice

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

□ Integers with the regular expression: digit+

□ Relational operators: <, >, <=, >=, ==, !=

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

□ Storing identifiers in symbol table

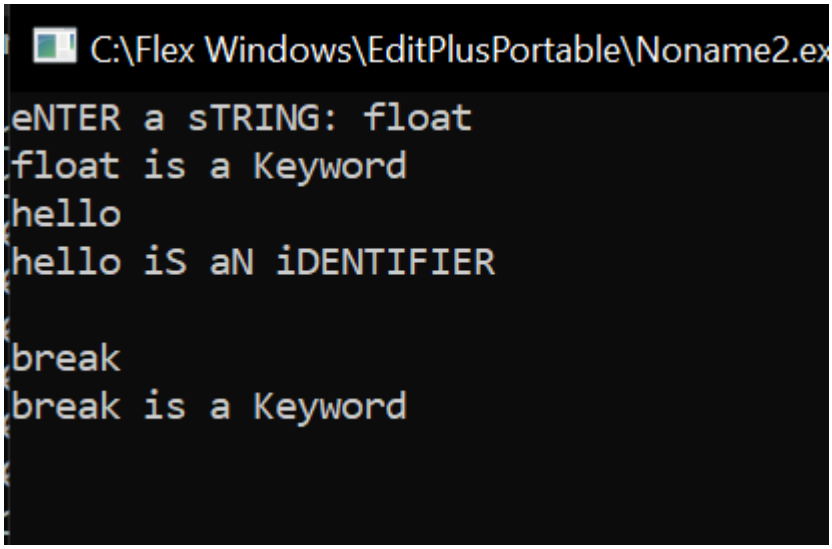
□ Using files for input and output.

Code:

```
%{
        #include<stdio.h>
%}

%%
auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern|retur
n|union|continue|for|signed|void|do|if|static|while|default|goto|sizeof|volatile|const|float|s
hort {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 rATIONAL oPERATOR\n",yytext);}
```

```
%%

int yywrap()
{
        return 1;
}

int main()
{
        printf("eNTER a sTRING: ");
        yylex();
        return 0;
}
```

Output:



Week 6: Implement Recursive Descent Parser for the Expression Grammar given below.

E → TE'

E'→ +TE' | є

T → FT'

T'→ *FT' | є

F → (E) | i

Code:

```
#include<stdio.h>
#include<string.h>
int S(),Ldash(),L();
 char *ip;
char string[50];
 int main()
 {
printf("Enter the string\n");
scanf("%s",string);
ip=string;
printf("\n\nInput\t\tAction\n");
if(S() && *ip=='\0')
 {
printf("\n String is successfully parsed\n");
 } else {
printf("Error in parsing String\n");
}
}
int S() {
if(*ip=='(')
{
printf("%s\t\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\t\tS->a \n",ip);
return 1;
}
 else {
return 0;
 }
}
int L()
{
printf("%s\t\tL->SL' \n",ip);
if(S())
{
 if(Ldash())
{
return 1;
}
else {
 return 0;
 }
}
else {
```

```
return 0;
}
}
int Ldash() {
 if(*ip==',')
 { printf("%s\t\tL'->,SL' \n",ip);
ip++;
if(S()) {
if(Ldash()) {
return 1;
 } else
 {
return 0;
}
}
else {
 return 0;
}
}
else {
printf("%s\t\tL'->^ \n",ip);
return 1;
}
}
```

Output:

```
/tmp/RgQlPR1XwQ.o
Enter the string
(a,(a,a))
Input       Action
(a,(a,a))       S->(L)
a,(a,a))        L->SL'
,(a,a))     S->a
,(a,a))     L'->,SL'
(a,a))      S->(L)
a,a))       L->SL'
,a))        S->a
,a))        L'->,SL'
))      S->a
))      L'->^
)       L'->^

 String is successfully parsed
```

Week 7: Write a C program for the computation of FIRST and FOLLOW for a given CFG

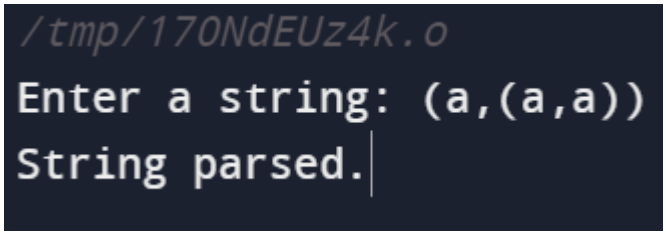
Code:

```
#include <stdio.h>
#include <string.h>

#define SIZE 100

char s[SIZE];

int a=0;

void S(), L(), L_();

int main() {
   printf("eNTER a sTRING: ");
```

```
    gets(s);
    S();

    if(a==strlen(s)) {
        printf("\nsTRING pARSED.");
    } else {
        printf("eRROR.");
    }
}

void S() {
    if(s[a]=='(') {
        a++;
        L();
        if(s[a]==')') {
            a++;
        }
    } else if(s[a]=='a') {
        a++;
    }
}

void L() {
    S();
    L_();
}

void L_() {
    if(s[a]==',') {
```

```
        i++;
        S();
        L_();
    }
    return;
}
```

Output:



Week 8: Implement non-recursive Predictive Parser for the grammar

S -> aBa

B -> bB | ε

a)

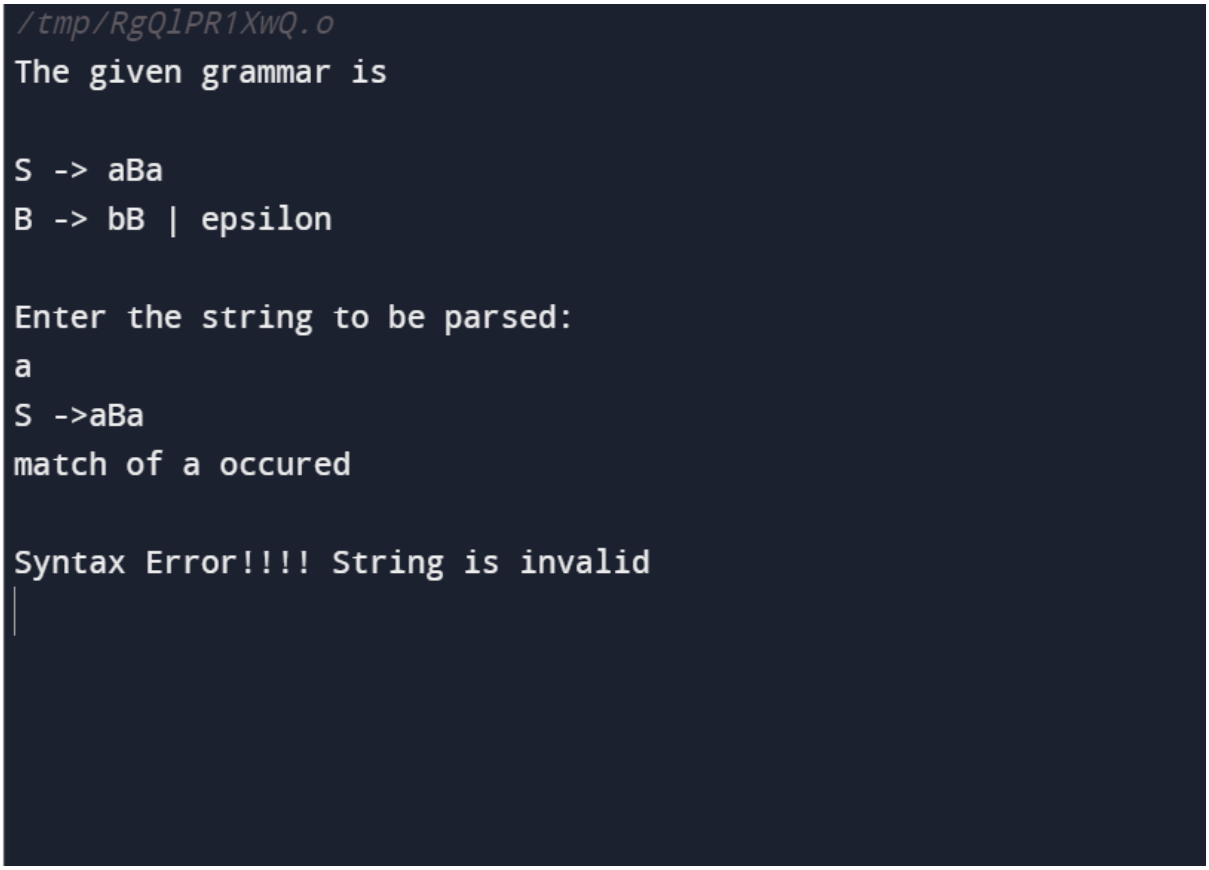
```
/*
Design of Non-Recursive Predictive Parsing
for the grammar

S -> aBa
B -> bB | epsilon
*/

#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
```

Output:



b) 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\nSyntax Error!!! String is invalid\n");
   getch();
   exit(0);
}

int main()
{
   int n;

   printf("The given grammar is\n\n");
   printf("E -> TC\n");
   printf("C -> +TC | epsilon\n");
   printf("T -> FD\n");
   printf("D -> *FD | epsilon\n");
```

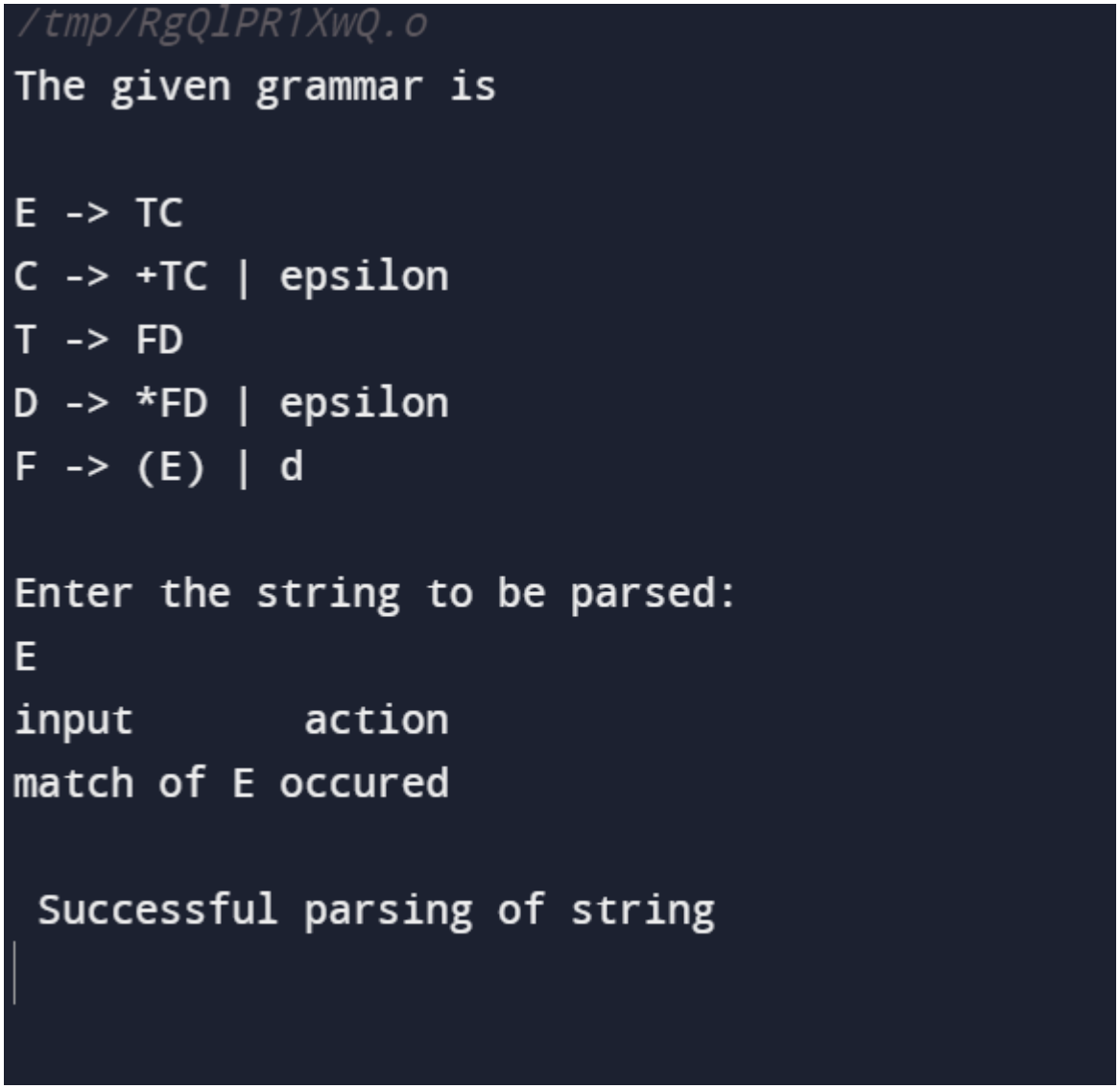
```
printf("F -> (E) | d\n\n");
printf("Enter the string to be parsed:\n");
scanf("%s",ip);
n=strlen(ip);
ip[n]='$';
ip[n+1]='\0';
push('$');
push('E');
printf("\ninput\t\taction\n");
while(ip[i]!='\0')
{
   if(ip[i]=='$' && stack[top-1]=='$')
   {
      printf("\n\n Successful parsing of string \n");
      return(1);

   }
   else if(ip[i]==stack[top-1])
        {
           printf("match of %c occured ",ip[i]);
           i++;
           pop();
        }
        else
        {
                  if(stack[top-1]=='E' && ip[i]=='d')
                  {
                     printf("\nE ->TC\t\t");
                     pop();
                     push('C');
                     push('T');
```

```
}
else if(stack[top-1]=='E' && ip[i]=='(')
{
   printf("\nE ->TC\t\t");
            pop();
            push('C');
            push('T');
}
else if(stack[top-1]=='C' && ip[i]=='+')
{
   printf("\nC -> +TC\t");
   pop();
   push('C');
   push('T');
   push('+');
}
else if(stack[top-1]=='C' && ip[i]==')')
{
   printf("\nC -> epsilon\t");
   pop();
}
else if(stack[top-1]=='C' && ip[i]=='$')
{
   printf("\nC -> epsilon\t");
   pop();
}
else if(stack[top-1]=='T' && ip[i]=='d')
{
   printf("\nT ->FD\t\t");
   pop();
   push('D');
```

```
    push('F');
}
else if(stack[top-1]=='T' && ip[i]=='(')
{
    printf("\nT ->FD\t\t");
            pop();
            push('D');
            push('F');
}
else if(stack[top-1]=='D' && ip[i]=='+')
{
    printf("\nD -> epsilon\t");
    pop();
}
else if(stack[top-1]=='D' && ip[i]=='*')
{
    printf("\nD -> *FD\t");
    pop();
    push('D');
    push('F');
    push('*');
}
else if(stack[top-1]=='D' && ip[i]==')')
{
    printf("\nD -> epsilon\t");
    pop();
}
else if(stack[top-1]=='D' && ip[i]=='$')
{
    printf("\nD -> epsilon\t");
    pop();
```

```
                }
                else if(stack[top-1]=='F' && ip[i]=='d')
                {
                   printf("\nF -> d\t\t");
                   pop();
                   push('d');
                }
                else if(stack[top-1]=='F' && ip[i]=='(')
                {
                   printf("\nF -> (E)\t");
                   pop();
                   push(')');
                   push('E');
                   push('(');
                }
                else
                {
                   error();
                }
          }
    }
}
```

Output:



Week 9: Implementation of Shift Reduce parser using C for the following grammar and illustrate the parser's actions for a valid and an invalid string.

E→E+E

E→E*E

E→(E)

E→d

```
#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("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",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]=='E' && stack[top-1]=='$') && (*(ip)=='$'))
{
display();
printf(" Valid\n\n\n");
break;
}
if(stack[top]=='$')
{
push(*ip);
ip++;
printf("Shift");
}
else if(stack[top]=='d')
{
display();
pop();
push('E');
printf("Reduce E->d");
}
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 if(*ip=='$')
{ printf(" Invalid\n\n\n");
break;
}
else
{
display();
push(*ip);
ip++;
printf("shift");
}
}while(1);
}
```

Output:

```
/tmp/kPrgjbuwvg.o
E->E+E
E->E*E
E->(E)
E->d
Enter the input string followed by $
E
STACK    BUFFER      ACTION
-----    -------     ------

$   E   Shift
$E      shift
$E      shift
$E      shift
$E      shift
$E      shift
$E      shift
$E      shift
$E      shift
```

Week 10: Implement LALR parser using LEX and YACC for the following Grammar:

E → E+T |T

E'→ T*F | F

F → (E) | d

Code:

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
yylval=atoi(yytext);
return (digit);
```

}

[\t];

[\n] return 0;

.return yytext[0];

%%

int yywrap(){

return 1;

}

Output: