



# **ST. ANNE'S COLLEGE OF ENGINEERING AND TECHNOLOGY**

(Approved by AICTE, New Delhi. Affiliated to Anna University, Chennai)

Accredited by NAAC

ANGUCHETTYPALAYAM, PANRUTI – 607 106.

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

## **LAB MANUAL**

**CS3501-COMPILER DESIGN**

**LABORATORY**

**(Regulation 2021)**

**Year / Semester : III / V**

**PREPARED BY**

**M.SUBATHIRADEVI, AP/CSE**

## INDEX

[illegible]

[illegible]

## **CS3501 COMPILER DESIGN LABORATORY**

### **COURSE OBJECTIVES:**

- To learn the various phases of compiler.
- To learn the various parsing techniques.
- To understand intermediate code generation and run-time environment.
- To learn to implement the front-end of the compiler.
- To learn to implement code generator.
- To learn to implement code optimization.

### **LIST OF EXPERIMENTS:**

1. Using the LEX tool, Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.). Create a symbol table, while recognizing identifiers.
2. Implement a Lexical Analyzer using LEX Tool
3. Generate YACC specification for a few syntactic categories.
  - a. Program to recognize a valid arithmetic expression that uses operator +, -, \* and /.
  - b. Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
  - c. Program to recognize a valid control structures syntax of C language (For loop, while loop, if-else, if-else-if, switch-case, etc.).
  - d. Implementation of calculator using LEX and YACC
4. Generate three address code for a simple program using LEX and YACC.
5. Implement type checking using Lex and Yacc.
6. Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)
7. Implement back-end of the compiler for which the three address code is given as input and the 8086 assembly language code is produced as output.

**30 PERIODS**

### **COURSE OUTCOMES:**

On Completion of the course, the students should be able to:

CO1:Understand the techniques in different phases of a compiler.

CO2:Design a lexical analyser for a sample language and learn to use the LEX tool.

CO3:Apply different parsing algorithms to develop a parser and learn to use YACC tool

CO4:Understand semantics rules (SDT), intermediate code generation and run-time environment.

CO5:Implement code generation and apply code optimization techniques.

**Ex: No: 1a**

## **Implementation of Symbol table**

**Date:**

**Aim:**

To write a C program to implement a symbol table.

**Algorithm:**

1. Start the Program.
2. Get the input from the user with the terminating symbol „\$“.
3. Allocate memory for the variable by dynamic memory allocation function.
4. If the next character of the symbol is an operator then only the memory is allocated.
5. While reading, the input symbol is inserted into symbol table along with its memory address.
6. The steps are repeated till ”\$” is reached.
7. To reach a variable, enter the variable to the searched and symbol table has been checked for corresponding variable, the variable along its address is displayed as result.
8. Stop the program.

**Program: Filename: symbol.c**

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<string.h>
#include<math.h>
#include<ctype.h>
void main()
{
    int i=0,j=0,x=0,n,flag=0;
    void *p,*add[15];
    char ch,srch,b[15],d[15],c;
    clrscr();
    printf("Expression terminated by $:");
    while((c=getchar())!='$')
    {
        b[i]=c; i++;
    }
    n=i-1;
    printf("Given Expression:");
    i=0;
    while(i<=n)
    {
        printf("%c",b[i]); i++;
    }
    printf("\nSYMBOL TABLE\n");
    printf("\nsymbol\taddr\ttype\n");
    while(j<=n)
    {
        c=b[j];
        if(isalpha(toascii(c)))
        {
            if(j==n)
            {
                p=malloc(c);
                add[x]=p;
                d[x]=c;
                printf("%c\t%d\tidentifier\n",c,p);
            }
            else
            {
                ch=b[j+1];
```

```

        if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
        {
            p=malloc(c);
            add[x]=p;
            d[x]=c;
            printf("%c\t%d\tidentifier\n",c,p);
            x++;
        }
    }
    j++;
}
printf("\nEnter the symbol to be searched\n");
srch=getch();
printf("\nThe Symbol forsearching is:%c\n",srch);
for(i=0;i<=x;i++)
{
    if(srch==d[i])
    {
        printf("symbol found\n");
        printf("%c\t%s\t%d\n",srch,"@address",add[i]);
        flag=1;
    }
}
if(flag==0)
printf("symbol not found\n");
getch();
}

```

Output:

```
≡ File Edit Search Run Compile Debug Project Options Window Help
[ ] Output 2=[ ]
Expression terminated by $:a=b+c+d$
Given Expression:a=b+c+d
SYMBOL TABLE

symbol  addr    type
a       1932    identifier
b       2034    identifier
c       2136    identifier
d       2240    identifier

Enter the symbol to be searched

The Symbol forsearching is:a
symbol found
a       @address    1932

F1 Help  ↑↓↔ Scroll
```

```
≡ File Edit Search Run Compile Debug Project Options Window Help
[ ] Output 2=[ ]
Expression terminated by $:a=b+c+d$
Given Expression:a=b+c+d
SYMBOL TABLE

symbol  addr    type
a       1932    identifier
b       2034    identifier
c       2136    identifier
d       2240    identifier

Enter the symbol to be searched

The Symbol forsearching is:e
symbol not found
-

F1 Help  ↑↓↔ Scroll
```

**Result**      Thus the C program to implement the symbol table was executed and the output is verified.

**Ex: No: 1b      DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C**

**Aim:**

To write a C program to develop a lexical analyzer to recognize a few patterns.

**Algorithm:**

1. Start the program
2. Include the header files.
3. Allocate memory for the variable by dynamic memory allocation function.
4. Use the file accessing functions to read the file.
5. Get the input file from the user.
6. Separate all the file contents as tokens and match it with the functions.
7. Define all the keywords in a separate file and name it as key.c
8. Define all the operators in a separate file and name it as oper.c
9. Give the input program in a file and name it as input.c
10. Finally print the output after recognizing all the tokens.
11. Stop the program.

### **Program: The main Program Pattern.c**

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
void main()
{
    FILE *fi,*fo,*fop,*fk;
    int flag=0,i=1;
    char c,t,a[15],ch[15],file[20];
    clrscr();
    printf("\n Enter the File Name:");
    scanf("%s",&file);
    fi=fopen(file,"r");
    fo=fopen("inter.c","w");
    fop=fopen("oper.c","r");
    fk=fopen("key.c","r");
    c=getc(fi);
    while(!feof(fi))
    {
        if(isalpha(c)||isdigit(c)||(c=='['||c==']'||c=='.'==1))
            fputc(c,fo);
        else
        {
            if(c=='\n')
                fprintf(fo,"\t$\t");
            else
                fprintf(fo,"\t%c\t",c);
        }
        c=getc(fi);
    }
    fclose(fi);
    fclose(fo);
    fi=fopen("inter.c","r");
    printf("\n Lexical Analysis");
    fscanf(fi,"%s",a);
    printf("\n Line: %d\n",i++);
    while(!feof(fi))
    {
        if(strcmp(a,"$")==0)
        {
            printf("\n Line: %d \n",i++);
            fscanf(fi,"%s",a);
        }
        fscanf(fop,"%s",ch);
        while(!feof(fop))
        {
            if(strcmp(ch,a)==0)
            {
```

```

        fscanf(fop,"%s",ch);
        printf("\t\t%s\t\t%s\n",a,ch);
        flag=1;
    }
    fscanf(fop,"%s",ch);
}
rewind(fop);
fscanf(fk,"%s",ch);
while(!feof(fk))
{
    if(strcmp(ch,a)==0)
    {
        fscanf(fk,"%k",ch);
        printf("\t\t%s\t\tKeyword\n",a);
        flag=1;
    }
    fscanf(fk,"%s",ch);
}
rewind(fk);
if(flag==0)
{
    if(isdigit(a[0]))
        printf("\t\t%s\t\tConstant\n",a);
    else
        printf("\t\t%s\t\tIdentifier\n",a);
}
flag=0;
fscanf(fi,"%s",a);
}
getch();
}

```

### **The Keywords are stored in separate key.c**

```

int
void
main
char
if
for
while
else
printf
scanf
FILE
include
stdio.h
conio.h
iostream.h

```

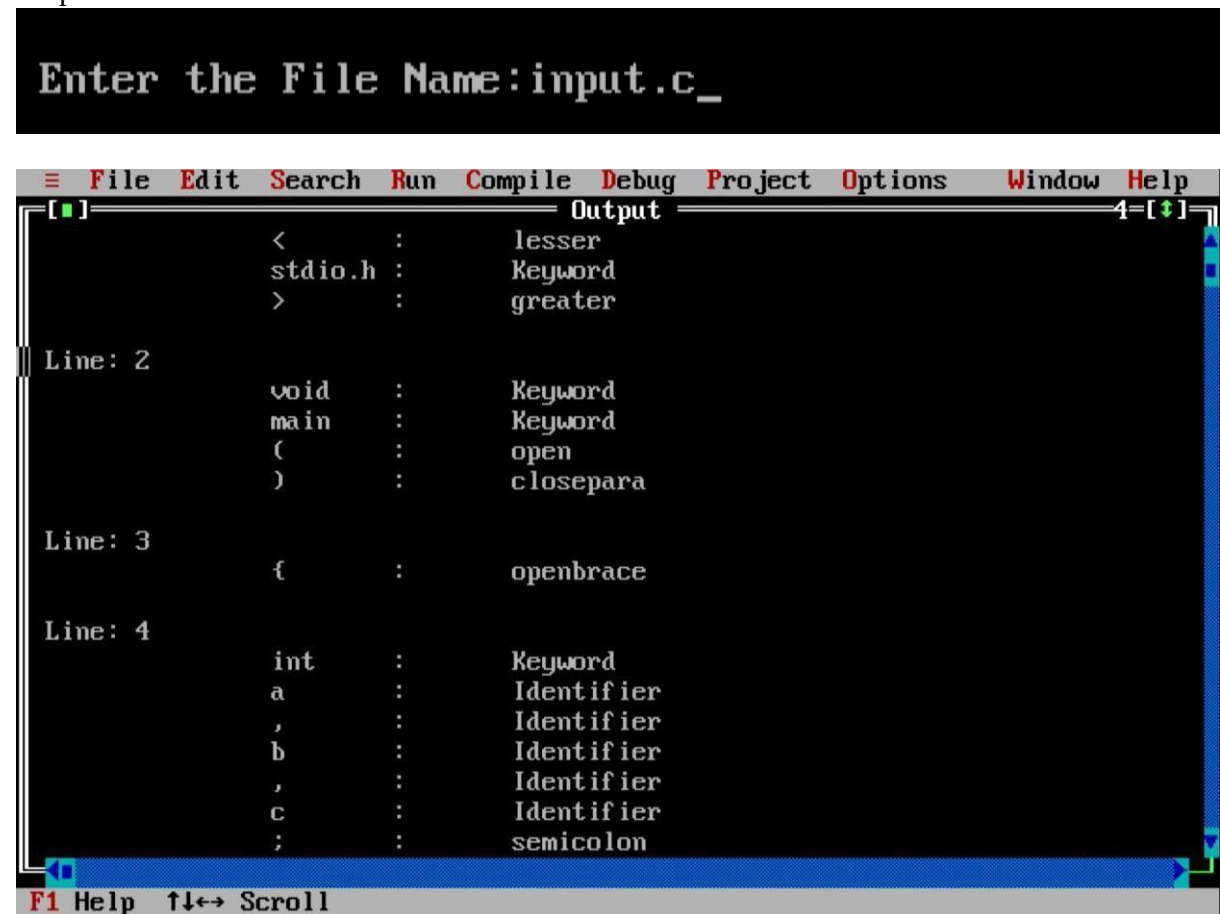
### **The Operators are stored in oper.c**

( openpara  
) closepara  
{ openbrace  
} closebrace  
< lesser  
> greater  
" doublequote  
' singlequote  
: colon  
; semicolon  
# preprocessor  
= equal  
== assign  
% percentage  
^ bitwise  
& reference  
\* star  
+ add  
- sub  
\ backslash  
/ slash

### **The file that should be processed is input.c**

```
#include <stdio.h>
void main()
{
int a,b,c;
}
```

Output:



```
Enter the File Name:input.c_

Output
Line: 1
<      : lesser
stdio.h : Keyword
>      : greater

Line: 2
void    : Keyword
main    : Keyword
(       : open
)       : closepara

Line: 3
{       : openbrace

Line: 4
int     : Keyword
a       : Identifier
,       : Identifier
b       : Identifier
,       : Identifier
c       : Identifier
;       : semicolon

F1 Help  ↑↓↔ Scroll
```

**Result:**

Thus the above program for developing the lexical the lexical analyzer and recognizing the few patterns in C is executed successfully and the output is verified.

**Ex: No: 2**  
**Date:**

## **Implementation of Lexical Analysis Using LEX Tool**

### **Aim**

To write a program to implement the function of Lexical Analyzer using LEX tool

### **Algorithm**

1. Declare and initialize any variable needed in the LEX program.
2. Set the rule for recognizing the Identifier.
3. Begin the Definition Part of the LEX Program with%%
4. Define the action to be performed when Pre-processor Directives appear beginning with a # symbol.
5. Define the action to be performed when any of the Keywords Appear in the input program (the set of the keywords can be of user's choice).
6. Define the action whenever comments appear opening with /\* and end with \*/.
7. Whenever an identifier followed by a '(' appears in the program, recognize the Token as function Declaration.
8. Recognize the beginning and ending of a block when '{' and '}' appear in the Input program.
9. Similarly recognize the strings, Numbers, Assignment Operator, Relational operators and define the actions to be taken thereafter.
10. Open the input file given by the user in the read mode.
11. Invoke the LEX Compiler. This produces a LEX.yy.c, which is a C Lexical Analyzer program

## Program: lexp2.l

```
/* program name is lexp2.l */
%{
/* program to recognize a c program */
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#. * { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
main |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
getch {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
"*/" {COMMENT = 0;}
{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n 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\t%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);}
\+ |
\ - |
\* |
```

```

\\ {if(!COMMENT) printf("\n\t%s is a arithmetic operator",yytext); }
%%
int main(int argc,char **argv)
{
if(argc>1)
{
FILE *file;
file=fopen(argv[1],"r");
if(!file)
{
printf("\n Could not open the file:%s",argv[1]);
exit(0);
}
yyin=file;
}
yylex();
printf("\n\n");
return 0;
}
int yywrap()
{
return 0;
}

```

### **Input File:test.c**

```

#include<stdio.h>
main()
{
int fact=1,n;
for(int i=1;i<=n;i++)
{ fact=fact*i; }
printf("Factorial Value of N is", fact);
getch();
}

```

**Output:**

```
$ flex lexp2.l
$ gcc lex.yy.c
$ ./a.out tset.c
#include<stdio.h> is a Preprocessor Directive
Function: main( )
Block Begins
int is a Keyword
fact is an Identifier
= is an Assignment Operator
1 is a Number
n is an Identifier
Function: for(
int is a Keyword
i is an Identifier
= is an Assignment Operator
1 is a Number
i is an Identifier
<= is a Relational Operator
n is an Identifier
i is an Identifier
);
Block Begins
fact is an Identifier
= is an Assignment Operator
fact is an Identifier
i is an Identifier
Block Ends
Function: printf(
"Factorial Value of N is" is a String
fact is an Identifier
);
Function: getch( );
Block Ends
```

**Result:**

Thus the LEX program was executed and the tokens of the program were classified.

**Ex: No: 3 a**  
**Date:**

**Generate YACC specification for a few syntactic categories**  
**Program to recognize a valid arithmetic expression that uses operator**

**Aim:**

To write a Yacc program to recognize a valid arithmetic expression that uses operators.

**Algorithm**

1. Start the program to recognize valid arithmetic expressions.
2. Define the pattern for digits in the lex file.
3. Assign yyval to return the token number as integer in the lex file.
4. Define the patterns for operators and expressions in the Yacc file.
5. Use yywrap() to print an error using yyerror() in the yacc file.
6. Use yyparse() to read a stream of value pairs in the yacc file.
7. Display the valid arithmetic expressions
8. Stop the program.

## **Program**

### **# lex program arith.l**

```
%{
/* This LEX program returns the tokens for the expression */
#include "y.tab.h"
%}

%%

"=" {printf("\n Operator is EQUAL");}
"+" {printf("\n Operator is PLUS");}
"-" {printf("\n Operator is MINUS");}
"/" {printf("\n Operator is DIVISION");}
"*" {printf("\n Operator is MULTIPLICATION");}

[a-zA-Z]*[0-9]* {
printf("\n Identifier is %s",yytext);
return ID;
}
return yytext[0];
\n return 0;
%%

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

## **YACC Program**

### **#YACC arith.y**

```
%{
#include
/* This YYAC program is for recognizing the Expression */
%}
%%
statement: A'='E
| E {
printf("\n Valid arithmetic expression");
$$ = $1;
};

E: E'+'ID
| E'-'ID
| E'*'ID
| E'/'ID
| ID
;
%%
extern FILE *yyin;
main()
```

```
{  
do  
{  
  yyparse();  
}while(!feof(yyin));  
}
```

```
yyerror(char*s)  
{  
}
```

**Output:**

```
$flex arith.l
$bison -dy arith.y
$gcc lex.yy.c t.tab.c
$ ./a.out
x=a+b;
Identifier is x
Operator is EQUAL
Identifier is a
Operator is PLUS
Identifier is b
```

**Result:**

The program is to recognize valid arithmetic expressions that are verified and executed successfully.

**Ex: No: 3b**

**Date:**

**Generate YACC specification for a few syntactic categories.  
Program to recognize a valid variable which starts with a letter followed  
by any number of letters or digits.**

**Aim:**

To Write a YACC program to recognize a valid variable which starts with a letter followed by any number of letters or digits..

**Algorithm:**

1. Start the program to recognize valid arithmetic expressions.
2. Define the pattern for letters and digits in the lex file.
3. Define the pattern for identifiers in the yacc file.
4. Use yywrap() to print an error using yyerror() in the yacc file.
5. Use yyparse() to read a stream of value pairs in the yacc file.
6. Print the valid identifiers using yylex().
7. Display the valid identifiers.
8. Stop the program.

**Program:**

```
#lex program ex3.l
%{
    #include "y.tab.h"
}%
%%

[a-zA-Z_][a-zA-Z_0-9]* return letter;
[0-9]          return digit;
.              return yytext[0];
\n            return 0;
%%

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

**YACC Program: ex3.y**

```
%{
    #include<stdio.h>
    int valid=1;
}%
%token digit letter
%%

start : letter s
s :    letter s
      | digit s
      |
      ;
%%

int yyerror()
{
    printf("\nIt's not an identifier!\n");
    valid=0;
    return 0;
}

int main()
{
    printf("\nEnter a name to tested for identifier ");
    yyparse();
    if(valid)
    {
        printf("\n It is an identifier!\n");
    }
}
```

## Output

```
$ flex ex3.l
$ bison -dy ex3.y
$ gcc lex.yy.c t.tab.c -w
$ ./a.out
Enter a name to tested for identifier abc
It is an identifier!
$ ./a.out
Enter a name to tested for identifier _abc
It is an identifier!
$ ./a.out
Enter a name to tested for identifier 854_a It's
not an identifier!
```

## Result:

Thus the given program to validate variables is implemented and executed successfully.

**Ex: No: 3c**

**Date:**

**Generate YACC specification for a few syntactic categories.  
Program to recognize valid control structures syntax of C language  
(For loop, while loop, if-else, if-else-if, etc.).**

**Aim:**

To write a Yacc program to validate control structures syntax.

**Algorithm:**

1. Start the program.
2. Read the statement
3. Validating the given statement according to the rule using Yacc.
4. Using the syntax rule print the result of the given syntax.
5. Stop the program.

**Program:**

```

#for.l
%{
#include<stdio.h>
#include "y.tab.h"
%}
alpha[A-Za-z]
digit[0-9]
%%
[\t\n]
for      return FOR;
{digit}+  return NUM;
{alpha}({alpha}|{digit})*  return ID;
"<="      return LE;
">="      return GE;
"=="      return EQ;
"!="      return NE;
"||"      return OR;
"&&"      return AND;
.         return yytext[0];
%%
yywrap()
{}

```

**Program Yacc**

```

#for.y
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token ID NUM FOR LE GE EQ NE OR AND
%right „=“
%left OR AND
%left „>“<“ LE GE EQ NE
%left „+““-“
%left „*“/“
%right UMINUS
%left „!“
%%
S      :ST {printf(“input accepted\n”);exit(0)};
ST     : FOR“(„E“:“E2“:“E“)” def
      ;
DEF    :“{„BODY“}“
      |E“:“
      |ST
      |
      ;
E      :ID „=“ E
      | E „+“ E

```

```

| E „-“ E
| E „*“ E
| E „/“ E
| E „<“ E
| E „>“ E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| E „+“ „+“
| E „-“ „-“
| ID
| NUM
;
E2 :E“<“E
| E „>“ E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
;
%%
main(){
    printf(“Enter the expression\n”);
    yyparse();
}
yyerror()
{
    printf(“\n Entered arithmetic expression is invalid\n”);
}

```

**Output:**

```
$ flex for.l
```

```
$ bison -dy for.y
```

```
$ gcc lex.yy.c t.tab.c
```

```
$ ./a.out
```

```
Enter the expression:
```

```
for(i=0;i<n;i--){i=i-1;}
```

```
Input accepted
```

### Program: WHILE

```
#while.l
%{
#include<stdio.h>
#include "y.tab.h"
%}
alpha[A-Za-z]
digit[0-9]
%%
[\t\n]
while return WHILE;
{digit}+ return NUM;
{alpha}({alpha}|{digit})* return ID;
"<=" return LE;
">=" return GE;
"==" return EQ;
"!=" return NE;
"||" return OR;
"&&" return AND;
. return yytext[0];
%%
yywrap()
{}
```

### Program Yacc

```
#while.y
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token ID NUM WHILE LE GE EQ NE OR AND
%right „=“
%left OR AND
%left „>“<“ LE GE EQ NE
%left „+““-“
%left „*“/“
%right UMINUS
%left „!“
%%
S :ST1 {printf(“input accepted\n”);exit(0)};
ST1 : WHILE “(„E2““ „(„ ST““
    ;
ST : ST ST
    | E“:“
    ;
E : ID „=“ E
    | E „+“ E
    | E „-“ E
    | E „*“ E
```

```

| E „/“ E
| E „<“ E
| E „>“ E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| E „+“ „+“
| E „-“ „-“
| ID
| NUM
;
E2 :E“<“E
| E „>“ E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| ID
| NUM
;
%%
main(){
    printf(“Enter the expression\n”);
    yyparse();
}
yyerror()
{
    printf(“\n Entered arithmetic expression is invalid\n”);
}

```

**Output:**

```
$ flex while.l
$ bison -dy while.y
$ gcc lex.yy.c t.tab.c
$ ./a.out
Enter the expression:
while(a>1){b=1;}
Input accepted
```

**Program: IF THEN ELSE**

```

#ifelse.l
%{
#include<stdio.h>
#include "y.tab.h"
%}
alpha[A-Za-z]
digit[0-9]
%%
[\t\n]
if      return IF;
then    return THEN;
else    return ELSE;
{digit}+ return NUM;
{alpha}({alpha}|{digit})* return ID;
"<="    return LE;
">="    return GE;
"=="    return EQ;
"!="    return NE;
"||"    return OR;
"&&"    return AND;
.        return yytext[0];
%%
yywrap()
{}

```

**Program Yacc**

```

#ifelse.y
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token ID NUM IF THEN LE GE EQ NE OR AND ELSE
%right „=“
%left OR AND
%left „>“<“ LE GE EQ NE
%left „+““-“,
%left „*““/“
%right UMINUS
%left „!“
%%
S      : ST {printf(“input accepted\n”);exit(0)};
ST     : IF „(, E2 „)“ THEN ST1 „,“ ELSE ST1 „,“
        | IF „(, E2 „)“ THEN ST1 “:“
        ;
ST1    : ST
        | E
        ;
E      : ID „=“ E

```

```

| E „+“ E
| E „-“ E
| E „*“ E
| E „/“ E
| E „<“ E
| E „>“ E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| ID
| NUM
;
E2 :E“<“E
| E „>“ E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| ID
| NUM
;
%%
main(){
    printf(“Enter the expression\n”);
    yyparse();
}
yyerror()
{
    printf(“\n Entered arithmetic expression is invalid\n”);
}

```

**Output:**

```
$ flex ifelse.l
```

```
$ bison -dy ifelse.y
```

```
$ gcc lex.yy.c t.tab.c
```

```
$ ./a.out
```

```
Enter the expression:
```

```
If(a==1)then b=1;else b=2;
```

```
Input accepted
```

### Program: ELSEIF

```
#elseif.l
%{
#include<stdio.h>
#include "y.tab.h"
%}
alpha[A-Za-z]
digit[0-9]
%%
[\t\n]
if      return IF;
then    return THEN;
else    return ELSE;
elseif  return ELSEIF;
{digit}+ return NUM;
{alpha}({alpha}|{digit})* return ID;
"<="    return LE;
">="    return GE;
"=="    return EQ;
"!="    return NE;
"||"    return OR;
"&&"    return AND;
.        return yytext[0];
%%
yywrap()
{}
```

### Program Yacc

```
#elseif.y
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token ID NUM IF THEN LE GE EQ NE OR AND ELSE
%right „=“
%left OR AND
%left „>“ „<“ LE GE EQ NE
%left „+“ „-“
%left „*“ „/“
%right UMINUS
%left „!“
%%
S      : ST {printf(“input accepted\n”);exit(0)};
ST     : IF „(, E2 „)“ THEN ST1 „,“ ELSEIF „(,E2“)“ THEN ST1 „,“ ELSE ST1 “:“
        | IF „(, E2 „)“ THEN ST1 “:“
        ;
ST1    : ST
        | E
        ;
```

```

E      : ID „=“ E
      | E „+“ E
      | E „-“ E
      | E „*“ E
      | E „/“ E
      | E „<“ E
      | E „>“ E
      | E LE E
      | E GE E
      | E EQ E
      | E NE E
      | E OR E
      | E AND E
      | ID
      | NUM
      ;

```

```

E2     :E“<“E
      | E „>“ E
      | E LE E
      | E GE E
      | E EQ E
      | E NE E
      | E OR E
      | E AND E
      | ID
      | NUM
      ;

```

```
%%
```

```

main(){
    printf(“Enter the expression\n”);
    yyparse();
}
yyerror()
{
    printf(“\n Entered arithmetic expression is invalid\n”);
}

```

**Output:**

```
$ flex elseif.l  
$ bison -dy elseif.y  
$ gcc lex.yy.c t.tab.c -w  
$ ./a.out  
Enter the expression:  
if(a<p)then b=1;elseif(a>c)then b=2;else b=1;  
Input accepted
```

**Result:**

Thus the given program to recognize control structures is executed successfully.

**Ex: No: 3d**  
**Date:**

## **Implementation of calculator using LEX and YACC**

### **Aim**

To write a lex and yacc program to implement a calculator.

### **Algorithm**

1. Start the program
2. In the declaration part of lex, includes declaration of regular definitions as digit.
3. In the translation rules part of lex, specifies the pattern and its action that is to be executed whenever a lexeme matched by pattern is found in the input in the calc.l
4. By use of Yacc program all the arithmetic operations are done such as +,-,\*,/.
5. Display error is persisting.
6. Verify the output.
7. Stop the Program.

**Program:**

```
#calc.l
%{
#include<stdio.h>
#include "y.tab.h"
extern int yyval;
}%
%%
[0-9]+{yyval=atoi(yytext);return NUMBER;}
[\n] return 0;
[\t];
. return yytext[0];
%%
yywrap()
{return 0;}
```

## Yacc Program

```
#calc.y
%{
#include<stdio.h>
Int yylex(void);
}%
%token NAME NUMBER
%left GE LE NE EQ „<“>““%/“
%left „-“ „+“
%left „*“ „/“
%nonassoc UMINUS
%%
Statement :NAME „=“ exp
           |exp {printf(“=%d\n”, $1);}
           ;
exp : NUMBER {$$ == $1;}
    | exp „+“ exp {$$ = $1 + $3;}
    | exp „-“ exp {$$ = $1 - $3;}
    | exp „*“ exp {$$ = $1 * $3;}
    | exp „/“ exp {$$ = $1 / $3;}
    | exp „-“ exp %prec UMINUS {$$ = -$2;}
    | „(„ exp „)“ {$$ = $2;}
    ;
%%
main()
{
printf(“Enter the expression.”);
yyvsparse();
}
yyerror()
{
printf(“\n Error Occurred\n”);
}
```

**Output:**

```
$ flex calc.l
$ bison -dy calc.y
$ gcc lex.yy.c t.tab.c -w
$ ./a.out
Enter the expression:2+2
4
$ ./a.out
Enter the expression:8-2
6
```

**Result:**

Thus the program to implement a calculator using lex is executed successfully.

**Ex: No: 4**  
**Date:**

### **Generate three address codes for a simple program**

#### **Aim**

To write a program to implement the three address code for the given expression.

#### **Algorithm**

1. Start a program.
2. Include all the header files.
3. Initialize all the functions and variables.
4. Get the expression from the user and separate into the tokens.
5. After separation, specify the identifiers, operators and number.
6. Print the output.
7. Stop the program

```
#include<stdio.h>
#include<string.h>
void pm();
void plus();
void div();
int i,ch,j,l;
char ex[10],ex1[10],exp1[10],ex2[10];
main()
{
    while(1)
    {
        printf("\n 1.Assignment\n 2.Arithmetic\n 3.exit \n ENTER THE CHOICE:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("\n Enter the expression with assignment operator:");
                scanf("%s",ex1);
                l=strlen(ex1);
                ex2[0]='\0';
                i=0;
                while(ex1[i]!='=')
                {
                    i++;
                }
                strncat(ex2,ex1,i);
                strev(ex1);
                exp1[0]='\0';
                strncat(exp1,ex1,l-(i+1));
                strev(exp1);
                printf("\n 3 address code:\n temp=%s \n %s=temp\n",exp1,ex2);
                break;
            case 2:
                printf("\n Enter the expression with arithmetic operator:");
                scanf("%s",ex);
                strcpy(ex1,ex);
                l=strlen(ex1);
                exp1[0]='\0';
                for(i=0;i<l;i++)
                {
                    if(ex1[i]=='+'||ex1[i]=='-')
                    {
                        if(ex1[i+2]=='/'||ex1[i+2]=='*')
                        {
                            pm();
                            break;
                        }
                    }
                    else
                }
            }
        }
    }
}
```

```

        {
            plus();
            break;
        }
    }
    else if(ex1[i]=='/'||ex1[i]=='*')
    {
        div();
        break;
    }
}
break;
case 3:
    exit(0);
}
}
}
void pm()
{
    strev(exp1);
    j=l-i-1;
    strncat(exp1,ex1,j);
    strev(exp1);
    printf("3 address code:\n temp=%s\n temp1=%c*%ctemp\n",exp1,ex1[j+2],ex1[j]);
}
void div()
{
    strncat(exp1,ex1,i+2);
    printf("3 address code:\n temp=%s\n temp1=temp%c+%c\n",exp1,ex1[l+2],ex1[i+3]);
}
void plus()
{
    strncat(exp1,ex1,i+2);
    printf("3 address code:\n temp=%s\n temp1=temp%c+%c\n",exp1,ex1[l+2],ex1[i+3]);
}

```

## Output:

```
C:\TURBOC3\BIN>TC
```

```
1.Assignment
2.Arithmetic
3.exit
ENTER THE CHOICE:1
```

```
Enter the expression with assignment operator:a=b
3 address code:
temp=b
a=temp
```

```
1.Assignment
2.Arithmetic
3.exit
ENTER THE CHOICE:_
```

```
1.Assignment
2.Arithmetic
3.exit
ENTER THE CHOICE:2
```

```
Enter the expression with arithmetic operator:a*b+c
3 address code:
temp=a*b
temp1=temp +c
```

```
1.Assignment
2.Arithmetic
3.exit
ENTER THE CHOICE:3_
```

## Result:

Thus the three address code is implemented and executed successfully.

**Ex: No: 4b**  
**Date:**

**Generate three address code for a simple program using  
LEX and YACC**

### **Aim**

Write a program to implement the three address code for the given expression using LEX and YACC.

### **Algorithm**

1. Start a program.
2. Include all the header files.
3. Initialize all the functions and variables.
4. Get the expression from the user and separate into the tokens.
5. After separation, specify the identifiers, operators and number.
6. Print the output.
7. Stop the program

### **Program lex: tac.l**

```
%{
#include"y.tab.h"
extern char yyval;
%}

%%

[0-9]+ {yyval.symbol=(char)(yytext[0]);return NUMBER;}
[a-z] {yyval.symbol= (char)(yytext[0]);return LETTER;}
. {return yytext[0];}
\n {return 0;}
%%
```

### **Program Yacc: tac.y**

```
%{
#include"y.tab.h"
#include<stdio.h>
char addtotable(char,char,char);
int index1=0;
char temp = 'A'-1;
struct expr{
char operand1;
char operand2;
char operator;
char result;
};
%}

%union{ char symbol; }
%left '+' '-'
%left '/' '*'
%token <symbol> LETTER NUMBER
%type <symbol> exp
%%
statement: LETTER '=' exp ';' {addtotable((char)$1,(char)$3,'=');};
exp: exp '+' exp {$$ = addtotable((char)$1,(char)$3,'+');}
    |exp '-' exp {$$ = addtotable((char)$1,(char)$3,'-');}
    |exp '/' exp {$$ = addtotable((char)$1,(char)$3,'/');}
    |exp '*' exp {$$ = addtotable((char)$1,(char)$3,'*');}
    | '(' exp ')' {$$ = (char)$2;}
    |NUMBER {$$ = (char)$1;}
    |LETTER {(char)$1;};
%%
struct expr arr[20];
void yyerror(char *s){
    printf("Error %s",s);
}
```

```

char addtotable(char a, char b, char o){
    temp++;
    arr[index1].operand1 =a;
    arr[index1].operand2 = b;
    arr[index1].operator = o;
    arr[index1].result=temp;
    index1++;
    return temp;
}
void threeAdd(){

    int i=0;
    char temp='A';
    while(i<index1){
        printf("%c:=\t",arr[i].result);
        printf("%c\t",arr[i].operand1);
        printf("%c\t",arr[i].operator);
        printf("%c\t",arr[i].operand2);
        i++;
        temp++;
        printf("\n");
    }
}

void fouradd(){
    int i=0;
    char temp='A';
    while(i<index1){
        printf("%c\t",arr[i].operator);
        printf("%c\t",arr[i].operand1);
        printf("%c\t",arr[i].operand2);
        printf("%c",arr[i].result);
        i++;
        temp++;
        printf("\n");
    }
}

int find(char l){
    int i;
    for(i=0;i<index1;i++)
        if(arr[i].result==l) break;
    return i;
}

void triple(){
    int i=0;
    char temp='A';
    while(i<index1){
        printf("%c\t",arr[i].operator);
        if(!isupper(arr[i].operand1))
            printf("%c\t",arr[i].operand1);
    }
}

```

```
        else{
            printf("pointer");
            printf("%d\t",find(arr[i].operand1));
        }
        if(!isupper(arr[i].operand2))
            printf("%c\t",arr[i].operand2);
        else{
            printf("pointer");
            printf("%d\t",find(arr[i].operand2));
        }
        i++;
        temp++;
        printf("\n");
    }
}
int yywrap(){
    return 1;
}
int main(){
    printf("Enter the expression: ");
    yyparse();
    threeAdd();
    printf("\n");
    fouradd();
    printf("\n");
    triple();
    return 0;
}
```

**Output:**

```
$flex tac.l
$bison -d tac.y
$ gcc y.tab.c lex.yy.c -w
$ ./a.out
Enter the expression: a=b*c+1/3-5*f;
A:= b * c
B:= 1 / 3
C:= A + B
D:= 5 * f
E:= C - D
F:= a = E

* b c A
/ 1 3 B
+ A B C
* 5 f D
- C D E
= a E F

* b c
/ 1 3
+ pointer0 pointer1
* 5 f
- pointer2 pointer3
= a pointer4
```

**Result:**

Thus the three address code is implemented and executed successfully.

**Ex No: 5**  
**Date:**

## **Implementation of Type Checking**

**Aim:**

To write a C program for implementing type checking for a given expression

**Algorithm:**

1. Start the program.
2. Include all the header files.
3. Initialize all the functions and variables.
4. Get the expression from the user and separate into tokens.
5. After separation, specify the identifiers, operators and numbers.
6. Print the output.
7. Stop the program.

## Program

```
#include<stdio.h>
#include<conio.h>
char str[50],opstr[75];
int f[2][9]={2,3,4,4,4,0,6,6,0,1,1,3,3,5,5,0,5,0};
int col,col1,col2; char c;
swt()
{
    switch(c)
    {
        case'+':
            col=0;
            break;
        case'-':
            col=1;
            break;
        case'*':
            col=2;
            break;
        case'/':
            col=3;
            break;
        case'^':
            col=4;
            break;
        case(':':
            col=5;
            break;
        case')':
            col=6;
            break;
        case'd':
            col=7;
            break;
        case'$':
            col=8;
            break;
        default:
            printf("\nTERMINAL MISSMATCH\n");
            exit(1);
    }
    // return 0;
}
main()
{
    int i=0,j=0,col1,cn,k=0;
    int t1=0,foundg=0;
    char temp[20];
    clrscr();
```

```

printf("\nEnter arithmetic expression:");
scanf("%s",&str);
while(str[i]!='\0')
    i++;
    str[i]='$';
    str[++i]='\0';
    printf("%s\n",str);
    come:
        i=0;
        opstr[0]='$'; j=1;
        c='$';
        swt();
        col1=col;
        c=str[i];
        swt();
        col2=col;
        if(f[1][col1]>f[2][col2])
        {
            opstr[j]='>'; j++;
        }
        else if(f[1][col1]<f[2][col2])
        {
        }
        else
        {
    }
    opstr[j]='<'; j++;
    opstr[j]='='; j++;
    while(str[i]!='$')
    {
        c=str[i];
        swt();
        col1=col;
        c=str[++i];
        swt();
        col2=col;
        opstr[j]=str[--i];
        j++;
        if(f[0][col1]>f[1][col2])
        {
            opstr[j]='>'; j++;
        }
        else if(f[0][col1]<f[1][col2])
        {
            opstr[j]='<'; j++;
        }
        else
        {
            opstr[j]='='; j++;
        }
    }

```

```

        i++;
    }
    opstr[j]='$';
    opstr[++j]='\0';
    printf("\nPrecedence Input:%s\n",opstr); i=0;
    j=0;
    while(opstr[i]!='\0')
    {
        foundg=0;
        while(foundg!=1)
        {
            if(opstr[i]=='\0')
                goto redone;
            if(opstr[i]=='>')
                foundg=1;

            t1=i;
            i++;
        }
        if(foundg==1)
            for(i=t1;i>0;i--)
                if(opstr[i]=='<')
                    break;
                if(i==0)
                {
                    printf("\nERROR\n");
                    exit(1);
                }
            cn=i; j=0; i=t1+1;
            while(opstr[i]!='\0')
            {
                temp[j]=opstr[i]; j++;i++;
            }
            temp[j]='\0';
            opstr[cn]='E';
            opstr[++cn]='\0';
            strcat(opstr,temp);
            printf("\n%s",opstr);
            i=1;
        }
    }
    redone:k=0;
    while(opstr[k]!='\0')
    {
        k++;
        if(opstr[k]=='<')
        {
            printf("\nError");
            exit(1);
        }
    }
}
if((opstr[0]=='$')&&(opstr[2]=='$'))

```

```
goto sue;
i=1
while(opstr[i]!='\0')
{
    c=opstr[i];
    if(c=='+'||c=='*'||c=='/'||c=='$')
    {
        temp[j]=c;
        j++;
    }
    i++;
}
temp[j]='\0';
strcpy(str,temp);
goto come;
sue:
printf("\n success");
return 0;
}
```

## Output

```
Enter arithmetic expression:(d*d)+d$  
(d*d)+d$$
```

```
Precedence Input:$<=(<d>*<d>)>+<d>$
```

```
$<=(E*<d>)>+<d>$
```

```
$<=(E*E)>+<d>$
```

```
$E+<d>$
```

```
$E+E$
```

```
Precedence Input:$<=$
```

```
Error_
```

```
Enter arithmetic expression:(d*d)$  
(d*d)$$
```

```
Precedence Input:$<=(<d>*<d>)>$
```

```
$<=(E*<d>)>$
```

```
$<=(E*E)>$
```

```
$E$
```

```
success_
```

## Result:

Thus the input is checked with the syntax and semantics and the output is verified successfully.

**Ex: No: 6**  
**Date:**

## **Implementation of Simple Code Optimization**

**Aim:**

To write a program using C for implementing Code Optimization Technique.

**Algorithm:**

1. Generate the program for factorial program using for and do-while loop to specify optimization technique.
2. In for loop variable initialization is activated first and the condition is checked next. If the condition is true the corresponding statements are executed and specified increment / decrement operation is performed.
3. The for loop operation is activated till the condition failure.
4. In do-while loop the variable is initialized and the statements are executed then the condition checking and increment / decrement operation is performed.
5. When comparing both for and do-while loop for optimization do while is best because first the statement execution is done then only the condition is checked.
6. During the statement execution itself we can find the inconvenience of the result and no need to wait for the specified condition result.
7. Finally when considering Code Optimization in loop do-while best with is respect to performance.

**Program:**

```
#include<stdio.h>
#include<string.h>
struct op
{
    char l;
    char r[20];
}
op[10],pr[10];
void main()
{
    int a,i,k,j,n,z=0,m,q;
    char *p,*l;
    char temp,t;
    char *tem;
    printf("Enter the Number of Values:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("left: ");
        scanf(" %c",&op[i].l);
        printf("right: ");
        scanf(" %s",&op[i].r);
    }
    printf("Intermediate Code\n");
    for(i=0;i<n;i++)
    {
        printf("%c=",op[i].l);
        printf("%s\n",op[i].r);
    }
    for(i=0;i<n-1;i++)
    {
        temp=op[i].l;
        for(j=0;j<n;j++)
        {
            p=strchr(op[j].r,temp);
            if(p)
            {
                pr[z].l=op[i].l;
                strcpy(pr[z].r,op[i].r);
                z++;
            }
        }
        pr[z].l=op[n-1].l;
        strcpy(pr[z].r,op[n-1].r);
        z++;
        printf("\nAfter Dead Code Elimination\n");
        for(k=0;k<z;k++)
        {
            printf("%c\t=",pr[k].l);
```

```

        printf("%s\n",pr[k].r);
    }
    for(m=0;m<z;m++)
    {
        tem=pr[m].r;
        for(j=m+1;j<z;j++)
        {
            p=strstr(tem,pr[j].r);
            if(p)
            {
                t=pr[j].l;
                pr[j].l=pr[m].l;
                for(i=0;i<z;i++)
                {
                    l=strchr(pr[i].r,t);
                    if(l)
                    {
                        a=l-pr[i].r;
                        printf("pos: %d\n",a);
                        pr[i].r[a]=pr[m].l;
                    }
                }
            }
        }
    }
    }printf("Eliminate Common Expression\n");
    for(i=0;i<z;i++)
    {
        printf("%c\t=",pr[i].l);
        printf("%s\n",pr[i].r);
    }
    for(i=0;i<z;i++)
    {
        for(j=i+1;j<z;j++)
        {
            q=strcmp(pr[i].r,pr[j].r);
            if((pr[i].l==pr[j].l)&&!q)
            {
                pr[i].l='\0';
            }
        }
    }
    }printf("Optimized Code\n");
    for(i=0;i<z;i++)
    {
        if(pr[i].l!='\0')
        {
            printf("%c=",pr[i].l);
            printf("%s\n",pr[i].r);
        }
    }
}

```

Output:

```
C:\TURBOC3\BIN>TC
Enter the Number of Values:5
left: a
right: 9
left: b
right: c+d
left: e
right: c+d
left: f
right: b+e
left: r
right: f_
```

```
File Edit Search Run Compile Debug Project Options Window Help
[ ] Output 2-[ ]
Intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f
```

```
After Dead Code Elimination
b      =c+d
e      =c+d
f      =b+e
r      =f
pos: 2
Eliminate Common Expression
b      =c+d
b      =c+d
f      =b+b
r      =f
Optimized Code
b=c+d
f=b+b
r=f
```

**Result:**

Thus the Simple code Optimization techniques are implemented and the output is verified successfully.

**Ex No: 7**

## **Implementing the Back end of the Compiler**

**Date:**

**Aim:**

To implement the back end of a compiler which takes the Three Address code and produce the 8086 Assembly language instructions Using a C program.

**Algorithm:**

1. Start the program
2. Include the necessary header files.
3. Declare the necessary character array for input and output and also a structure to include it.
4. Get the Intermediate Code (Three Address Code) as the input.
5. Display the options to do various operations and use a switch case to implement that operation.
6. Stop the program.

**Program:**

```
#include<stdio.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    char icode[10][30],str[20],opr[10];
    int i=0;
    clrscr();
    printf("\n Enter the set of intermediate code (terminated by exit):\n");
    do
    {
        scanf("%s",icode[i]);
    } while(strcmp(icode[i++],"exit")!=0);
    printf("\n target code generation");
    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);
    getch();
}
```

**Output:**

```
Enter the set of intermediate code (terminated by exit):  
t=2/3  
c=4/5  
a=2*c  
exit
```

```
target code generation
```

```
    Mov 2,R0
```

```
    DIV3,R0
```

```
    Mov R0,t
```

```
    Mov 4,R1
```

```
    DIV5,R1
```

```
    Mov R1,c
```

```
    Mov 2,R2
```

```
    MULc,R2
```

```
    Mov R2,a_
```

**Result**

Thus the program was implemented a Target code from a Three Address Code and successfully verified the output.