



Stay Ahead

BANNARI AMMAN INSTITUTE OF TECHNOLOGY

An Autonomous Institution Affiliated to Anna University Chennai - Approved by AICTE - Accredited by NAAC with "A+" Grade

SATHYAMANGALAM - 638 401 ERODE DISTRICT TAMIL NADU INDIA

Ph: 04295-226000 / 221289 Fax: 04295-226666 E-mail: stayahead@bitsathy.ac.in Web: www.bitsathy.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

22CB501 – COMPILER DESIGN

REGULATION 2022

INDEX

EX.NO	NAME OF THE EXPERIMENT	PAGE NO.
1	DEVELOP LEXICAL ANALYZER AIMS TO BREAK DOWN THE ADDITION OF N NUMBERS SOURCE CODE INTO MEANINGFUL UNITS (TOKENS) LIKE IDENTIFIERS, CONSTANTS, OPERATORS, AND KEYWORDS, SKIPPING WHITESPACE AND COMMENTS, TO PREPARE THE CODE FOR LATER STAGES.	4
2	IMPLEMENT LEX PROGRAMS FOR THE FOLLOWING: A. COUNT THE NUMBER OF CHARACTERS, WORDS, SPACES AND LINES IN A GIVEN SENTENCE. B. CHECK VALID MOBILE NUMBER C. ACCEPT VALID EMAIL	10
3	IMPLEMENT A PROGRAM FOR SYNTAX CHECKING USING LEX AND YACC. a. CHECKING FOR LOOPING STATEMENTS USED IN SORTING ALGORITHMS (MERGE SORT, INSERTION SORT ETC...) b. CHECKING FOR CONTROL STATEMENTS USED IN SORTING ALGORITHMS (MERGE SORT, INSERTION SORT ETC...)	14
4	IMPLEMENT A PROGRAM FOR SYNTAX CHECKING USING LEX AND YACC a. VARIABLE DECLARATIONS FOR PERFORMING SORTING OPERATION USED IN SORTING ALGORITHMS (MERGE SORT, INSERTION SORT, QUICK SORT ETC...) b. FUNCTIONS CREATED FOR PERFORMING SORTING OPERATION USED IN SORTING ALGORITHMS (MERGE SORT, INSERTION SORT, QUICK SORT ETC...)	22
5	DEVELOP A DESK CALCULATOR PROGRAM USING LEX AND YACC TO EVALUATE ARITHMETIC EXPRESSIONS. THE CALCULATOR SHOULD SUPPORT THE ARITHMETIC OPERATIONS AND ERROR HANDLING	29
6	DEVELOP A PROGRAM USING LEX AND YACC TO GENERATE THREE ADDRESS CODE (TAC) FOR A QUICK SORT FUNCTION IN A CUSTOM-DEFINED	32

	PROGRAMMING LANGUAGE. THE TAC SHOULD REPRESENT INTERMEDIATE CODE THAT CAN BE USED FOR FURTHER OPTIMIZATIONS OR TRANSLATION INTO MACHINE CODE.	
7	DEVELOP A PROGRAM TO IMPLEMENT AND DEMONSTRATE VARIOUS CODE OPTIMIZATION TECHNIQUES (CONSTANT FOLDING, DEAD CODE ELIMINATION, LOOP OPTIMIZATION, STRENGTH REDUCTION) ON A QUICK SORT ALGORITHM WRITTEN IN A HIGH-LEVEL PROGRAMMING LANGUAGE.	39
8	DEVELOP A PROGRAM USING LEX AND YACC TO IMPLEMENT CODE GENERATION TECHNIQUES FOR GENERATING OPTIMIZED INTERMEDIATE CODE FOR THE QUICK SORT ALGORITHM FROM A CUSTOM-DEFINED HIGH-LEVEL LANGUAGE.	44

EX.NO:1	DEVELOP LEXICAL ANALYZER AIMS TO BREAK DOWN THE ADDITION OF N NUMBERS SOURCE CODE INTO MEANINGFUL UNITS (TOKENS) LIKE IDENTIFIERS, CONSTANTS, OPERATORS, AND KEYWORDS, SKIPPING WHITESPACE AND COMMENTS, TO PREPARE THE CODE FOR LATER STAGES.
DATE:	

AIM:

To write a C program to develop a lexical analyzer to recognize a few patterns in c. **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:

```
#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();

```

}

Key.C:

int

void

main

char

if

for

while

else

printf

scanf

FILE

include

stdio.h

conio.h

iostream.h

Oper.C:

(open para

) closepara

{ openbrace

} closebrace

< lesser

> greater

" doublequote

' singlequote

: colon

; semicolon

preprocessor

= equal

== asign

% percentage

^ bitwise
 & reference
 * star
 + add
 - sub
 \ backslash
 / slash

Input.C:

```

#include "stdio.h"
#include "conio.h"
void main()
{
int a=10,b,c;
a=b*c;
getch();
}

```

OUTPUT:

Enter the File Name: Input.C

Line: 1

: Preprocessor

include : keyword

< : lesser

stdio.h : keyword

> : greater

Line: 2

: Preprocessor

include : keyword

< : lesser

conio.h : keyword

> : greater

Line: 3

void : keyword

main : keyword

(: openpara

) : closepara

Line: 4

{ : openbrace

Line: 5

int : keyword

a : identifier

= : equal

10 : constant

, : identifier

b : identifier

, : identifier

c : identifier

; : semicolon

Line: 6

a : identifier

= : equal

b : identifier

* : star

c : identifier

; : semicolon

Line: 8

} : close brace

RESULT:

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

EX.NO:2	IMPLEMENT LEX PROGRAMS FOR THE FOLLOWING:
DATE:	

	A. COUNT THE NUMBER OF CHARACTERS, WORDS, SPACES AND LINES IN A GIVEN SENTENCE. B. CHECK VALID MOBILE NUMBER C. ACCEPT VALID EMAIL
--	---

AIM:

To write a program to implement the Lexical Analyzer using Lex Tool.

ALGORITHM:

1. Start the program.
2. Declare variables, manifest constants and regular definitions in the declaration section
3. Declare regular expression in the translation rules section.
4. Declare the main function and call yylex() function.
5. Compile the lex program using following command

```
$ flex filename.l
$ gcc lex.yy.c
$ ./a.out
```
6. Feed the input and retrieve the output.
7. Stop the program.

PROGRAM:**a. Count the number of characters, words, spaces and lines**

```
% {
#include<stdio.h>
int lc=0, sc=0, tc=0, ch=0; /*Global variables*/
% }

/*Rule Section*/

%%

\n lc++; //line counter
([ ])+ sc++; //space counter
\t tc++; //tab counter
. ch++; //characters counter
%%

int main()
```

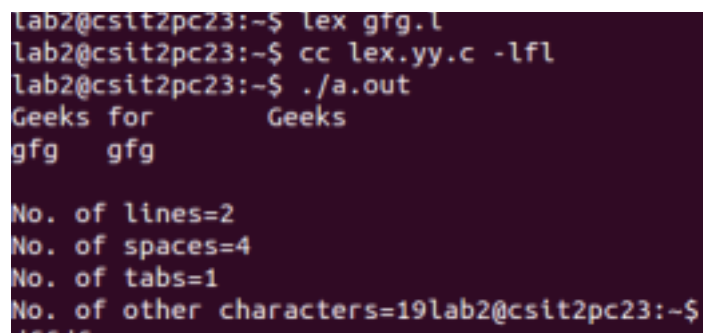
```

{
// The function that starts the analysis
yylex();

printf("\nNo. of lines=%d", lc);
printf("\nNo. of spaces=%d", sc);
printf("\nNo. of tabs=%d", tc);
printf("\nNo. of other characters=%d", ch);

}

```

OUTPUT:


```

lab2@csit2pc23:~$ lex gfg.l
lab2@csit2pc23:~$ cc lex.yy.c -lfl
lab2@csit2pc23:~$ ./a.out
Geeks for      Geeks
gfg  gfg

No. of lines=2
No. of spaces=4
No. of tabs=1
No. of other characters=19lab2@csit2pc23:~$

```

b. Check valid Mobile Number

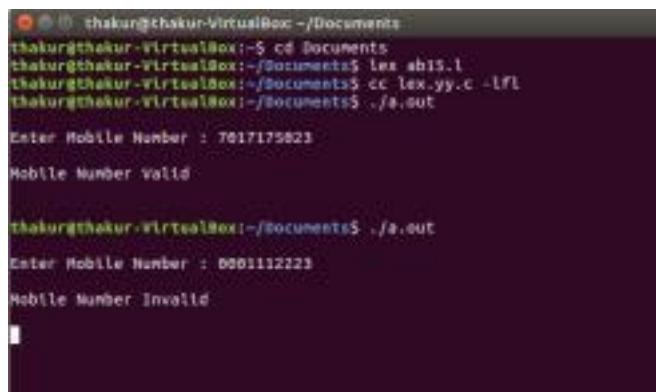
```
% {
/* Definition section */
% }

/* Rule Section */
%%

[1-9][0-9]{9} {printf("\nMobile
Number Valid\n");} .+
{printf("\nMobile Number Invalid\n");}

%%

// driver code
int main()
{
printf("\nEnter Mobile Number : ");
yylex();
printf("\n");
return 0;
}
```

OUTPUT:


```
thakur@thakur-VirtualBox: ~/Documents
thakur@thakur-VirtualBox:~$ cd Documents
thakur@thakur-VirtualBox:~/Documents$ lex abis.l
thakur@thakur-VirtualBox:~/Documents$ cc lex.yy.c -lfl
thakur@thakur-VirtualBox:~/Documents$ ./a.out

Enter Mobile Number : 7017175821

Mobile Number Valid

thakur@thakur-VirtualBox:~/Documents$ ./a.out

Enter Mobile Number : 0001112222

Mobile Number Invalid
```

c. Accept valid email

```

%
{
int flag = 0; %
} %
% [a - z.0 - 9 _] + @[a - z] + ".com" | ".in"
flag = 1; %
%
main() {
yylex();
if (flag == 1)
printf("Accepted");
else
printf("Not Accepted");
}

```

OUTPUT:

```

X _ □ Terminal File Edit View Search Terminal Help
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out
csetannayjain@gmail.com
Accepted
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out
123@33.com
Accepted
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out
123@33.com
Accepted
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out
Not Accepted
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out
abc@def.com
Not Accepted
pc@pc:/media/pc/Shared/Compiler Lab$

```

RESULT:

Thus the program is executed successfully.

EX.NO:3.a	IMPLEMENT A PROGRAM FOR SYNTAX CHECKING USING LEX AND YACC. a. CHECKING FOR LOOPING STATEMENTS USED IN SORTING ALGORITHMS (MERGE SORT, INSERTION SORT ETC...)
DATE:	

AIM:

To write Yacc specification to validate the syntax for looping statements.

ALGORITHM:

1. Start the program.
2. Declare regular expression for alphabets, digits, for loop and relational operators in lex specification.
3. Declare grammar for looping statements in the translation rules section of YACC specification.
4. Declare the main function and call yylex() function.

5. Compile the lex program using following command

```
$ flex filename.l
```

```
$ gcc lex.yy.c
```

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

6. Feed the input and retrieve the output.

7. Stop the program.

PROGRAM:

```
// Lex file: for.l
```

```
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];
```

```
%%
```

```
// Yacc file: 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
```

```
    |
```

```
;
```

```
BODY : BODY 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
 ;

%%


```
#include "lex.yy.c"

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

Output:

Enter the expression:

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

i=i+1;

Input accepted

RESULT:

Thus the program is executed successfully.

EX.NO:3.b	IMPLEMENT A PROGRAM FOR SYNTAX CHECKING USING LEX AND YACC. b. CHECKING FOR CONTROL STATEMENTS USED IN SORTING ALGORITHMS (MERGE SORT, INSERTION SORT ETC...)
DATE:	

AIM:

To write Yacc specification to validate the syntax for control statements.

ALGORITHM:

1. Start the program.
2. Declare regular expression for alphabets, digits, if-else in lex specification. 3
Declare grammar for control statements in the translation rules section of YACC specification. 4. Declare the main function and call yylex() function.

5. Compile the lex program using following command

```
$ flex filename.l
```

```
$ gcc lex.yy.c
```

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

6. Feed the input and retrieve the output.

7. Stop the program.

PROGRAM:**Program:**

(Lex Program: ift.l)

```
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];
```

```
%%
```

```
(Yacc Program: ift.y)
```

```
{
```

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

```
#include <stdlib.h>
```

```
}
```

```
%token ID NUM IF THEN LE GE EQ NE OR AND ELSE
```

```
%right '='
```

```
%left AND OR
```

```
%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
```

```

;
```

```

%%
```

```

#include "lex.yy.c"
```

```

main()
```

```

{
```

```

    printf("Enter the exp: ");
```

```

    yyparse();
```

```

}
```

Output:

```
students@cselab-desktop:~$ lex  
ift.lex students@cselab-  
desktop:~$ yacc ift.y  
students@cselab-desktop:~$ gcc  
y.tab.c -ll -ly students@cselab-  
desktop:~$ ./a.out Enter the exp:  
if(a==1) then b=1; else b=2;  
Input accepted.  
students@cselab-desktop:~$
```

RESULT:

Thus the program is executed successfully.

EX.NO:4.a	IMPLEMENT A PROGRAM FOR SYNTAX CHECKING USING LEX AND YACC a. VARIABLE DECLARATIONS FOR PERFORMING SORTING OPERATION USED IN SORTING ALGORITHMS (MERGE SORT, INSERTION SORT, QUICK SORT ETC...)
DATE:	

AIM:

To write Yacc specification to validate the syntax for variable declaration statements.

ALGORITHM:

1. Start the program.
2. Declare regular expression for alphabets, digits, for loop and relational operators in lex specification.
3. Declare regular expression for variable declaration in the translation rules section.
4. Declare the main function and call yylex() function.
5. Compile the lex program using following command

```
$ flex filename.l
```

```
$ gcc lex.yy.c
```

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

6. Stop the program.

PROGRAM:

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

DIGIT [0-9]
REAL {DIGIT}+[.]{DIGIT}*
LETTER [A-Za-z]
ASSIGN =
%%

[ \t ] ;

int {printf("%s\t==> DataType\n",yytext);return (INT);}
float {printf("%s\t==> DataType\n",yytext);return (FLOAT);}
char {printf("%s\t==> DataType\n",yytext);return (CHAR);}
boolean {printf("%s\t==> DataType\n",yytext);return (BL);}
```

```

true|false { printf("%s\t==> BOOLEAN VAL\n",yytext);return BLVAL;}
[']^[^\\t\\n]['] { printf("%s\t==> CHAR VALUE\n",yytext);return CHVAL;}
[a-zA-z]+[a-zA-z0-9_]* {printf("%s\t==> ID\n",yytext);return ID;}
{REAL} { printf("%s\t==> REAL NUMBER\n",yytext);return REAL;}
{DIGIT}+ { printf("%s\t==> INT NUMBER\n",yytext);return NUM;}
"," {printf("%s\t==> COMMA\n",yytext);return COMMA;}
";" {printf("%s\t==> SC\n",yytext);return SC;}
{ASSIGN} {printf("%s\t==> ASSIGN\n",yytext);return AS;}
\n return NL;
. ;
%%

```

```

int yywrap()
{
return 1;
}

```

```

%{
#include<stdio.h>
void yyerror(char*);
int yylex();
//FILE* yyin;
%}

%token ID SC INT CHAR FLOAT BL BLVAL CHVAL REAL AS NUM COMMA NL

%%

s: type1|type2|type3|type4
;

type1:INT varlist SC NL { printf("valid INT Variable declaration");
return 0;} /// for "int a" Test case(without SC ;) NL is added at end
otherwise it waits for input ;

type2:FLOAT varlist2 SC NL{ printf("valid FLOAT Variable
declaration");return 0;} ;

```

```
type3:CHAR varlist3 SC NL{ printf("valid CHAR Variable
declaration");return 0;} ;
```

```
type4:BL varlist4 SC NL{ printf("valid BOOLEAN Variable
declaration");return 0;} ;
```

```
varlist: ID | ID COMMA varlist | ID AS NUM |ID AS NUM COMMA varlist |
//THIS IS FOR EPSILON CASE (EMPTY)
```

```
;
```

```
varlist2: ID | ID COMMA varlist2 | ID AS REAL |ID AS REAL
COMMA varlist2 | ;
```

```
varlist3: ID | ID COMMA varlist3 | ID AS CHVAL | ID AS CHVAL
COMMA varlist3 | ;
```

```
varlist4: ID | ID COMMA varlist4 | ID AS BLVAL | AS BLVAL
COMMA varlist4 | ;
```

```
%%
```

```
void yyerror(char *s )
```

```
{
```

```
fprintf(stderr,
"ERROR: %s\n",s);
}
```

```
int main()
```

```
{
```

```
    //yyin=fopen("input.txt","r");
```

```
    yyparse();
```

```
    //fclose(yyin);
```

```
    return 0;
```

```
}
```


OUTPUT:

```
int    ==> TYPE
a      ==> ITIFIER
,      ==> COLON
b      ==> ITIFIER
,      ==> COLON
c      ==> ITIFIER
,      ==> COLON
d      ==> ITIFIER
=      ==> ASSIGN
10     ==> NUMBER
;      ==> SC
```

RESULT:

Thus the program is executed successfully.

EX.NO:4.b	IMPLEMENT A PROGRAM FOR SYNTAX CHECKING USING LEX AND YACC b. FUNCTIONS CREATED FOR PERFORMING SORTING OPERATION USED IN SORTING ALGORITHMS (MERGE SORT, INSERTION SORT, QUICK SORT ETC...)
DATE:	

AIM:

To write Yacc specification to validate the syntax for functions.

ALGORITHM:

1. Start the program.
2. Declare regular expression for function definition.
3. Declare grammar for function definition in the translation rules section of YACC specification.
4. Compile the lex program using following command

```
$ flex filename.l
```

```
$ gcc lex.yy.c
```

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

PROGRAM:

(Lex Program : fundf.l)

```
alpha [A-Za-z]
```

```
digit [0-9]
```

```
%%
```

```
[\t \n] ;
```

```
int|float|void|char return TYPE;
```

```
return return RETURN;
```

```
{digit}+ return NUM;
```

```
{alpha}({alpha}|{digit})* return ID;
```

```
. return yytext[0];
```

```
%%
```

(Yacc Program : fundf.y)

```
%{
```

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

```
#include <stdlib.h>
```

```
% }
```

```

%token TYPE RETURN ID NUM
%right "="
%left '+' '-'
%left '*' '/'
%right UMINUS
%left '!'

%%

S : FUN {printf("Input accepted\n"); exit(0);}
FUN : TYPE ID '(' PARAM ')' '{' BODY '}'
    ;
PARAM : PARAM ',' TYPE ID
    | TYPE ID
    |
    ;
BODY : BODY BODY
    | PARAM ';'
    | E ';'
    | RETURN E ';'
    |
    ;
E : ID '=' E
    | E '+' E
    | E '-' E
    | E '*' E
    | E '/' E
    | ID
    | NUM
    ;

%%

#include "lex.yy.c"
main()

```

```
{
    printf("Enter the
expression:\n");
    yyparse();
}
```

Output :

```
nn@linuxmint ~ $ lex fund.l
nn@linuxmint ~ $ yacc fund.y

conflicts: 17 shift/reduce, 11
reduce/reduce nn@linuxmint
~ $ gcc y.tab.c -ll -ly
nn@linuxmint ~ $ ./a.out

Enter the expression:

float sum(int term)
{
    float result;
    result=result+term;
    return result;
}

Input accepted
nn@linuxmint ~ $
```

RESULT:

Thus the program is executed successfully.

EX.NO:5	DEVELOP A DESK CALCULATOR PROGRAM USING LEX AND YACC TO EVALUATE ARITHMETIC EXPRESSIONS. THE CALCULATOR SHOULD SUPPORT THE ARITHMETIC OPERATIONS AND ERROR HANDLING
DATE:	

AIM:

To write a program for implementing a calculator for computing the given expression using semantic

rules of the YACC tool and LEX.

ALGORITHM:

1. Start the program.
2. Declare regular expression for digits and operators in lex specification.
3. Declare tokens and grammar to perform arithmetic operations.
4. Declare the main function and call yylex() function.
5. Compile the lex program using following command

```
$ flex filename.l
```

```
$ gcc lex.yy.c
```

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

PROGRAM CODE:

//Implementation of calculator using LEX and YACC

LEX PART:

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

```
. return yytext[0];
```

```
%%
```

```
int yywrap()
```

```
{
```

```
return 1;
```

```
}
```

YACC PART:

```
%{
```

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

```
int flag=0;
```

```
%}
```

```
%token NUMBER
```

```
%left '+' '-'
```

```
%left '*' '/' '%'
```

```
%left '(' ')'
```

```
%%
```

```
ArithmeticExpression: E{
```

```
printf("\nResult=%d\n", $$);
```

```
return 0;
```

```
};
```

```
E: E '+' E { $$ = $1 + $3; }
```

```
| E '-' E { $$ = $1 - $3; }
```

```
| E '*' E { $$ = $1 * $3; }
```

```
| E '/' E { $$ = $1 / $3; }
```

```
| E '%' E { $$ = $1 % $3; }
```

```
| '(' E ')' { $$ = $2; }
```

```
| NUMBER { $$ = $1; }
```

```
;
```

```
%%
```

```
void main()
```

```
{
```

```
printf("\nEnter Any Arithmetic Expression which can have operations Addition,
```

Subtraction, Multiplication, Division, Modulus and Round brackets:\n");

```
yyvsparse();
```

```
if(flag==0)
```

```
printf("\nEntered arithmetic expression is Valid\n\n");
```

```
}
```

```
void yyerror()
```

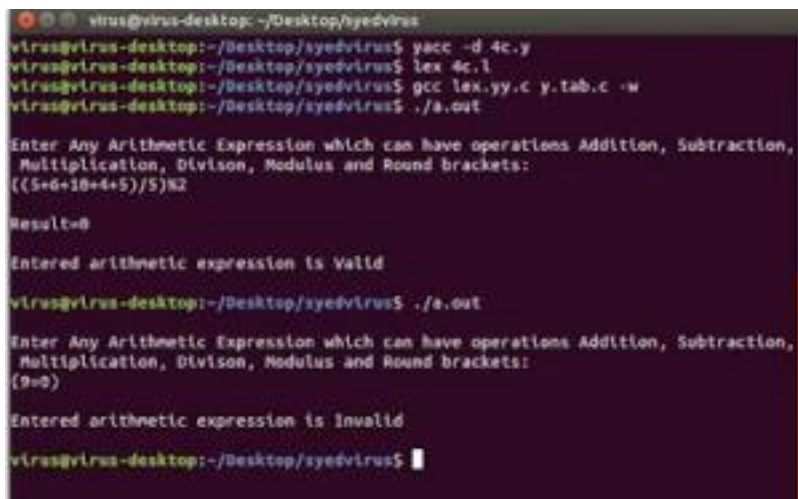
```
{
```

```
printf("\nEntered arithmetic expression is Invalid\n\n");
```

```
flag=1;
```

```
}
```

OUTPUT



```
virus@virus-desktop: ~/Desktop/syedvirus
virus@virus-desktop:~/Desktop/syedvirus$ yacc -d 4c.y
virus@virus-desktop:~/Desktop/syedvirus$ lex 4c.l
virus@virus-desktop:~/Desktop/syedvirus$ gcc lex.yy.c y.tab.c -w
virus@virus-desktop:~/Desktop/syedvirus$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
Multiplication, Division, Modulus and Round brackets:
((5+6+10+4*5)/5)%2

Result=0
Entered arithmetic expression is Valid

virus@virus-desktop:~/Desktop/syedvirus$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
Multiplication, Division, Modulus and Round brackets:
9=0

Entered arithmetic expression is Invalid

virus@virus-desktop:~/Desktop/syedvirus$
```

RESULT

Thus the program is executed successfully.

EX.NO:6	DEVELOP A PROGRAM USING LEX AND YACC TO GENERATE THREE ADDRESS CODE (TAC) FOR A QUICK SORT FUNCTION IN A CUSTOM-DEFINED PROGRAMMING LANGUAGE. THE TAC SHOULD REPRESENT INTERMEDIATE CODE THAT CAN BE USED FOR FURTHER OPTIMIZATIONS OR TRANSLATION INTO MACHINE CODE.
DATE:	

AIM:

To write a program to generate three address code for a simple program using LEX and YACC.

ALGORITHM:

1. Start the program.
2. Declare regular expression for alphabets and digits in lex specification.
3. Declare grammar for generating three address code in YACC specification.
4. Declare the main function and call yylex() function.
5. Compile the lex program using following command

```
$ flex filename.l
```

```
$ gcc lex.yy.c
```

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

PROGRAM CODE:**LEX**

```
%{
#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;}
%%
```

YACC

```
%{
```



```

#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].o
        perand1);
        printf("%c\t",arr[i].o
        perator);
        printf("%c\t",arr[i].o
        perand2); i++;

        temp++;
        printf("\n");
    }
}

```

```

void fouradd(){
    int i=0;
    char temp='A';
    while(i<index1){

        printf("%c\t",arr[i].o
        perator);
        printf("%c\t",arr[i].o
        perand1);
        printf("%c\t",arr[i].o
        perand2);

        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].o
        perator);
        if(!isupper(arr[i].oper

```

```

and1))
printf("%c\t",arr[i].o
perand1); else{

    printf("pointer");

    printf("%d\t",find(arr[i].opera
nd1)); }

if(!isupper(arr[i].oper
and2))
printf("%c\t",arr[i].o
perand2); else{

    printf("pointer");

    printf("%d\t",find(arr[i].opera
nd2)); }

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:

```
[duke@duke-pc a5]$
yacc -d yacc.y
[duke@duke-pc a5]$
lex lex.l

[duke@duke-pc a5]$ gcc
y.tab.c lex.yy.c -w
[duke@duke-pc a5]$ ./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 program is executed successfully.

EX.NO:7	DEVELOP A PROGRAM TO IMPLEMENT AND DEMONSTRATE VARIOUS CODE OPTIMIZATION TECHNIQUES (CONSTANT FOLDING, DEAD CODE ELIMINATION, LOOP OPTIMIZATION, STRENGTH REDUCTION) ON A QUICK SORT ALGORITHM WRITTEN IN A HIGH-LEVEL PROGRAMMING LANGUAGE.
DATE:	

AIM:

To write a program for implementation of Code Optimization Technique. **ALGORITHM:**

Step1: Start the program.

Step2: Enter the number of expressions

Step3: Read the expression.

Step4: Identify dead codes and eliminate.

Step5: Identify common subexpression and eliminate.

Step6: Print the optimized code.

Step7: Stop the program.

PROGRAM CODE:

//Code Optimization Technique

```
#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:

```

virus@virus-desktop: ~/Desktop/syedvirus
virus@virus-desktop:~/Desktop/syedvirus$ gcc codeop.c -w
virus@virus-desktop:~/Desktop/syedvirus$ ./a.out
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
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
virus@virus-desktop:~/Desktop/syedvirus$

```

RESULT:

Thus the program is executed successfully.

EX.NO:8	DEVELOP A PROGRAM USING LEX AND YACC TO IMPLEMENT CODE GENERATION TECHNIQUES FOR GENERATING OPTIMIZED INTERMEDIATE CODE FOR THE QUICK SORT ALGORITHM FROM A CUSTOM-DEFINED HIGH-LEVEL LANGUAGE.
DATE:	

AIM:

To write a program for implementation of Code Generation Technique.

ALGORITHM:

1. Start
2. Get address code sequence.
3. Determine current location of operand using address (for 1st operand).
4. If current location not already exist generate move (B,O).
5. Update address of A(for 2nd operand).
6. If current value of B not null , check the operator.
7. If the operator is +,-,* or / , print ADD, SUB, MUL, DIV respectively.
8. Store the move instruction in memory.
9. Stop.

//C program to implement Simple Code Generator.

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

```
#include<conio.h>
```

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

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

```
#include<graphics.h>
```

```
typedef struct
```

```
{ char var[10];
```

```
int alive;
```

```
}
```

```
regist;
```

```

regist preg[10];

void substring(char exp[],int st,int end)

{ int i,j=0;

char dup[10]="";
for(i=st;i<end;i++)

dup[j++]=exp[i];

dup[j]='0';

strcpy(exp,dup);

} int

getregister(c

har var[]) {

int i;

for(i=0;i<10;i++)

{

if(preg[i].alive==0)

{

strcpy(preg[i].var,var);

break;

}}

return(i);

}

void getvar(char

exp[],char v[]) {

int i,j=0;

char var[10]="";

```

```

for(i=0;ex
p[i]!='\0';i
++)
if(isalpha(
exp[i]))

var[j++]=exp[i];

else
break;

strcpy(v,var);

}
void main()

{ char basic[10][10],var[10][10],fstr[10],op; int i,j,k,reg,vc,flag=0;

clrscr();

printf("\nEnter the Three
Address Code:\n");

for(i=0;;i++)

{

gets(basic[i]);

if(strcmp(basic[i],"exit")==0)

break;

}

printf("\nThe Equivalent
Assembly Code is:\n");

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

{

```

```

getvar(basic[j],var[vc++]);

strcpy(fstr,var[vc-1]);

substring(basic[j],strlen(var[vc-
1])+1,strlen(basic[j]));

getvar(basic[j],var[vc++]);

reg=getregister(var[vc-1]);

if(preg[reg].alive==0)

{

printf("\nMov R%d,%s",reg,var[vc-1]);

preg[reg].alive=1;

}

op=basic[j][strlen(var[vc-1])];

substring(basic[j],strlen(var[vc-
1])+1,strlen(basic[j]));

getvar(basic[j],var[vc++]);

switch(op)

{ case '+': printf("\nAdd"); break;

case '-': printf("\nSub"); break;

case '*': printf("\nMul"); break;

case '/': printf("\nDiv"); break;

}

flag=1;

for(k=0;k<=reg;k++)

{ if(strcmp(preg[k].var,var[vc-1])==0)

{

```

```

printf("R%d, R%d",k,reg);

preg[k].alive=0;

flag=0;

break;

}} if(flag)

{

printf(" %s,R%d",var[vc-1],reg);

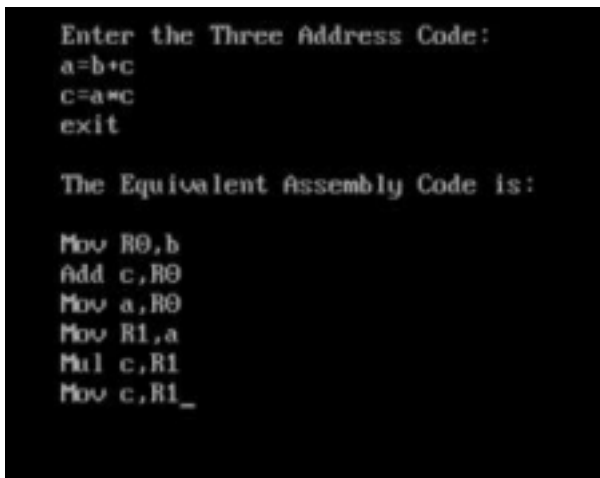
printf("\nMov %s,R%d",fstr,reg);

}strcpy(preg[reg].var,var[vc-3]);

getch();

}}

```

OUTPUT:


```

Enter the Three Address Code:
a=b+c
c=a*c
exit

The Equivalent Assembly Code is:

Mov R0,b
Add c,R0
Mov a,R0
Mov R1,a
Mul c,R1
Mov c,R1_

```

RESULT:

Thus the program is executed successfully.