CD:

SET-9:

(a) Design a Lexical analyzer for C language for Checking operators.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
char s[5];
printf("\n Enter any operator:");
gets(s);
switch(s[0])
{
case'>':
if(s[1]=='=')
printf("\n Greater than or equal");
else printf("\n Greater than");
break;
case'<':
if(s[1]=='=')
printf("\n Less than or equal");
else printf("\nLess than");
break;
case'=':
if(s[1]=='=')
printf("\nEqual to");
else
printf("\nAssignment");
break;
case'!':
if(s[1]=='=')
printf("\nNot Equal");
else
printf("\n Bit Not");
break;
case'&':
if(s[1]=='&')
printf("\nLogical AND");
else printf("\n Bitwise AND");
break;
case'|':
if(s[1]=='|')
printf("\nLogical OR");
else
printf("\nBitwise OR");
break;
case'+':
printf("\n Addition");
break;
case'-':
printf("\nSubstraction");
break;
case'*':
printf("\nMultiplication");
break;
case'/':
printf("\nDivision");
break;
case'%':
printf("Modulus");
break;
default:
printf("\n Not a operator");
}
}
```

O/P:

```
Enter any operator:&
Bitwise AND
```

(b) Implement the lexical analyzer using lex.

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

%%
(\+\+|--|==|!=|<=|>=|\|\||&&|\*|\/|%|<<|>>|&|\||\^|!|~|\+|-|<|>|=) {
   printf("OPERATOR: %s\n", yytext);
}
.|\n    /* Ignore any other characters */

%%

int main() {
   yylex();
   return 0;
}
```

```
lex lexer.l
gcc lex.yy.c -o lexer -ll
```

plain text:

```
++ -- == != <= >= || && * / % << >> & | ^ ! ~ + - < > =
```

Bash:

```
./lexer < input.txt
```

Expected output:

```
OPERATOR: ++
OPERATOR: --
OPERATOR: ==
OPERATOR: !=
OPERATOR: <=
OPERATOR: >=
OPERATOR: ||
OPERATOR: &&
OPERATOR: *
OPERATOR: /
OPERATOR: %
OPERATOR: <<
OPERATOR: >>
OPERATOR: &
OPERATOR: |
OPERATOR: ^
OPERATOR: !
OPERATOR: ~
OPERATOR: +
OPERATOR: -
OPERATOR: <
OPERATOR: >
OPERATOR: =
```

O/P:

```
 lex lexer.l
gcc lex.yy.c -o lexer -ll
./lexer
```

SET-8:

a)Design a Lexical analyzer for C language for given string Is constant or not?

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
char s[50];
printf("enter string:");
gets(s);
if(atoi(s))
printf("given string is contant");
else
printf("given string is not constant");
}
```

o/p: enter string:hhdh

given string is not constant

b) Design LALR bottom up parser for a given language.

Week4.l

```
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+ {yylval.dval=atof(yytext);
return DIGIT;
}
\n|. return yytext[0];
%%
```

Week4.y

```
%{
/*This YACC specification file generates the LALR parser for the program
considered in experiment 4.*/
#include<stdio.h>
%}
%union
{
double dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
line: expr '\n' {
printf("%g\n",$1);
}
;
expr: expr '+' term {$$=$1 + $3 ;}
| term
;
term: term '*' factor {$$=$1 * $3 ;}
| factor
;
factor: '(' expr ')' {$$=$2 ;}
```

```
| DIGIT
;
%%
int main()
{
yyparse();
}
yyerror(char *s)
{
printf("%s",s);
}
```

Input: 3 + (4 * 5)
Output: 23

To run your code, you need to generate the lexer and parser using the following commands:

```
lex Week4.l
yacc -d Week4.y
gcc lex.yy.c y.tab.c -o parser -ll
```

Then, you can run the parser: ./parser

Enter your expression, and when you press Enter, it should print the result:

```
 3 + (4 * 5)
23
```

SET-7:

a) Design a Lexical analyzer for C language for Checking identifiers

```
#include<stdio.h>
//#include<stdlib.h>
#include<string.h>
void main()
{
char s[50];
printf("enter input:");
gets(s);
int flag=0;
if(isalpha(s[0])||s[0]=='_')
{
for(int i=0;i<strlen(s);i++){
if(isdigit(s[i])|| isalpha(s[i])|| s[i]=='_')
flag=1;
else
break;
}
}
if(flag==1)
printf("valid");
else
printf("invalid");
}
```

o/p: enter input:bwve

valid

b) Design Predictive parser for a given language.

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

char prol[7][10] = {"S", "A", "A", "B", "B", "C", "C"};
char pror[7][10] = {"A", "Bb", "Cd", "aB", "@", "Cc", "@"};
char prod[7][10] = {"S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C->@"};
char first[7][10] = {"abcd", "ab", "cd", "a@", "@", "c@", "@"};
char follow[7][10] = {"$", "$", "$", "a$", "b$", "c$", "d$"};
char table[5][6][10];

int numr(char c)
{
    switch (c)
    {
    case 'S':
        return 0;
    case 'A':
        return 1;
    case 'B':
        return 2;
    case 'C':
        return 3;
    case 'a':
        return 0;
    case 'b':
        return 1;
    case 'c':
        return 2;
    case 'd':
        return 3;
    case '$':
        return 4;
    }
    return 2;
}

int main()
{
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 6; j++)
            strcpy(table[i][j], " ");

    printf("The following grammar is used for Parsing Table:\n");
    for (int i = 0; i < 7; i++)
        printf("%s\n", prod[i]);

    printf("\nPredictive parsing table:\n");

    for (int i = 0; i < 7; i++)
    {
        int k = strlen(first[i]);
        for (int j = 0; j < 10; j++)
            if (first[i][j] != '@')
                strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1], prod[i]);
    }

    for (int i = 0; i < 7; i++)
    {
        if (strlen(pror[i]) == 1 && pror[i][0] == '@')
        {
            int k = strlen(follow[i]);
            for (int j = 0; j < k; j++)
                strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1], prod[i]);
        }
    }

    strcpy(table[0][0], " ");
    strcpy(table[0][1], "a");
    strcpy(table[0][2], "b");
    strcpy(table[0][3], "c");
    strcpy(table[0][4], "d");
    strcpy(table[0][5], "$");
    strcpy(table[1][0], "S");
    strcpy(table[2][0], "A");
    strcpy(table[3][0], "B");
    strcpy(table[4][0], "C");

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

    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 6; j++)
            printf("%-10s", table[i][j]);
        printf("\n--------------------------------------------------------\n");
    }

    return 0;
}
```

```
o/p:
/tmp/DKCVCvMtKX.o
The following grammar is used for Parsing Table:
S->A
A->Bb
A->Cd
B->aB
B->@
C->Cc
C->@

Predictive parsing table:

--------------------------------------------------------
          a         b         c         d         $
--------------------------------------------------------
S         S->A      S->A      S->A      S->A
--------------------------------------------------------
A         A->Bb     A->Bb     A->Cd     A->Cd
--------------------------------------------------------
B         B->aB     B->@      B->@                B->@
--------------------------------------------------------
C                             C->@      C->@      C->@
```

SET-6:

(a) Design a Lexical analyzer for C language for checking keywords.

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

```
#include <string.h>

int main() {
   int flag = 0, m;
   char s[5][10] = {"if", "else", "goto", "continue", "return"}, st[10];

   printf("\nEnter the string: ");
   gets(st);

   for (int i = 0; i < 5; i++) {
      m = strcmp(st, s[i]);
      if (m == 0) {
         flag = 1;
         break;
      }
   }

   if (flag == 0)
      printf("\nIt is not a keyword");
   else
      printf("\nIt is a keyword");

   return 0;
}
```

o/p:

```
Enter the string: if
It is a keyword
```

SET-2:

a)Design a Lexical analyzer for C language for Checking comment lines.

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

int main() {
   char s[50];
   printf("Enter input: ");
   gets(s);

   if (s[0] == '/') {
      if (s[1] == '/')
         printf("Given statement is a constant");
      else if (s[1] == '*') {
         int n = strlen(s) - 1;
         if (s[n] == '/' && s[n - 1] == '*')
            printf("Given statement is a comment");
         else
            printf("Given statement is not a comment");
      } else {
         printf("Given statement is not a comment");
      }
   } else {
      printf("Given statement is not a comment");
   }
```

```
    return 0;
}
o/p:
Enter input: //ahhjj
Given statement is a constant
```

SET-1:

c) Construct YACC code to perform Arithmetic Operations

```
Week6.l
%{
 #include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)? return num;
[+/*] return op;
. return yytext[0];
\n return 0;
%%
int yywrap()
{
return 1;
}
Week6.y
%{
 #include<stdio.h>
 int valid=1;
%}
%token num id op
%%
start : id '=' s ';'
s : id x
 | num x
 | '-' num x
 | '(' s ')' x
 ;
x : op s
 | '-' s
 |
 ;
%%
int yyerror()
{
 valid=0;
 printf("\nInvalid expression!\n");
 return 0;
}
int main()
{
 printf("\nEnter the expression:\n");
 yyparse();
 if(valid)
 {
 printf("\nValid expression!\n");
 }
}
```

Input: variable = 3 * (5 - 2);

Output:

```
 Enter the expression:
Valid expression!
```

commands:

```
lex Week6.l
yacc -d Week6.y
gcc lex.yy.c y.tab.c -o parser -ll
```

Then, you can run the parser: ./parser