SYLLABUS

Course No.	Course Name	L-T-P - Credits	Year of Introduction
CS431	**COMPILER DESIGN LAB**	**0-0-3-1**	**2018**

Pre-requisite : CS331 System Software Lab

Course Objectives

1. Learn to implement the different Phases of compiler.
2. Learn simple optimization techniques.
3. Be exposed to compiler writing tools.

List of Exercises/Experiments :

1. Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.
2. Implementation of Lexical Analyzer using Lex Tool
3. Generate YACC specification for a few syntactic categories.

a) Program to recognize a valid arithmetic expression that uses operator +, – , * and /.

b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

c) Implementation of Calculator using LEX and YACC

d) Convert the BNF rules into YACC form and write code to generate abstract syntax tree

4. Write program to find ε – closure of all states of any given NFA with ε transition.
5. Write program to convert NFA with ε transition to NFA without ε transition.
6. Write program to convert NFA to DFA
7. Write program to minimize any given DFA.
8. Develop an operator precedence parser for a given language.
9. Write program to find Simulate First and Follow of any given grammar.
10. Construct a recursive descent parser for an expression.
11. Construct a Shift Reduce Parser for a given language.
12. Write a program to perform loop unrolling.
13. Write a program to perform constant propagation.
14. Implement Intermediate code generation for simple expressions.
15. Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc.

Expected Outcome:

After completing the course, students will be able to

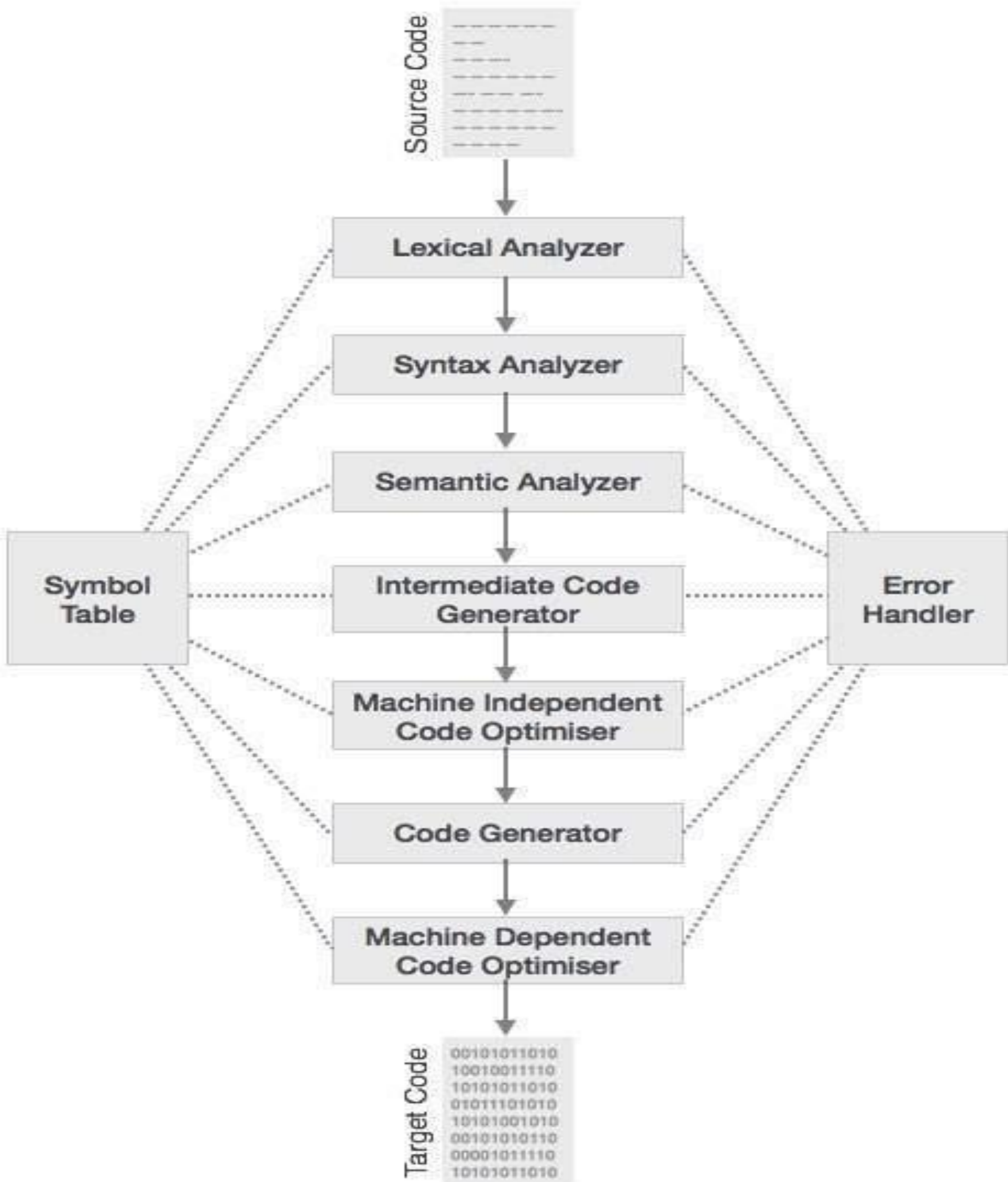
1. Implement the techniques of Lexical Analysis and Syntax Analysis.
2. Apply the knowledge of Lex & Yacc tools to develop programs.
3. Generate intermediate code.
4. Implement Optimization techniques and generate machine level code.

LIST OF EXPERIMENTS

SL. NO.	**NAME OF THE EXPERIMENT**
1	STUDY ABOUT PHASES OF COMPILER
2	LEXICAL ANALYSER
3	DFA IMPLEMENTATION
4	NFA IMPLEMENTATION
5	NFA TO DFA CONVERSION
6	RECURSIVE DECENT PARSING
7	INTERMEDIATE CODE GENERATION
8	IDENTIFICATION OF FLOATING POINT NUMBER, STRING AND INTEGER USING LEX
9	COUNTING NUMBER OF CHARACTERS, WORDS, SPACE AND LINE NUMBER USING LEX
10	CHECKING TOTAL NUMBER OF VOWELS USING LEX
11	CALCULATOR USING YACC
12	8085 CODE GENERATION

EXPERIMENT 1
PHASES OF COMPILER

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



1) Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Token: Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.

• **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token. ,

• **Pattern:** Pattern describes the rule that the lexemes of a token takes. It is the structure that must be matched by strings.

• Once a token is generated the corresponding entry is made in the symbol table.

Input: stream of characters

Output: Token

Lexemes and tokens

Lexemes	**Tokens**
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

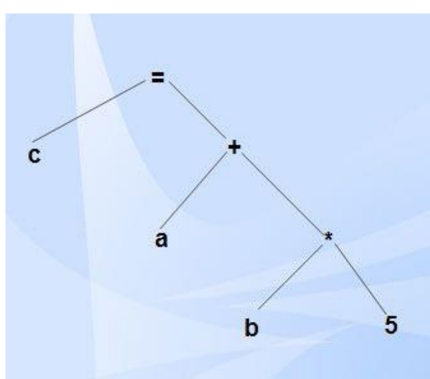
Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

2) Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct. Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator.

Input: *Tokens*

Output: *Syntax tree*

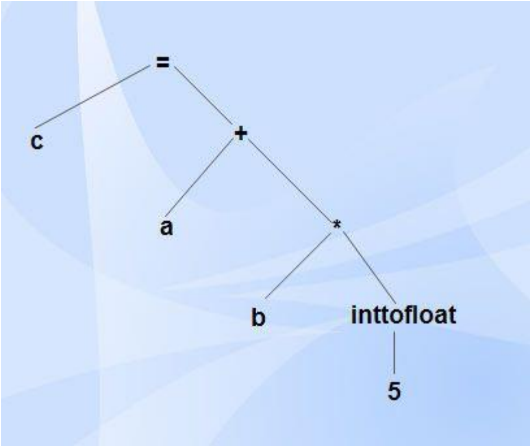


3) Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Input: *Syntax Tree*

Output: *Annotated Syntax Tree*



4) Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Intermediate code generation produces intermediate representations for the source program which are of the following forms:

- o Postfix notation
- o Three address code
- o Syntax tree

Most commonly used form is the three address code.

t_1 = inttofloat (5)

t_2 = id_3* tl

t_3 = id_2 + t_2

id_1 = t_3

5) Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

During the code optimization, the result of the program is not affected.

• To improve the code generation, the optimization involves

- o Deduction and removal of dead code (unreachable code).
- o Calculation of constants in expressions and terms.
- o Collapsing of repeated expression into temporary string.
- o Loop unrolling.
- o Moving code outside the loop.
- o Removal of unwanted temporary variables.

t_1 = id_3* 5.0

id_1 = id_2 + t_1

6) Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

The code generation involves

- o Allocation of register and memory.
- o Generation of correct references.
- o Generation of correct data types.
- o Generation of missing code.

LDF R_2, id_3

MULF R_2, # 5.0

LDF R_1, id_2

ADDF R_1, R_2

STF id_1, R_1

7) Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Example

int a, b; float c; char z;

Symbol name	Type	Address
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Example

```
extern double test (double x);
 double sample (int count)
{   double sum= 0.0;
  for (int i = 1; i < = count; i++)
  sum+= test((double) i);
  return sum;
 }
```

Symbol name	Type	Scope
test	function, double	extern
x	double	function parameter
sample	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

8) Error Handling

Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

• In lexical analysis, errors occur in separation of tokens.

• In syntax analysis, errors occur during construction of syntax tree.

• In semantic analysis, errors may occur at the following cases:

(i) When the compiler detects constructs that have right syntactic structure but no meaning

(ii) During type conversion.

• In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when code is missing etc.

Figure illustrates the translation of source code through each phase, considering the statement

c =*a*+ *b* * 5.

Error Encountered in Different Phases

Each phase can encounter errors. After detecting an error, a phase must some how deal with the error, so that compilation can proceed.

A program may have the following kinds of errors at various stages:

Lexical Errors

It includes incorrect or misspelled name of some identifier i.e., identifiers typed incorrectly.

Syntactical Errors

It includes missing semicolon or unbalanced parenthesis. Syntactic errors are handled by syntax analyzer (parser).

When an error is detected, it must be handled by parser to enable the parsing of the rest of the input. In general, errors may be expected at various stages of compilation but most of the errors are syntactic errors and hence the parser should be able to detect and report those errors in the program.

The goals of error handler in parser are:

• Report the presence of errors clearly and accurately.
• Recover from each error quickly enough to detect subsequent errors.
• Add minimal overhead to the processing of correcting programs.

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

o Panic mode.
o Statement level.
o Error productions.
o Global correction.

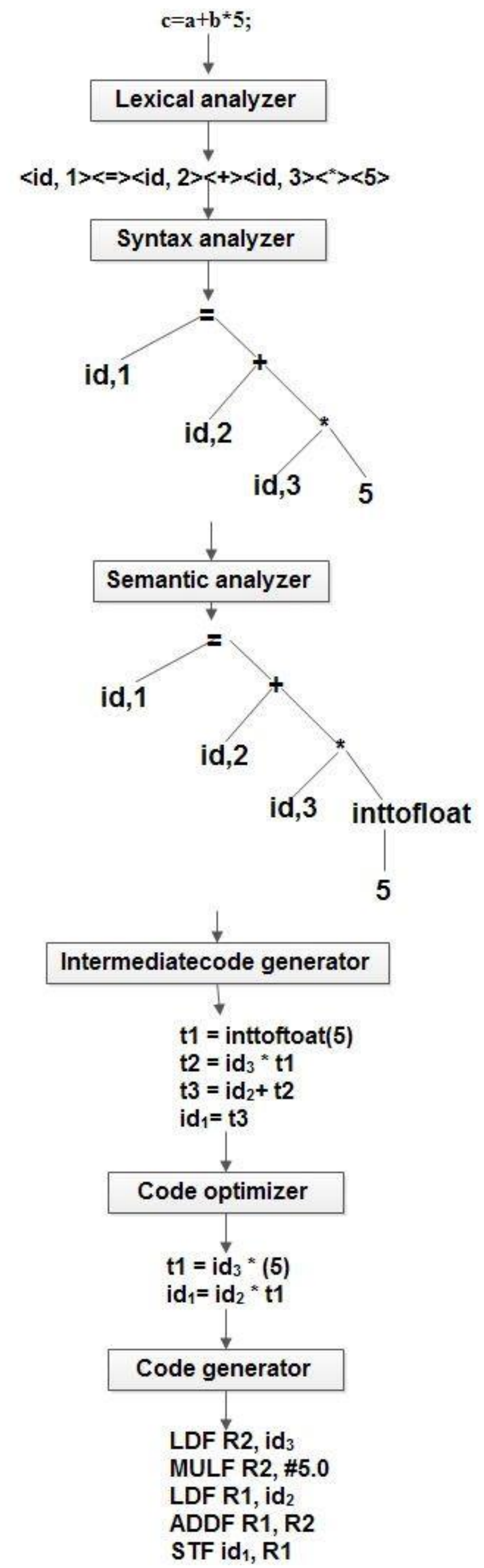
Semantical Errors

These errors are a result of incompatible value assignment. The semantic errors that the semantic analyzer is expected to recognize are:

• Type mismatch.
• Undeclared variable.
• Reserved identifier misuse.
• Multiple declaration of variable in a scope.
• Accessing an out of scope variable.
• Actual and formal parameter mismatch.

Logical errors

These errors occur due to not reachable code-infinite loop.


c=a+b*5;
Lexical analyzer
<id, 1><=><id, 2><+><id, 3><*><5>
Syntax analyzer
=
id,1
+
id,2
*
id,3
5
Semantic analyzer
=
id,1
+
id,2
*
id,3
inttofloat
5
Intermediatecode generator
t1 = inttoftoat(5)
t2 = id_3 * t1
t3 = id_2+ t2
id_1= t3
Code optimizer
t1 = id_3 * (5)
id_1= id_2 * t1
Code generator
LDF R2, id_3
MULF R2, #5.0
LDF R1, id_2
ADDF R1, R2
STF id_1, R1

EXPERIMENT 2
LEXICAL ANALYSER

Aim: Write a program to implement lexical analyser.

Program

```
#include<ctype.h>
#include<stdio.h>
#include<string.h>
void main()
{
FILE *f1;
char c,str[10];
int lineno=0,num=0,i=0;
f1=fopen("input.txt","r");
while((c=getc(f1))!=EOF) // TO READ THE GIVEN INPUT FILE
{
if(isdigit(c)) // TO RECOGNIZE NUMBERS
{
num=c-48;//That is, subtracting by 48 translates the char values '0'..'9' to the int values 0..9
c=getc(f1);
while(isdigit(c))
{
num=num*10+(c-48);
c=getc(f1);
}
printf("%d is a number \n",num);
ungetc(c,f1);//ungetc() returns previously read character back to the stream so that it could be read again.
}
else if(isalpha(c)) // TO RECOGNIZE KEYWORDS AND IDENTIFIERS
{
str[i++]=c;
c=getc(f1);
while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
{
str[i++]=c;
c=getc(f1);
}
str[i++]='\0';
if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||
strcmp("double",str)==0||strcmp("static",str)==0||
strcmp("switch",str)==0||strcmp("case",str)==0) // TYPE 32 KEYWORDS
printf("%s is a keyword\n",str);
```

```
else
printf("%s is a identifier\n",str);
ungetc(c,f1);
i=0;
}
else if(c==' '||c=='\t') // TO IGNORE THE SPACE
printf("\n");
else if(c=='\n') // TO COUNT LINE NUMBER IN INPUT FILE
lineno++;
else // TO FIND SPECIAL SYMBOL
printf("%c is a special symbol\n",c);
}
printf("Total no. of lines in input file are: %d\n",lineno);
fclose(f1);
}
```

Input

input.txt

```
#include<string.h>
void main()
{
int a,b,c;
c=a+123;
}
```

Output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ gcc lexical.c
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out

# is a special symbol
include is a identifier
< is a special symbol
string is a identifier
. is a special symbol
h is a identifier
> is a special symbol
void is a identifier

main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
```

a is a identifier
, is a special symbol
b is a identifier
, is a special symbol
c is a identifier
; is a special symbol
c is a identifier
= is a special symbol
a is a identifier
+ is a special symbol
123 is a number
; is a special symbol
} is a special symbol
Total no. of lines in input file are: 6

Result

The program was successfully executed and output was verified.

EXPERIMENT 3
DFA IMPLEMENTATION

Aim: Write a program to perform DFA implementation.

Program

```
#include<stdio.h>
void main()
{
int state[10];
int str[10],input[10];
char ch;
int x[20];
int s,n,k=0,j,a,i,l,t,q=0,fs,b,nxt,z;
printf("enter the no. states\n");
scanf("%d",&s);
printf("enter the no.of i/ps\n");
scanf("%d",&n);
for(i=0;i<s;i++)
{
printf("enter the state\n");
scanf("%d",&state[i]);
printf("is it final state?... .y..1/n..0\n");
scanf("%d",&a);
if(a==1)
fs=state[i];
}
printf("enter the i/ps\n");
for(i=0;i<n;i++)
scanf("%d",&input[i]);
printf("transition state\n");

for(i=0;i<s;i++)
{
for(j=0;j<n;j++)
{
printf("(q%d,%d)=q",state[i],input[j]);
scanf("%d",&b);
x[k]=b; k++;
}
}
do
{
printf("enter the length of string\n");
scanf("%d",&l);
```

```
printf("enter the i/p string\n");
for(i=0;i<l;i++)
scanf("%d",&str[i]);
for(i=0;i<l;i++)
{
t=0;
do
{
if(str[i]==input[t])
{
nxt=x[n*q+t];
for(j=0;j<s;j++)
{
if(nxt==state[j])
q=j;
}
t++;
}

else
t++;
}
while(t!=n);
}
if(nxt==fs)
printf("\n string is accepted\n");
else
printf("\n not accepted\n");
printf("do you want to continue...if yes press 1 else 2");
scanf("%d",&z);
}
while(z!=2);
}
```

input and output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ gcc dfaa.c
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out
enter the no. states
3
enter the no.of i/ps
2
enter the state
0
is it final state?... .y..1/n..0
0
```

```
enter the state
1
is it final state?... .y..1/n..0
0
enter the state
2
is it final state?... .y..1/n..0
1
enter the i/ps
0
1
transition state
(q0,0)=q0
(q0,1)=q1
(q1,0)=q2
(q1,1)=q1
(q2,0)=q2
(q2,1)=q2
enter the length of string
4
enter the i/p string
0 1 1 0

string is accepted
do you want to continue...if yes press 1 else 21
enter the length of string
5
enter the i/p string
0 1 1 1 1

string is rejected
do you want to continue...if yes press 1 else 2 1
enter the length of string
3
enter the i/p string
1 1 1

string is rejected
do you want to continue...if yes press 1 else 2 2
```

Result

The program was successfully executed and output was verified.

EXPERIMENT 4
NFA IMPLEMENTATION

Aim: Write a program to perform NFA implementation.

Program

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

#define fl(i,a,b) for(i=a; i<b; i++)
#define scan(a) scanf("%d", &a)
#define nline printf("\n")

#define MAX 1000

int states, symbols, symdir[20], final_states, mark[20], mat[20][20][20];
char str1[MAX];

int curr[20], t[20];
int ind, ind1;
int l1;

int main()
{
int i, j, k;

fl(i,0,20)
fl(j,0,20)
fl(k,0,20)
mat[i][j][k]=-1;

printf("Enter the number of states : ");
scan(states);
printf("Enter the number of symbols : ");
scan(symbols);
printf("Enter the symbols : ");
nline;
fl(i,0,symbols)
{
printf("Enter the symbol number %d : ", i);
scan(symdir[i]);
}
printf("Enter the number of final states : ");
scan(final_states);
printf("Enter the number of the states which are final : ");
```

```
fl(i,0,final_states)
{
int temp;
scan(temp);
mark[temp]=1;
}
printf("Define the relations for the states and symbols : ");
nline;
fl(i,0,states)
{
fl(j,0,symbols)
{
int ntemp;
printf("Enter the number of relations for Q(%d) -> %d : ", i, symdir[j]);
scan(ntemp);
fl(k,0,ntemp)
{
printf("Enter the relation number %d for Q(%d) -> %d : ", k, i, symdir[j]);
scan(mat[i][symdir[j]][k]);
}
}
}
//-------------------------------------------------------//
int cases;
printf("Enter the number of strings to be tested : ");
scan(cases);
fl(k,0,cases)
{
printf("Enter the string to be tested : ");
scanf("%s", str1);

curr[0]=0;
ind=1;
int limit=strlen(str1);

fl(i,0,limit)
{
int ele=(int)(str1[i]-'0');

ind1=0;
fl(l1,0,ind)
{
j=0;
while(mat[curr[l1]][ele][j]!=-1)
{
t[ind1++]=mat[curr[l1]][ele][j];
```

```
j++;
}
}

fl(l1,0,ind1)
{
curr[l1]=t[l1];
}

ind=ind1;
}
int flag=0;

fl(i,0,ind)
{
if(mark[curr[i]]==1)
{
flag=1;
break;
}
}
printf("The entered string is ");

if(flag==1)
printf("Accepted");
else
printf("Rejected");
nline;
}

return 0;
}
```

input and output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ gcc nfa2.c
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out

Enter the number of states : 3
Enter the number of symbols : 2
Enter the symbols :
Enter the symbol number 0 : 0
Enter the symbol number 1 : 1
Enter the number of final states : 1
Enter the number of the states which are final : 2
Define the relations for the states and symbols :
```

Enter the number of relations for Q(0) -> 0 : 1
Enter the relation number 0 for Q(0) -> 0 : 0
Enter the number of relations for Q(0) -> 1 : 2
Enter the relation number 0 for Q(0) -> 1 : 0
Enter the relation number 1 for Q(0) -> 1 : 1
Enter the number of relations for Q(1) -> 0 : 2
Enter the relation number 0 for Q(1) -> 0 : 1
Enter the relation number 1 for Q(1) -> 0 : 2
Enter the number of relations for Q(1) -> 1 : 0
Enter the number of relations for Q(2) -> 0 : 0
Enter the number of relations for Q(2) -> 1 : 0
Enter the number of strings to be tested : 2
Enter the string to be tested : 0100
The entered string is Accepted
Enter the string to be tested : 0110
The entered string is Rejected

Result

The program was successfully executed and output was verified.

EXPERIMENT 5
NFA TO DFA CONVERSION

Aim: Write a program to perform any NFA to DFA conversion.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN 100

char NFA_FILE[MAX_LEN];
char buffer[MAX_LEN];
int zz = 0;

// Structure to store DFA states and their
// status ( i.e new entry or already present)
struct DFA {
  char *states;
  int count;
} dfa;

int last_index = 0;
FILE *fp;
int symbols;

/* reset the hash map*/
void reset(int ar[], int size) {
  int i;

  // reset all the values of
  // the mapping array to zero
  for (i = 0; i < size; i++) {
    ar[i] = 0;
  }
}

// Check which States are present in the e-closure

/* map the states of NFA to a hash set*/
void check(int ar[], char S[]) {
  int i, j;

  // To parse the individual states of NFA
  int len = strlen(S);
```

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

    // Set hash map for the position
    // of the states which is found
    j = ((int)(S[i]) - 65);
    ar[j]++;
  }
}

// To find new Closure States
void state(int ar[], int size, char S[]) {
  int j, k = 0;

  // Combine multiple states of NFA
  // to create new states of DFA
  for (j = 0; j < size; j++) {
    if (ar[j] != 0)
      S[k++] = (char)(65 + j);
  }

  // mark the end of the state
  S[k] = '\0';
}

// To pick the next closure from closure set
int closure(int ar[], int size) {
  int i;

  // check new closure is present or not
  for (i = 0; i < size; i++) {
    if (ar[i] == 1)
      return i;
  }
  return (100);
}

// Check new DFA states can be
// entered in DFA table or not
int indexing(struct DFA *dfa) {
  int i;

  for (i = 0; i < last_index; i++) {
    if (dfa[i].count == 0)
      return 1;
  }
  return -1;
```

```
}

/* To Display epsilon closure*/
void Display_closure(int states, int closure_ar[],
                     char *closure_table[],
                     char *NFA_TABLE[][symbols + 1],
                     char *DFA_TABLE[][symbols]) {
  int i;
  for (i = 0; i < states; i++) {
    reset(closure_ar, states);
    closure_ar[i] = 2;

    // to neglect blank entry
    if (strcmp(&NFA_TABLE[i][symbols], "-") != 0) {

      // copy the NFA transition state to buffer
      strcpy(buffer, &NFA_TABLE[i][symbols]);
      check(closure_ar, buffer);
      int z = closure(closure_ar, states);

      // till closure get completely saturated
      while (z != 100)
      {
        if (strcmp(&NFA_TABLE[z][symbols], "-") != 0) {
          strcpy(buffer, &NFA_TABLE[z][symbols]);

          // call the check function
          check(closure_ar, buffer);
        }
        closure_ar[z]++;
        z = closure(closure_ar, states);
      }
    }

    // print the e closure for every states of NFA
    printf("\n e-Closure (%c) :\t", (char)(65 + i));

    bzero((void *)buffer, MAX_LEN);
    state(closure_ar, states, buffer);
    strcpy(&closure_table[i], buffer);
    printf("%s\n", &closure_table[i]);
  }
}

/* To check New States in DFA */
int new_states(struct DFA *dfa, char S[]) {
```

```
  int i;

  // To check the current state is already
  // being used as a DFA state or not in
  // DFA transition table
  for (i = 0; i < last_index; i++) {
    if (strcmp(&dfa[i].states, S) == 0)
      return 0;
  }

  // push the new
  strcpy(&dfa[last_index++].states, S);

  // set the count for new states entered
  // to zero
  dfa[last_index - 1].count = 0;
  return 1;
}

// Transition function from NFA to DFA
// (generally union of closure operation )
void trans(char S[], int M, char *clsr_t[], int st,
               char *NFT[][symbols + 1], char TB[]) {
  int len = strlen(S);
  int i, j, k, g;
  int arr[st];
  int sz;
  reset(arr, st);
  char temp[MAX_LEN], temp2[MAX_LEN];
  char *buff;
 // Transition function from NFA to DFA
  for (i = 0; i < len; i++) {

    j = ((int)(S[i] - 65));
    strcpy(temp, &NFT[j][M]);

    if (strcmp(temp, "-") != 0) {
      sz = strlen(temp);
      g = 0;

      while (g < sz) {
        k = ((int)(temp[g] - 65));
        strcpy(temp2, &clsr_t[k]);
        check(arr, temp2);
        g++;
```

```
      }
    }
  }

  bzero((void *)temp, MAX_LEN);
  state(arr, st, temp);
  if (temp[0] != '\0') {
    strcpy(TB, temp);
  } else
    strcpy(TB, "-");
}
/* Display DFA transition state table*/
void Display_DFA(int last_index, struct DFA *dfa_states,
                 char *DFA_TABLE[][symbols]) {
  int i, j;
  printf("\n\n********************************************************\n\n");
  printf("\t\t DFA TRANSITION STATE TABLE \t\t \n\n");
  printf("\n STATES OF DFA :\t\t");

  for (i = 1; i < last_index; i++)
    printf("%s, ", &dfa_states[i].states);
  printf("\n");
  printf("\n GIVEN SYMBOLS FOR DFA: \t");

  for (i = 0; i < symbols; i++)
    printf("%d, ", i);
  printf("\n\n");
  printf("STATES\t");

  for (i = 0; i < symbols; i++)
    printf("|%d\t", i);
  printf("\n");
 // display the DFA transition state table
  printf("--------------------------------\n");
  for (i = 0; i < zz; i++) {
    printf("%s\t", &dfa_states[i + 1].states);
    for (j = 0; j < symbols; j++) {
      printf("|%s \t", &DFA_TABLE[i][j]);
    }
    printf("\n");
  }
}

// Driver Code
int main() {
  int i, j, states;
```

```
  char T_buf[MAX_LEN];

  // creating an array dfa structures
  struct DFA *dfa_states = malloc(MAX_LEN * (sizeof(dfa)));
  states = 6, symbols = 2;

  printf("\n STATES OF NFA :\t\t");
  for (i = 0; i < states; i++)

    printf("%c, ", (char)(65 + i));
 printf("\n");
  printf("\n GIVEN SYMBOLS FOR NFA: \t");

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

    printf("%d, ", i);
  printf("eps");
  printf("\n\n");
  char *NFA_TABLE[states][symbols + 1];

  // Hard coded input for NFA table
  char *DFA_TABLE[MAX_LEN][symbols];
  strcpy(&NFA_TABLE[0][0], "FC");
  strcpy(&NFA_TABLE[0][1], "-");
  strcpy(&NFA_TABLE[0][2], "BF");
  strcpy(&NFA_TABLE[1][0], "-");
  strcpy(&NFA_TABLE[1][1], "C");
  strcpy(&NFA_TABLE[1][2], "-");
  strcpy(&NFA_TABLE[2][0], "-");
  strcpy(&NFA_TABLE[2][1], "-");
  strcpy(&NFA_TABLE[2][2], "D");
  strcpy(&NFA_TABLE[3][0], "E");
  strcpy(&NFA_TABLE[3][1], "A");
  strcpy(&NFA_TABLE[3][2], "-");
  strcpy(&NFA_TABLE[4][0], "A");
  strcpy(&NFA_TABLE[4][1], "-");
  strcpy(&NFA_TABLE[4][2], "BF");
  strcpy(&NFA_TABLE[5][0], "-");
  strcpy(&NFA_TABLE[5][1], "-");
  strcpy(&NFA_TABLE[5][2], "-");
  printf("\n NFA STATE TRANSITION TABLE \n\n\n");
  printf("STATES\t");

  for (i = 0; i < symbols; i++)
    printf("|%d\t", i);
  printf("eps\n");
```

```
 // Displaying the matrix of NFA transition table
 printf("-------------------------------------------\n");
 for (i = 0; i < states; i++) {
   printf("%c\t", (char)(65 + i));

   for (j = 0; j <= symbols; j++) {
     printf("|%s \t", &NFA_TABLE[i][j]);
   }
   printf("\n");
 }
 int closure_ar[states];
 char *closure_table[states];

 Display_closure(states, closure_ar, closure_table, NFA_TABLE, DFA_TABLE);
 strcpy(&dfa_states[last_index++].states, "-");

 dfa_states[last_index - 1].count = 1;
 bzero((void *)buffer, MAX_LEN);

 strcpy(buffer, &closure_table[0]);
 strcpy(&dfa_states[last_index++].states, buffer);

 int Sm = 1, ind = 1;
 int start_index = 1;

 // Filling up the DFA table with transition values
 // Till new states can be entered in DFA table
 while (ind != -1) {
   dfa_states[start_index].count = 1;
   Sm = 0;
   for (i = 0; i < symbols; i++) {

     trans(buffer, i, closure_table, states, NFA_TABLE, T_buf);
// storing the new DFA state in buffer
     strcpy(&DFA_TABLE[zz][i], T_buf);

     // parameter to control new states
     Sm = Sm + new_states(dfa_states, T_buf);
   }
   ind = indexing(dfa_states);
   if (ind != -1)
     strcpy(buffer, &dfa_states[++start_index].states);
   zz++;
 }
 // display the DFA TABLE
```

```
 Display_DFA(last_index, dfa_states, DFA_TABLE);
  return 0;
}
```

Input an output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ gcc nfatodfa.c
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out

STATES OF NFA :        A, B, C, D, E, F,
GIVEN SYMBOLS FOR NFA:     0, 1, eps

 NFA STATE TRANSITION TABLE

STATES        |0        |1        |eps
--------------------------------------------
A             FC        |-        |BF
B             |-        |C        |-
C             |-        |-        |D
D             |E        |A        |-
E             |A        |-        |BF
F             |-        |-        |-

 e-Closure (A) :    ABF
 e-Closure (B) :    B
 e-Closure (C) :    CD
 e-Closure (D) :    D
 e-Closure (E) :    BEF
 e-Closure (F) :    F

********************************************************

DFA TRANSITION STATE TABLE

STATES OF DFA :        ABF, CDF, CD, BEF,
GIVEN SYMBOLS FOR DFA:     0, 1,

STATES        |0        |1
-------------------------------
ABF           |CDF      |CD
CDF           |BEF      |ABF
CD            |BEF      |ABF
BEF           |ABF      |CD
```

Result

The program was successfully executed and output was verified.

EXPERIMENT 6
RECURSIVE DECENT PARSING

Aim: Write a program to perform Recursive Decent Parsing.

Program

```
#include<stdio.h>
#include<string.h>
char input[10];
int i=0,error=0;
void E();
void T();
void Eprime();
void Tprime();
void F();

void main()
{
printf("Enter an arithmetic expression :\n");
gets(input);
E();
if(strlen(input)==i&&error==0)
printf("\nAccepted..!!!");
else
printf("\nRejected..!!!");
}

void E()
{
T();
Eprime();
}

void Eprime()
{
if(input[i]=='+')
{
i++;
T();
Eprime();
}
}

void T()
{
```

```
F();
Tprime();
}

void Tprime()
{
if(input[i]=='*')
{
i++;
F();
Tprime();
}
}

void F()
{
if(input[i]=='(')
{
i++;
E();
if(input[i]==')')
i++;
}
else if(isalpha(input[i]))
{
i++;
while(isalnum(input[i])||input[i]=='_')
i++;
}
else
error=1;
}
```

input and output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ gcc rdpp.c
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out

Enter an arithmetic expression :
a*(n+m)

Accepted..!!!
```

Result

The program was successfully executed and output was verified.

EXPERIMENT 7
INTERMEDIATE CODE GENERATION

Aim: Write a program to implement Intermediate Code Generation.

Program

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
int isp(char item);
void output(char item);
void push(char item);
char pop(void);
void quad(void);
char exp[20];
char res[20];
char a[20],opr[20],opd1[20],opd2[20],result[20];
int st[20],value[20];
int top=0,z=0,i=0,op1,op2,k,j,p,l;
char x,item;

void main()
{
printf("Enter the infix expression: ");
gets(exp);
l=strlen(exp);
push('#');
while((item=exp[i])!='\0')
{
if(isalpha(exp[i]))
output(item);
else if(item=='+'||item=='-'||item=='*'||item=='/'||item=='^')
push(item);
else if(item=='(')
push(item);
else if(item==')')
{
while((x=pop())!='(')
output(x);
}
else if(isp(x=pop())<isp(item))
{
push(x);
push(item);
}
```

```
else
{
output(x);
push(item);
}
i++;
}
while((x=pop())!='#')
output(x);
printf("Postfix expression: ");
puts(res);
quad();
}

int isp(char item)
{
if((item=='+')||(item=='-'))
return(1);
else if((item=='*')||(item=='/'))
return(2);
else if((item=='^'))
return(3);
else
return(0);
}
void output(char item)
{
res[z++]=item;
}
void push(char item)
{
a[++top]=item; }
char pop(void)
{
item=a[top--];
return(item); }
void quad()
{
int i,x=0;
char m,n,p,temp,str1[5],str2[5];
printf("\noperator\top1\top2\tresult\n");
printf("---------------------------------------");
for(i=0;i<l;i++)
{
if(isalnum(res[i]))
{
```

```
push(res[i]); }
else
{
if(isalpha(m=pop()))
{
str1[0]=m;
str1[1]='\0'; }
else
{
str1[0]='t';
str1[1]=m;
str1[2]='\0'; }
if(isalpha(n=pop()))
{
str2[0]=n;
str2[1]='\0'; }
else
{ str2[2]='\0'; }
x++;
printf("\n%c\t\t%s\t%s\tt%d\n",res[i],str2,str1,x);
temp=x+'0';
push(temp);
} } }
```

input and output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ gcc icg.c
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out
Enter the infix expression: a+(b*c)-d
Postfix expression: abc*d-+

Operator        op1     op2     result
---------------------------------------
*               b       c       t1

-               b       d       t2

+               a       t2      t3

                a       t3      t4

                a       t4      t5
```

Result

The program was successfully executed and output was verified.

EXPERIMENT 8
LEX PROGRAM 1

Aim: Write a program for the identification of floating point number, string and integer using LEX.

Program

```
%{
#include<stdio.h>
%}
%%
[\t];
[0-9]+\.[0-9]+ {printf("Found a floating point number %s \n",yytext);}
[0-9]+ {printf("Found an integer number %s \n",yytext);}
[a-zA-Z0-9]+ {printf("Found a string %s \n",yytext);}
%%
main()
{
//lex throug the input
yylex();
}
```

input and output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ flex x.lex
ilm@ilm-HCL-Desktop:~/Downloads/lija$ cc lex.yy.c -lfl
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out

23
Found an integer number 23

12.5
Found a floating point number 12.5

hello world
Found a string hello
Found a string world
```

Result

The program was successfully executed and output was verified.

EXPERIMENT 9
LEX PROGRAM 2

Aim: Write a program for counting number of characters, words, space and line number in a paragraph using LEX.

Program

```
%{
#include<stdio.h>
int c=0,w=0,s=0,l=0;
%}
WORD [^ \t\n,\.:]+
EOL [\n]
BLANK [ ]
%%
{WORD} {w++;c=c+yyleng;}
{BLANK} {s++;}
{EOL} {l++;}
. {c++;}
%%
int yywrap()
{return 1;}
main(int argc,char *argv[])
{if(argc!=2)
{
printf("Usage:<./a.out><sourcefile>\n");
exit(0);
}
yyin=fopen(argv[1],"r");//open the file in read mode... argv is the first word after a.out ...that is the file name
yylex();
printf("No: of characters=%d\nNo: of words=%d\nNo: of spaces=%d\nNo: of lines =%d",c,w,s,l);}
```

input and output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ flex count.l
ilm@ilm-HCL-Desktop:~/Downloads/lija$ cc lex.yy.c -lfl
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out input.txt
No: of characters=255
No: of words=50
No: of spaces=49
No: of lines =5
```

Result

The program was successfully executed and output was verified.

EXPERIMENT 10
LEX PROGRAM 3

Aim: Write a program for counting the number of vowels in the string using LEX.

Program

```
%{
#include<stdio.h>
int num_vowel=0;
%}
%%
[\t];
[aeiouAEIOU] {num_vowel++;}
%%
int yywrap()
{
return 1;
}
int main()
{
//lex throug the input
yylex();
printf("\n\nNumber of vowels in the given string: %d \n",num_vowel);
return 0;
}
```

input and output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ flex vowelcount.l
ilm@ilm-HCL-Desktop:~/Downloads/lija$ cc lex.yy.c -lfl
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out

hello how are you my dear friend
hll hw r y my dr frnd

Number of vowels in the given string: 10
```

Result

The program was successfully executed and output was verified.

EXPERIMENT 11
CALCULATROR USING YACC

Aim: Write a program for the working of arithmetic expression (calculator) using YACC.

Program

```
%{
#include<stdio.h>
%}
%union {int a_number;}
%start line /* to indicate the start symbol*/
%type <a_number> exp term factor number digit /* to assign the type int to nonterminals*/
%%
line : exp ';' '\n' {printf("Result is %d\n",$1);}
;
exp : term {$$ = $1;}
| exp '+' term {$$ = $1 + $3;}
| exp '-' term {$$ = $1 - $3;}
;
term : factor {$$ = $1;}
| term '*' factor {$$ = $1 * $3;}
| term '/' factor {$$ = $1 / $3;}
;
factor : number {$$ = $1;}
|'('exp')' {$$ = $2;}
;
number : digit {$$ = $1;}
|number digit {$$ = $1*10 + $2;}
;
digit : '0' {$$ = 0;}
| '1' {$$ = 1;}
| '2' {$$ = 2;}
| '3' {$$ = 3;}
| '4' {$$ = 4;}
| '5' {$$ = 5;}
| '6' {$$ = 6;}
| '7' {$$ = 7;}
| '8' {$$ = 8;}
| '9' {$$ = 9;}
;
%%

int main() {return yyparse();}
int yylex(){ return getchar();}
void yyerror() {printf("abc"); }
```

input and output

ilm@ilm-HCL-Desktop:~/Downloads/lija$ yacc arthexp.y
ilm@ilm-HCL-Desktop:~/Downloads/lija$ cc y.tab.c
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out

23*2;
Result is 46

./a.out

22/2;
Result is 11

Result

The program was successfully executed and output was verified.

EXPERIMENT 12
8085 code generation

Aim: Write a program to perform 8085 code generation.

Program

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

int isp(char item);
void output(char item);
void push(char item);
char pop(void);
char exp[20];
char res[20];
char a[20];
int st[20], value[20];
int top=0,z=0,i=0,op1,op2,l,t,s;
int pc=0x5000;
char x,item;

void main()
{
printf("Enter the infix expression: ");
gets(exp);
l=strlen(exp);
push('#');
while((item=exp[i])!='\0')
{
if(isalpha(exp[i]))
output(item);
else if(item=='+'||item=='*'||item=='-'||item=='/'||item=='^')
push(item);
else if(item=='(')
push(item);
else if(item==')')
{
while((x=pop())!='(')
output(x);
}
else if(isp(x=pop())<isp(item))
{
push(x);
push(item);
```

```
}
else
{
output(x);
push(item);
}
i++;
}
while((x=pop())!='#')
output(x);
printf("Postfix expression: ");
puts(res);
printf("8085 Code Generation:\n");
for(i=0;i<l;i++)
{
if(isalpha(res[i]))
{
printf("%x MVI A,%c\n",pc,res[i]);
pc=pc+2;
printf("%x PUSH A\n",pc);
pc=pc+1;
}
else
{
printf("%x POP B\n",pc);
pc=pc+1;
printf("%x POP A\n",pc);
pc=pc+1;
if(res[i]=='+')
{
printf("%x ADD B\n",pc);
pc=pc+1;
printf("%x PUSH B\n",pc);
pc=pc+1;
}
else if(res[i]=='-')
{
printf("%x SUB B\n",pc);
pc=pc+1;
printf("%x PUSH B\n",pc);
pc=pc+1;
}
else if(res[i]=='*')
{
printf("%x MOV C,A\n",pc);
pc=pc+1;
```

```
printf("%x MVI A,00\n",pc);
pc=pc+2;
t=pc;
printf("%x ADD B\n",pc);
pc=pc+1;
printf("%x DCR C\n",pc);
pc=pc+1;
printf("%x JNZ %x\n",pc,t);
pc=pc+3;
printf("%x PUSH A\n",pc);
pc=pc+1;
}
else if(res[i]=='/')
{
printf("%x MVI C,00\n",pc);
pc=pc+2;
printf("%x INR C\n",pc);
s=pc;
pc=pc+1;
printf("%x SUB B\n",pc);
pc=pc+1;
printf("%x JP %x\n",pc,s);
pc=pc+3;
printf("%x MOV C,A\n",pc);
pc=pc+1;
printf("%x PUSH A\n",pc);
pc=pc+1;
}
else
{
printf("Invalid operator!!!\n");
}
}
}
}

int isp(char item)
{
if((item=='+')||(item=='-'))
return(1);
else if((item=='*')||(item=='/'))
return(2);
else if((item=='^'))
return(3);
else
return(0);
```

```
}

void output(char item)
{
res[z++]=item;
}

void push(char item)
{
a[++top]=item;
}

char pop(void)
{
item=a[top--];
return(item);
}
```

input and output

```
ilm@ilm-HCL-Desktop:~/Downloads/lija$ gcc codgen.c
ilm@ilm-HCL-Desktop:~/Downloads/lija$ ./a.out

Enter the infix expression: a*b
Postfix expression: ab*

8085 Code Generation:

5000 MVI A,a
5002 PUSH A
5003 MVI A,b
5005 PUSH A
5006 POP B
5007 POP A
5008 MOV C,A
5009 MVI A,00
500b ADD B
500c DCR C
500d JNZ 500b
5010 PUSH A
```

Result

The program was successfully executed and output was verified.