# COLLEGE OF ENGINEERING TRIVANDRUM


DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

---

**CSL411**
**COMPILER DESIGN LAB**

---

Bhagya Manjula Sanil Kumar
TVE21CS039
S7 CSE
(2021-2025 BATCH)

# COLLEGE OF ENGINEERING TRIVANDRUM


DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

---

**CSL411**
**COMPILER DESIGN LAB RECORD**

---

Certified that this is a bonafide record of **Bhagya Manjula Sanil Kumar, TVE21CS039** of **S7 CSE**, done as part of Compiler Design Lab during the year 2024-2025.

**FACULTY IN CHARGE** **EXTERNAL EXAMINER**

| Experiment Number | Experiment Name | Date | Signature |
|---|---|---|---|
| 1 | Design and Implement Lexical Analyzer using C | 08/08/2024 | |
| 2 | Implement Lexical Analyzer Using LEX | 08/08/2024 | |
| 3 | LEX Program to Display Number of Lines and Words | 25/07/2024 | |
| 4 | LEX Program to Convert the Substring abc to ABC | 25/07/2024 | |
| 5 | LEX Program to find the Number of Vowels and Consonants | 25/07/2024 | |
| 6 | Generate YACC Specification to Recognize a Valid Arithmetic Expression | 01/08/2024 | |
| 7 | Generate YACC Specification to Recognize a Valid Identifier | 01/08/2024 | |
| 8 | Implementation of Calculator Using LEX and YACC | 01/08/2024 | |
| 9 | Convert BNF Rules into YACC and Write Abstract Syntax Tree | 08/08/2024 | |
| 10 | Program to Find ε-Closure of All States of NFA | 22/08/2024 | |
| 11 | Program to Convert NFA With ε Transition to NFA Without ε Transitions | 22/08/2024 | |
| 12 | Program to Convert NFA to DFA | 22/08/2024 | |
| 13 | Program to Minimise Any Given DFA | 22/08/2024 | |
| 14 | Program to Find First and Follow of Any Grammar | 29/08/2024 | |
| 15 | Design and Implement Recursive Descent Parser | 29/08/2024 | |
| 16 | Construct Shift Reduce Parser | 29/08/2024 | |
| 17 | Program to Perform Constant Propagation | 05/09/2024 | |
| 18 | Program for Intermediate Code | 05/09/2024 | |

| | | | |
|---|---|---|---|
| | Generation | | |
| 19 | Implementation of Back-end Compiler | 12/09/2024 | |

# Contents

# CYCLE 1

# 1 Design and Implement Lexical Analyser Using C

## 1.1 Aim

To Design and implement a lexical analyzer using C language to recognize all valid tokens in the input program. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments

## 1.2 Algorithm

1. Set ‘left‘ and ‘right‘ pointers to 0.
2. Determine the length of the input string.
3. While the right pointer is less than or equal to the length of the string:
    a. Move the ‘right‘ pointer to the right as long as the character at ‘right‘ is not a delimiter.
    b. If a delimiter is encountered and ‘left‘ equals ‘right‘:
        i. Check if the delimiter is an operator. If so, print that it’s an operator.
        ii. Move both pointers forward.
    c. If a delimiter is encountered and ‘left‘ is not equal to ‘right‘:
        i. Extract the substring from ‘left‘ to ‘right - 1‘.
        ii. Determine the type of the substring:
            a. If it’s a keyword, print that it’s a keyword.
            b. If it’s an integer, print that it’s an integer.
            c. If it’s a real number, print that it’s a real number.
            d. If it’s a valid identifier and does not end with a delimiter,
            print that it’s a valid identifier.
            e. If it’s not a valid identifier and does not end with a delimiter,
            print that it’s not a valid identifier.
        iii. Update the ‘left‘ pointer to ‘right‘ and free any allocated memory for the substring.
4. The process continues until the entire string is parsed.

## 1.3 Program

```

#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] <= '9' || isDelimiter(str[0]))
        return false;
    return true;
}

bool isKeyword(char* str) {
    const char* keywords[] = {"if", "else", "while", "do", "break", "continue", "int",
                              "double", "float", "return", "char", "case", "sizeof",
                              "long", "short", "typedef", "switch", "unsigned",
                              "void", "static", "struct", "goto"};
    for (int i = 0; i < sizeof(keywords)/sizeof(keywords[0]); i++) {
        if (!strcmp(str, keywords[i]))
```

```
            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] > '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] > '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]))
            right++;

        if (isDelimiter(str[right]) && left == right) {
            if (isOperator(str[right]))
                printf("'%c' IS AN OPERATOR\n", str[right]);

            right++;
            left = right;
        } else if ((isDelimiter(str[right]) && left != right) || (right == len && left !=
    right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr))
                printf("'%s' IS A KEYWORD\n", subStr);
            else if (isInteger(subStr))
                printf("'%s' IS AN INTEGER\n", subStr);
            else if (isRealNumber(subStr))
                printf("'%s' IS A REAL NUMBER\n", subStr);
            else if (validIdentifier(subStr) && !isDelimiter(str[right - 1]))
                printf("'%s' IS A VALID IDENTIFIER\n", subStr);
            else if (!validIdentifier(subStr) && !isDelimiter(str[right - 1]))
                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

            left = right;
            free(subStr); // Free the allocated memory
        }
    }
}

```

```c
int main() {
    // char str[100] = "int g = 9.8*f;";
    char str[100];
    printf("Enter input in a line: ");
    scanf("%[^\n]s", str);
    parse(str);
    return 0;
}
```

## 1.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp7.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out
'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'c' IS A VALID IDENTIFIER
'char' IS A KEYWORD
'ch' IS A VALID IDENTIFIER
```

## 1.5 Result

Lexical Analyzer for a given program was successfully implemented using C .

# 2 Implement Lexical Analyser Using LEX

## 2.1 Aim

Implement a Lexical Analyzer for a given program using Lex Tool

## 2.2 Algorithm

1. Lex program contains three sections: definitions, rules,
 and user subroutines. Each section must be separated from the others by a line containing
 only the delimiter, %%. The format is as follows:
 definitions
 \%\%
 rules
 \%\%
 user_subroutines
2. In definition section, the variables make up the left column, and their definitions make up
the right column. Any C statements should be enclosed in \%{..}\%.
 Identifier is defined such that the first letter of an identifier is alphabet and remaining
 letters are alphanumeric.
3. In rules section, the left column contains the pattern to be recognized in an input file to
yylex(). The right column contains the C program fragment executed when that pattern is recognized.
 The various patterns are keywords, operators, new line character, number, string, identifier,
 beginning and end of block, comment statements, preprocessor directive statements etc.
4. Each pattern may have a corresponding action, that is, a fragment of C source code to execute
when the pattern is matched.
5. When yylex() matches a string in the input stream, it copies the matched text to an external
character array, yytext, before it executes any actions in the rules section.
6. In user subroutine section, main routine calls yylex().
 yywrap() is used to get more input.
7. The lex command uses the rules and actions contained in file to generate a program,
lex.yy.c, which can be compiled with the cc command.
 That program can then receive input, break the input into the logical pieces defined by
 the rules in file, and run program fragments contained in the actions in file.

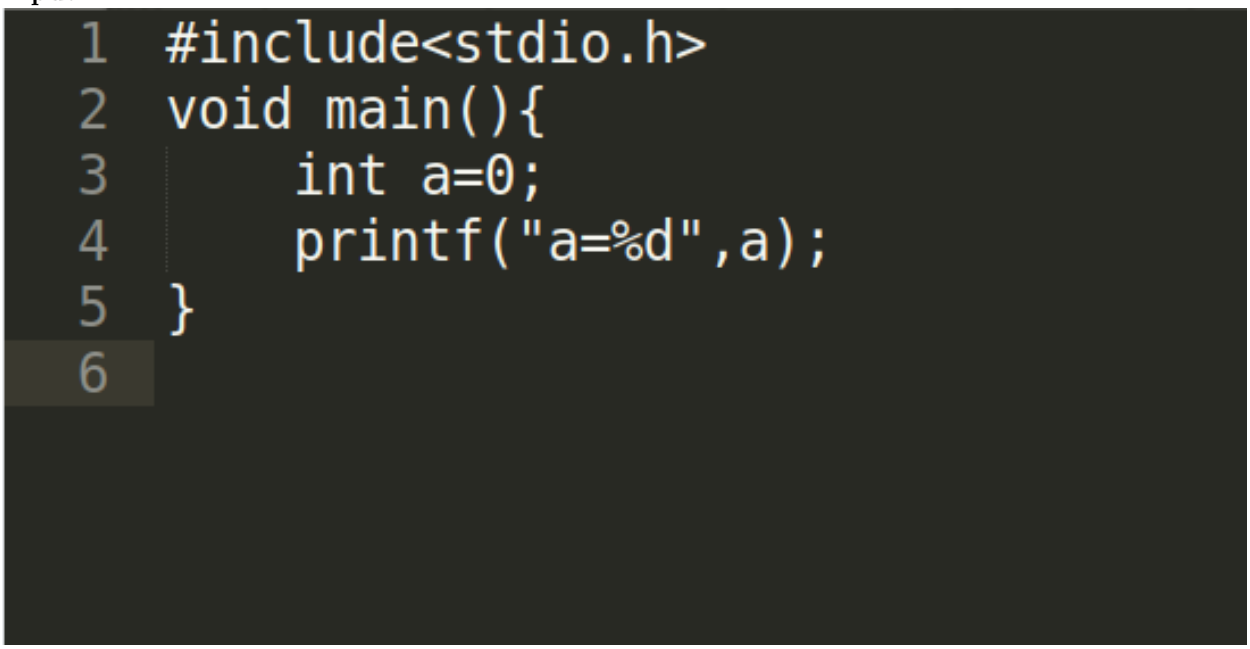
## 2.3 Program

```
1
2  %{
3  int COMMENT=0;
4  %}
5  identifier [a-zA-Z][a-zA-Z0-9]*
6  %%
7  #.* {printf("\n%s is a preprocessor directive",yytext);}
8  int |
9  float |
10 char |
11 double |
12 while |
13 for |
14 struct |
15 typedef |
16 do |
17 if |
18 break |
19 continue |
20 void |
21 switch |
22 return |
23 else |
24 goto {printf("\n\t%s is a keyword",yytext);}
25 "/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}
26 {identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
27 \{  {if(!COMMENT)printf("\n BLOCK BEGINS");}
28 \}  {if(!COMMENT)printf("BLOCK ENDS ");}
```

```
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\( ECHO;
\+ |
\- |
\/ |
\* {if(!COMMENT)printf("\n\t%s is an ARITHMETIC OPERATOR", yytext);}
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

## 2.4 Output

**Input**



**Output**

```
s21a23@administrator-rusa:~/cd_lab$ flex exp9.l
s21a23@administrator-rusa:~/cd_lab$ gcc lex.yy.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out

 #include<stdio.h> is a preprocessor

 void is a keyword
 Function main(
        )

 Block begins

 int is a keyword
 identifier a
 assignment operator =
         0 is a Number;

 Function printf(
         "a=%d" is a String,
 identifier a
        );


 Block ends


 Total No. Of comments are 0
```

## 2.5 Result

Lexical Analyzer for a given program was successfully implemented using Lex Tool and correct output was obtained.

# 3 LEX Program to Display Number of Lines and Words

## 3.1 Aim

Write a lex program to display the number of lines, words and characters in an input text

## 3.2 Algorithm

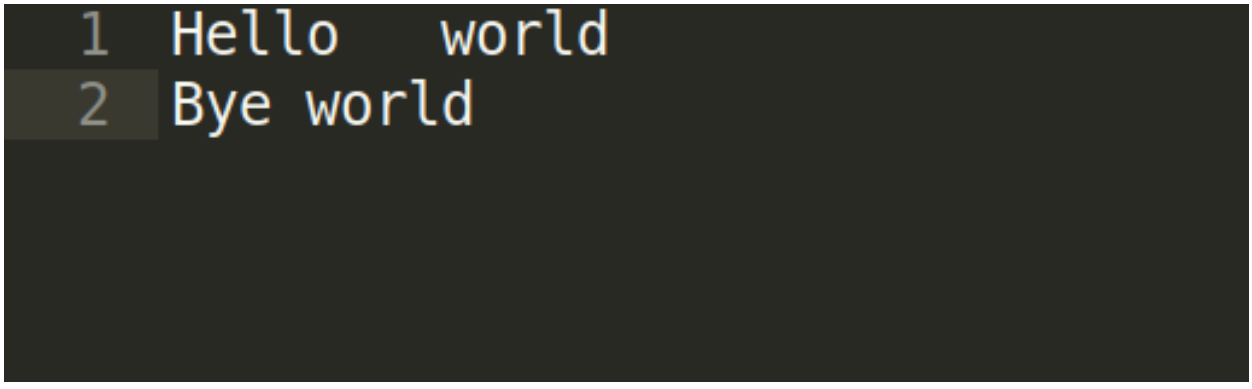
1. Read input.
2. Initialize chars = 0, words = 0 and lines = 0.
3. If token equals [\t\n ]+ , increment words and chars = chars + yylen.
4. Else if token = [\n] increment lines, chars.
5. Else if token = [ ]* increment chars.
6. Print number of words, characters and lines.
7. EXIT

## 3.3 Program

```
%{
int lines_count=1;
int words_count =0;
int chars_count =0;
%}
%%
[\n] { lines_count++; chars_count += yyleng; }
[ \t] { chars_count += yyleng; }
[^\t\n ]+ { words_count++; chars_count += yyleng ; }
%%
int yywrap(){
  return 1;
}
int main()
{
  FILE *file;
  file = fopen("input.txt","r");
  if(!file){
    printf("File error\n");
    return 1;
  }
  yyin = file;
  yylex();
  printf("\n");
  printf("Lines : %d\n", lines_count);
  printf("Words : %d\n", words_count);
  printf("Characters : %d\n", chars_count);
  return 0;

}
```

## 3.4 Output

**input.txt**



### Output

```
s21a23@administrator-rusa:~/cd_lab$ lex exp1.l
s21a23@administrator-rusa:~/cd_lab$ gcc lex.yy.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out
The number of lines = 1
The number of spaces = 2
The number of words = 4
The number of characters are = 21
```

## 3.5 Result

Lex program to display the number of lines, words and characters in an input text was successfully implemented and correct output was obtained.

# 4 LEX Program to Convert the Substring abc to ABC

## 4.1 Aim

Write a LEX Program to convert the substring abc to ABC from the given input string

## 4.2 Algorithm

1. Read input string.
2. If a substring "abc" is identified using (abc), replace it by
   using the string "ABC".
3. EXIT

## 4.3 Program

```
%{
  #include <stdio.h>
%}
%%
"\n" {printf("\n"); return 0;}
"abc" printf("ABC");
. printf("%s",yytext);
%%
int yywrap(){return 1;}
int main(){
printf("Enter string: ");
yylex();
return 0;
}
```

## 4.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ flex exp3.l
s21a23@administrator-rusa:~/cd_lab$ gcc lex.yy.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out

Enter string: helloabc
helloABC
```

## 4.5 Result

LEX Program to convert the substring abc to ABC from the given input string was successfully implemented and correct output was obtained.

# 5 LEX Program to find the Number of Vowels and Consonants

## 5.1 Aim

Write a lex program to find out the total number of vowels and consonants from the given input string

## 5.2 Algorithm

1. Read Input string.
2. Initialize vowels =0, consonants = 0.
3. If character matches vowels in uppercase or lowercase
   [aeiouAEIOU] increment vowels.
4. Else if character belongs to [a-zA-Z] increment consonants.
5. Print number of vowels and consonants.
6. EXIT

## 5.3 Program

```
%{
int vow_count=0;
int const_count =0;
%}
%%
[aeiouAEIOU] {vow_count++;}
[a-zA-Z] {const_count++;}
"\n" {return 0;}
%%
int yywrap(){}
int main()
{
  printf("Enter the string : ");
  yylex();
  printf("Vowel count : %d\n", vow_count);
  printf("Consonant count: %d\n", const_count);
  return 0;

}
```

## 5.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ lex exp2.l
s21a23@administrator-rusa:~/cd_lab$ gcc lex.yy.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out
Enter the string: hello world
 Number of vowels: 3
Number of consonants: 7
```

## 5.5 Result

Lex program to find out the total number of vowels and consonants from the given input string was successfully implemented and correct output was obtained.

# 6 Generate YACC Specification to Recognize a Valid Arithmetic Expression

## 6.1 Aim

Generate a YACC specification to recognize a valid arithmetic expression that uses operators +, – , *,/ and parenthesis.

## 6.2 Algorithm

```
Grammar rules are as follows:
E->T
T->T+T | T-T | T*T | T/T | -ID | -NUMBER
    | NUMBER | ID | (T)

Lex Program:
1.Begin
2.If 0-9 return number
3.Else if a-z A-Z return ID
4.Else if \t,\n return 0
5.else return yytext[0]
6.yywrap()
YACC Program:
1.Identify number and ID as token
2.Define +,-,*,/ to do operations from left side
3.Define E AS E+E, E-E, E*E, E/E, or (E) or number or ID
4.Enter expression from user
5.yyparse() function is called
6.If valid print "expression is valid"
7.If yyerror() produce error, print "Expression is invalid"
```

## 6.3 Program

**lex code**

```
%{
  #include "y.tab.h"
%}
%%
[0-9]+ { return NUMBER ;}
[a-zA-Z0-9]+ { return ID ;}
\n { return NL ;}
. { return yytext[0];}
%%
int yywrap() {
return 1;
}
```

**yacc code**

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%%
valid : E NL { printf("Expression is valid !!\n"); exit(0);}
E : E '+' E
| E '-' E
| E '*' E
| E '/' E
| NUMBER
| ID
| '(' E ')' ;
%%
```

```c
int main() {
printf ("Enter the expression : ") ;
yyparse() ;
}

int yyerror () {
 printf ("Invalid expression\n") ;
 exit(1) ;
}
```

## 6.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ ./a.out
Enter the expression: a+b

Expression is valid !!
^C
s21a23@administrator-rusa:~/cd_lab$ ./a.out
Enter the expression: a++
Invalid expressions21a23@administrator-rusa:~/cd_lab$
```

## 6.5 Result

YACC specification to recognize a valid arithmetic expression that uses operators +, – , *,/ and parenthesis was implemented and correct output was obtained.

# 7 Generate YACC Specification to Recognize a Valid Identifier

## 7.1 Aim

Generate a YACC specification to recognize a valid identifier which starts with a letter followed by any number of letters or digits.

## 7.2 Algorithm

```
1. Read the expression
2. Evaluate the expression using grammar rules specified using YACC.
The grammar rules are as follows:
stmt->variable NL
variable->LETTER alphanumeric
alphanumeric->LETTER alphanumeric
 | DIGIT alphanumeric
 | UND alphanumeric
 | LETTER
 | DIGIT
 | UND
 3. Print the result
```

## 7.3 Program

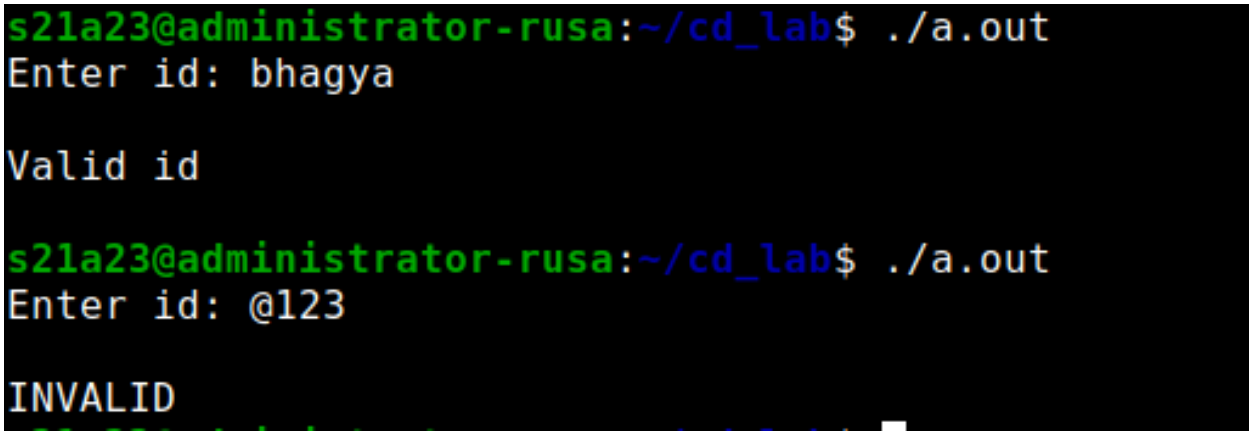
### lex code

```

%{
# include "y.tab.h"
%}

%%
[a-zA-Z] { return LETTER ;}
[0-9] { return DIGIT ;}
[\n] { return NL ;}
[_] { return UND ;}
. { return yytext [0];}
%%
```

### yacc code

```
%token DIGIT LETTER NL UND

%%
stmt : variable NL {
printf ("\nValid id\n\n"); exit (0);}
;
variable : LETTER alphanumeric
;
alphanumeric : LETTER alphanumeric
| DIGIT alphanumeric
| UND alphanumeric
| LETTER
| DIGIT
| UND
;
%%
int yyerror( char * msg )
{
 printf ("\nINVALID \n") ;
 exit(0) ;
}
void main ()
{
 printf (" Enter id : ") ;
 yyparse () ;
}
```

### 7.4 Output




### 7.5 Result

YACC specification to recognize a valid identifier was implemented and correct output was obtained.

# 8 Calculator using LEX and YACC

## 8.1 Aim

Implementation of Calculator using LEX and YACC.

## 8.2 Algorithm

1. Read the expression
2. Evaluate the expression using grammar rules specified using YACC. The grammar rules are as follows:
E->T
T->T+T
| T-T | T*T | T/T | -ID | -NUMBER
| NUMBER | ID | (T)
3. At each stage evaluate the expression value
4. Print the result

## 8.3 Program

**lex code**

```
%{
  /* Definition section*/
  #include "y.tab.h"
  extern yylval;
%}

%%
[0-9]+ {
       yylval = atoi(yytext);
       return NUMBER;
       }

[a-zA-Z]+ { return ID; }
[ \t]+     ; /*For skipping whitespaces*/

\n     { return 0; }
.    { return yytext[0]; }

%%
```

**yacc code**

```
%{
  /* Definition section */
#include <stdio.h>
%}

%token NUMBER ID
// setting the precedence
// and associativity of operators
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

/* Rule Section */
%%
E : T
{
  printf("Result = %d\n", $$);
  return 0;
}

T :
  T '+' T { $$ = $1 + $3; }
  | T '-' T { $$ = $1 - $3; }
  | T '*' T { $$ = $1 * $3; }
  | T '/' T { $$ = $1 / $3; }
  | T '%' T { $$ = $1 % $3; }
  | '-' NUMBER { $$ = -$2; }
  | '-' ID { $$ = -$2; }
```

```
  | '(' T ')' { $$ = $2; }
  | NUMBER { $$ = $1; }
  | ID { $$ = $1; };
%%

int main() {
  printf("Enter the expression\n");
  yyparse();
}

/* For printing error messages */
int yyerror(char* s) {
  printf("\nExpression is invalid\n");
}
```

## 8.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ ./a.out

Enter expression: 3+4

Result: 7

Expression is valid
```

## 8.5 Result

Implementation of Calculator using LEX and YACC was successfully done and correct output was obtained.

# 9 Convert BNF Rules into YACC and Write Abstract Syntax Tree

## 9.1 Aim

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

## 9.2 Algorithm

1. Start
2. Read the input file line by line.
3. Convert it in to abstract syntax tree using three address code.
4. Represent three address code in the form of quadruple tabular form.
5. Stop

## 9.3 Program

### lex code

```
%{
#include "y.tab.h"
#include <stdio.h>
#include <string.h>
int LineNo = 1;
%}

identifier [a-zA-Z][a-zA-Z0-9_]*
number [0-9]+|([0-9]*\.[0-9]+)

%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int |
char |
float return TYPE;
{identifier} { strcpy(yylval.var, yytext); return VAR; }
{number} { strcpy(yylval.var, yytext); return NUM; }
\< |
\> |
\>= |
\<= |
== { strcpy(yylval.var, yytext); return RELOP; }
[ \t] ;
\n LineNo++;
. return yytext[0];
%%
```

### yacc code

```
%{
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

struct quad {
    char op[5];
    char arg1[10];
    char arg2[10];
    char result[10];
} QUAD[30];

struct stack {
    int items[100];
    int top;
} stk;

int Index = 0, tIndex = 0, StNo, Ind, tInd;
extern int LineNo;

void AddQuadruple(char op[5], char arg1[10], char arg2[10], char result[10]);
```

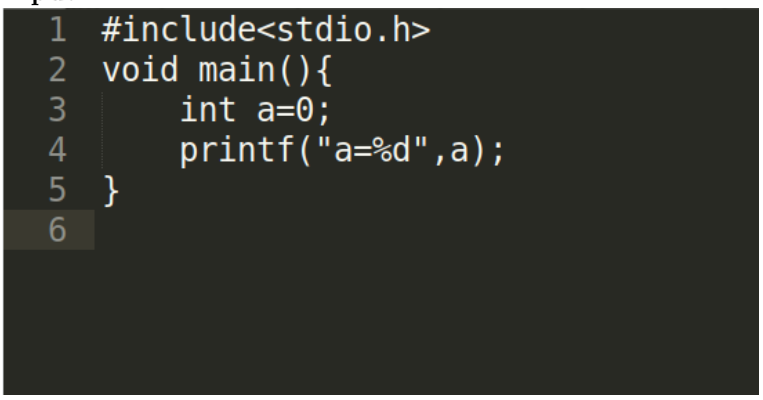
```
int pop();
void push(int data);
int yyerror();
int yylex();
%}

%union { char var[10]; }
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'

%%
PROGRAM : MAIN BLOCK;
BLOCK : '{' CODE '}';
CODE : BLOCK
     | STATEMENT CODE
     | STATEMENT;
STATEMENT : DESCT ';'
          | ASSIGNMENT ';'
          | CONDST
          | WHILEST;
DESCT : TYPE VARLIST;
VARLIST : VAR ',' VARLIST
        | VAR;
ASSIGNMENT : VAR '=' EXPR {
    strcpy(QUAD[Index].op, "=");
    strcpy(QUAD[Index].arg1, $3);
    strcpy(QUAD[Index].arg2, "");
    strcpy(QUAD[Index].result, $1);
    strcpy($$, QUAD[Index++].result);
};
EXPR : EXPR '+' EXPR { AddQuadruple("+", $1, $3, $$); }
     | EXPR '-' EXPR { AddQuadruple("-", $1, $3, $$); }
     | EXPR '*' EXPR { AddQuadruple("*", $1, $3, $$); }
     | EXPR '/' EXPR { AddQuadruple("/", $1, $3, $$); }
     | '-' EXPR { AddQuadruple("UMIN", $2, "", $$); }
     | '(' EXPR ')' { strcpy($$, $2); }
     | VAR
     | NUM;
CONDST : IFST {
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", Index);
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", Index);
}
| IFST ELSEST;
IFST : IF '(' CONDITION ')' {
    strcpy(QUAD[Index].op, "==");
    strcpy(QUAD[Index].arg1, $3);
    strcpy(QUAD[Index].arg2, "FALSE");
    strcpy(QUAD[Index].result, "-1");
    push(Index);
    Index++;
}
BLOCK {
    strcpy(QUAD[Index].op, "GOTO");
    strcpy(QUAD[Index].arg1, "");
    strcpy(QUAD[Index].arg2, "");
    strcpy(QUAD[Index].result, "-1");
    push(Index);
    Index++;
};
ELSEST : ELSE {
    tInd = pop();
    Ind = pop();
    push(tInd);
    sprintf(QUAD[Ind].result, "%d", Index);
}
BLOCK {
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", Index);
};
CONDITION : VAR RELOP VAR { AddQuadruple($2, $1, $3, $$); StNo = Index - 1; }
```

```
          | VAR
          | NUM;
WHILEST : WHILELOOP {
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", StNo);
    Ind = pop();
    sprintf(QUAD[Ind].result, "%d", Index);
};
WHILELOOP : WHILE '(' CONDITION ')' {
    strcpy(QUAD[Index].op, "==");
    strcpy(QUAD[Index].arg1, $3);
    strcpy(QUAD[Index].arg2, "FALSE");
    strcpy(QUAD[Index].result, "-1");
    push(Index);
    Index++;
}
BLOCK {
    strcpy(QUAD[Index].op, "GOTO");
    strcpy(QUAD[Index].arg1, "");
    strcpy(QUAD[Index].arg2, "");
    strcpy(QUAD[Index].result, "-1");
    push(Index);
    Index++;
};
%%

extern FILE *yyin;
int main(int argc, char *argv[]) {
    FILE *fp;
    int i;
    if (argc > 1) {
        fp = fopen(argv[1], "r");
        if (!fp) {
            printf("\n File not found");
            exit(0);
        }
        yyin = fp;
    }
    yyparse();
    printf("\n\n\t\t--------------------------------------\n\t\tPos\tOperator\tArg1\tArg2\
    tResult\n\t\t--------------------------------------");
    for (i = 0; i < Index; i++) {
        printf("\n\t\t%d\t%s\t%s\t%s\t%s", i, QUAD[i].op, QUAD[i].arg1, QUAD[i].arg2, QUAD[
    i].result);
    }
    printf("\n\t\t----------------------");
    printf("\n\n");
    return 0;
}

void push(int data) {
    stk.top++;
    if (stk.top == 100) {
        printf("\n Stack overflow\n");
        exit(0);
    }
    stk.items[stk.top] = data;
}

int pop() {
    int data;
    if (stk.top == -1) {
        printf("\n Stack underflow\n");
        exit(0);
    }
    data = stk.items[stk.top--];
    return data;
}

void AddQuadruple(char op[5], char arg1[10], char arg2[10], char result[10]) {
    strcpy(QUAD[Index].op, op);
    strcpy(QUAD[Index].arg1, arg1);
    strcpy(QUAD[Index].arg2, arg2);
    sprintf(QUAD[Index].result, "t%d", tIndex++);
    strcpy(result, QUAD[Index++].result);
```

```
}

int yyerror() {
    printf("\n Error on line no: %d", LineNo);
}
```

## 9.4 Output

**Input**



**Output**

```
s21a23@administrator-rusa:~/cd_lab$ flex exp8.l
s21a23@administrator-rusa:~/cd_lab$ yacc -d exp8.y
s21a23@administrator-rusa:~/cd_lab$ gcc lex.yy.c y.tab.c -ll
s21a23@administrator-rusa:~/cd_lab$ ./a.out exp8.c

Error on line no: 9

                ----------------------------------------
                Pos Operator Arg1 Arg2 Result
                ----------------------------------------
                0       =       10              a
                1       =       20              b
                2       +       a       b       t0
                3       =       t0              c
                ----------------------------------------
```

## 9.5 Result

The program using YACC has been executed succesfully.

# CYCLE 2

# 10 Program to find $\epsilon$ – closure of All States of NFA

## 10.1 Aim

Program to find  – closure of all states of any given NFA with  transition.

## 10.2 Algorithm

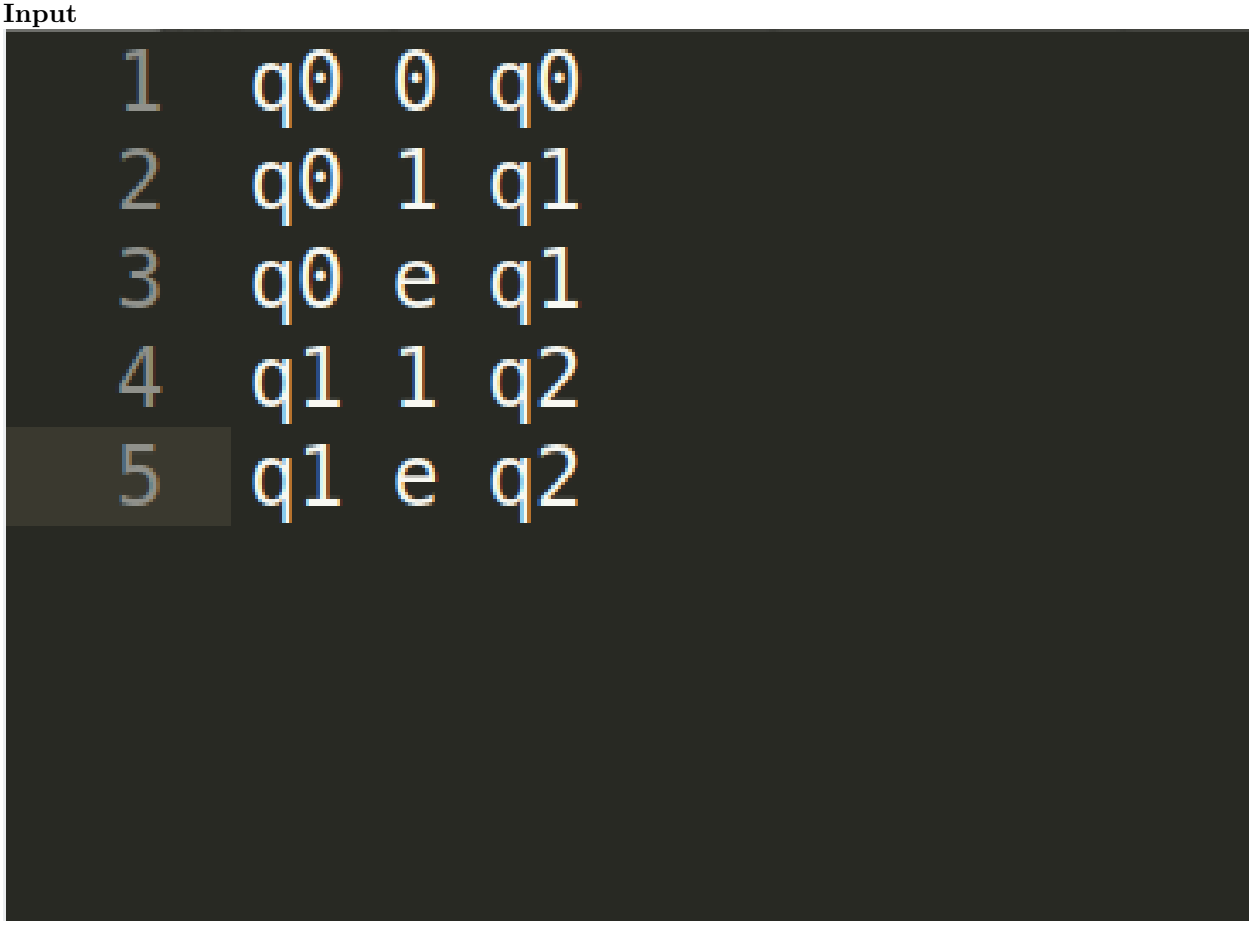
1. Start.
2. Input the no of states as n according to the input text file.
3. Input the states of NFA.
4. For each state, add itself to result as an epsilon transition.
5. While reading the input file, check if there is an epsilon transition.
6. If there is an epsilon transition,add that state to the result.
7. Print result.
8. Repeat 4 to 7 for all n.
9. Stop.

## 10.3 Program

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

int n; //number of states

void closure(int state, int matrix[][n]){
  for(int i=0;i<n;i++){
    if(matrix[state][i] == 1){
      // calculated in main if the i is reachable from the state
      printf(" , q%d",i );
      closure(i,matrix);
      // if the ith state reachable then null or epsilion transitions from i also appended

    }
  }
  return;
}

int main() {
  FILE* INPUT = fopen("input.txt","r");
  char state1[10],input[10],state2[10];
  int s1,s2;

  printf("Enter the number of states : ");
  scanf("%d",&n);
  int matrix[n][n];
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < n; ++j)
    {
      /* code */
      matrix[i][j]=0;
    }
  }
  while(fscanf(INPUT,"%s%s%s", state1,input,state2) != EOF){
    if(strcmp(input,"e")==0){
      s1 = state1[1]-'0';
      s2 = state2[1]-'0';
      matrix[s1][s2] = 1;
    }
  }
  printf("epsilion closure\n");
  for (int i = 0; i < n; ++i)
  {
    printf("q%d: q%d", i,i);
    closure(i,matrix);
    printf("\n");
  }
  return 0;
}
```

## 10.4 Output

**Input**




**Output**

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp10.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out
Enter the number of states: 3
Epsilon closure of
q0 : q0, q1, q2
q1 : q1, q2
q2 : q2
s21a23@administrator-rusa:~/cd_lab$
```

## 10.5 Result

Successfully implemented closure generation of epsilon transitions of all states of a NFA.

# 11 Program to Convert NFA WIth $\epsilon$ Transition to NFA Without $\epsilon$ Transition

## 11.1 Aim

Write a program to convert NFA with epsilon transition to NFA without epsilon transition.

## 11.2 Algorithm

1. Start.
2. Input the no of nodes.
3. Input no. of alphabets.
4. Input no. of transitions.
5. Input the state table.
6. Find epsilon-closure of each state.
7. Find transitions using epsilon-closure .
8. Step 7 is repeated for each input symbol and for each state of given NFA.
9. By using the resultant status,the transition table for equivalent NFA without epsilon is built.
10. Stop

## 11.3 Program

```

#include <stdio.h>
#include <stdlib.h>

// represents transitions in the NFA
struct node {
  int st;
  struct node* link;
};

// hold a set of each DFA states
struct node1{
  int nst[20];
};

void insert(int,char,int);
int findalpha(char);
void findfinalstate(void);
int insertdfastate(struct node1);
int compare(struct node1, struct node1);
void printnewstate(struct node1);
static int set[20],nostate,noalpha,notransition,nofinal,start,finalstate[20],r,s;
int complete = -1;
char c,alphabet[20];
// array to store DFA states
struct node1 hash[20];
// transition table
struct node* transition[20][20] = {NULL};
void main(){
  printf("Enter the number of alphabets: ");
  scanf("%d",&noalpha);
  printf("Enter each alphabet\n");
  getchar();
  for (int i = 0; i < noalpha; ++i)
  {
    alphabet[i] = getchar();
    getchar();
  }

  printf("Enter the number of states: ");
  scanf("%d",&nostate);


  printf("Enter the start state: ");
  scanf("%d",&start);


```

```
  printf("Enter the number of final states: ");
  scanf("%d",&nofinal);
  printf("Enter the final states: \n");
  for (int i = 0; i < nofinal; ++i)
  {
    scanf("%d",&finalstate[i]);
  }

  printf("Enter the number of transitions: ");
  scanf("%d",&notransition);

  for (int i = 0; i < notransition; ++i)
  {
    scanf("%d %c %d",&r,&c,&s);
    insert(r,c,s);
  }

  // preparing to store
  for (int i = 0; i < 20; ++i)
  {
    for (int j = 0; j < 20; ++j)
    {
      hash[i].nst[j] = 0;
    }
  }


  complete=-1; // track last state
  // indicate number of DFA identified and stored
  int i=-1;
  // indicating if all states explored
  printf("Equivalent DFA ....\n");
  printf("Transitions of DFA \n");

  struct node1 newstate={0};
  struct node1 tmpstate={0};
  struct node* temp;
  int c,l;

  newstate.nst[start] = start;
  insertdfastate(newstate);
  while(i != complete){
    i++;
    newstate=hash[i];
    for (int k = 0; k < noalpha; ++k)
    {
      c=0;
      for (int j = 1; j <= nostate; ++j)
      {
        set[j]=0;
      }
      for (int j = 1; j <= nostate; ++j)
      {
        l = newstate.nst[j];
        if(l != 0){
          temp = transition[l][k];
          while(temp != NULL){
            if (set[temp-> st] == 0){
              c++;
              set[temp->st] = temp->st;
            }
            temp = temp->link;
          }
        }
      }
      printf("\n");
      if(c != 0){
        for (int m = 1; m <= nostate; ++m)
        {
          tmpstate.nst[m] = set[m];
        }
        insertdfastate(tmpstate);
        printnewstate(newstate);

        printf("%c\t",alphabet[k] );
```

```
        printnewstate(tmpstate);
        printf("\n");
      }
      else{
        printnewstate(newstate);
        printf("%c\t",alphabet[k]);
        printf("NULL\n");
      }
    }
  }
  printf("\n");
  printf("States of DFA\n");
  for (int i = 0; i <= complete; ++i)
  {
    printnewstate(hash[i]);

  }
  printf("alphabets: ");
  for (int j = 0; j < noalpha; ++j)
  {
    printf("%c \t", alphabet[j]);
  }
  printf("\n");
  printf("Start state: q%d\n",start );
  printf("Final states: ");
  findfinalstate();
  printf("\n");

}
// adds a new DFA state into hash if not already present
int insertdfastate(struct node1 newstate){
  for (int i = 0; i <= complete; ++i)
  {
    if(compare(hash[i],newstate)){
      return 0;
    }
  }
  complete++;
  hash[complete] = newstate;
  return 1;
}

// two DFA states compared
int compare(struct node1 a, struct node1 b){
  for (int i = 1; i <= nostate; ++i)
  {
    if(a.nst[i] != b.nst[i]){
      return 0;
    }
  }
  return 1;
}
// adds transition to transition table
void insert(int r,char c, int s){
  struct node* temp;
  int j = findalpha(c);
  if(j==999){
    printf("Error\n");
    exit(0);
  }
  temp = (struct node*)malloc(sizeof(struct node));
  temp->st = s;
  temp->link = transition[r][j];
  transition[r][j] = temp;
}

// finds index of given alphabet
int findalpha(char c){
  for (int i = 0; i < noalpha; ++i)
  {
    if(alphabet[i] == c){
      return i;
    }
  }
  return 999;
```

```
}

// identifies and prints final states
void findfinalstate(){
  for (int i = 0; i <= complete; ++i)
  {
    for (int j = 1; j <= nostate; ++j)
    {
      for (int k = 0; k < nofinal; ++k)
      {
        if(hash[i].nst[j] == finalstate[k]){
          printnewstate(hash[i]);
          printf("\t");
          j = nostate;
          break;
        }
      }
    }
  }
  printf("\n");
}

// print in readable format
void printnewstate(struct node1 state){
  printf("{");
  for (int i = 1; i <= nostate; ++i)
  {
    if(state.nst[i] != 0){
      printf("q%d, ",state.nst[i] );
    }
  }
  printf("}\t");
}
```

## 11.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp11.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out
Enter the number of alphabets:
4

Enter alphabets: a b c e
Enter the number of states?
3
Enter the start state:
1
Enter the number of final states?
1
Enter the final states:
3
Enter the number of transitions:
5

Enter transitions:
1 a 1
1 e 2
2 b 2
2 e 3
3 c 3

Equivalent NFA without epsilon
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Start state: {q1,q2,q3,}
Alphabets: a b c e
States: {q1,q2,q3,}     {q2,q3,}        {q3,}
Transitions are:

{q1,q2,q3,}     a       {q1,q2,q3,}
{q1,q2,q3,}     b       {q2,q3,}
{q1,q2,q3,}     c       {q3,}
{q2,q3,}        a       {}
{q2,q3,}        b       {q2,q3,}
{q2,q3,}        c       {q3,}
{q3,}   a       {}
{q3,}   b       {}
{q3,}   c       {q3,}
Final states: {q1,q2,q3,}       {q2,q3,}        {q3,}
```

## 11.5 Result

Successfully implemented a program to convert NFA with epsilon transition to NFA without epsilon transition.

# 12 Program to Convert NFA to DFA

## 12.1 Aim

Write a program to convert NFA to DFA.

## 12.2 Algorithm

1. Start
2. Input the required array ie, set of alphabets, set of states, initial state, set of final states, transitions.
3. Initially Q' = $\phi$
4. Add q0 of NFA to Q'. Then find the transitions from this start state.
5. In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.
6. In DFA, the final state will be all the states which contain F(final states of NFA)
7. Stop

## 12.3 Program

```

#include <stdio.h>
#include <stdlib.h>

// represents transitions in the NFA
struct node {
  int st;
  struct node* link;
};

// hold a set of each DFA states
struct node1{
  int nst[20];
};

void insert(int,char,int);
int findalpha(char);
void findfinalstate(void);
int insertdfastate(struct node1);
int compare(struct node1, struct node1);
void printnewstate(struct node1);
static int set[20],nostate,noalpha,notransition,nofinal,start,finalstate[20],r,s;
int complete = -1;
char c,alphabet[20];
// array to store DFA states
struct node1 hash[20];
// transition table
struct node* transition[20][20] = {NULL};
void main(){
  printf("Enter the number of alphabets: ");
  scanf("%d",&noalpha);
  printf("Enter each alphabet\n");
  getchar();
  for (int i = 0; i < noalpha; ++i)
  {
    alphabet[i] = getchar();
    getchar();
  }

  printf("Enter the number of states: ");
  scanf("%d",&nostate);


  printf("Enter the start state: ");
  scanf("%d",&start);


  printf("Enter the number of final states: ");
  scanf("%d",&nofinal);
  printf("Enter the final states: \n");
```

```
  for (int i = 0; i < nofinal; ++i)
  {
    scanf("%d",&finalstate[i]);
  }

  printf("Enter the number of transitions: ");
  scanf("%d",&notransition);

  for (int i = 0; i < notransition; ++i)
  {
    scanf("%d %c %d",&r,&c,&s);
    insert(r,c,s);
  }

  // preparing to store
  for (int i = 0; i < 20; ++i)
  {
    for (int j = 0; j < 20; ++j)
    {
      hash[i].nst[j] = 0;
    }
  }


  complete=-1; // track last state
  // indicate number of DFA identified and stored
  int i=-1;
  // indicating if all states explored
  printf("Equivalent DFA ....\n");
  printf("Transitions of DFA \n");

  struct node1 newstate={0};
  struct node1 tmpstate={0};
  struct node* temp;
  int c,l;

  newstate.nst[start] = start;
  insertdfastate(newstate);
  while(i != complete){
    i++;
    newstate=hash[i];
    for (int k = 0; k < noalpha; ++k)
    {
      c=0;
      for (int j = 1; j <= nostate; ++j)
      {
        set[j]=0;
      }
      for (int j = 1; j <= nostate; ++j)
      {
        l = newstate.nst[j];
        if(l != 0){
          temp = transition[l][k];
          while(temp != NULL){
            if (set[temp-> st] == 0){
              c++;
              set[temp->st] = temp->st;
            }
            temp = temp->link;
          }
        }
      }
      printf("\n");
      if(c != 0){
        for (int m = 1; m <= nostate; ++m)
        {
          tmpstate.nst[m] = set[m];
        }
        insertdfastate(tmpstate);
        printnewstate(newstate);

        printf("%c\t",alphabet[k] );
        printnewstate(tmpstate);
        printf("\n");
      }
```

```
      else{
        printnewstate(newstate);
        printf("%c\t",alphabet[k]);
        printf("NULL\n");
      }
    }
  }
  printf("\n");
  printf("States of DFA\n");
  for (int i = 0; i <= complete; ++i)
  {
    printnewstate(hash[i]);

  }
  printf("alphabets: ");
  for (int j = 0; j < noalpha; ++j)
  {
    printf("%c \t", alphabet[j]);
  }
  printf("\n");
  printf("Start state: q%d\n",start );
  printf("Final states: ");
  findfinalstate();
  printf("\n");

}
// adds a new DFA state into hash if not already present
int insertdfastate(struct node1 newstate){
  for (int i = 0; i <= complete; ++i)
  {
    if(compare(hash[i],newstate)){
      return 0;
    }
  }
  complete++;
  hash[complete] = newstate;
  return 1;
}

// two DFA states compared
int compare(struct node1 a, struct node1 b){
  for (int i = 1; i <= nostate; ++i)
  {
    if(a.nst[i] != b.nst[i]){
      return 0;
    }
  }
  return 1;
}
// adds transition to transition table
void insert(int r,char c, int s){
  struct node* temp;
  int j = findalpha(c);
  if(j==999){
    printf("Error\n");
    exit(0);
  }
  temp = (struct node*)malloc(sizeof(struct node));
  temp->st = s;
  temp->link = transition[r][j];
  transition[r][j] = temp;
}

// finds index of given alphabet
int findalpha(char c){
  for (int i = 0; i < noalpha; ++i)
  {
    if(alphabet[i] == c){
      return i;
    }
  }
  return 999;
}

// identifies and prints final states
```

```
void findfinalstate(){
  for (int i = 0; i <= complete; ++i)
  {
    for (int j = 1; j <= nostate; ++j)
    {
      for (int k = 0; k < nofinal; ++k)
      {
        if(hash[i].nst[j] == finalstate[k]){
          printnewstate(hash[i]);
          printf("\t");
          j = nostate;
          break;
        }
      }
    }
  }
  printf("\n");
}

// print in readable format
void printnewstate(struct node1 state){
  printf("{");
  for (int i = 1; i <= nostate; ++i)
  {
    if(state.nst[i] != 0){
      printf("q%d, ",state.nst[i] );
    }
  }
  printf("}\t");
}
```

## 12.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ ./a.out

Enter No of alphabets and alphabets :
2
a
b
Enter the number of states :
4
Enter the start state :
1
Enter the number of final states :
2
Enter the final states :
3 4
Enter no of transition :
8

Enter transition :
1 a 1
1 b 1
1 a 2
2 b 2
2 a 3
3 a 4
3 b 4
4 b 3

Equivalent DFA .....
Transitions of DFA

{q1,}   a       {q1,q2,}

{q1,}   b       {q1,}

{q1,q2,}        a       {q1,q2,q3,}

{q1,q2,}        b       {q1,q2,}

{q1,q2,q3,}     a       {q1,q2,q3,q4,}

{q1,q2,q3,}     b       {q1,q2,q4,}

{q1,q2,q3,q4,}  a       {q1,q2,q3,q4,}

{q1,q2,q3,q4,}  b       {q1,q2,q3,q4,}

{q1,q2,q4,}     a       {q1,q2,q3,}

{q1,q2,q4,}     b       {q1,q2,q3,}

States of DFA :
{q1,}   {q1,q2,}        {q1,q2,q3,}     {q1,q2,q3,q4,}  {q1,q2,q4,}
Alphabets :
a       b
Start State :
q1
Final states :
{q1,q2,q3,}             {q1,q2,q3,q4,}          {q1,q2,q4,}
s21a23@administrator-rusa:~/cd_lab$
```

## 12.5 Result

Successfully implemented conversion of NFA to DFA.

# 13 Program to Minimise Any Given DFA

## 13.1 Aim

Write a program to minimise any given DFA.

## 13.2 Algorithm

1. Start
2. Input the required array ie, set of alphabets, set of states, initial state, set of final states, transitions.
3. Initially Q' = $\phi$
4. Add q0 of NFA to Q'. Then find the transitions from this start state.
5. In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.
6. In DFA, the final state will be all the states which contain F(final states of NFA)
7. Stop

## 13.3 Program

```

#include <stdio.h>
#include <stdlib.h>

// represents transitions in the NFA
struct node {
  int st;
  struct node* link;
};

// hold a set of each DFA states
struct node1{
  int nst[20];
};

void insert(int,char,int);
int findalpha(char);
void findfinalstate(void);
int insertdfastate(struct node1);
int compare(struct node1, struct node1);
void printnewstate(struct node1);
static int set[20],nostate,noalpha,notransition,nofinal,start,finalstate[20],r,s;
int complete = -1;
char c,alphabet[20];
// array to store DFA states
struct node1 hash[20];
// transition table
struct node* transition[20][20] = {NULL};
void main(){
  printf("Enter the number of alphabets: ");
  scanf("%d",&noalpha);
  printf("Enter each alphabet\n");
  getchar();
  for (int i = 0; i < noalpha; ++i)
  {
    alphabet[i] = getchar();
    getchar();
  }

  printf("Enter the number of states: ");
  scanf("%d",&nostate);


  printf("Enter the start state: ");
  scanf("%d",&start);


  printf("Enter the number of final states: ");
  scanf("%d",&nofinal);
  printf("Enter the final states: \n");
```

```
  for (int i = 0; i < nofinal; ++i)
  {
    scanf("%d",&finalstate[i]);
  }

  printf("Enter the number of transitions: ");
  scanf("%d",&notransition);

  for (int i = 0; i < notransition; ++i)
  {
    scanf("%d %c %d",&r,&c,&s);
    insert(r,c,s);
  }

  // preparing to store
  for (int i = 0; i < 20; ++i)
  {
    for (int j = 0; j < 20; ++j)
    {
      hash[i].nst[j] = 0;
    }
  }


  complete=-1; // track last state
  // indicate number of DFA identified and stored
  int i=-1;
  // indicating if all states explored
  printf("Equivalent DFA ....\n");
  printf("Transitions of DFA \n");

  struct node1 newstate={0};
  struct node1 tmpstate={0};
  struct node* temp;
  int c,l;

  newstate.nst[start] = start;
  insertdfastate(newstate);
  while(i != complete){
    i++;
    newstate=hash[i];
    for (int k = 0; k < noalpha; ++k)
    {
      c=0;
      for (int j = 1; j <= nostate; ++j)
      {
        set[j]=0;
      }
      for (int j = 1; j <= nostate; ++j)
      {
        l = newstate.nst[j];
        if(l != 0){
          temp = transition[l][k];
          while(temp != NULL){
            if (set[temp-> st] == 0){
              c++;
              set[temp->st] = temp->st;
            }
            temp = temp->link;
          }
        }
      }
      printf("\n");
      if(c != 0){
        for (int m = 1; m <= nostate; ++m)
        {
          tmpstate.nst[m] = set[m];
        }
        insertdfastate(tmpstate);
        printnewstate(newstate);

        printf("%c\t",alphabet[k] );
        printnewstate(tmpstate);
        printf("\n");
      }
```

```
      else{
        printnewstate(newstate);
        printf("%c\t",alphabet[k]);
        printf("NULL\n");
      }
    }
  }
  printf("\n");
  printf("States of DFA\n");
  for (int i = 0; i <= complete; ++i)
  {
    printnewstate(hash[i]);

  }
  printf("alphabets: ");
  for (int j = 0; j < noalpha; ++j)
  {
    printf("%c \t", alphabet[j]);
  }
  printf("\n");
  printf("Start state: q%d\n",start );
  printf("Final states: ");
  findfinalstate();
  printf("\n");

}
// adds a new DFA state into hash if not already present
int insertdfastate(struct node1 newstate){
  for (int i = 0; i <= complete; ++i)
  {
    if(compare(hash[i],newstate)){
      return 0;
    }
  }
  complete++;
  hash[complete] = newstate;
  return 1;
}

// two DFA states compared
int compare(struct node1 a, struct node1 b){
  for (int i = 1; i <= nostate; ++i)
  {
    if(a.nst[i] != b.nst[i]){
      return 0;
    }
  }
  return 1;
}
// adds transition to transition table
void insert(int r,char c, int s){
  struct node* temp;
  int j = findalpha(c);
  if(j==999){
    printf("Error\n");
    exit(0);
  }
  temp = (struct node*)malloc(sizeof(struct node));
  temp->st = s;
  temp->link = transition[r][j];
  transition[r][j] = temp;
}

// finds index of given alphabet
int findalpha(char c){
  for (int i = 0; i < noalpha; ++i)
  {
    if(alphabet[i] == c){
      return i;
    }
  }
  return 999;
}

// identifies and prints final states
```

```
void findfinalstate(){
  for (int i = 0; i <= complete; ++i)
  {
    for (int j = 1; j <= nostate; ++j)
    {
      for (int k = 0; k < nofinal; ++k)
      {
        if(hash[i].nst[j] == finalstate[k]){
          printnewstate(hash[i]);
          printf("\t");
          j = nostate;
          break;
        }
      }
    }
  }
  printf("\n");
}

// print in readable format
void printnewstate(struct node1 state){
  printf("{");
  for (int i = 1; i <= nostate; ++i)
  {
    if(state.nst[i] != 0){
      printf("q%d, ",state.nst[i] );
    }
  }
  printf("}\t");
}
```

## 13.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp13.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out

DFA : STATE TRANSITION TABLE
 |  0  1
- - - - -+ - - ----- -----
 A |  B  C
 B |  E  F
 C |  A  A
 D |  F  E
 E |  D  F
 F |  D  E
 Final states = EF

EQUIV. CLASS CANDIDATE ==>  0:[ABCD] 1:[EF]
 0:[ABCD]       -> [BEAF] (0101)
 0:[ABCD]       -> [CFAE] (0101)
 1:[EF] -> [DD] (00)
 1:[EF] -> [FE] (11)

EQUIV. CLASS CANDIDATE ==>  0:[AC] 1:[BD] 2:[EF]
 0:[AC] -> [BA] (10)
 0:[AC] -> [CA] (00)
 1:[BD] -> [EF] (22)
 1:[BD] -> [FE] (22)
 2:[EF] -> [DD] (11)
 2:[EF] -> [FE] (22)

EQUIV. CLASS CANDIDATE ==>  0:[A] 1:[BD] 2:[C] 3:[EF]
 0:[A]  -> [B] (1)
 0:[A]  -> [C] (2)
 1:[BD] -> [EF] (33)
 1:[BD] -> [FE] (33)
 2:[C]  -> [A] (0)
 2:[C]  -> [A] (0)
 3:[EF] -> [DD] (11)
 3:[EF] -> [FE] (33)

DFA : STATE TRANSITION TABLE
 |  0  1
- - - - -+ - - ----- -----
 A |  B  C
 B |  D  D
 C |  A  A
 D |  B  D
 Final states = D
```

## 13.5 Result

Successfully implemented a program to minimise DFA.

# CYCLE 3

# 14 Program to Find First and Follow of Any Grammar

## 14.1 Aim

Write a program to find First and Follow of any given grammar

## 14.2 Algorithm

1. Start
2. Calculating first, $\alpha \rightarrow$ t $\beta$
3. if $\alpha$ is a terminal, then FIRST($\alpha$) = $\alpha$ .
4. if $\alpha$ is a non-terminal and $\alpha \rightarrow \epsilon$ is a production,
then FIRST($\alpha$) = $\epsilon$ .
5. if $\alpha$ is a non-terminal and $\alpha \rightarrow \gamma 1\ \gamma 2\ \gamma 3$ ...
$\gamma$n and any FIRST($\gamma$) contains t
then t is in FIRST($\alpha$).
6. Calculating follow
7. if $\alpha$ is a start symbol, then FOLLOW() = $
8. if $\alpha$ is a non-terminal and has a production $\alpha \rightarrow$ AB, then FIRST(B) is in
FOLLOW(A) except $\epsilon$.
9. if $\alpha$ is a non-terminal and has a production $\alpha \rightarrow$ AB, where B $\epsilon$,
then FOLLOW(A) is in FOLLOW($\alpha$).
10. Stop

## 14.3 Program

```c

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// n number of productions
// m track of number of elements in FIRST or FOLLOW
// a store grammar productions
// f store FIRST or FOLLOW for non terminals
int n, m = 0, p, i = 0, j = 0;
char a[10][10], f[10];

void follow(char c);
void first(char c);

int main() {
    int i, z;
    char c, ch;

    printf("Enter the number of productions:\n");
    scanf("%d", &n);

    printf("Enter the productions:\n");
    for (i = 0; i < n; i++)
        scanf("%s%c", a[i], &ch);

    do {
        m = 0;
        printf("Enter the elements whose first & follow is to be found: ");
        scanf(" %c", &c);  // Note: space before %c to skip any whitespace
        first(c);
        printf("First(%c)={", c);
        for (i = 0; i < m; i++)
            printf("%c", f[i]);
        printf("}\n");

        strcpy(f, " ");
        m = 0;
        follow(c);
        printf("Follow(%c)={", c);
        for (i = 0; i < m; i++)
```

```
            printf("%c", f[i]);
        printf("}\n");

        printf("Continue (0/1)? ");
        scanf("%d%c", &z, &ch);  // Note: space before %c to skip any whitespace
    } while (z == 1);

    return 0;
}

void first(char c) {
    int k;

    // is c terminal then not uppercase
    // add it to first set f
    if (!isupper(c)) {
        f[m++] = c;
        return;
    }
    // for each production rule where c is LHS

    for (k = 0; k < n; k++) {
        if (a[k][0] == c) {
            if (a[k][2] == '$') {
              // indicate epsilon transition so follow
                follow(a[k][0]);
            } else if (islower(a[k][2])) {
              // RHS starts with terminal then add to first
                f[m++] = a[k][2];
            } else {
              // RHS starts with non terminal recursively calculate
                first(a[k][2]);
            }
        }
    }
}

void follow(char c) {
    int k;

    if (a[0][0] == c)
      // starting symbol
        f[m++] = '$';

    for (i = 0; i < n; i++) {
      // each production rule
        for (j = 2; j < strlen(a[i]); j++) {
            if (a[i][j] == c) {
                if (a[i][j + 1] != '\0') {
                  // if symbol follwing c add first of that symbol
                    first(a[i][j + 1]);
                }
                if (a[i][j + 1] == '\0' && a[i][0] != c) {
                  // end of RHS add follow of LHS non terminal
                    follow(a[i][0]);
                }
            }
        }
    }
}
```

## 14.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp14.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out
Enter the number of productions:
5
Enter the productions:
S=AbCd
A=Cf
A=a
C=gE
E=h
Enter the elements whose first & follow is to be found: S
First(S) = {ga}
Follow(S) = {$}
Continue (0/1)? 1
Enter the elements whose first & follow is to be found: C
First(C) = {g}
Follow(C) = {df}
Continue (0/1)? 0
```

## 14.5 Result

Implemented a program to find First and Follow of any given grammar.

# 15 Recursive Descent Parser

## 15.1 Aim

Design and implement a recursive descent parser for a given grammar.

## 15.2 Algorithm

1. Start
2. Input the expression
3. Grammar without left recursion is added to the program
4. The grammar which had been given already is substituted with the right productions until the input expression is developed.
5. Stop

## 15.3 Program

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

#define SUCCESS 1
#define FAILURE 0

int E(),E_prime(),T(),T_prime(),F();

char *cursor;
char s[64];

int main(){
  printf("Enter string\n");
  scanf("%s",s);
  cursor = s;
  printf("\n");
  printf("Input\tAction\n");
  printf("--------------------\n");
  if(E() && *cursor =='\0'){
    printf("--------------------\n");
    printf("Parsed successfully\n");
    return 0;
  }
  else{
    printf("-------------------------\n");
    printf("Error \n");
    return 1;
  }
}

// E-> TE'
int E(){
  printf("%-16s E->TE'\n",cursor );
  if(T()){
    if(E_prime())
      return SUCCESS;

  }
  return FAILURE;
}

// E'-> +TE' | $
int E_prime(){
  if(*cursor == '+'){
    printf("%-16s E'->+TE'\n",cursor );
    cursor++;
    if(T()){
      if(E_prime())
        return SUCCESS;
    }
  }
  else{
    printf("%-16s E'-> $\n", cursor);
    return SUCCESS;
```

```
  }
  return FAILURE;
}

// T-> FT'

int T(){
  printf("%-16s T->FT'\n",cursor );
  if(F()){
    if(T_prime())
      return SUCCESS;

  }
  return FAILURE;
}

// T'-> *FT' | $
int T_prime(){
  if(*cursor == '*'){
    printf("%-16s T'->*FT'\n",cursor );
    cursor++;
    if(F()){
      if(T_prime())
        return SUCCESS;
    }
  }
  else{
    printf("%-16s T'-> $\n", cursor);
    return SUCCESS;
  }
  return FAILURE;
}

// F -> (E) | i

int F(){
  if(*cursor == '('){
    printf("%-16s F->(E)\n",cursor );
    cursor++;
    if(E()){
      if(*cursor == ')'){
        cursor++;
        return SUCCESS;
      }
    }
  }
  else if(*cursor == 'i'){
    printf("%-16s F->I\n",cursor );
    cursor++;
    return SUCCESS;

  }
  return FAILURE;
}
```

## 15.4 Output

```
^Cs21a23@administrator-rusa:~/cd_lab$ gcc exp15.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out

Grammar without left recursion
                E -> TE'
                E' -> +TE' | e
                T -> FT'
                T' -> *FT' | e
                F -> (E) | i
Enter the input expression:i+i
Expressions     Sequence of production rules
E=TE'                       E -> TE'
E=FT'E'                     T -> FT'
E=iT'E'                     F -> i
E=ieE'                      T' -> e
E=i+TE'                     E' -> +TE'
E=i+FT'E'                   T -> FT'
E=i+iT'E'                   F -> i
E=i+ieE'                    T' -> e
E=i+ie                      E' -> e
```

## 15.5 Result

Implemented a recursive descent parser for a given grammar.

# 16 Construct Shift Reduce Parser

## 16.1 Aim

Construct a Shift Reduce Parser for a given language

## 16.2 Algorithm

```
1. START
2. Input the set of productions , symbols and expressions .
3. Read each symbol of the expression .
4. Parse method () is called for each non terminal symbol in the
productions .
5. A non terminal in the right hand side of rewrite rule leads to a
call to parse method for that non - terminal .
6. A terminal symbol on the right hand side of a rewrite rule leads to
consuming that token from input token string .
7. l in the CFG leads to "If else " in the parser .
8. If symbol is not expanded correctly are to input expression ,
backtrack .
9. STOP
Procedure parser
1. from j =1 to t , repeat
2. choose a production A -> x1 , x2 , ... , xi ;
3. from i -1 to k repeat
4. if( xi is a non - terminal )
5. call procedure xi () ;
6. else if( xi equals current input a )
7. advance input to next symbol
8. else backtrack input and reset pointer .
```

## 16.3 Program

```c
#include <stdio.h>
#include <string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main(){
  printf("GRAMMAR is \n E->E+E \n E->E*E \n E->(E) \n E->id\n");
  printf("Enter input string: \n");
  scanf("%s",a);
  c = strlen(a);
  strcpy(act,"SHIFT->");
  printf("Stack \t\t\t Input \t\t Action\n");
  for (i = 0,k=0; j<c; i++,k++,j++)
  {
    if(a[j] == 'i' && a[j+1] == 'd'){
      stk[i] = a[j];
      stk[i+1] = a[j+1];
      stk[i+2] = '\0';
      a[j] = ' ';
      a[j+1] = ' ';
      printf("\n$%s\t\t%s$\t\t%sid\n",stk,a,act );
      check();
    }
    else{
      stk[i] = a[j];
      stk[i+1] = '\0';
      a[j] = ' ';
      printf("\n$%s\t\t%s$\t\t%s%c\n",stk,a,act,stk[i] );
      check();

    }
  }
  printf("\n");

```

```
}

void check(){
  strcpy(ac,"REDUCE TO E");
  for (z=0; z < c; ++z)
  {
    if (stk[z] == 'i' && stk[z+1] == 'd')
    {
      stk[z] = 'E';
      stk[z+1] = '\0';
      printf("\n$%s\t\t%s$\t\t%s",stk,a,ac);
      j++;
    }
  }
  for (z=0; z < c; ++z)
  {
    if (stk[z] == 'E' && stk[z+1] == '+' && stk[z+2] == 'E')
    {
      stk[z] = 'E';
      stk[z+1] = '\0';
      stk[z+2] = '\0';
      printf("\n$%s\t\t%s$\t\t%s",stk,a,ac);
      i-=2;
    }
  }
  for (z=0; z < c; ++z)
  {
    if (stk[z] == 'E' && stk[z+1] == '*' && stk[z+2] == 'E')
    {
      stk[z] = 'E';
      stk[z+1] = '\0';
      stk[z+2] = '\0';
      printf("\n$%s\t\t%s$\t\t%s",stk,a,ac);
      i-=2;
    }
  }
  for (z=0; z < c; ++z)
  {
    if (stk[z] == '(' && stk[z+1] == 'E' && stk[z+2] == ')')
    {
      stk[z] = 'E';
      stk[z+1] = '\0';
      stk[z+2] = '\0';
      printf("\n$%s\t\t%s$\t\t%s",stk,a,ac);
      i-=2;
    }
  }


}
```

## 16.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp16.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out
GRAMMAR is E -> E+E
E -> E*E
E -> (E)
E -> id
enter input string
id+id*id+id
stack   input   action

$id       +id*id+id$    SHIFT -> id
$E        +id*id+id$    REDUCE TO E
$E+        id*id+id$    SHIFT ->symbols
$E+id        *id+id$    SHIFT -> id
$E+E         *id+id$    REDUCE TO E
$E           *id+id$    REDUCE TO E
$E*           id+id$    SHIFT ->symbols
$E*id           +id$    SHIFT -> id
$E*E            +id$    REDUCE TO E
$E              +id$    REDUCE TO E
$E+              id$    SHIFT ->symbols
$E+id              $    SHIFT -> id
$E+E               $    REDUCE TO E
$E                 $    REDUCE TO E
s21a23@administrator-rusa:~/cd_lab$
```

## 16.5 Result

Successfully implemented a Shift Reduce Parser for a given language.

# CYCLE 4

# 17 Constant Propogation

## 17.1 Aim

Write a program to perform constant propagation

## 17.2 Algorithm

1. Start
2. Construct a control flow graph (CFG).
3. Associate transfer functions with the edges of the CFG.
4. At every node (program point) we maintain the values of the program's variables at that point. We initialize those to $\perp$.
5. Iterate until the values of the variables stabilize.
6. Stop

## 17.3 Program

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

void input();
void output();
void change(int p,char *res);
void constant();

struct exp{
  char op[2], op1[5], op2[5], res[5];
  int flag;
} arr[10];

int n;

void main(){
  input();
  constant();
  output();
}

void input(){
  int i;
  printf("Enter the maximum number of expressions: \n");
  scanf("%d",&n);
  printf("Enter the input: \n");
  for ( i = 0; i < n; ++i)
  {
    scanf("%s",arr[i].op);
    scanf("%s",arr[i].op1);
    scanf("%s",arr[i].op2);
    scanf("%s",arr[i].res);
    arr[i].flag = 0;
  }
}

void constant(){
  int i;
  int op1,op2,res;
  char op,res1[5];
  for ( i = 0; i < n; ++i)
  {
    if (isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"=") == 0  )
    {
      // if both digits store in variables
      op1 = atoi(arr[i].op1);
      op2 = atoi(arr[i].op2);
      op = arr[i].op[0];
      switch(op){
        case '+':
          res = op1+op2;
```

```
          break;
        case '-':
          res = op1-op2;
          break;
        case '*':
          res = op1*op2;
          break;
        case '/':
          res = op1/op2;
          break;
        case '=':
          res = op1;
          break;
      }
      sprintf(res1,"%d", res);
      arr[i].flag = 1;
      change(i,res1);
    }
  }
}

void output(){
  int i=0;
  printf("Optimized code\n");
  for (i = 0; i < n; ++i)
  {
    if(!arr[i].flag){
      printf("%s %s %s %s\n", arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);

    }
  }
}

void change(int p, char *res){
  int i;
  for ( i = p+1; i < n; ++i)
  {
    if(strcmp(arr[p].res, arr[i].op1) == 0){
      strcpy(arr[i].op1,res);
    }
    else if(strcmp(arr[p].res,arr[i].op2) == 0){
      strcpy(arr[i].op2,res);
    }
  }
}
```

## 17.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp17.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out


Enter the maximum number of expressions: 4

Enter the input:
= 3 - a
+ a b t1
+ a c t2
+ t1 t2 t3

Optimized code is:
+ 3 b t1
+ 3 c t2
+ t1 t2 t3
s21a23@administrator-rusa:~/cd_lab$
```

## 17.5 Result

Successfully implemented a program to perform constant propagation.

# CYCLE 5

# 18 Program for Intermediate Code Generation

## 18.1 Aim

Implement Intermediate code generation for simple expressions

## 18.2 Algorithm

```
1. Start
2. Open the input file in read mode.
3. Open the output file in write mode.
4. In input file scan for operator, argument1, argument2 and result.
5. If the operator is +
   Move arg1 to R0
   Add arg2 and R0
   Move R0 to result
6. If the operator is -
   Move arg1 to R0
   Subtract arg2 and R0
   Move R0 to result
7. If the operator is *
   Move arg1 to R0
   Multiply arg2 and R0
   Move R0 to result
8. If the operator is /
   Move arg1 to R0
   Divide arg2 and R0
   Move R0 to result
9. If the operator is =
   Move arg1 to R0
   Move R0 to result
10.Close both the files.
11.Stop
```

## 18.3 Program

```

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

void code_op(char *inp, char op, char *reg){
  int i=0,j=0;
  char temp[100];
  while(inp[i] != '\0'){
    if(inp[i] == op){
      printf("%c\t%c\t%c\t%c\n\n", op, *reg, inp[i-1],inp[i+1]);
      temp[j-1] = *reg;
      i+=2;
      (*reg)--;
      continue;
    }
    temp[j] = inp[i];
    i++;
    j++;
  }
  temp[++j] = '\0';
  strcpy(inp,temp);
}

void gen_code(char *inp){
  char reg = 'Z';
  code_op(inp,'/',&reg);

  code_op(inp,'*',&reg);

  code_op(inp,'+',&reg);

  code_op(inp,'-',&reg);

  code_op(inp,'=',&reg);
}
```

```
36
37 void main(){
38   char inp[100];
39   printf("Enter expression: \n");
40   scanf("%s",inp);
41   printf("Oper\tDestn\tOp1\tOp2\n");
42   gen_code(inp);
43 }
```

## 18.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp18.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out
Enter expression: a+b-c/d*6
Oprtr   Destn   Op1     Op2
/       Z       c       d
*       Y       Z       6
+       X       a       b
-       W       X       Y
s21a23@administrator-rusa:~/cd_lab$
```

## 18.5 Result

Successfully implemented Intermediate code generation for simple expressions in c.

# CYCLE 6

# 19 Implementation of Back-end Compiler

## 19.1 Aim

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.

## 19.2 Algorithm

```
1. Start
2. Open the source file and store the contents as quadruples.
3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if
    assignment operator generates it, else perform unary minus on register C.
4. Write the generated code to output definition of the file.
5. Print the output.
6. Stop
```

## 19.3 Program

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

void main(){
  char icode[10][30], str[20], opr[10];
  int i=0;
  printf("Enter the set of intermediate code (terminated by exit) \n");
  do{
    scanf("%s", icode[i]);

  }while(strcmp(icode[i++] , "exit") != 0);

  printf("Target code \n");
  i=0;
  do{
    strcpy(str, icode[i]);
    switch(str[3]){
      case '+':
        strcpy(opr,"ADD");
        break;
      case '-':
        strcpy(opr,"SUB");
        break;
      case '*':
        strcpy(opr,"MUL");
        break;
      case '/':
        strcpy(opr,"DIV");
        break;
    }
    printf("MOV %c,R%d\n",str[2],i );

    printf("%s %c,R%d\n",opr,str[4],i );

    printf("MOV R%d,%c\n",i,str[0] );

  }while(strcmp(icode[++i] , "exit") != 0);
  printf("\n");
}
```

## 19.4 Output

```
s21a23@administrator-rusa:~/cd_lab$ gcc exp19.c
s21a23@administrator-rusa:~/cd_lab$ ./a.out

Enter the set of intermediate code (terminated by exit):
a=a+3
b=a*5
exit

Target code generation
******************
Mov a, R0
ADD  3, R0
Mov R0, a
Mov a, R1
MUL  5, R1
Mov R1, b
s21a23@administrator-rusa:~/cd_lab$
```

## 19.5 Result

Successfully implemented 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.