

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

1	Design and Implement Lexical Analyser Using C	2
1.1	Aim	2
1.2	Algorithm	2
1.3	Program	2
1.4	Output	4
1.5	Result	4
2	Implement Lexical Analyser Using LEX	5
2.1	Aim	5
2.2	Algorithm	5
2.3	Program	5
2.4	Output	6
2.5	Result	7
3	LEX Program to Display Number of Lines and Words	8
3.1	Aim	8
3.2	Algorithm	8
3.3	Program	8
3.4	Output	8
3.5	Result	9
4	LEX Program to Convert the Substring abc to ABC	10
4.1	Aim	10
4.2	Algorithm	10
4.3	Program	10
4.4	Output	10
4.5	Result	10
5	LEX Program to find the Number of Vowels and Consonants	11
5.1	Aim	11
5.2	Algorithm	11
5.3	Program	11
5.4	Output	11
5.5	Result	11
6	Generate YACC Specification to Recognize a Valid Arithmetic Expression	12
6.1	Aim	12
6.2	Algorithm	12
6.3	Program	12
6.4	Output	13
6.5	Result	13
7	Generate YACC Specification to Recognize a Valid Identifier	14
7.1	Aim	14
7.2	Algorithm	14
7.3	Program	14
7.4	Output	15
7.5	Result	15
8	Calculator using LEX and YACC	16
8.1	Aim	16
8.2	Algorithm	16
8.3	Program	16
8.4	Output	17
8.5	Result	17

9	Convert BNF Rules into YACC and Write Abstract Syntax Tree	18
9.1	Aim	18
9.2	Algorithm	18
9.3	Program	18
9.4	Output	21
9.5	Result	22
10	Program to find ϵ – closure of All States of NFA	24
10.1	Aim	24
10.2	Algorithm	24
10.3	Program	24
10.4	Output	25
10.5	Result	25
11	Program to Convert NFA With ϵ Transition to NFA Without ϵ Transition	26
11.1	Aim	26
11.2	Algorithm	26
11.3	Program	26
11.4	Output	30
11.5	Result	30
12	Program to Convert NFA to DFA	31
12.1	Aim	31
12.2	Algorithm	31
12.3	Program	31
12.4	Output	35
12.5	Result	35
13	Program to Minimise Any Given DFA	36
13.1	Aim	36
13.2	Algorithm	36
13.3	Program	36
13.4	Output	40
13.5	Result	40
14	Program to Find First and Follow of Any Grammar	42
14.1	Aim	42
14.2	Algorithm	42
14.3	Program	42
14.4	Output	44
14.5	Result	44
15	Recursive Descent Parser	45
15.1	Aim	45
15.2	Algorithm	45
15.3	Program	45
15.4	Output	47
15.5	Result	47
16	Construct Shift Reduce Parser	48
16.1	Aim	48
16.2	Algorithm	48
16.3	Program	48
16.4	Output	50
16.5	Result	50

17 Constant Propagation	52
17.1 Aim	52
17.2 Algorithm	52
17.3 Program	52
17.4 Output	54
17.5 Result	54
18 Program for Intermediate Code Generation	56
18.1 Aim	56
18.2 Algorithm	56
18.3 Program	56
18.4 Output	57
18.5 Result	57
19 Implementation of Back-end Compiler	59
19.1 Aim	59
19.2 Algorithm	59
19.3 Program	59
19.4 Output	60
19.5 Result	60

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

```
1
2 #include <stdbool.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include <stdlib.h>
6
7 bool isDelimiter(char ch) {
8     if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
9         ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
10        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
11        ch == '[' || ch == ']' || ch == '{' || ch == '}')
12        return true;
13    return false;
14 }
15
16 bool isOperator(char ch) {
17     if (ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
18         ch == '>' || ch == '<' || ch == '=')
19        return true;
20    return false;
21 }
22
23 bool validIdentifier(char* str) {
24     if (str[0] >= '0' && str[0] <= '9' || isDelimiter(str[0]))
25        return false;
26    return true;
27 }
28
29 bool isKeyword(char* str) {
30     const char* keywords[] = {"if", "else", "while", "do", "break", "continue", "int",
31                                "double", "float", "return", "char", "case", "sizeof",
32                                "long", "short", "typedef", "switch", "unsigned",
33                                "void", "static", "struct", "goto"};
34     for (int i = 0; i < sizeof(keywords)/sizeof(keywords[0]); i++) {
35         if (!strcmp(str, keywords[i]))
```

```

36         return true;
37     }
38     return false;
39 }
40
41 bool isInteger(char* str) {
42     int i, len = strlen(str);
43     if (len == 0)
44         return false;
45     for (i = 0; i < len; i++) {
46         if (str[i] < '0' || str[i] > '9' || (str[i] == '-' && i > 0))
47             return false;
48     }
49     return true;
50 }
51
52 bool isRealNumber(char* str) {
53     int i, len = strlen(str);
54     bool hasDecimal = false;
55     if (len == 0)
56         return false;
57     for (i = 0; i < len; i++) {
58         if ((str[i] < '0' || str[i] > '9') && str[i] != '.' ||
59             (str[i] == '-' && i > 0))
60             return false;
61         if (str[i] == '.')
62             hasDecimal = true;
63     }
64     return hasDecimal;
65 }
66
67 char* subString(char* str, int left, int right) {
68     int i;
69     char* subStr = (char*)malloc(sizeof(char) * (right - left + 2));
70     for (i = left; i <= right; i++)
71         subStr[i - left] = str[i];
72     subStr[right - left + 1] = '\0';
73     return subStr;
74 }
75
76 void parse(char* str) {
77     int left = 0, right = 0;
78     int len = strlen(str);
79
80     while (right <= len && left <= right) {
81         if (!isDelimiter(str[right]))
82             right++;
83
84         if (isDelimiter(str[right]) && left == right) {
85             if (isOperator(str[right]))
86                 printf("%c' IS AN OPERATOR\n", str[right]);
87
88             right++;
89             left = right;
90         } else if ((isDelimiter(str[right]) && left != right) || (right == len && left !=
91             right)) {
92             char* subStr = subString(str, left, right - 1);
93
94             if (isKeyword(subStr))
95                 printf("%s' IS A KEYWORD\n", subStr);
96             else if (isInteger(subStr))
97                 printf("%s' IS AN INTEGER\n", subStr);
98             else if (isRealNumber(subStr))
99                 printf("%s' IS A REAL NUMBER\n", subStr);
100             else if (validIdentifier(subStr) && !isDelimiter(str[right - 1]))
101                 printf("%s' IS A VALID IDENTIFIER\n", subStr);
102             else if (!validIdentifier(subStr) && !isDelimiter(str[right - 1]))
103                 printf("%s' IS NOT A VALID IDENTIFIER\n", subStr);
104
105             left = right;
106             free(subStr); // Free the allocated memory
107         }
108     }
109 }

```

```

110 int main() {
111     // char str[100] = "int g = 9.8*f;";
112     char str[100];
113     printf("Enter input in a line: ");
114     scanf("%[^\n]s", str);
115     parse(str);
116     return 0;
117 }

```

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 ");}
```

```

29 {identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
30 \".*\\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
31 [0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
32 \(\(:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
33 \(\ ECHO;
34 \+ |
35 \- |
36 \| |
37 \* {if(!COMMENT)printf("\n\t%s is an ARITHMETIC OPERATOR", yytext);}
38 = {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
39 \<= |
40 \>= |
41 \< |
42 == |
43 \> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
44 %%
45 int main(int argc, char **argv)
46 {
47 FILE *file;
48 file=fopen("var.c","r");
49 if(!file)
50 {
51 printf("could not open the file");
52 exit(0);
53 }
54 yyin=file;
55 yylex();
56 printf("\n");
57 return(0);
58 }
59 int yywrap()
60 {
61 return(1);
62 }

```

2.4 Output

Input

```

1  #include<stdio.h>
2  void main(){
3      int a=0;
4      printf("a=%d",a);
5  }
6

```

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

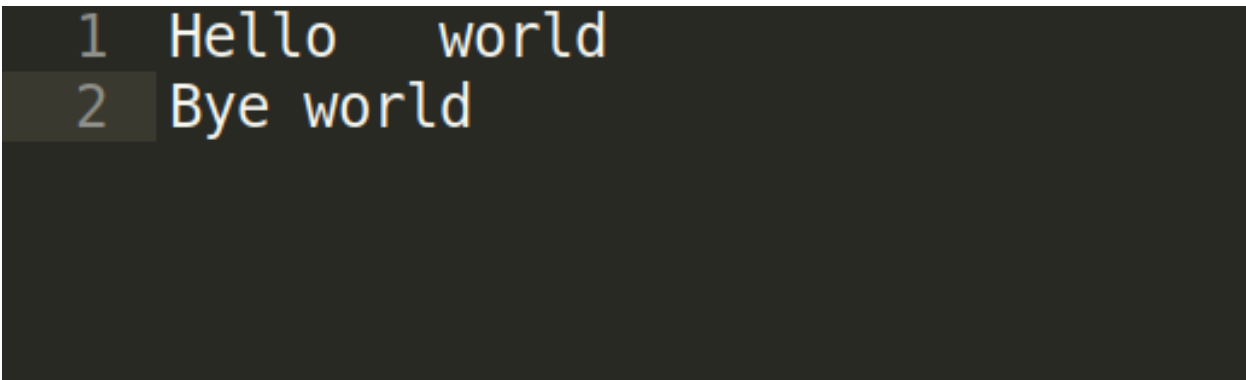
1. Read input.
2. Initialize chars = 0, words = 0 and lines = 0.
3. If token equals [\t\n]+ , increment words and chars = chars + yylen.
4. Else if token = [\n] increment lines, chars.
5. Else if token = []* increment chars.
6. Print number of words, characters and lines.
7. EXIT

3.3 Program

```
1 %{
2 int lines_count=1;
3 int words_count =0;
4 int chars_count =0;
5 %}
6 %%
7 [\n] { lines_count++; chars_count += yylen; }
8 [ \t] { chars_count += yylen; }
9 [^\t\n]+ { words_count++; chars_count += yylen ; }
10 %%
11 int yywrap(){
12     return 1;
13 }
14 int main()
15 {
16     FILE *file;
17     file = fopen("input.txt","r");
18     if(!file){
19         printf("File error\n");
20         return 1;
21     }
22     yyin = file;
23     yylex();
24     printf("\n");
25     printf("Lines : %d\n", lines_count);
26     printf("Words : %d\n", words_count);
27     printf("Characters : %d\n", chars_count);
28     return 0;
29 }
30 }
```

3.4 Output

input.txt



```
1 Hello world
2 Bye world
```

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

```
1 %{
2   #include <stdio.h>
3   %}
4   %%
5   "\n" {printf("\n"); return 0;}
6   "abc" printf("ABC");
7   . printf("%s",yytext);
8   %%
9   int yywrap(){return 1;}
10  int main(){
11  printf("Enter string: ");
12  yylex();
13  return 0;
14 }
```

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

```
1 %{
2 int vow_count=0;
3 int const_count =0;
4 %}
5 %%
6 [aeiouAEIOU] {vow_count++;}
7 [a-zA-Z] {const_count++;}
8 "\n" {return 0;}
9 %%
10 int yywrap(){ }
11 int main()
12 {
13     printf("Enter the string : ");
14     yylex();
15     printf("Vowel count : %d\n", vow_count);
16     printf("Consonant count: %d\n", const_count);
17     return 0;
18 }
19 }
```

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

```
1 %{
2     #include "y.tab.h"
3 }%
4 %%
5 [0-9]+ { return NUMBER ;}
6 [a-zA-Z0-9]+ { return ID ;}
7 \n { return NL ;}
8 . { return yytext[0];}
9 %%
10 int yywrap() {
11     return 1;
12 }
```

yacc code

```
1 %{
2     #include <stdio.h>
3     #include <stdlib.h>
4 }%
5 %token NUMBER ID NL
6 %left '+' '-'
7 %left '*' '/'
8 %%
9 valid : E NL { printf("Expression is valid !!\n"); exit(0);}
10 E : E '+' E
11   | E '-' E
12   | E '*' E
13   | E '/' E
14   | NUMBER
15   | ID
16   | '(' E ')';
17 %%
```

```

18 int main() {
19     printf ("Enter the expression : ") ;
20     yyparse() ;
21 }
22
23 int yyerror () {
24     printf ("Invalid expression\n") ;
25     exit(1) ;
26 }

```

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

```
1
2 %{
3 # include "y.tab.h"
4 %}
5
6 %%
7 [a-zA-Z] { return LETTER ;}
8 [0-9] { return DIGIT ;}
9 [\n] { return NL ;}
10 [_] { return UND ;}
11 . { return yytext [0];}
12 %%
```

yacc code

```
1 %token DIGIT LETTER NL UND
2
3 %%
4 stmt : variable NL {
5 printf ("\nValid id\n\n"); exit (0);}
6 ;
7 variable : LETTER alphanumeric
8 ;
9 alphanumeric : LETTER alphanumeric
10 | DIGIT alphanumeric
11 | UND alphanumeric
12 | LETTER
13 | DIGIT
14 | UND
15 ;
16 %%
17 int yyerror( char * msg )
18 {
19 printf ("\nINVALID \n") ;
20 exit(0) ;
21 }
22 void main ()
23 {
24 printf (" Enter id : ") ;
25 yyparse () ;
26 }
```

7.4 Output

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

Valid id

s21a23@administrator-rusa:~/cd_lab$ ./a.out
Enter id: @123

INVALID
```

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:

$E \rightarrow T$

$T \rightarrow T + T$

| $T - T$ | $T * T$ | T / T | $-ID$ | $-NUMBER$

| $NUMBER$ | ID | (T)

3. At each stage evaluate the expression value

4. Print the result

8.3 Program

lex code

```
1 %{
2     /* Definition section*/
3     #include "y.tab.h"
4     extern yylval;
5 }
6
7 %%
8 [0-9]+ {
9     yylval = atoi(yytext);
10    return NUMBER;
11 }
12
13 [a-zA-Z]+ { return ID; }
14 [ \t]+      ; /*For skipping whitespaces*/
15
16 \n         { return 0; }
17 .          { return yytext[0]; }
18
19 %%
```

yacc code

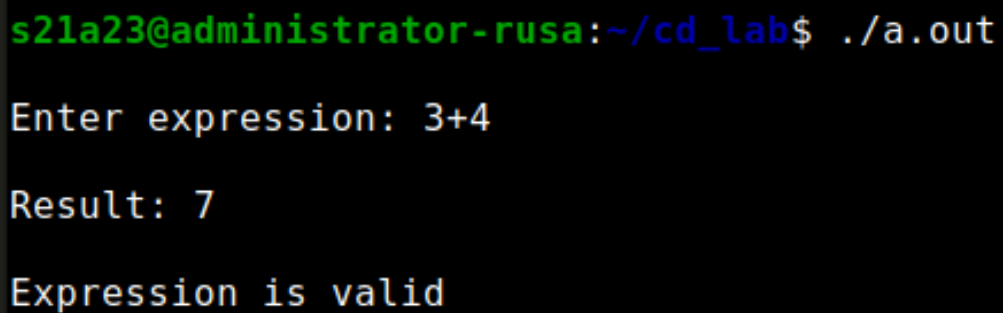
```
1 %{
2     /* Definition section */
3     #include <stdio.h>
4 }
5
6 %token NUMBER ID
7 // setting the precedence
8 // and associativity of operators
9 %left '+' '-'
10 %left '*' '/' '%'
11 %left '(' ')'
12
13 /* Rule Section */
14 %%
15 E : T
16 {
17     printf("Result = %d\n", $$);
18     return 0;
19 }
20
21 T :
22   '+' T { $$ = $1 + $3; }
23   '-' T { $$ = $1 - $3; }
24   '*' T { $$ = $1 * $3; }
25   '/' T { $$ = $1 / $3; }
26   '%' T { $$ = $1 % $3; }
27   '-' NUMBER { $$ = -$2; }
28   '-' ID { $$ = -$2; }
```

```

29 | '(' T ')' { $$ = $2; }
30 | NUMBER { $$ = $1; }
31 | ID { $$ = $1; };
32 %%
33
34 int main() {
35     printf("Enter the expression\n");
36     yyparse();
37 }
38
39 /* For printing error messages */
40 int yyerror(char* s) {
41     printf("\nExpression is invalid\n");
42 }

```

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

```
1 %{
2 #include "y.tab.h"
3 #include <stdio.h>
4 #include <string.h>
5 int LineNo = 1;
6 %}
7
8 identifier [a-zA-Z][a-zA-Z0-9_]*
9 number [0-9]+|([0-9]*\.[0-9]+)
10
11 %%
12 main\(\) return MAIN;
13 if return IF;
14 else return ELSE;
15 while return WHILE;
16 int |
17 char |
18 float return TYPE;
19 {identifier} { strcpy(yylval.var, yytext); return VAR; }
20 {number} { strcpy(yylval.var, yytext); return NUM; }
21 \< |
22 \> |
23 \>= |
24 \<= |
25 == { strcpy(yylval.var, yytext); return RELOP; }
26 [ \t] ;
27 \n LineNo++;
28 . return yytext[0];
29 %%
```

yacc code

```
1 %{
2 #include <string.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5
6 struct quad {
7     char op[5];
8     char arg1[10];
9     char arg2[10];
10    char result[10];
11 } QUAD[30];
12
13 struct stack {
14     int items[100];
15     int top;
16 } stk;
17
18 int Index = 0, tIndex = 0, StNo, Ind, tInd;
19 extern int LineNo;
20
21 void AddQuadruple(char op[5], char arg1[10], char arg2[10], char result[10]);
```

```

22 int pop();
23 void push(int data);
24 int yyerror();
25 int yylex();
26 %}
27
28 %union { char var[10]; }
29 %token <var> NUM VAR RELOP
30 %token MAIN IF ELSE WHILE TYPE
31 %type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
32 %left '-', '+'
33 %left '*', '/'
34
35 %%
36 PROGRAM : MAIN BLOCK;
37 BLOCK : '{' CODE '}';
38 CODE : BLOCK
39       | STATEMENT CODE
40       | STATEMENT;
41 STATEMENT : DESCT ';'
42           | ASSIGNMENT ';'
43           | CONDST
44           | WHILEST;
45 DESCT : TYPE VARLIST;
46 VARLIST : VAR ',' VARLIST
47          | VAR;
48 ASSIGNMENT : VAR '=' EXPR {
49     strcpy(QUAD[Index].op, "=");
50     strcpy(QUAD[Index].arg1, $3);
51     strcpy(QUAD[Index].arg2, "");
52     strcpy(QUAD[Index].result, $1);
53     strcpy($$, QUAD[Index++].result);
54 };
55 EXPR : EXPR '+' EXPR { AddQuadruple("+", $1, $3, $$); }
56      | EXPR '-' EXPR { AddQuadruple("-", $1, $3, $$); }
57      | EXPR '*' EXPR { AddQuadruple("*", $1, $3, $$); }
58      | EXPR '/' EXPR { AddQuadruple("/", $1, $3, $$); }
59      | '-' EXPR { AddQuadruple("UMIN", $2, "", $$); }
60      | '(' EXPR ')' { strcpy($$, $2); }
61      | VAR
62      | NUM;
63 CONDST : IFST {
64     Ind = pop();
65     sprintf(QUAD[Ind].result, "%d", Index);
66     Ind = pop();
67     sprintf(QUAD[Ind].result, "%d", Index);
68 }
69 | IFST ELSEST;
70 IFST : IF '(' CONDITION ')' {
71     strcpy(QUAD[Index].op, "==");
72     strcpy(QUAD[Index].arg1, $3);
73     strcpy(QUAD[Index].arg2, "FALSE");
74     strcpy(QUAD[Index].result, "-1");
75     push(Index);
76     Index++;
77 }
78 BLOCK {
79     strcpy(QUAD[Index].op, "GOTO");
80     strcpy(QUAD[Index].arg1, "");
81     strcpy(QUAD[Index].arg2, "");
82     strcpy(QUAD[Index].result, "-1");
83     push(Index);
84     Index++;
85 };
86 ELSEST : ELSE {
87     tInd = pop();
88     Ind = pop();
89     push(tInd);
90     sprintf(QUAD[Ind].result, "%d", Index);
91 }
92 BLOCK {
93     Ind = pop();
94     sprintf(QUAD[Ind].result, "%d", Index);
95 };
96 CONDITION : VAR RELOP VAR { AddQuadruple($2, $1, $3, $$); StNo = Index - 1; }

```

```

97         | VAR
98         | NUM;
99 WHILEST : WHILELOOP {
100     Ind = pop();
101     sprintf(QUAD[Index].result, "%d", StNo);
102     Ind = pop();
103     sprintf(QUAD[Index].result, "%d", Index);
104 };
105 WHILELOOP : WHILE '(' CONDITION ')' {
106     strcpy(QUAD[Index].op, "=");
107     strcpy(QUAD[Index].arg1, $3);
108     strcpy(QUAD[Index].arg2, "FALSE");
109     strcpy(QUAD[Index].result, "-1");
110     push(Index);
111     Index++;
112 }
113 BLOCK {
114     strcpy(QUAD[Index].op, "GOTO");
115     strcpy(QUAD[Index].arg1, "");
116     strcpy(QUAD[Index].arg2, "");
117     strcpy(QUAD[Index].result, "-1");
118     push(Index);
119     Index++;
120 };
121 %%
122
123 extern FILE *yyin;
124 int main(int argc, char *argv[]) {
125     FILE *fp;
126     int i;
127     if (argc > 1) {
128         fp = fopen(argv[1], "r");
129         if (!fp) {
130             printf("\n File not found");
131             exit(0);
132         }
133         yyin = fp;
134     }
135     yyparse();
136     printf("\n\n\t\t-----\n\t\tPos\tOperator\tArg1\tArg2\t\n\t\tResult\n\t\t-----");
137     for (i = 0; i < Index; i++) {
138         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);
139     }
140     printf("\n\t\t\t-----");
141     printf("\n\n");
142     return 0;
143 }
144
145 void push(int data) {
146     stk.top++;
147     if (stk.top == 100) {
148         printf("\n Stack overflow\n");
149         exit(0);
150     }
151     stk.items[stk.top] = data;
152 }
153
154 int pop() {
155     int data;
156     if (stk.top == -1) {
157         printf("\n Stack underflow\n");
158         exit(0);
159     }
160     data = stk.items[stk.top--];
161     return data;
162 }
163
164 void AddQuadruple(char op[5], char arg1[10], char arg2[10], char result[10]) {
165     strcpy(QUAD[Index].op, op);
166     strcpy(QUAD[Index].arg1, arg1);
167     strcpy(QUAD[Index].arg2, arg2);
168     sprintf(QUAD[Index].result, "t%d", tIndex++);
169     strcpy(result, QUAD[Index++].result);

```

```
170 }  
171  
172 int yyerror() {  
173     printf("\n Error on line no: %d", LineNo);  
174 }
```

9.4 Output

Input

```
1 #include<stdio.h>  
2 void main(){  
3     int a=0;  
4     printf("a=%d",a);  
5 }  
6
```

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 ϵ – closure of All States of NFA

10.1 Aim

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

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int n; //number of states
5
6 void closure(int state, int matrix[][n]){
7     for(int i=0;i<n;i++){
8         if(matrix[state][i] == 1){
9             // calculated in main if the i is reachable from the state
10            printf("    , q%d",i );
11            closure(i,matrix);
12            // if the ith state reachable then null or epsilon transitions from i also appended
13        }
14    }
15 }
16 return;
17 }
18
19 int main() {
20     FILE* INPUT = fopen("input.txt","r");
21     char state1[10],input[10],state2[10];
22     int s1,s2;
23
24     printf("Enter the number of states : ");
25     scanf("%d",&n);
26     int matrix[n][n];
27     for (int i = 0; i < n; ++i)
28     {
29         for (int j = 0; j < n; ++j)
30         {
31             /* code */
32             matrix[i][j]=0;
33         }
34     }
35     while(fscanf(INPUT,"%s%s%s", state1,input,state2) != EOF){
36         if(strcmp(input,"e")==0){
37             s1 = state1[1]-'0';
38             s2 = state2[1]-'0';
39             matrix[s1][s2] = 1;
40         }
41     }
42     printf("epsilon closure\n");
43     for (int i = 0; i < n; ++i)
44     {
45         printf("q%d: q%d", i,i);
46         closure(i,matrix);
47         printf("\n");
48     }
49     return 0;
50 }
```

10.4 Output

Input

1	q0	0	q0
2	q0	1	q1
3	q0	e	q1
4	q1	1	q2
5	q1	e	q2

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 ϵ Transition to NFA Without ϵ 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

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 // represents transitions in the NFA
6 struct node {
7     int st;
8     struct node* link;
9 };
10
11 // hold a set of each DFA states
12 struct node1{
13     int nst[20];
14 };
15
16 void insert(int,char,int);
17 int findalpha(char);
18 void findfinalstate(void);
19 int insertdfastate(struct node1);
20 int compare(struct node1, struct node1);
21 void printnewstate(struct node1);
22 static int set[20],nostate,noalpha,notransition,nofinal,start,finalstate[20],r,s;
23 int complete = -1;
24 char c,alphabet[20];
25 // array to store DFA states
26 struct node1 hash[20];
27 // transition table
28 struct node* transition[20][20] = {NULL};
29 void main(){
30     printf("Enter the number of alphabets: ");
31     scanf("%d",&noalpha);
32     printf("Enter each alphabet\n");
33     getchar();
34     for (int i = 0; i < noalpha; ++i)
35     {
36         alphabet[i] = getchar();
37         getchar();
38     }
39
40     printf("Enter the number of states: ");
41     scanf("%d",&nostate);
42
43
44     printf("Enter the start state: ");
45     scanf("%d",&start);
46
47
```

```

48 printf("Enter the number of final states: ");
49 scanf("%d",&nofinal);
50 printf("Enter the final states: \n");
51 for (int i = 0; i < nofinal; ++i)
52 {
53     scanf("%d",&finalstate[i]);
54 }
55
56 printf("Enter the number of transitions: ");
57 scanf("%d",&notransition);
58
59 for (int i = 0; i < notransition; ++i)
60 {
61     scanf("%d %c %d",&r,&c,&s);
62     insert(r,c,s);
63 }
64
65 // preparing to store
66 for (int i = 0; i < 20; ++i)
67 {
68     for (int j = 0; j < 20; ++j)
69     {
70         hash[i].nst[j] = 0;
71     }
72 }
73
74
75 complete=-1; // track last state
76 // indicate number of DFA identified and stored
77 int i=-1;
78 // indicating if all states explored
79 printf("Equivalent DFA ....\n");
80 printf("Transitions of DFA \n");
81
82 struct node1 newstate={0};
83 struct node1 tmpstate={0};
84 struct node* temp;
85 int c,l;
86
87 newstate.nst[start] = start;
88 insertdfastate(newstate);
89 while(i != complete){
90     i++;
91     newstate=hash[i];
92     for (int k = 0; k < noalpha; ++k)
93     {
94         c=0;
95         for (int j = 1; j <= nostate; ++j)
96         {
97             set[j]=0;
98         }
99         for (int j = 1; j <= nostate; ++j)
100         {
101             l = newstate.nst[j];
102             if(l != 0){
103                 temp = transition[l][k];
104                 while(temp != NULL){
105                     if (set[temp->st] == 0){
106                         c++;
107                         set[temp->st] = temp->st;
108                     }
109                     temp = temp->link;
110                 }
111             }
112         }
113         printf("\n");
114         if(c != 0){
115             for (int m = 1; m <= nostate; ++m)
116             {
117                 tmpstate.nst[m] = set[m];
118             }
119             insertdfastate(tmpstate);
120             printnewstate(newstate);
121
122             printf("%c\t",alphabet[k] );

```

```

123     printnewstate(tmpstate);
124     printf("\n");
125 }
126 else{
127     printnewstate(newstate);
128     printf("%c\t",alphabet[k]);
129     printf("NULL\n");
130 }
131 }
132 }
133 printf("\n");
134 printf("States of DFA\n");
135 for (int i = 0; i <= complete; ++i)
136 {
137     printnewstate(hash[i]);
138 }
139 printf("alphabets: ");
140 for (int j = 0; j < noalpha; ++j)
141 {
142     printf("%c \t", alphabet[j]);
143 }
144 printf("\n");
145 printf("Start state: q%d\n",start );
146 printf("Final states: ");
147 findfinalstate();
148 printf("\n");
149 }
150 }
151 // adds a new DFA state into hash if not already present
152 int insertdfastate(struct node1 newstate){
153     for (int i = 0; i <= complete; ++i)
154     {
155         if(compare(hash[i],newstate)){
156             return 0;
157         }
158     }
159     complete++;
160     hash[complete] = newstate;
161     return 1;
162 }
163 }
164 // two DFA states compared
165 int compare(struct node1 a, struct node1 b){
166     for (int i = 1; i <= nostate; ++i)
167     {
168         if(a.nst[i] != b.nst[i]){
169             return 0;
170         }
171     }
172     return 1;
173 }
174 // adds transition to transition table
175 void insert(int r,char c, int s){
176     struct node* temp;
177     int j = findalpha(c);
178     if(j==999){
179         printf("Error\n");
180         exit(0);
181     }
182     temp = (struct node*)malloc(sizeof(struct node));
183     temp->st = s;
184     temp->link = transition[r][j];
185     transition[r][j] = temp;
186 }
187 // finds index of given alphabet
188 int findalpha(char c){
189     for (int i = 0; i < noalpha; ++i)
190     {
191         if(alphabet[i] == c){
192             return i;
193         }
194     }
195     return 999;
196 }
197

```

```

198 }
199
200 // identifies and prints final states
201 void findfinalstate(){
202     for (int i = 0; i <= complete; ++i)
203     {
204         for (int j = 1; j <= nostate; ++j)
205         {
206             for (int k = 0; k < nofinal; ++k)
207             {
208                 if(hash[i].nst[j] == finalstate[k]){
209                     printnewstate(hash[i]);
210                     printf("\t");
211                     j = nostate;
212                     break;
213                 }
214             }
215         }
216     }
217     printf("\n");
218 }
219
220 // print in readable format
221 void printnewstate(struct node1 state){
222     printf("{");
223     for (int i = 1; i <= nostate; ++i)
224     {
225         if(state.nst[i] != 0){
226             printf("q%d, ",state.nst[i] );
227         }
228     }
229     printf("}\t");
230 }

```

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 $Q' = \emptyset$
4. Add q_0 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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // represents transitions in the NFA
5  struct node {
6      int st;
7      struct node* link;
8  };
9
10 // hold a set of each DFA states
11 struct node1{
12     int nst[20];
13 };
14
15 void insert(int,char,int);
16 int findalpha(char);
17 void findfinalstate(void);
18 int insertdfastate(struct node1);
19 int compare(struct node1, struct node1);
20 void printnewstate(struct node1);
21 static int set[20],nostate,noalpha,notransition,nofinal,start,finalstate[20],r,s;
22 int complete = -1;
23 char c,alphabet[20];
24 // array to store DFA states
25 struct node1 hash[20];
26 // transition table
27 struct node* transition[20][20] = {NULL};
28 void main(){
29     printf("Enter the number of alphabets: ");
30     scanf("%d",&noalpha);
31     printf("Enter each alphabet\n");
32     getchar();
33     for (int i = 0; i < noalpha; ++i)
34     {
35         alphabet[i] = getchar();
36         getchar();
37     }
38
39     printf("Enter the number of states: ");
40     scanf("%d",&nostate);
41
42
43     printf("Enter the start state: ");
44     scanf("%d",&start);
45
46
47     printf("Enter the number of final states: ");
48     scanf("%d",&nofinal);
49     printf("Enter the final states: \n");
```

```

51  for (int i = 0; i < nofinal; ++i)
52  {
53      scanf("%d",&finalstate[i]);
54  }
55
56  printf("Enter the number of transitions: ");
57  scanf("%d",&notransition);
58
59  for (int i = 0; i < notransition; ++i)
60  {
61      scanf("%d %c %d",&r,&c,&s);
62      insert(r,c,s);
63  }
64
65  // preparing to store
66  for (int i = 0; i < 20; ++i)
67  {
68      for (int j = 0; j < 20; ++j)
69      {
70          hash[i].nst[j] = 0;
71      }
72  }
73
74
75  complete=-1; // track last state
76  // indicate number of DFA identified and stored
77  int i=-1;
78  // indicating if all states explored
79  printf("Equivalent DFA ....\n");
80  printf("Transitions of DFA \n");
81
82  struct node1 newstate={0};
83  struct node1 tmpstate={0};
84  struct node* temp;
85  int c,l;
86
87  newstate.nst[start] = start;
88  insertdfastate(newstate);
89  while(i != complete){
90      i++;
91      newstate=hash[i];
92      for (int k = 0; k < noalpha; ++k)
93      {
94          c=0;
95          for (int j = 1; j <= nostate; ++j)
96          {
97              set[j]=0;
98          }
99          for (int j = 1; j <= nostate; ++j)
100          {
101              l = newstate.nst[j];
102              if(l != 0){
103                  temp = transition[l][k];
104                  while(temp != NULL){
105                      if (set[temp->st] == 0){
106                          c++;
107                          set[temp->st] = temp->st;
108                      }
109                      temp = temp->link;
110                  }
111              }
112          }
113          printf("\n");
114          if(c != 0){
115              for (int m = 1; m <= nostate; ++m)
116              {
117                  tmpstate.nst[m] = set[m];
118              }
119              insertdfastate(tmpstate);
120              printnewstate(newstate);
121
122              printf("%c\t",alphabet[k] );
123              printnewstate(tmpstate);
124              printf("\n");
125          }

```

```

126         else{
127             printnewstate(newstate);
128             printf("%c\t",alphabet[k]);
129             printf("NULL\n");
130         }
131     }
132 }
133 printf("\n");
134 printf("States of DFA\n");
135 for (int i = 0; i <= complete; ++i)
136 {
137     printnewstate(hash[i]);
138 }
139 }
140 printf("alphabets: ");
141 for (int j = 0; j < noalpha; ++j)
142 {
143     printf("%c \t", alphabet[j]);
144 }
145 printf("\n");
146 printf("Start state: q%d\n",start );
147 printf("Final states: ");
148 findfinalstate();
149 printf("\n");
150 }
151 // adds a new DFA state into hash if not already present
152 int insertdfastate(struct node1 newstate){
153     for (int i = 0; i <= complete; ++i)
154     {
155         if(compare(hash[i],newstate)){
156             return 0;
157         }
158     }
159     complete++;
160     hash[complete] = newstate;
161     return 1;
162 }
163 }
164 // two DFA states compared
165 int compare(struct node1 a, struct node1 b){
166     for (int i = 1; i <= nostate; ++i)
167     {
168         if(a.nst[i] != b.nst[i]){
169             return 0;
170         }
171     }
172     return 1;
173 }
174 // adds transition to transition table
175 void insert(int r,char c, int s){
176     struct node* temp;
177     int j = findalpha(c);
178     if(j==999){
179         printf("Error\n");
180         exit(0);
181     }
182     temp = (struct node*)malloc(sizeof(struct node));
183     temp->st = s;
184     temp->link = transition[r][j];
185     transition[r][j] = temp;
186 }
187 // finds index of given alphabet
188 int findalpha(char c){
189     for (int i = 0; i < noalpha; ++i)
190     {
191         if(alphabet[i] == c){
192             return i;
193         }
194     }
195     return 999;
196 }
197 // identifies and prints final states

```



```

201 void findfinalstate(){
202     for (int i = 0; i <= complete; ++i)
203     {
204         for (int j = 1; j <= nostate; ++j)
205         {
206             for (int k = 0; k < nofinal; ++k)
207             {
208                 if(hash[i].nst[j] == finalstate[k]){
209                     printnewstate(hash[i]);
210                     printf("\t");
211                     j = nostate;
212                     break;
213                 }
214             }
215         }
216     }
217     printf("\n");
218 }
219
220 // print in readable format
221 void printnewstate(struct node1 state){
222     printf("{");
223     for (int i = 1; i <= nostate; ++i)
224     {
225         if(state.nst[i] != 0){
226             printf("q%d, ",state.nst[i] );
227         }
228     }
229     printf("}\t");
230 }

```

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 q_0 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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // represents transitions in the NFA
5  struct node {
6      int st;
7      struct node* link;
8  };
9
10 // hold a set of each DFA states
11 struct node1{
12     int nst[20];
13 };
14
15 void insert(int,char,int);
16 int findalpha(char);
17 void findfinalstate(void);
18 int insertdfastate(struct node1);
19 int compare(struct node1, struct node1);
20 void printnewstate(struct node1);
21 static int set[20],nostate,noalpha,notransition,nofinal,start,finalstate[20],r,s;
22 int complete = -1;
23 char c,alphabet[20];
24 // array to store DFA states
25 struct node1 hash[20];
26 // transition table
27 struct node* transition[20][20] = {NULL};
28 void main(){
29     printf("Enter the number of alphabets: ");
30     scanf("%d",&noalpha);
31     printf("Enter each alphabet\n");
32     getchar();
33     for (int i = 0; i < noalpha; ++i)
34     {
35         alphabet[i] = getchar();
36         getchar();
37     }
38
39     printf("Enter the number of states: ");
40     scanf("%d",&nostate);
41
42
43     printf("Enter the start state: ");
44     scanf("%d",&start);
45
46
47     printf("Enter the number of final states: ");
48     scanf("%d",&nofinal);
49     printf("Enter the final states: \n");
```

```

51  for (int i = 0; i < nofinal; ++i)
52  {
53      scanf("%d",&finalstate[i]);
54  }
55
56  printf("Enter the number of transitions: ");
57  scanf("%d",&notransition);
58
59  for (int i = 0; i < notransition; ++i)
60  {
61      scanf("%d %c %d",&r,&c,&s);
62      insert(r,c,s);
63  }
64
65  // preparing to store
66  for (int i = 0; i < 20; ++i)
67  {
68      for (int j = 0; j < 20; ++j)
69      {
70          hash[i].nst[j] = 0;
71      }
72  }
73
74
75  complete=-1; // track last state
76  // indicate number of DFA identified and stored
77  int i=-1;
78  // indicating if all states explored
79  printf("Equivalent DFA ....\n");
80  printf("Transitions of DFA \n");
81
82  struct node1 newstate={0};
83  struct node1 tmpstate={0};
84  struct node* temp;
85  int c,l;
86
87  newstate.nst[start] = start;
88  insertdfastate(newstate);
89  while(i != complete){
90      i++;
91      newstate=hash[i];
92      for (int k = 0; k < noalpha; ++k)
93      {
94          c=0;
95          for (int j = 1; j <= nostate; ++j)
96          {
97              set[j]=0;
98          }
99          for (int j = 1; j <= nostate; ++j)
100          {
101              l = newstate.nst[j];
102              if(l != 0){
103                  temp = transition[l][k];
104                  while(temp != NULL){
105                      if (set[temp->st] == 0){
106                          c++;
107                          set[temp->st] = temp->st;
108                      }
109                      temp = temp->link;
110                  }
111              }
112          }
113          printf("\n");
114          if(c != 0){
115              for (int m = 1; m <= nostate; ++m)
116              {
117                  tmpstate.nst[m] = set[m];
118              }
119              insertdfastate(tmpstate);
120              printnewstate(newstate);
121
122              printf("%c\t",alphabet[k] );
123              printnewstate(tmpstate);
124              printf("\n");
125          }

```

```

126         else{
127             printnewstate(newstate);
128             printf("%c\t",alphabet[k]);
129             printf("NULL\n");
130         }
131     }
132 }
133 printf("\n");
134 printf("States of DFA\n");
135 for (int i = 0; i <= complete; ++i)
136 {
137     printnewstate(hash[i]);
138 }
139 }
140 printf("alphabets: ");
141 for (int j = 0; j < noalpha; ++j)
142 {
143     printf("%c \t", alphabet[j]);
144 }
145 printf("\n");
146 printf("Start state: q%d\n",start );
147 printf("Final states: ");
148 findfinalstate();
149 printf("\n");
150 }
151 // adds a new DFA state into hash if not already present
152 int insertdfastate(struct node1 newstate){
153     for (int i = 0; i <= complete; ++i)
154     {
155         if(compare(hash[i],newstate)){
156             return 0;
157         }
158     }
159     complete++;
160     hash[complete] = newstate;
161     return 1;
162 }
163 }
164 // two DFA states compared
165 int compare(struct node1 a, struct node1 b){
166     for (int i = 1; i <= nostate; ++i)
167     {
168         if(a.nst[i] != b.nst[i]){
169             return 0;
170         }
171     }
172     return 1;
173 }
174 // adds transition to transition table
175 void insert(int r,char c, int s){
176     struct node* temp;
177     int j = findalpha(c);
178     if(j==999){
179         printf("Error\n");
180         exit(0);
181     }
182     temp = (struct node*)malloc(sizeof(struct node));
183     temp->st = s;
184     temp->link = transition[r][j];
185     transition[r][j] = temp;
186 }
187 // finds index of given alphabet
188 int findalpha(char c){
189     for (int i = 0; i < noalpha; ++i)
190     {
191         if(alphabet[i] == c){
192             return i;
193         }
194     }
195     return 999;
196 }
197 // identifies and prints final states

```

```

201 void findfinalstate(){
202     for (int i = 0; i <= complete; ++i)
203     {
204         for (int j = 1; j <= nostate; ++j)
205         {
206             for (int k = 0; k < nofinal; ++k)
207             {
208                 if(hash[i].nst[j] == finalstate[k]){
209                     printnewstate(hash[i]);
210                     printf("\t");
211                     j = nostate;
212                     break;
213                 }
214             }
215         }
216     }
217     printf("\n");
218 }
219
220 // print in readable format
221 void printnewstate(struct node1 state){
222     printf("{");
223     for (int i = 1; i <= nostate; ++i)
224     {
225         if(state.nst[i] != 0){
226             printf("q%d, ",state.nst[i] );
227         }
228     }
229     printf("}\t");
230 }

```

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 α is a terminal, then $\text{FIRST}(\alpha) = \alpha$.
4. if α is a non-terminal and $\alpha \rightarrow \epsilon$ is a production, then $\text{FIRST}(\alpha) = \epsilon$.
5. if α is a non-terminal and $\alpha \rightarrow \gamma_1 \gamma_2 \gamma_3 \dots \gamma_n$ and any $\text{FIRST}(\gamma_i)$ contains t then t is in $\text{FIRST}(\alpha)$.
6. Calculating follow
7. if α is a start symbol, then $\text{FOLLOW}(\alpha) = \$$
8. if α is a non-terminal and has a production $\alpha \rightarrow AB$, then $\text{FIRST}(B)$ is in $\text{FOLLOW}(\alpha)$ except ϵ .
9. if α is a non-terminal and has a production $\alpha \rightarrow AB$, where $B \in$, then $\text{FOLLOW}(\alpha)$ is in $\text{FOLLOW}(B)$.
10. Stop

14.3 Program

```
1
2 #include <stdio.h>
3 #include <math.h>
4 #include <string.h>
5 #include <ctype.h>
6 #include <stdlib.h>
7
8 // n number of productions
9 // m track of number of elements in FIRST or FOLLOW
10 // a store grammar productions
11 // f store FIRST or FOLLOW for non terminals
12 int n, m = 0, p, i = 0, j = 0;
13 char a[10][10], f[10];
14
15 void follow(char c);
16 void first(char c);
17
18 int main() {
19     int i, z;
20     char c, ch;
21
22     printf("Enter the number of productions:\n");
23     scanf("%d", &n);
24
25     printf("Enter the productions:\n");
26     for (i = 0; i < n; i++)
27         scanf("%s%c", a[i], &ch);
28
29     do {
30         m = 0;
31         printf("Enter the elements whose first & follow is to be found: ");
32         scanf(" %c", &c); // Note: space before %c to skip any whitespace
33         first(c);
34         printf("First(%c)={", c);
35         for (i = 0; i < m; i++)
36             printf("%c", f[i]);
37         printf("}\n");
38
39         strcpy(f, " ");
40         m = 0;
41         follow(c);
42         printf("Follow(%c)={", c);
43         for (i = 0; i < m; i++)
```

```

44     printf("%c", f[i]);
45     printf("}\n");
46
47     printf("Continue (0/1)? ");
48     scanf("%d%c", &z, &ch); // Note: space before %c to skip any whitespace
49 } while (z == 1);
50
51     return 0;
52 }
53
54 void first(char c) {
55     int k;
56
57     // is c terminal then not uppercase
58     // add it to first set f
59     if (!isupper(c)) {
60         f[m++] = c;
61         return;
62     }
63     // for each production rule where c is LHS
64
65     for (k = 0; k < n; k++) {
66         if (a[k][0] == c) {
67             if (a[k][2] == '$') {
68                 // indicate epsilon transition so follow
69                 follow(a[k][0]);
70             } else if (islower(a[k][2])) {
71                 // RHS starts with terminal then add to first
72                 f[m++] = a[k][2];
73             } else {
74                 // RHS starts with non terminal recursively calculate
75                 first(a[k][2]);
76             }
77         }
78     }
79 }
80
81 void follow(char c) {
82     int k;
83
84     if (a[0][0] == c)
85         // starting symbol
86         f[m++] = '$';
87
88     for (i = 0; i < n; i++) {
89         // each production rule
90         for (j = 2; j < strlen(a[i]); j++) {
91             if (a[i][j] == c) {
92                 if (a[i][j + 1] != '\0') {
93                     // if symbol following c add first of that symbol
94                     first(a[i][j + 1]);
95                 }
96                 if (a[i][j + 1] == '\0' && a[i][0] != c) {
97                     // end of RHS add follow of LHS non terminal
98                     follow(a[i][0]);
99                 }
100             }
101         }
102     }
103 }

```

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

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #define SUCCESS 1
5 #define FAILURE 0
6
7 int E(),E_prime(),T(),T_prime(),F();
8
9 char *cursor;
10 char s[64];
11
12 int main(){
13     printf("Enter string\n");
14     scanf("%s",s);
15     cursor = s;
16     printf("\n");
17     printf("Input\tAction\n");
18     printf("-----\n");
19     if(E() && *cursor == '\0'){
20         printf("-----\n");
21         printf("Parsed successfully\n");
22         return 0;
23     }
24     else{
25         printf("-----\n");
26         printf("Error \n");
27         return 1;
28     }
29 }
30
31 // E-> TE'
32 int E(){
33     printf("%-16s E->TE'\n",cursor );
34     if(T()){
35         if(E_prime())
36             return SUCCESS;
37     }
38     return FAILURE;
39 }
40
41 // E'-> +TE' | $
42 int E_prime(){
43     if(*cursor == '+'){
44         printf("%-16s E'->+TE'\n",cursor );
45         cursor++;
46         if(T()){
47             if(E_prime())
48                 return SUCCESS;
49         }
50     }
51     else{
52         printf("%-16s E'-> $\n", cursor);
53         return SUCCESS;
54     }
```

```

55     }
56     return FAILURE;
57 }
58
59 // T-> FT'
60
61 int T(){
62     printf("%-16s T->FT'\n",cursor );
63     if(F()){
64         if(T_prime())
65             return SUCCESS;
66     }
67     return FAILURE;
68 }
69
70
71 // T'-> *FT' | $
72 int T_prime(){
73     if(*cursor == '*'){
74         printf("%-16s T'->*FT'\n",cursor );
75         cursor++;
76         if(F()){
77             if(T_prime())
78                 return SUCCESS;
79         }
80     }
81     else{
82         printf("%-16s T'-> $\n", cursor);
83         return SUCCESS;
84     }
85     return FAILURE;
86 }
87
88 // F -> (E) | i
89
90 int F(){
91     if(*cursor == '('){
92         printf("%-16s F->(E)\n",cursor );
93         cursor++;
94         if(E()){
95             if(*cursor == ')'){
96                 cursor++;
97                 return SUCCESS;
98             }
99         }
100     }
101     else if(*cursor == 'i'){
102         printf("%-16s F->I\n",cursor );
103         cursor++;
104         return SUCCESS;
105     }
106     return FAILURE;
107 }
108 }

```

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. 1 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 $A \rightarrow x_1, x_2, \dots, x_i$;
3. from i =1 to k repeat
4. if(x_i is a non - terminal)
5. call procedure x_i () ;
6. else if(x_i equals current input a)
7. advance input to next symbol
8. else backtrack input and reset pointer .

16.3 Program

```
1 #include <stdio.h>
2 #include <string.h>
3 int k=0,z=0,i=0,j=0,c=0;
4 char a[16],ac[20],stk[15],act[10];
5 void check();
6 int main(){
7     printf("GRAMMAR is \n E->E+E \n E->E*E \n E->(E) \n E->id\n");
8     printf("Enter input string: \n");
9     scanf("%s",a);
10    c = strlen(a);
11    strcpy(act,"SHIFT->");
12    printf("Stack \t\t\t Input \t\t Action\n");
13    for (i = 0,k=0; j<c; i++,k++,j++)
14    {
15        if(a[j] == 'i' && a[j+1] == 'd'){
16            stk[i] = a[j];
17            stk[i+1] = a[j+1];
18            stk[i+2] = '\0';
19            a[j] = ' ';
20            a[j+1] = ' ';
21            printf("\n%s\t\t%s\t\t%sid\n",stk,a,act );
22            check();
23        }
24        else{
25            stk[i] = a[j];
26            stk[i+1] = '\0';
27            a[j] = ' ';
28            printf("\n%s\t\t%s\t\t%s%c\n",stk,a,act,stk[i] );
29            check();
30        }
31    }
32 }
33 printf("\n");
34
```

```

35 }
36
37 void check(){
38     strcpy(ac,"REDUCE TO E");
39     for (z=0; z < c; ++z)
40     {
41         if (stk[z] == 'i' && stk[z+1] == 'd')
42         {
43             stk[z] = 'E';
44             stk[z+1] = '\0';
45             printf("\n%s\t\t%s\t\t%s",stk,a,ac);
46             j++;
47         }
48     }
49     for (z=0; z < c; ++z)
50     {
51         if (stk[z] == 'E' && stk[z+1] == '+' && stk[z+2] == 'E')
52         {
53             stk[z] = 'E';
54             stk[z+1] = '\0';
55             stk[z+2] = '\0';
56             printf("\n%s\t\t%s\t\t%s",stk,a,ac);
57             i-=2;
58         }
59     }
60     for (z=0; z < c; ++z)
61     {
62         if (stk[z] == 'E' && stk[z+1] == '*' && stk[z+2] == 'E')
63         {
64             stk[z] = 'E';
65             stk[z+1] = '\0';
66             stk[z+2] = '\0';
67             printf("\n%s\t\t%s\t\t%s",stk,a,ac);
68             i-=2;
69         }
70     }
71     for (z=0; z < c; ++z)
72     {
73         if (stk[z] == '(' && stk[z+1] == 'E' && stk[z+2] == ')')
74         {
75             stk[z] = 'E';
76             stk[z+1] = '\0';
77             stk[z+2] = '\0';
78             printf("\n%s\t\t%s\t\t%s",stk,a,ac);
79             i-=2;
80         }
81     }
82
83
84 }

```


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 Propagation

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

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <ctype.h>
4 #include <stdlib.h>
5
6 void input();
7 void output();
8 void change(int p,char *res);
9 void constant();
10
11 struct exp{
12     char op[2], op1[5], op2[5], res[5];
13     int flag;
14 } arr[10];
15
16 int n;
17
18 void main(){
19     input();
20     constant();
21     output();
22 }
23
24 void input(){
25     int i;
26     printf("Enter the maximum number of expressions: \n");
27     scanf("%d",&n);
28     printf("Enter the input: \n");
29     for ( i = 0; i < n; ++i)
30     {
31         scanf("%s",arr[i].op);
32         scanf("%s",arr[i].op1);
33         scanf("%s",arr[i].op2);
34         scanf("%s",arr[i].res);
35         arr[i].flag = 0;
36     }
37 }
38
39 void constant(){
40     int i;
41     int op1,op2,res;
42     char op,res1[5];
43     for ( i = 0; i < n; ++i)
44     {
45         if (isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"=") == 0 )
46         {
47             // if both digits store in variables
48             op1 = atoi(arr[i].op1);
49             op2 = atoi(arr[i].op2);
50             op = arr[i].op[0];
51             switch(op){
52                 case '+':
53                     res = op1+op2;
```

```

54         break;
55     case '-':
56         res = op1-op2;
57         break;
58     case '*':
59         res = op1*op2;
60         break;
61     case '/':
62         res = op1/op2;
63         break;
64     case '=':
65         res = op1;
66         break;
67     }
68     sprintf(res1,"%d", res);
69     arr[i].flag = 1;
70     change(i,res1);
71 }
72 }
73 }
74
75 void output(){
76     int i=0;
77     printf("Optimized code\n");
78     for (i = 0; i < n; ++i)
79     {
80         if(!arr[i].flag){
81             printf("%s %s %s %s\n", arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);
82         }
83     }
84 }
85 }
86
87 void change(int p, char *res){
88     int i;
89     for ( i = p+1; i < n; ++i)
90     {
91         if(strcmp(arr[p].res, arr[i].op1) == 0){
92             strcpy(arr[i].op1,res);
93         }
94         else if(strcmp(arr[p].res,arr[i].op2) == 0){
95             strcpy(arr[i].op2,res);
96         }
97     }
98 }

```

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 1. Start
2 2. Open the input file in read mode.
3 3. Open the output file in write mode.
4 4. In input file scan for operator, argument1, argument2 and result.
5 5. If the operator is +
6     Move arg1 to R0
7     Add arg2 and R0
8     Move R0 to result
9 6. If the operator is -
10    Move arg1 to R0
11    Subtract arg2 and R0
12    Move R0 to result
13 7. If the operator is *
14    Move arg1 to R0
15    Multiply arg2 and R0
16    Move R0 to result
17 8. If the operator is /
18    Move arg1 to R0
19    Divide arg2 and R0
20    Move R0 to result
21 9. If the operator is =
22    Move arg1 to R0
23    Move R0 to result
24 10. Close both the files.
25 11. Stop
```

18.3 Program

```
1
2 #include <stdio.h>
3 #include <string.h>
4
5 void code_op(char *inp, char op, char *reg){
6     int i=0,j=0;
7     char temp[100];
8     while(inp[i] != '\0'){
9         if(inp[i] == op){
10             printf("%c\t%c\t%c\t%c\t%c\n\n", op, *reg, inp[i-1],inp[i+1]);
11             temp[j-1] = *reg;
12             i+=2;
13             (*reg)--;
14             continue;
15         }
16         temp[j] = inp[i];
17         i++;
18         j++;
19     }
20     temp[++j] = '\0';
21     strcpy(inp,temp);
22 }
23
24 void gen_code(char *inp){
25     char reg = 'Z';
26     code_op(inp,'/',&reg);
27
28     code_op(inp,'*',&reg);
29
30     code_op(inp,'+',&reg);
31
32     code_op(inp,'-',&reg);
33
34     code_op(inp,'=',&reg);
35 }
```

```

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
Opertr   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 1. Start
2 2. Open the source file and store the contents as quadruples.
3 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 4. Write the generated code to output definition of the file.
5 5. Print the output.
6 6. Stop
```

19.3 Program

```
1 #include <stdio.h>
2 #include <string.h>
3
4 void main(){
5     char icode[10][30], str[20], opr[10];
6     int i=0;
7     printf("Enter the set of intermediate code (terminated by exit) \n");
8     do{
9         scanf("%s", icode[i]);
10
11     }while(strcmp(icode[i++] , "exit") != 0);
12
13     printf("Target code \n");
14     i=0;
15     do{
16         strcpy(str, icode[i]);
17         switch(str[3]){
18             case '+':
19                 strcpy(opr, "ADD");
20                 break;
21             case '-':
22                 strcpy(opr, "SUB");
23                 break;
24             case '*':
25                 strcpy(opr, "MUL");
26                 break;
27             case '/':
28                 strcpy(opr, "DIV");
29                 break;
30         }
31         printf("MOV  %c,R%d\n",str[2],i );
32
33         printf("%s  %c,R%d\n",opr,str[4],i );
34
35         printf("MOV  R%d,%c\n",i,str[0] );
36
37     }while(strcmp(icode[++i] , "exit") != 0);
38     printf("\n");
39 }
```

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.