

CONTENTS

| Experiment Number | Experiment Name | Page Number | Date | Signature |
|-------------------|--|-------------|------------|-----------|
| 1 | Design and Implement Lexical Analyzer Using C | 4 | 28/09/2022 | |
| 2 | Implement Lexical Analyzer Using LEX | 7 | 28/09/2022 | |
| 3 | LEX Program to Display Number of Lines and Words | 11 | 28/09/2022 | |
| 4 | LEX Program to Convert the Substring abc to ABC | 12 | 28/09/2022 | |
| 5 | LEX Program to find the Number of Vowels and Consonants | 13 | 28/09/2022 | |
| 6 | Generate YACC Specification to Recognize a Valid Arithmetic Expression | 14 | 12/10/2022 | |
| 7 | Generate YACC Specification to Recognize a Valid Identifier | 17 | 12/10/2022 | |
| 8 | Implementation of Calculator Using LEX and YACC | 19 | 12/10/2022 | |
| 9 | Convert BNF Rules into YACC and Write Abstract Syntax Tree | 21 | 12/10/2022 | |
| 10 | Program to Find ϵ -Closure of All States of NFA | 25 | 19/10/2022 | |
| 11 | Program to Convert NFA With ϵ Transition to NFA Without ϵ Transition | 28 | 19/10/2022 | |
| 12 | Program to Convert NFA to DFA | 32 | 26/10/2022 | |
| 13 | Program to Minimize Any Given DFA | 39 | 26/10/2022 | |
| 14 | Program to Find First and Follow of Any Grammar | 45 | 02/11/2022 | |
| 15 | Design and Implement Recursive Descent Parser | 49 | 02/11/2022 | |
| 16 | Construct Shift Reduce Parser | 54 | 09/11/2022 | |
| 17 | Program to Perform Constant Propagation | 57 | 16/11/2022 | |
| 18 | Program for Intermediate Code Generation | 60 | 23/11/2022 | |
| 19 | Implementation of Back-end Compiler | 62 | 30/11/2022 | |



CSL 411 - Compiler Lab

CYCLE 1 : Application of LEX and YACC Tools

1 Experiment 1

1.1 Aim

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. Read input file using file operator.
2. Traverse the file one character at a time.
3. For each character, check if the character is an operator.
4. If yes, print operator.
5. Else check if the character is an alphabet or number.
6. If so, append the character to a buffer string.
7. Append until a space or negative character is encountered.
8. Check whether the contents inside is identifier or keyword.
9. Print the corresponding token and empty the buffer string.
10. Continue reading file until EOF.
11. EXIT

1.3 Program

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.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 (isdigit(str[0]) || isDelimiter(str[0]) == true)
        return (false);
}
```

```

    return (true);
}

bool isKeyword(char* str){
    if (!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") || !strcmp(str,
        "do") ||
        !strcmp(str, "break") || !strcmp(str, "continue") || !strcmp(str, "int") ||
        !strcmp(str, "double")
        || !strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char") || !strcmp(str, "sizeof") ||
        !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef") || !strcmp(str, "switch") ||
        !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static") || !strcmp(str, "struct") ||
        !strcmp(str, "goto"))
        return (true);
    return (false);
}

char* subString(char* str, int left, int right){
    int i;
    char* subStr = (char*)malloc(sizeof(char) * (right - left + 2));
    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

void parse(char* str){
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

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

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

            if (isKeyword(subStr) == true)
                printf("'%s' IS A KEYWORD\n", subStr);

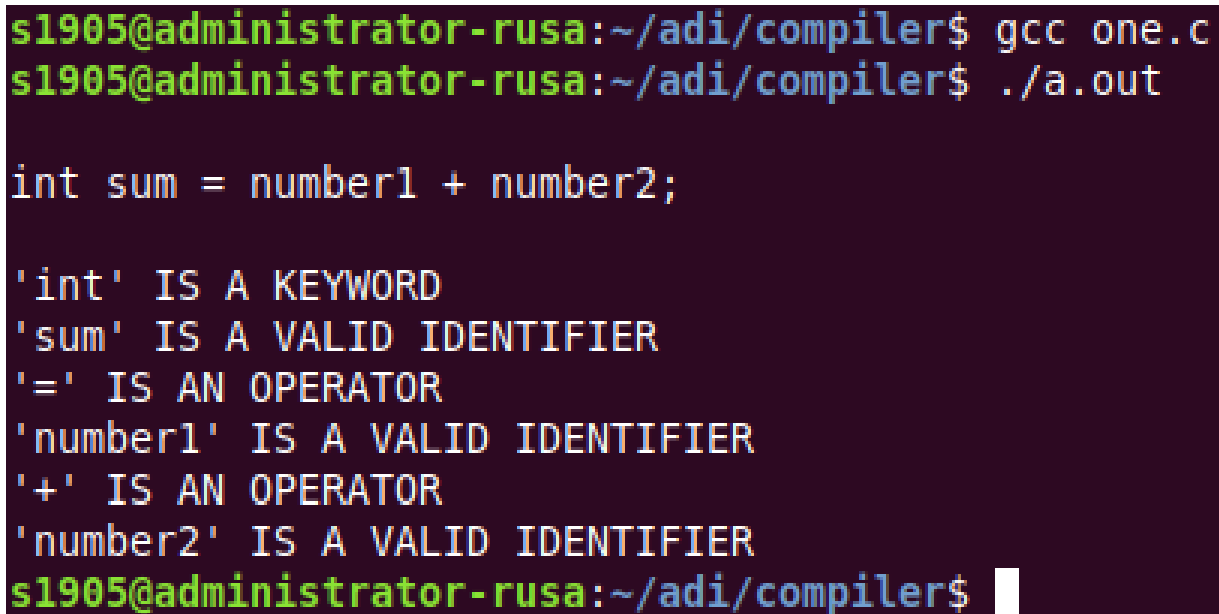
            else if (validIdentifier(subStr) == true
                && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS A VALID IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false
                && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
            left = right;
        }
    }
    return;
}

```

```
int main()
{
    char str[100] = "int sum = number1 + number2; ";
    printf("\n%s\n\n",str);
    parse(str);
    return (0);
}
```

1.4 Output



```
s1905@administrator-rusa:~/adi/compiler$ gcc one.c
s1905@administrator-rusa:~/adi/compiler$ ./a.out

int sum = number1 + number2;

'int' IS A KEYWORD
'sum' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'number1' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'number2' IS A VALID IDENTIFIER
s1905@administrator-rusa:~/adi/compiler$
```

1.5 Result

Implemented a lexical analyzer using C language to recognize all valid tokens in the input program.

2 Experiment 2

2.1 Aim

Implement a Lexical Analyzer for a given program using Lex Tool

2.2 Algorithm

Step1: 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

Step2: 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.

Step3: 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.

Step4: Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

Step5: 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.

Step6: In user subroutine section, main routine calls `yylex()`. `yywrap()` is used to get more input.

Step7: 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

```
%{  
int COMMENT=0;  
%}  
identifier [a-zA-Z][a-zA-Z0-9]*  
%%  
#.* {printf("\n%s is a preprocessor directive",yytext);}
```

```

int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
\{ {if(!COMMENT)printf("\n BLOCK BEGINS");}
\} {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 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 File

```
s1905@administrator-rusa:~/adi/compiler$ cat var.c
main(){
    int a,b;
    a = b;
}s1905@administrator-rusa:~/adi/compiler$
```

Output

```
s1905@administrator-rusa:~/adi/compiler$ lex lextool.l
s1905@administrator-rusa:~/adi/compiler$ cc lex.yy.c -ll
s1905@administrator-rusa:~/adi/compiler$ ./a.out

FUNCTION
    main(
)

BLOCK BEGINS

    int is a keyword
a IDENTIFIER,
b IDENTIFIER;

a IDENTIFIER
    = is an ASSIGNMENT OPERATOR
b IDENTIFIER;
BLOCK ENDS
s1905@administrator-rusa:~/adi/compiler$
```

2.5 Result

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

3 Experiment 3

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 chars=0,words=0,lines=0;
}%

%%
[^\t\n ]+ {words++;
           chars=chars+yylen;}
[ ]* {chars++;}
[\n] {lines++;
      chars++;}
%%

int main(){
    yyin = fopen("input.txt","r");
    yylex();
    fclose(yyin);
    printf("\nwords = %d",words);
    printf("\ncharacters = %d",chars);
    printf("\nlines = %d\n",lines);
    return 0;
}
```


3.4 Output

```
s1905@administrator-rusa:~/adi/compiler$ cat input.txt
lorem inpsum
sample text
kkk kkk
classroom good boy
s1905@administrator-rusa:~/adi/compiler$ lex len.l
s1905@administrator-rusa:~/adi/compiler$ cc lex.yy.c -ll
s1905@administrator-rusa:~/adi/compiler$ ./a.out
Words = 9
Characters = 52
spaces = 5
lines = 4s1905@administrator-rusa:~/adi/compiler$
```

3.5 Result

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

4 Experiment 4

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

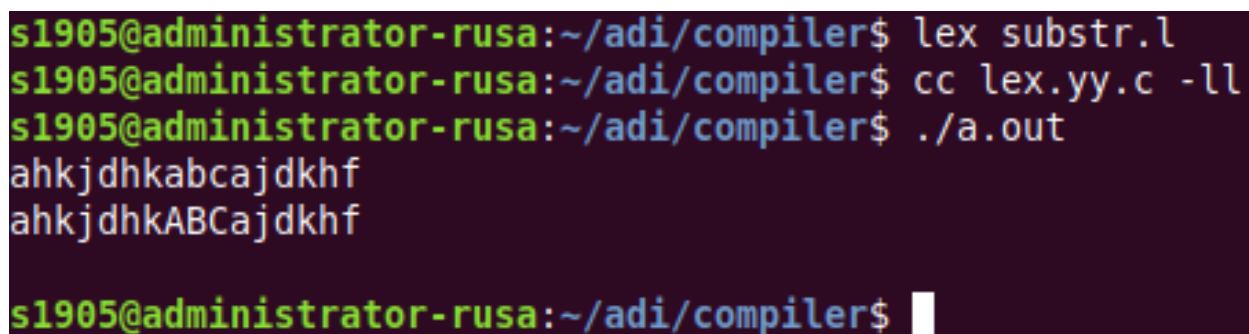
4.3 Program

```
%{
%}

%%
[a-zA-Z]* {
    for(int i=0;i<yyleng-2; i++){
        if(yytext[i] == 'a' && yytext[i+1] == 'b' && yytext[i+2] == 'c'){
            yytext[i] = 'A';
            yytext[i+1] = 'B';
            yytext[i+2] = 'C';
        }
    }
    printf("%s\n",yytext);
}
%%

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

4.4 Output



```
s1905@administrator-rusa:~/adi/compiler$ lex substr.l
s1905@administrator-rusa:~/adi/compiler$ cc lex.yy.c -ll
s1905@administrator-rusa:~/adi/compiler$ ./a.out
ahkjdhkabcajdkhf
ahkjdhkABCajdkhf

s1905@administrator-rusa:~/adi/compiler$
```

4.5 Result

Implemented LEX Program to convert the substring abc to ABC from the given input string.

5 Experiment 5

5.1 Aim

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

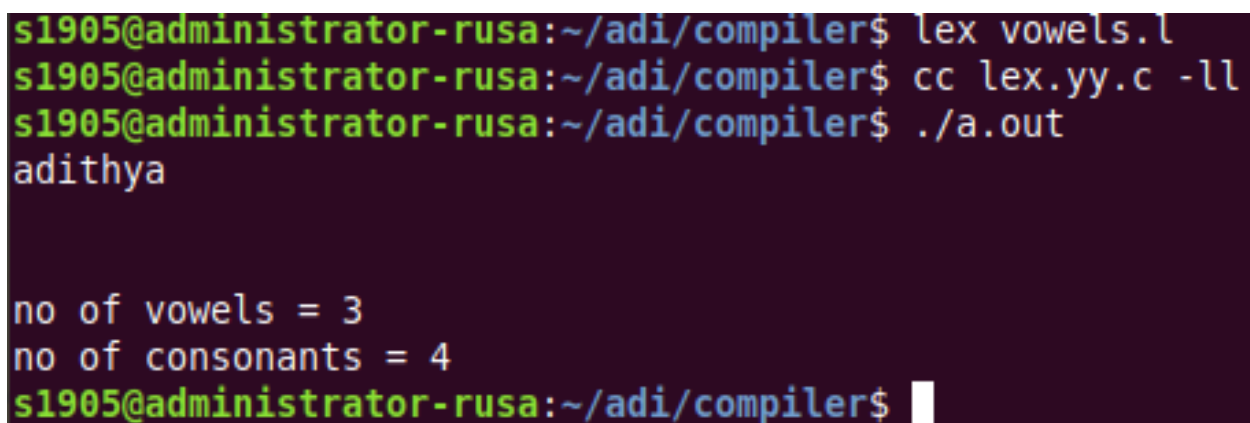
5.2 Algorithm

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 vowels=0,cons=0;  
}%  
  
%%  
[aeiouAEIOU] { vowels++;}  
[a-zA-Z] { cons++; }  
%%  
  
int main(){  
    yylex();  
    printf("\nno of vowels = %d",vowels);  
    printf("\nno of consonants = %d\n",cons);  
    return 0;  
}
```

5.4 Output



```
s1905@administrator-rusa:~/adi/compiler$ lex vowels.l  
s1905@administrator-rusa:~/adi/compiler$ cc lex.yy.c -ll  
s1905@administrator-rusa:~/adi/compiler$ ./a.out  
adithya  
  
no of vowels = 3  
no of consonants = 4  
s1905@administrator-rusa:~/adi/compiler$
```

5.5 Result

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

6 Experiment 6

6.1 Aim

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

6.2 Algorithm

1. Start
2. Read an expression
3. Checking the validation of the given expression according to the rule using yacc.
4. if valid expression print VALID EXPRESSION
5. if error print INVALID EXPRESSION
6. Stop

6.3 Program

YACC Code

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

%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%left '(' ')'

%%
valid : e NL{printf("\n valid expression!\n");}
e : e '+' e
  | e '-' e
  | e '*' e
  | e '/' e
  | '(' e ')'
  | NUMBER
  | ID ;
%%

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

int yyerror(char *s)
{
    printf("\n invalid expression\n");
    exit(1);
}
```

LEX Code

```
%{
    #include "y.tab.h"
    int yylval;
}%
```

```

%%
[0-9]+ {yyval=atoi(yytext);
        return NUMBER;}
[a-zA-Z]+ {return ID;}
[\t]+;
\n {return NL;}
. {return yytext[0];}
%%

int yywrap()
{
    return 1;
}

```

6.4 Output

```

s1905@administrator-rusa:~/adi/compiler$ yacc -d 6.y
s1905@administrator-rusa:~/adi/compiler$ lex 6.l
s1905@administrator-rusa:~/adi/compiler$ cc y.tab.c lex.yy.c -w
s1905@administrator-rusa:~/adi/compiler$ ./a.out

Enter an expression: 4+5

valid expression!
s1905@administrator-rusa:~/adi/compiler$ ./a.out

Enter an expression: 45+

invalid expression
s1905@administrator-rusa:~/adi/compiler$ █

```

6.5 Result

Implemented YACC and LEX Program to recognize valid arithmetic expression.

7 Experiment 7

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

7.3 Program

YACC Code

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

%token DIGIT LETTER NL UND

%%

valid : id NL {printf("\n valid identifier!\n");}
id : LETTER alphanum;
alphanum : LETTER alphanum
          | DIGIT alphanum
          | UND alphanum
          | LETTER
          | DIGIT
          | UND ;

%%

int main()
{
    printf("\n Enter an identifier: ");
    yyparse();
}

int yyerror()
{
    printf("\n invalid identifier\n");
    exit(1);
}
```

LEX Code

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

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

```
int yywrap()
{
    return 1;
}
```

7.4 Output

```
s1905@administrator-rusa:~/adi/compiler$ yacc -d 7.y
s1905@administrator-rusa:~/adi/compiler$ lex 7.l
s1905@administrator-rusa:~/adi/compiler$ cc y.tab.c lex.yy.c -w
s1905@administrator-rusa:~/adi/compiler$ ./a.out

Enter an identifier: val1

valid identifier!
s1905@administrator-rusa:~/adi/compiler$ ./a.out

Enter an identifier: 1val

invalid identifier
s1905@administrator-rusa:~/adi/compiler$
```

7.5 Result

YACC and LEX Programs to recognize valid identifiers were successfully implemented.

8 Experiment 8

8.1 Aim

Implementation of Calculator using LEX and YACC

8.2 Algorithm

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

8.3 Program

YACC Code

```
%{
    #include<stdio.h>
    int flag=0;
}%

%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%

valid : e{printf("\n Result=%d\n",);}
e : e '+' e {=1+3;}
  | e '-' e {=1-3;}
  | e '*' e {=1*3;}
  | e '/' e {=1/3;}
  | e '%' e {=1|('e')'=2;}
  | NUMBER {=
```

LEX Code

```
%{
    #include<stdio.h>
    #include "y.tab.h"
    int yylval;
}%

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

int yywrap()
{
    return 1;
}
```


8.4 Output

```
s1905@administrator-rusa:~/adi/compiler$ yacc -d 8.y
s1905@administrator-rusa:~/adi/compiler$ lex 8.l
s1905@administrator-rusa:~/adi/compiler$ cc y.tab.c lex.yy.c -w
s1905@administrator-rusa:~/adi/compiler$ ./a.out

Enter an expression: 1+5

Result=6
Expression is valid
s1905@administrator-rusa:~/adi/compiler$ ./a.out

Enter an expression: +14

invalid expression
s1905@administrator-rusa:~/adi/compiler$
```

8.5 Result

Calculator was successfully implemented using LEX and YACC.

9 Experiment 9

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 an identifier
3. Checking the validation of the given identifier according to the rule using yacc.
4. If valid identifier set validid = 1
5. if validid = 1 print VALID IDENTIFIER
6. if error print INVALID IDENTIFIER
7. Stop

9.3 Program

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 ';'
| CONDEST
| WHILEST
;
DESCRIPT: 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
;
CONDEST: 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
        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,

```

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];
%%

```

9.4 Output

```
s1905@administrator-rusa:~/adi/compiler$ yacc -d 9.y
s1905@administrator-rusa:~/adi/compiler$ lex 9.l
s1905@administrator-rusa:~/adi/compiler$ cc y.tab.c lex.yy.c -ll
s1905@administrator-rusa:~/adi/compiler$ ./a.out test.c
```

```
-----
Pos Operator Arg1 Arg2 Result
-----
0      <      a      b      t0
1      ==     t0     FALSE  5
2      +      a      b      t1
3      =      t1     a
4      GOTO
5      <      a      b      t2
6      ==     t2     FALSE 10
7      +      a      b      t3
8      =      t3     a
9      GOTO
10     <=     a      b      t4
11     ==     t4     FALSE 15
12     -      a      b      t5
13     =      t5     c
14     GOTO
15     +      a      b      t6
16     =      t6     c
-----
```

```
s1905@administrator-rusa:~/adi/compiler$ █
```

9.5 Result

The program using YACC has been executed successfully.

CYCLE 2 : Application problems using NFA and DFA

10 Experiment 10

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

```
#include<stdio.h>
#include<string.h>
char result[20][20], copy[3], states[20][20];
void add_state(char a[3], int i) {
    strcpy(result[i], a);
}
void display(int n) {
    int k = 0;
    printf("Epsilon closure of %s = { ", copy);
    while (k < n) {
        printf(" %s", result[k]);
        k++;
    }
    printf(" }\n");
}
int main() {
    FILE * INPUT;
    INPUT = fopen("input.dat", "r");
    char state[3];
    int end, i = 0, n, k = 0;
    char state1[3], input[3], state2[3];
    printf("Enter the no of states: ");
    scanf("%d", & n);
    printf("Enter the states :");
    for (k = 0; k < 3; k++) {
        scanf("%s", states[k]);
    }
    for (k = 0; k < n; k++) {
        i = 0;
        strcpy(state, states[k]);
        strcpy(copy, state);
        add_state(state, i++);
        while (1) {
            end = fscanf(INPUT, "%s%s%s", state1, input, state2);
```

```

        if (end == EOF) {
            break;
        }
        if (strcmp(state, state1) == 0) {
            if (strcmp(input, "e") == 0) {
                add_state(state2, i++);
                strcpy(state, state2);
            }
        }
    }
    display(i);
    rewind(INPUT);
}
return 0;
}

```

10.4 Output

```

s1905@administrator-rusa:~/adi/compiler$ gcc 10.c
s1905@administrator-rusa:~/adi/compiler$ ./a.out
Enter the no of states: 3
Enter the states :q0 q1 q2
Epsilon closure of q0 = {  q0 q1 q2 }
Epsilon closure of q1 = {  q1 q2 }
Epsilon closure of q2 = {  q2 }
s1905@administrator-rusa:~/adi/compiler$ cat input.dat
q0 0 q0
q0 1 q1
q0 e q1
q1 1 q2
q1 e q2s1905@administrator-rusa:~/adi/compiler$

```

10.5 Result

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

11 Experiment 11

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
10. Stop

11.3 Program

```
#include<stdio.h>

typedef struct Node{
    int transition[5];
}Node;

Node node[10];
int e_arr[10];
int e_sub_arr[10];
char alphabets[10];

int map(char a){

    if( a == 'e' )
        return 0;
    int i;
    i=a-96;
    return i;
}

int e_closure(int e, int res[]){
    int count=0, i;
    res[count++] = e; // e closure contain itself
    while(node[e].transition[ map('e') ] !=-1 ){
        i = node[e].transition[ map('e') ];
        res[count++] = i;
        e = i;
    }
    return count;
}

int get_transition(int e,char a){
    if(node[e].transition[ map(a) ] !=-1 ){
        int i = node[e].transition[ map(a) ];
        return i;
    }
    return -1;
}
```



```

}

void perform(int n,int alpha)
{
    int i,c,j,r[50],b,p,k,w,copy[50],q;
    for(i=0;i<n;i++){
        c=e_closure(i,e_arr);// finding e-closure

        // finding transition for each alphabet
        for(k=0;k<alpha;k++){
            int cnt=0;
            for(j=0;j<c;j++){
                int flag;
                flag=get_transition(e_arr[j],alphabets[k]);
                if(flag!=-1){
                    r[cnt++]=flag;
                }
            } // finding transition of it for each alphabet
            int count=0;
            for( w=0;w<cnt;w++){
                b = e_closure(r[w],e_sub_arr);
                for(q=0;q<b;q++){
                    copy[count++] = e_sub_arr[q];
                }
            }

            printf("\nState→ %d for alphabet %c >> { ",i,alphabets[k]);
            int print;
            for(p=0;p<count;p++){
                print=1;
                for(int r=p-1;r>0;r--){
                    if(copy[p] == copy[r])
                        print=0;
                }
                if( print==1)
                    printf(" %d",copy[p]);
            }
            printf(" } ");
        }
    }
}

int main(){
    /*variables in main()
    * INTEGERS
    * n - no of nodes
    * t - no of transitions
    * a - no of alphabets
    * -1 # indicates No transition for that input alphabet
    */
    int n,t,a,state1,state2,i,j;
    char alpha;

    printf("\n Enter the no of nodes :");
    scanf("%d",&n);
    printf("\n Enter the no of alphabets: ");
    scanf("%d",&a);
    for(i=0;i<a;i++)

```

```

scanf(" %c",&alphabets[i]);

for(i=0;i<n;i++){
    for(j=0;j<a+1;j++)
        node[i].transition[j] = -1;
}
printf("\n Enter the no of transitions: ");
scanf("%d", &t);
printf("\nEnter the state table:>>nSTATE ALPHABET STATE\n");
for(i=0;i<t;i++){
    scanf("%d %c %d",&state1,&alpha,&state2);
    node[state1].transition[ map(alpha) ] = state2;
}
perform(n,a);
printf("\n");
}

```

11.4 Output

```
s1905@administrator-rusa:~/adi/compiler$ gcc 11.c
s1905@administrator-rusa:~/adi/compiler$ ./a.out

Enter the no of nodes :3

Enter the no of alphabets: 2
a b

Enter the no of transitions: 5

Enter the state table:>>nSTATE ALPHABET STATE
0 a 0
0 b 1
0 e 1
1 b 2
1 e 2

State-> 0 for alphabet a >> { 0 1 2 }
State-> 0 for alphabet b >> { 1 2 }
State-> 1 for alphabet a >> { }
State-> 1 for alphabet b >> { 2 }
State-> 2 for alphabet a >> { }
State-> 2 for alphabet b >> { }
s1905@administrator-rusa:~/adi/compiler$
```

11.5 Result

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

12 Experiment 12

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 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

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int st;
    struct node *link;
};
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,s,notransition,nofinal,start,finalstate[20],
r,buffer[20];
int complete=-1;
char c,alphabet[20];
static int eclosure[20][20]={0};
struct node1 hash[20];
struct node * transition[20][20]={NULL};
void main()
{
    int i,j,k,m,t,n,l;
    struct node *temp;
    struct node1 newstate={0},tmpstate={0};

    printf("Enter the number of alphabets?\n");
    printf("NOTE:- [ use letter e as epsilon]\n");
    printf("NOTE:- [e must be last character ,if it is present]\n");
    printf("\nEnter No of alphabets and alphabets?\n");
```

```

scanf("%d",&noalpha);
getchar();
for(i=0;i<noalpha;i++)
{

alphabet[i]=getchar();
getchar();
}
printf("Enter the number of states?\n");
scanf("%d",&nostate);
printf("Enter the start state?\n");
scanf("%d",&start);
printf("Enter the number of final states?\n");
scanf("%d",&nofinal);
printf("Enter the final states?\n");
for(i=0;i<nofinal;i++)
scanf("%d",&finalstate[i]);
printf("Enter no of transition?\n");

scanf("%d",&notransition);
printf("NOTE:- [Transition is in the form> qno alphabet qno] \n");
printf("NOTE:- [States number must be greater than zero]\n");
printf("\nEnter transition?\n");

for(i=0;i<notransition;i++)
{

scanf("%d %c%d",&r,&c,&s);
insert(r,c,s);

}
for(i=0;i<20;i++)
{
for(j=0;j<20;j++)
hash[i].nst[j]=0;
}
complete=-1;
i=-1;
printf("\nEquivalent DFA....\n");
printf("Trnsitions of DFA\n");

newstate.nst[start]=start;
insertdfastate(newstate);
while(i!=complete)
{
i++;
newstate=hash[i];
for(k=0;k<noalpha;k++)
{
c=0;
for(j=1;j<=nostate;j++)
set[j]=0;
for(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(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("\nStates of DFA:\n");
for(i=0;i<=complete;i++)
    printnewstate(hash[i]);
printf("\n Alphabets:\n");
for(i=0;i<noalpha;i++)
    printf("%c\t",alphabet[i]);
printf("\n Start State:\n");
printf("q%d",start);
printf("\nFinal states:\n");
findfinalstate();

}

int insertdfastate(struct node1 newstate)
{
    int i;
    for(i=0;i<=complete;i++)
    {
        if(compare(hash[i],newstate))
            return 0;
    }
    complete++;
    hash[complete]=newstate;
    return 1;
}

int compare(struct node1 a,struct node1 b)
{
    int i;

```

```

    for(i=1;i<=nostate;i++)
    {
        if(a.nst[i]!=b.nst[i])
            return 0;
    }
    return 1;

}

void insert(int r,char c,int s)
{
    int j;
    struct node *temp;
    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;
}

int findalpha(char c)
{
    int i;
    for(i=0;i<noalpha;i++)
    if(alphabet[i]==c)
        return i;

    return(999);
}

void findfinalstate()
{
    int i,j,k,t;

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

```

```

printf("\n");
}

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

```

12.4 Output

```

s1905@administrator-rusa:~/adi/compiler$ gcc 12.c
s1905@administrator-rusa:~/adi/compiler$ ./a.out
Enter the number of alphabets?
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

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
NOTE:- [Transition is in the form-> qno alphabet qno]
NOTE:- [States number must be greater than zero]

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

```



```

Equivalent DFA.....
Trnsitions of DFA

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

States of DFA:
{q1,}    {q1,q2,}    {q1,q2,q3,}    {q1,q2,q4,}    {q1,q2,q3,q4,}
Alphabets:
a        b
Start State:
q1
Final states:
{q1,q2,q3,}    {q1,q2,q4,}    {q1,q2,q3,q4,}
s1905@administrator-rusa:~/adi/compiler$ █

```

12.5 Result

Successfully implemented conversion of NFA to DFA.

13 Experiment 13

13.1 Aim

Write a program to minimise any given DFA.

13.2 Algorithm

1. Remove all the states that are unreachable from the initial start state.
2. Create a table of pairs (p, q) where p, q denotes some two states of Q. Initially, all the table cells are unmarked.
3. Mark all the pairs (p, q) such that p belongs to F and q does not belong to F, or vice versa.
4. If (p, q) is unmarked and there exists a symbol such that $\delta(p, a), \delta(q, a)$ is marked, then mark p and q.
5. Repeat step-4 until no new pairs get marked.
6. After completing the above process, p is equivalent to q if and only if (p, q) is unmarked. The equivalent states can be compressed to get the minimum number of states.

13.3 Program

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

#define STATES 99
#define SYMBOLS 20

int N_symbols; /* number of input symbols */
int N_DFA_states; /* number of DFA states */
char *DFA_finals; /* final-state string */
int DFAtab[STATES][SYMBOLS];
char StateName[STATES][STATES + 1]; /* state-name table */
int N_optDFA_states; /* number of optimized DFA states */
int OptDFA[STATES][SYMBOLS];
char NEW_finals[STATES + 1];

void print_dfa_table(
    int tab[][SYMBOLS], /* DFA table */
    int nstates, /* number of states */
    int nsymbols, /* number of input symbols */
    char *finals)
{
    int i, j;
    puts("\nDFA: STATE TRANSITION TABLE");
    printf("    | ");
    for (i = 0; i < nsymbols; i++) printf(" %c ", '0' + i);
    printf("\n-----+--");
    for (i = 0; i < nsymbols; i++) printf("-----");
    printf("\n");
    for (i = 0; i < nstates; i++) {
        printf(" %c | ", 'A' + i); /* state */
        for (j = 0; j < nsymbols; j++)
            printf(" %c ", tab[i][j]); /* next state */
        printf("\n");
    }
    printf("Final states = %s\n", finals);
}

void load_DFA_table()
```

```

{
    DFAatab[0][0] = 'B'; DFAatab[0][1] = 'C';
    DFAatab[1][0] = 'E'; DFAatab[1][1] = 'F';
    DFAatab[2][0] = 'A'; DFAatab[2][1] = 'A';
    DFAatab[3][0] = 'F'; DFAatab[3][1] = 'E';
    DFAatab[4][0] = 'D'; DFAatab[4][1] = 'F';
    DFAatab[5][0] = 'D'; DFAatab[5][1] = 'E';
    DFA_finals = "EF";
    N_DFA_states = 6;
    N_symbols = 2;
}

void get_next_state(char *nextstates, char *cur_states,
                   int dfa[STATES][SYMBOLS], int symbol)
{
    int i, ch;
    for (i = 0; i < strlen(cur_states); i++)
        *nextstates++ = dfa[cur_states[i] - 'A'][symbol];
    *nextstates = '\0';
}

char equiv_class_ndx(char ch, char stnt[][STATES + 1], int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (strchr(stnt[i], ch)) return i + '0';
    return -1; /* next state is NOT defined */
}

char is_one_nextstate(char *s)
{
    char equiv_class; /* first equiv. class */
    while (*s == '@') s++;
    equiv_class = *s++; /* index of equiv. class */
    while (*s) {
        if (*s != '@' && *s != equiv_class) return 0;
        s++;
    }
    return equiv_class; /* next state: char type */
}

int state_index(char *state, char stnt[][STATES + 1], int n, int *pn,
               int cur) /* 'cur' is added only for 'printf()' */
{
    int i;
    char state_flags[STATES + 1]; /* next state info. */
    if (!*state) return -1; /* no next state */
    for (i = 0; i < strlen(state); i++)
        state_flags[i] = equiv_class_ndx(state[i], stnt, n);
    state_flags[i] = '\0';
    printf(" %d:[%s]\t→ [%s] (%s)\n",
           cur, stnt[cur], state, state_flags);
    if (i = is_one_nextstate(state_flags))
        return i - '0'; /* deterministic next states */
    else {
        strcpy(stnt[*pn], state_flags); /* state-division info */
        return (*pn)++;
    }
}

int init_equiv_class(char statename[][STATES + 1], int n, char *finals)

```

```

{
    int i, j;
    if (strlen(finals) == n) { /* all states are final states */
        strcpy(statename[0], finals);
        return 1;
    }
    strcpy(statename[1], finals); /* final state group */
    for (i = j = 0; i < n; i++) {
        if (i == *finals - 'A') {
            finals++;
        } else statename[0][j++] = i + 'A';
    }
    statename[0][j] = '\0';
    return 2;
}

int get_optimized_DFA(char stnt[][STATES + 1], int n,
                     int dfa[][SYMBOLS], int n_sym, int newdfa[][SYMBOLS])
{
    int n2 = n; /* 'n' + <num. of state-division info> */
    int i, j;
    char nextstate[STATES + 1];
    for (i = 0; i < n; i++) { /* for each pseudo-DFA state */
        for (j = 0; j < n_sym; j++) { /* for each input symbol */
            get_next_state(nextstate, stnt[i], dfa, j);
            newdfa[i][j] = state_index(nextstate, stnt, n, &n2, i) + 'A';
        }
    }
    return n2;
}

void chr_append(char *s, char ch)
{
    int n = strlen(s);
    *(s + n) = ch;
    *(s + n + 1) = '\0';
}

void sort(char stnt[][STATES + 1], int n)
{
    int i, j;
    char temp[STATES + 1];
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++)
            if (stnt[i][0] > stnt[j][0]) {
                strcpy(temp, stnt[i]);
                strcpy(stnt[i], stnt[j]);
                strcpy(stnt[j], temp);
            }
}

int split_equiv_class(char stnt[][STATES + 1],
                     int i1, /* index of 'i1'-th equiv. class */
                     int i2, /* index of equiv. vector for 'i1'-th class */
                     int n, /* number of entries in 'stnt' */
                     int n_dfa) /* number of source DFA entries */
{
    char *old = stnt[i1], *vec = stnt[i2];
    int i, n2, flag = 0;
    char newstates[STATES][STATES + 1]; /* max. 'n' subclasses */
    for (i = 0; i < STATES; i++) newstates[i][0] = '\0';
}

```

```

for (i = 0; vec[i]; i++)
    chr_append(newstates[vec[i] - '0'], old[i]);
for (i = 0, n2 = n; i < n_dfa; i++) {
    if (newstates[i][0]) {
        if (!flag) { /* stnt[i1] = s1 */
            strcpy(stnt[i1], newstates[i]);
            flag = 1; /* overwrite parent class */
        } else /* newstate is appended in 'stnt' */
            strcpy(stnt[n2++], newstates[i]);
    }
}

sort(stnt, n2); /* sort equiv. classes */
return n2; /* number of NEW states(equiv. classes) */
}

int set_new_equiv_class(char stnt[][STATES + 1], int n,
                        int newdfa[][SYMBOLS], int n_sym, int n_dfa)
{
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n_sym; j++) {
            k = newdfa[i][j] - 'A'; /* index of equiv. vector */
            if (k >= n) /* equiv. class 'i' should be segmented */
                return split_equiv_class(stnt, i, k, n, n_dfa);
        }
    }
    return n;
}

void print_equiv_classes(char stnt[][STATES + 1], int n)
{
    int i;
    printf("\nEQUIV. CLASS CANDIDATE ==>");
    for (i = 0; i < n; i++)
        printf(" %d:[%s]", i, stnt[i]);
    printf("\n");
}

int optimize_DFA(
    int dfa[][SYMBOLS], /* DFA state-transition table */
    int n_dfa, /* number of DFA states */
    int n_sym, /* number of input symbols */
    char *finals, /* final states of DFA */
    char stnt[][STATES + 1], /* state name table */
    int newdfa[][SYMBOLS]) /* reduced DFA table */
{
    char nextstate[STATES + 1];
    int n; /* number of new DFA states */
    int n2; /* 'n' + <num. of state-dividing info> */
    n = init_equiv_class(stnt, n_dfa, finals);
    while (1) {
        print_equiv_classes(stnt, n);
        n2 = get_optimized_DFA(stnt, n, dfa, n_sym, newdfa);
        if (n != n2)
            n = set_new_equiv_class(stnt, n, newdfa, n_sym, n_dfa);
        else break; /* equiv. class segmentation ended!!! */
    }
    return n; /* number of DFA states */
}

```

```

int is_subset(char *s, char *t)
{
    int i;
    for (i = 0; *t; i++)
        if (!strchr(s, *t++)) return 0;
    return 1;
}

void get_NEW_finals(
    char *newfinals, /* new DFA finals */
    char *oldfinals, /* source DFA finals */
    char stnt[][STATES + 1], /* state name table */
    int n) /* number of states in 'stnt' */
{
    int i;
    for (i = 0; i < n; i++)
        if (is_subset(oldfinals, stnt[i])) *newfinals++ = i + 'A';
    *newfinals++ = '\0';
}

void main()
{
    load_DFA_table();
    print_dfa_table(DFAstab, N_DFA_states, N_symbols, DFA_finals);
    N_optDFA_states = optimize_DFA(DFAstab, N_DFA_states,
                                   N_symbols, DFA_finals, StateName, OptDFA);
    get_NEW_finals(NEW_finals, DFA_finals, StateName, N_optDFA_states);
    print_dfa_table(OptDFA, N_optDFA_states, N_symbols, NEW_finals);
}

```

13.4 Output

```
s1905@administrator-rusa:~/adi/compiler$ gcc 13.c
s1905@administrator-rusa:~/adi/compiler$ ./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

```
s1905@administrator-rusa:~/adi/compiler$
```

13.5 Result

Successfully implemented a program to minimise DFA.

CYCLE 3 : Implementation of Parsers

14 Experiment 14

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

14.3 Program

```
#include<stdio.h>
#include<math.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.h>
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;
    //clrscr();
    printf("Enter the no of prooductions:\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 elemets whose fisrt & follow is to be found:");
            scanf("%c",&c);
            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);
    }while(z==1);
    return(0);
}

void first(char c)
{
    int k;
    if(!isupper(c))
        f[m++]=c;
    for(k=0;k<n;k++)
    {
        if(a[k][0]==c)
        {
            if(a[k][2]=='')follow(a[k][0]);elseif(islower(a[k][2]))f[m++] = a[k][2];elsefirst(a[k][2]);voidfollow(charc)if
            ,

;
        for(i=0;i<n;i++)
        {
            for(j=2;j<strlen(a[i]);j++)
            {
                if(a[i][j]==c)
                {
                    if(a[i][j+1]!='\0')
                        first(a[i][j+1]);
                    if(a[i][j+1]=='\0' && c!=a[i][0])
                        follow(a[i][0]);
                }
            }
        }
    }
}

```

14.4 Output

```
s1905@administrator-rusa:~/adi/compiler$ gcc 14.c
s1905@administrator-rusa:~/adi/compiler$ ./a.out
Enter the no of prooductions:
5
Enter the productions:
S=AbCd
A=Cf
A=a
C=gE
E=h
Enter the elemets whose fisrt & follow is to be found:A
First(A)={ga}
Follow(A)={b}
Continue(0/1)?1
Enter the elemets whose fisrt & follow is to be found:S
First(S)={ga}
Follow(S)={$}
Continue(0/1)?C
Enter the elemets whose fisrt & follow is to be found:First(C)={g}
Follow(C)={df}
Continue(0/1)?E
Enter the elemets whose fisrt & follow is to be found:First(E)={h}
Follow(E)={df}
Continue(0/1)?0
s1905@administrator-rusa:~/adi/compiler$ |
```

14.5 Result

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

15 Experiment 15

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

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

char input[10];
int i, error;
void E();
void T();
void Eprime();
void Tprime();
void F();
int main()
{
    i = 0;
    error = 0;
    printf("Enter an arithmetic expression : "); // Eg: a+a*a
    scanf("%s",input);
    E();
    if (strlen(input) == i && error == 0)
        printf("\nAccepted..!!!\n");
    else printf("\nRejected..!!!\n");
    return 0;
}

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

```

```

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

        else error = 1;
    }

    else error = 1;
}

```

15.4 Output

```

s1905@administrator-rusa:~/adi/compiler$ gcc 15.c
s1905@administrator-rusa:~/adi/compiler$ ./a.out
Enter an arithmetic expression : a+a*a

Accepted..!!!
s1905@administrator-rusa:~/adi/compiler$ ./a.out
Enter an arithmetic expression : a+a-

Rejected..!!!
s1905@administrator-rusa:~/adi/compiler$ |

```

15.5 Result

Implemented a recursive descent parser for a given grammar.

16.1 Aim

16.2 Algorithm

- ## 16.3 Program

48

16.4 Output

```
s1905@administrator-rusa:~/adi/compiler$ gcc 16.c
s1905@administrator-rusa:~/adi/compiler$ ./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
s1905@administrator-rusa:~/adi/compiler$ |
```

16.5 Result

Successfully implemented a Shift Reduce Parser for a given language.

CYCLE 4 : Simulation of code optimization Techniques

17 Experiment 17

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.
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 expr
{
    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("\n\nEnter the maximum number of expressions : ");
    scanf("%d", &n);
    printf("\nEnter 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 them 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("\nOptimized code is : \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

```
s1905@administrator-rusa:~/adi/compiler$ gcc 17.c
s1905@administrator-rusa:~/adi/compiler$ ./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
s1905@administrator-rusa:~/adi/compiler$ |
```

17.5 Result

Successfully implemented a program to perform constant propagation.

CYCLE 5 : Implementation of Intermediate Code Generation

18 Experiment 18

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. Give more precedence to / and * when evaluating the expression
6. Give less precedence to + and -.
7. write operator operand1 operand2 result into the output file
10. Close both the files.
11. Stop

18.3 Program

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
int main()
{
    FILE *fin, *fout;
    char ch = 'Z';
    fin = fopen("input.txt", "r");
    fout = fopen("output.txt", "w");
    char buff[20];
    char signs[10], op[10];
    int signsl, opl;
    signsl = opl = 0;
    printf("op\targ1\targ2\tres\n");
    while (1)
    {
        signsl = opl = -1;
        fscanf(fin, "%s", buff);
        for (int i = 0; i < strlen(buff); i++)
        {
            if (isdigit(buff[i]))
            {
                signs[++signsl] = buff[i];
            }
            else
            {
                if (buff[i] == '/' || buff[i] == '*')
                {
                    while (opl >= 0 && (op[opl] == '/' || op[opl] == '*'))
                    {
```

```

        printf("%c\t%c\t%c\t%c\n", op[opl], signs[signsl - 1], signs[signsl],
            ch);
        signsl--;
        signs[signsl] = ch;
        opl--;
        ch--;
    }
}
else
{
    while (opl >= 0 && (op[opl] == '/' || op[opl] == '*' || op[opl] == '-' ||
        op[opl] == '+'))
    {
        printf("%c\t%c\t%c\t%c\n", op[opl], signs[signsl - 1], signs[signsl],
            ch);
        signsl--;
        signs[signsl] = ch;
        opl--;
        ch--;
    }
    op[++opl] = buff[i];
}
}
while (opl >= 0)
{
    printf("%c\t%c\t%c\t%c\n", op[opl], signs[signsl - 1], signs[signsl], ch);
    signsl--;
    signs[signsl] = ch;
    opl--;
    ch--;
}
if(!feof(fin))
    break;
}

return 0;
}

```

18.4 Output

input.txt

a+b*c

```
PS C:\Users\adith\Downloads> cat input.txt
a+b*c
PS C:\Users\adith\Downloads> gcc .\18.c
PS C:\Users\adith\Downloads> .\a.exe
op      arg1      arg2      res
*        b        c        Z
+        a        Z        Y
PS C:\Users\adith\Downloads> |
```

18.5 Result

Successfully implemented Intermediate code generation for simple expressions in c.

CYCLE 6 : Implementation of the back end of the compiler

19 Experiment 19

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

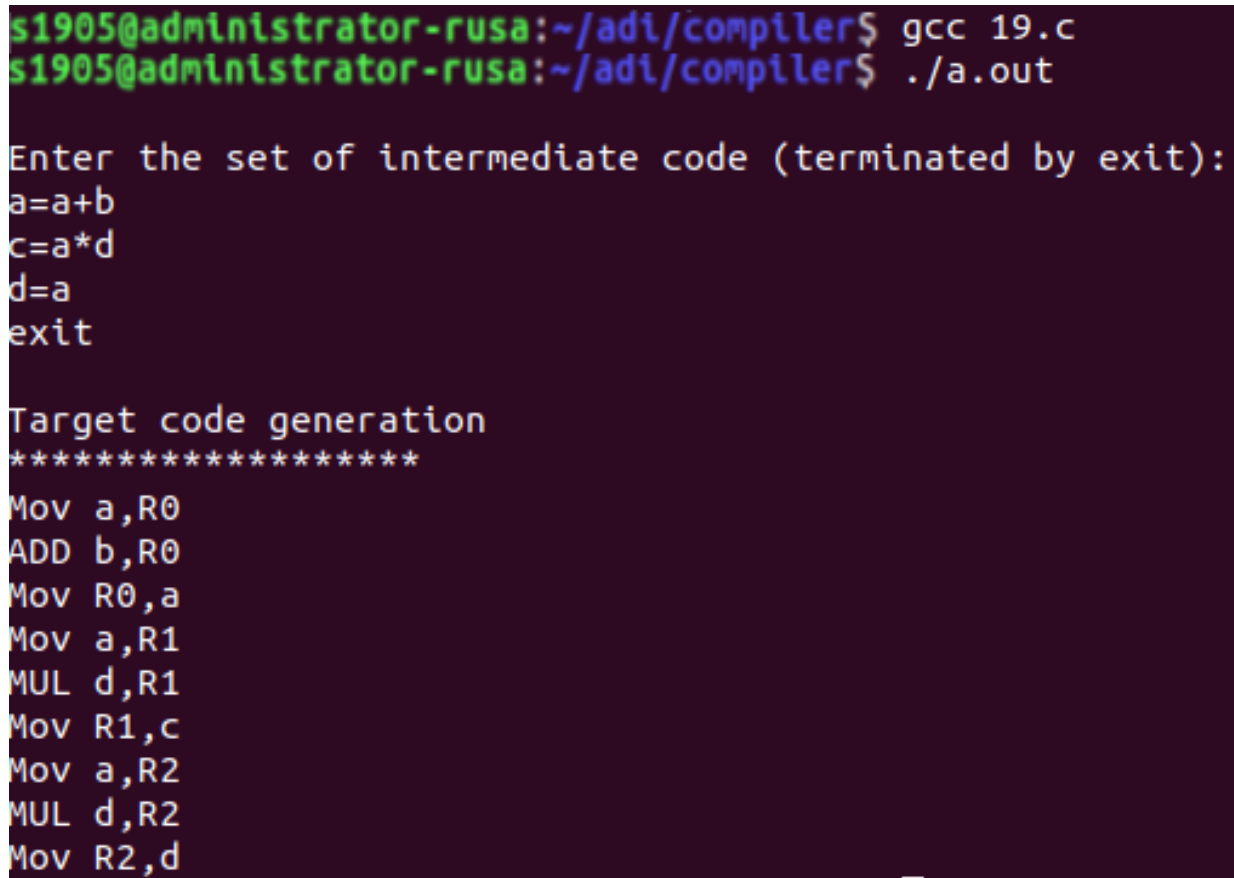
19.3 Program

```
#include<stdio.h>
#include<string.h>
void main() {
    char icode[10][30], str[20], opr[10];
    int i = 0;
    printf("\nEnter the set of intermediate code (terminated by exit):\n");
    do {
        scanf("%s", icode[i]);
    }
    while (strcmp(icode[i++], "exit") != 0);
    printf("\nTarget code generation");
    printf("\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("\n\tMov %c,R%d", str[2], i);
        printf("\n\t%s %c,R%d", opr, str[4], i);
        printf("\n\tMov R%d,%c", i, str[0]);
```

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

19.4 Output



```
s1905@administrator-rusa:~/adi/compiler$ gcc 19.c  
s1905@administrator-rusa:~/adi/compiler$ ./a.out  
  
Enter the set of intermediate code (terminated by exit):  
a=a+b  
c=a*d  
d=a  
exit  
  
Target code generation  
*****  
Mov  a,R0  
ADD  b,R0  
Mov  R0,a  
Mov  a,R1  
MUL  d,R1  
Mov  R1,c  
Mov  a,R2  
MUL  d,R2  
Mov  R2,d
```

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.