SHANMUGANATHAN ENGINEERING COLLEGE

Arasampatti (Po), Thirumayam (Tk),

Pudukkottai (Dist.) – 622 507.

Department of Computer Science & Engineering

CS3501-Compiler Design

CS3501 - COMPILER DESIGN LABORATORY

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 a 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.

Total: 30 PERIODS

EX. NO: 1

IMPLEMENTATION OF SYMBOL TABLE

AIM:

To write a C program to implement a symbol table.

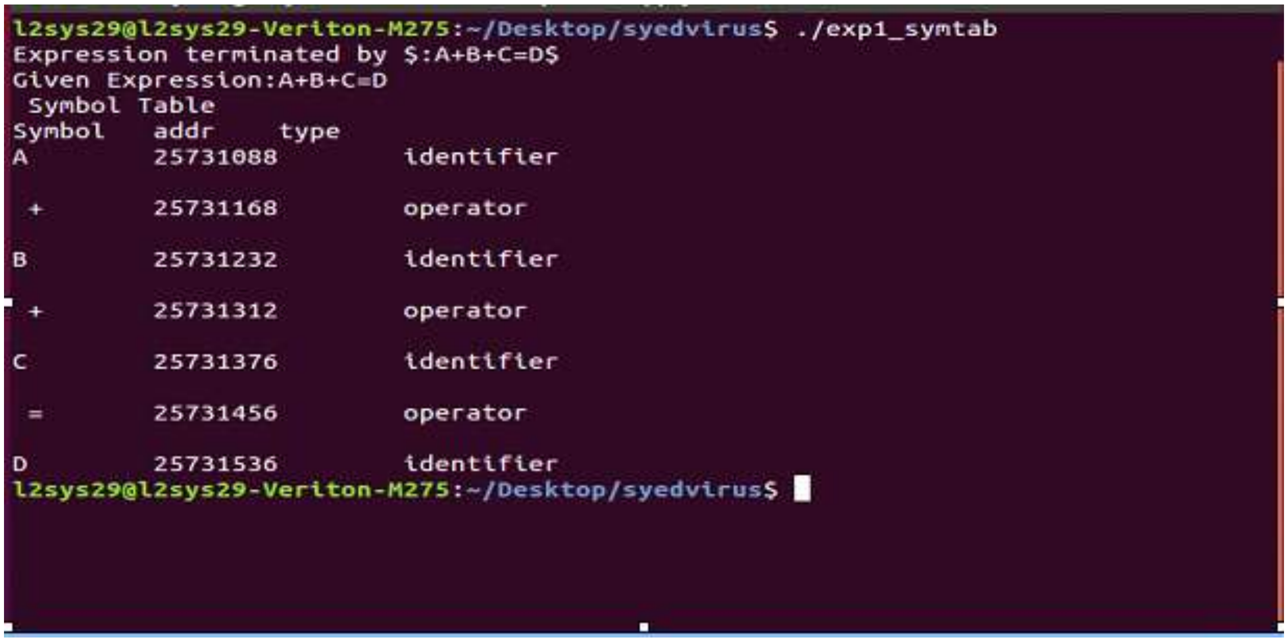
Algorithm

1: Start the program for performing insert, display, delete, search and modify option in symbol table
2: Define the structure of the Symbol Table
3: Enter the choice for performing the operations in the symbol Table
4: If the entered choice is 1, search the symbol table for the symbol to be inserted. If the symbol is already present, it displays "Duplicate Symbol". Else, insert the symbol and the corresponding address in the symbol table.
5: If the entered choice is 2, the symbols present in the symbol table are displayed.
6: If the entered choice is 3, the symbol to be deleted is searched in the symbol table.
7: If it is not found in the symbol table it displays "Label Not found". Else, the symbol is deleted.
8: If the entered choice is 5, the symbol to be modified is searched in the symbol table.

PROGRAM CODE:

```
//Implementation of symbol table
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
void main()
{
 int i=0,j=0,x=0,n;
 void *p,*add[5];
 char ch,srch,b[15],d[15],c;
 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("\n Symbol Table\n");
 printf("Symbol \t addr \t type");
 while(j<=n)
 {
  c=b[j];
  if(isalpha(toascii(c)))
  {
   p=malloc(c);
   add[x]=p;
   d[x]=c;
   printf("\n%c \t %d \t identifier\n",c,p);
   x++;
   j++;
  }
  else
  {
   ch=c;
   if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
   {
    p=malloc(ch);
    add[x]=p;
    d[x]=ch;
    printf("\n %c \t %d \t operator\n",ch,p);
    x++;
  j++;
 }}} }
```

OUTPUT:



```
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ ./exp1_symtab
Expression terminated by $:A+B+C=D$
Given Expression:A+B+C=D
 Symbol Table
Symbol   addr     type
A        25731088        identifier

 +       25731168        operator

B        25731232        identifier

 +       25731312        operator

C        25731376        identifier

 =       25731456        operator

D        25731536        identifier
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ 
```

RESULT:

Thus the program for symbol table has been executed successfully.

EX. NO: 2

IMPLEMENT A LEXICAL ANALYZER USING LEX TOOL

Aim:

To Implement a Lexical Analyzer using Lex Tool

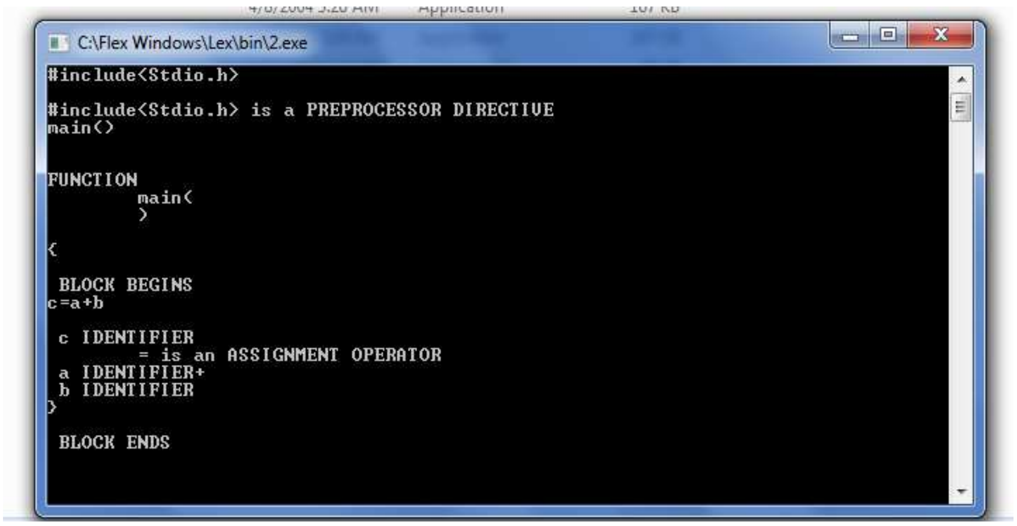
ALGORITHM:

1. Begin by defining any necessary definitions or declarations, such as including header files or defining global variables, at the start of the Lex file.
2. Specify the regular expressions and corresponding actions for token recognition in the %rules section of the Lex file. These rules define patterns and associated actions to perform when a specific pattern is matched.
3. Within the action code blocks of each rule, perform any required actions such as setting the value of token attributes or storing token information.
4. If needed, define auxiliary functions or data structures to support the lexical analysis process.
5. Compile the Lex file using the Lex tool, which generates a C source file.
6. Include the generated C source file in your C/C++ project.
7. Write the main program code in your C/C++ file to call the lexer functions and handle the tokens.
8. Run the program and observe the output to verify the correctness of the lexical analysis.

Program

```
%{
/* program to recognize a c program */
int COMMENT=0;
int cnt=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
"*/" {COMMENT = 0; cnt++;}
{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);}
%%
int main(int argc,char **argv)
{
if (argc > 1)
{
FILE *file;
file = fopen(argv[1],"r");
if(!file)
{
printf("could not open %s \n",argv[1]);
exit(0);
}
yyin = file;
}
yylex();
printf("\n\n Total No.Of comments are %d",cnt);
return 0;
}
int yywrap()
{
return 1;
}
```

OUTPUT:



```
#include<Stdio.h>

#include<Stdio.h> is a PREPROCESSOR DIRECTIVE
main()

FUNCTION
        main(
        )

{

 BLOCK BEGINS
c=a+b

 c IDENTIFIER
        = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER
}

 BLOCK ENDS
```

RESULT:

Thus the program for implementing a lexical analyzer using lex tool has been executed successfully.

EX. NO: 3a

PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +, -, * AND /

AIM:

To write a program for recognizing a valid arithmetic expression that uses operator +, -, * and / .

ALGORITHM:

LEX

1. Declare the required header file and variable declaration with in '%{' and '%}'.

2. LEX requires regular expressions to identify valid arithmetic expression token of lexemes.

3. LEX call yywrap() function after input is over. It should return 1 when work is done or should return 0 when more processing is required.

YACC

1. Declare the required header file and variable declaration with in '%{' and '%}'.

2. Define tokens in the first section and also define the associativity of the operations

3. Mention the grammar productions and the action for each production.

4. $$ refer to the top of the stack position while $1 for the first value, $2 for the second value in the stack.

5. Call yyparse() to initiate the parsing process.

6. yyerror() function is called when all productions in the grammar in second section doesn't match to the input statement.

Program:

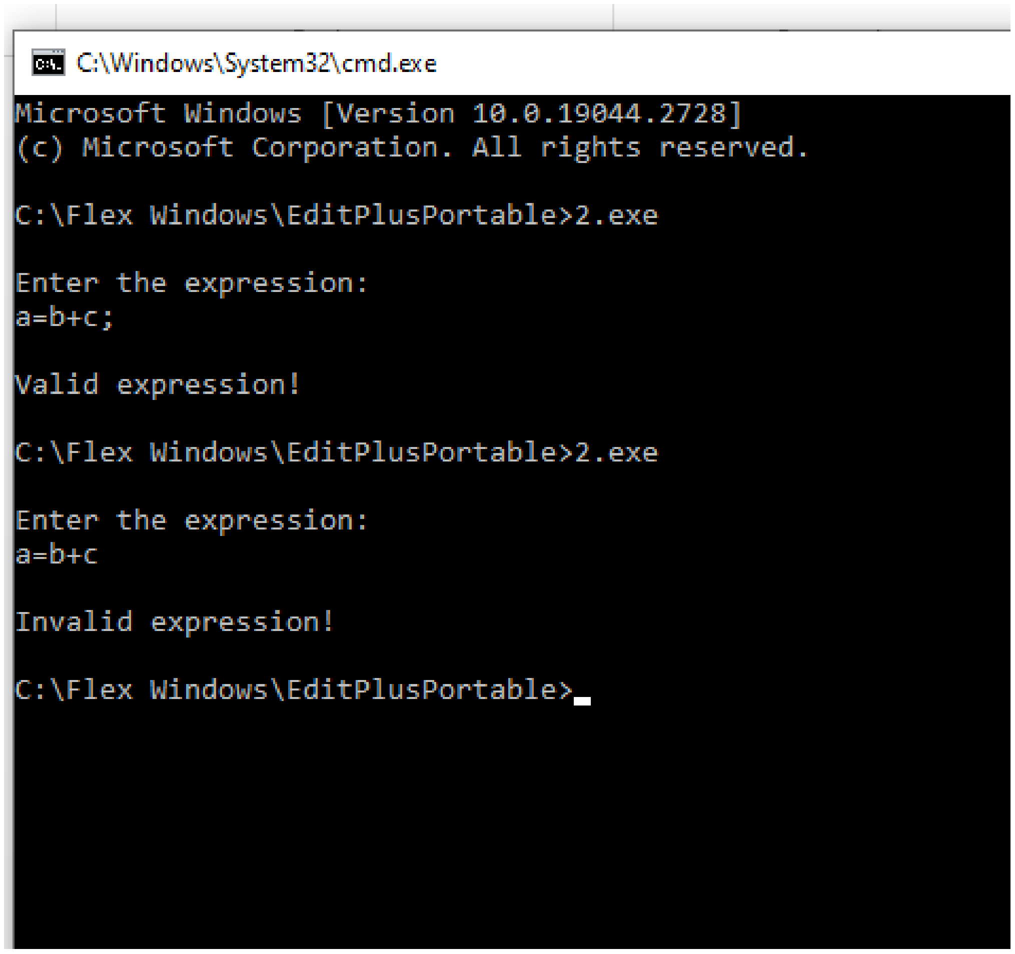
LEX PART:

```
%{
   #include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)?      return num;
[+/*]                  return op;
.                    return yytext[0];
\n                    return 0;
%%
int yywrap()
{
return 1;
}
```

YACC PART:

```
%{
   #include<stdio.h>
   int valid=1;
%}
%token num id op
%%
start : id '=' s ';'
s :     id x
      | num x
      | '-' num x
      | '(' s ')' x
      ;
x :     op s
      | '-' s
      |
      ;
%%
int yyerror()

{
   valid=0;
   printf("\nInvalid expression!\n");
   return 0;
}
int main()
{
   printf("\nEnter the expression:\n");
   yyparse();
   if(valid)
   {
      printf("\nValid expression!\n");
   }
}
```

Output:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19044.2728]
(c) Microsoft Corporation. All rights reserved.

C:\Flex Windows\EditPlusPortable>2.exe

Enter the expression:
a=b+c;

Valid expression!

C:\Flex Windows\EditPlusPortable>2.exe

Enter the expression:
a=b+c

Invalid expression!

C:\Flex Windows\EditPlusPortable>
```

RESULT:

Thus the program to recognize a valid arithmetic expression that uses operator +, - , * and / using YACC tool was executed and verified successfully.

EX. NO: 3b

PROGRAM TO RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS.

AIM:

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

ALGORITHM:

LEX

1. Declare the required header file and variable declaration with in '%{' and '%}'.
2. LEX requires regular expressions or patterns to identify token of lexemes for recognize a valid variable.
3. Lex call yywrap() function after input is over. It should return 1 when work is done or should return 0 when more processing is required.

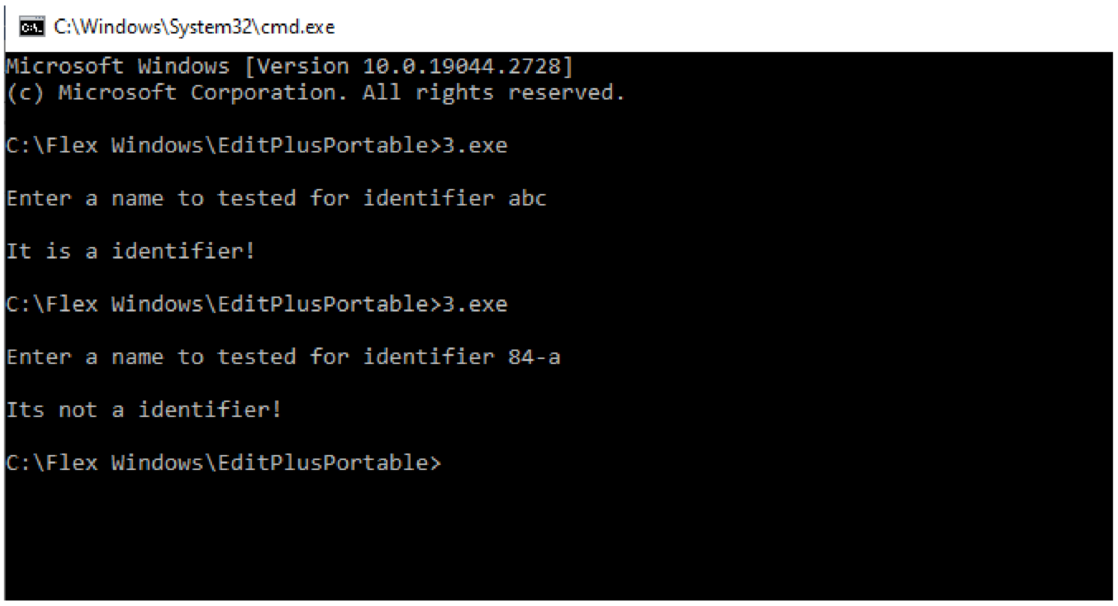
YACC

1. Declare the required header file and variable declaration with in '%{' and '%}'.
2. Define tokens in the first section and also define the associativity of the operations
3. Mention the grammar productions and the action for each production.
4. $$ refer to the top of the stack position while $1 for the first value, $2 for the second value in the stack.
5. Call yyparse() to initiate the parsing process.
6. yyerror() function is called when all productions in the grammar in second section doesn't match to the input statement

Program:

```
LEX PART:
%{
   #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 PART:
%{
   #include<stdio.h>
   int valid=1;
%}
%token digit letter
%%
start : letter s
s :     letter s
      | digit s
      |
      ;

%%
int yyerror()
{
  printf("\nIts not a identifier!\n");
   valid=0;
   return 0;
}
int main()
{
   printf("\nEnter a name to tested for identifier ");
   yyparse();
  if(valid)
   {
      printf("\nIt is a identifier!\n");
   }
}
```

Output:



C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.19044.2728]
(c) Microsoft Corporation. All rights reserved.

C:\Flex Windows\EditPlusPortable>3.exe

Enter a name to tested for identifier abc

It is a identifier!

C:\Flex Windows\EditPlusPortable>3.exe

Enter a name to tested for identifier 84-a

Its not a identifier!

C:\Flex Windows\EditPlusPortable>
```

Result:

Thus program to recognize a valid variable which starts with a letter followed by any number of letters or digits using YACC tool was implemented successfully.

EX. NO: 3C

PROGRAM TO RECOGNIZE A VALID CONTROL STRUCTURES SYNTAX OF C LANGUAGE

Aim:

To write a program to recognize a valid control structures syntax of c language

Algorithm:

Lexer (Lex):

1. Define regular expressions and associated actions to recognize tokens such as keywords (if, else, while, for), operators (>, <, ==, !=), identifiers, numbers, and punctuation symbols (;, {, }).
2. On recognizing a token, assign the token type to yylval (if required) and return the corresponding token code.

Parser (YACC):

1. Define the grammar rules for valid control structures in the C language, including if-statements, while-loops, and for-loops.
2. Specify the precedence and associativity of operators.
3. Implement actions associated with each grammar rule to perform semantic actions or generate an abstract syntax tree (AST).
4. Use semantic actions to handle the recognized control structures appropriately (e.g., printing, executing, or generating intermediate code).

Program:

Lex part:

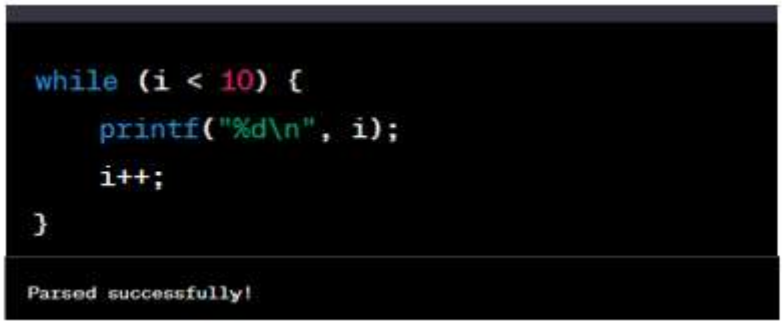
```
%{

#include "y.tab.h"

%}


%%

"if"       { return IF; }

"else"     { return ELSE; }

"while"    { return WHILE; }

"for"      { return FOR; }

"do"       { return DO; }

"switch"   { return SWITCH; }

"case"     { return CASE; }

"break"    { return BREAK; }

"{"        { return LBRACE; }

"}"        { return RBRACE; }

";"        { return SEMICOLON; }

[ \t\n]+   ; /* ignore whitespace and newlines */

.          ; /* ignore all other characters */

%%

}
```

YACC part:

```
%{
#include <stdio.h>
int yylex();
void yyerror(const char *s);
int yywrap();

int main() {
    yyparse();
```

```
   return 0;
}
%}
%token IF ELSE WHILE FOR DO SWITCH CASE BREAK LBRACE RBRACE SEMICOLON
%%
program : control_structure
        | program control_structure
        ;
control_structure : if_statement
                  | while_statement
                  | for_statement
                  | do_while_statement
                  | switch_statement
                  | SEMICOLON
                  ;
if_statement : IF '(' expression ')' control_structure
             | IF '(' expression ')' control_structure ELSE control_structure
             ;
while_statement : WHILE '(' expression ')' control_structure
                ;
for_statement : FOR '(' expression SEMICOLON expression SEMICOLON expression ')'
control_structure
              ;

do_while_statement : DO control_structure WHILE '(' expression ')' SEMICOLON
                   ;
switch_statement : SWITCH '(' expression ')' '{' case_statements '}'
                 ;
case_statements : CASE expression ':' control_structure
                | CASE expression ':' control_structure case_statements
                | BREAK SEMICOLON
                ;
expression : /* define your expression grammar here */
           ;
%%
void yyerror(const char *s) {
   printf("Error: %s\n", s);
}
int yywrap() {
   return 1;
}
```

Output:



```
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

```
Parsed successfully!
```

Result:

Thus program to recognize a valid control structures syntax of c language are implemented successfully.

EX. NO: 3d

IMPLEMENTATION OF CALCULATOR USING LEX & YACC

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: A Yacc source program has three parts as follows:

Declarations %% translation rules %% supporting C routines

2: Declarations Section: This section contains entries that:

i. Include standard I/O header file.

ii. Define global variables.

iii. Define the list rule as the place to start processing.

iv. Define the tokens used by the parser. v. Define the operators and their precedence.

3: Rules Section: The rules section defines the rules that parse the input steam. Each rule of a grammar production and the associated semantic action.

4: Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.

5: Main- The required main program that calls the yyparse subroutine to start the program.

6: yyerror(s) -This error-handling subroutine only prints a syntax error message.

7: yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The calc.lex file contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The y.tab.h file contains definitions for the tokens that the parser program uses.

8: calc.lex contains the rules to generate these tokens from the input stream.

PROGRAM CODE:

//*Implementation of calculator using LEX and YACC*

LEX PART:

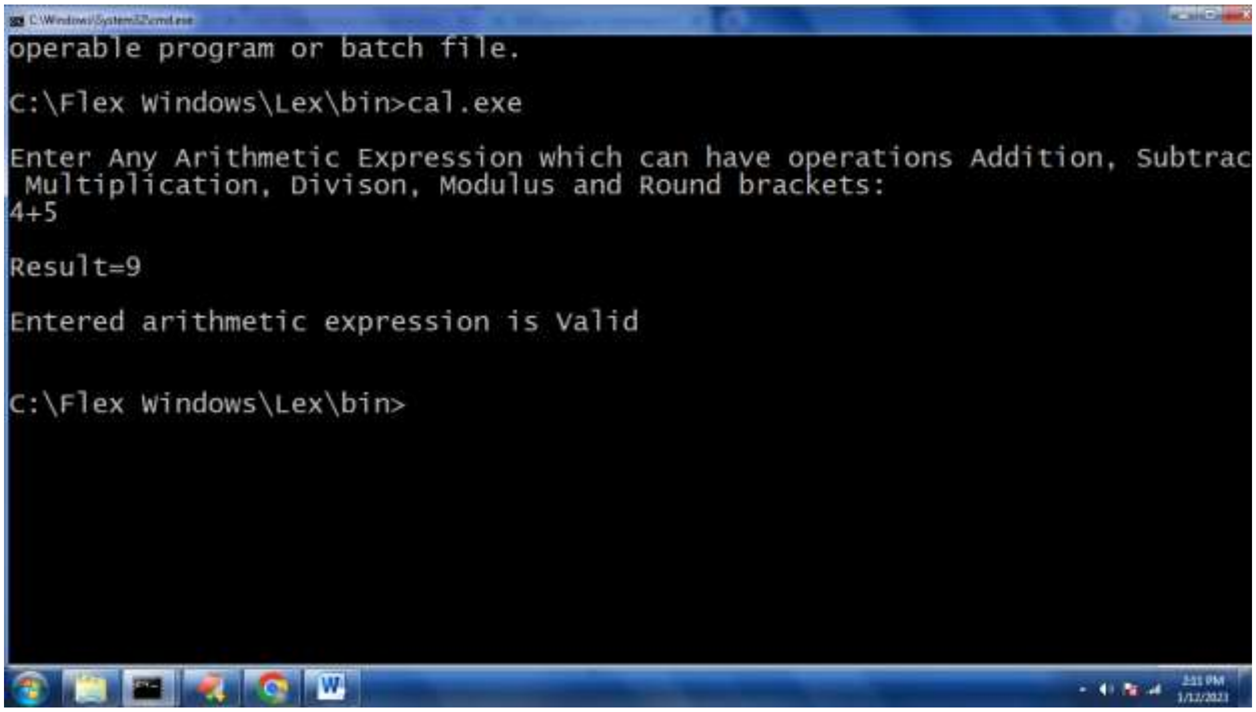
```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+
{
        yylval=atoi(yytext);
        return NUMBER;
      }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

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, Divison, Modulus and Round brackets:\n");
   yyparse();
  if(flag==0)
   printf("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror()
{
   printf("\nEntered arithmetic expression is Invalid\n\n");
   flag=1;
}
```

OUTPUT:



```
operable program or batch file.

C:\Flex Windows\Lex\bin>cal.exe

Enter Any Arithmetic Expression which can have operations Addition, Subtrac
 Multiplication, Divison, Modulus and Round brackets:
4+5

Result=9

Entered arithmetic expression is Valid


C:\Flex Windows\Lex\bin>
```

RESULT

Thus the program for implementing a calculator for computing the given expression using semantic rules of the YACC tool and LEX is executed successfully.

Ex No:4

GENERATE THREE ADDRESS CODES FOR A SIMPLE PROGRAM USING LEX AND YACC.

Aim :

To generate three address codes for a simple program using LEX and YACC.

Algorithm:

1. Define the token patterns and corresponding actions in a Lex file (usually with a .l extension). These patterns should recognize identifiers, constants, operators, and other relevant symbols.
2. Specify any necessary definitions or declarations at the beginning of the Lex file.
3. Specify the regular expressions and corresponding actions using the %rules section in the Lex file. These rules define patterns and associated actions to perform when a specific pattern is matched.
4. Within the action code blocks of each rule, perform any required actions such as setting the value of token attributes or storing token information.
5. If needed, define auxiliary functions or data structures to support the parsing and code generation process.
6. Write the Yacc file (usually with a .y extension) that specifies the grammar rules and actions for parsing the program and generating three-address code.
7. Define the grammar rules in the Yacc file that describe the program's syntax. Each rule should specify how to combine or reduce smaller expressions into larger ones while generating three-address code.
8. Within the action code blocks of each rule, generate the corresponding three-address code based on the parsed expressions.
9. If needed, define additional helper functions or data structures to assist in the code generation process.
10. Compile the Lex and Yacc files using the Lex and Yacc tools to generate the corresponding C source files.
11. Include the generated C source files in your C/C++ project.
12. Write the main program code in your C/C++ file to interact with the user, call the parser functions, and display the generated three-address code.
13. Run the program and test it with simple programs to verify the parsing and code generation functionality.

Program:

Lex:

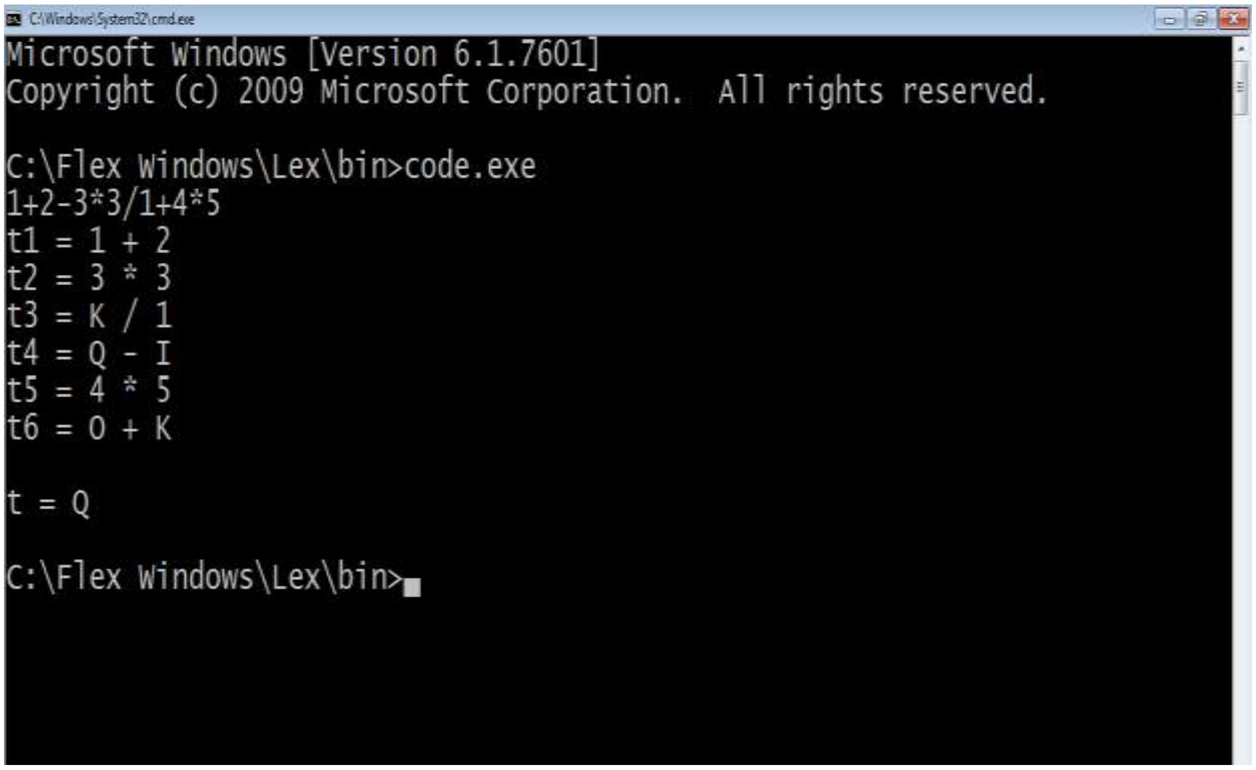
```
%{
   #include<stdio.h>
   #include"y.tab.h"
   int k=1;
%}

%%
[0-9]+ {
yylval.dval=yytext[0];
return NUM;
}

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

void yyerror(char* str)
{
      printf("\n%s",str);
}
char *gencode(char word[],char first,char op,char second)
{
   char temp[10];
   sprintf(temp,"%d",k);
   strcat(word,temp);
   k++;
   printf("%s = %c %c %c\n",word,first,op,second);

   return word; //Returns variable name like t1,t2,t3... properly
}
int yywrap()
{
   return 1;
}

main()
{
      yyparse();
      return 0;
}
```

YACC:

```
%{
#include<stdio.h>
int aaa;
%}
%union{
    char dval;
}
%token <dval> NUM
%type <dval> E
%left '+' '-'
%left '*' '/' '%'
%%
statement : E {printf("\nt = %c \n",$1);}
          ;
E : E '+' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'+',$3);
        $$=test;
    }
  | E '-' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'-',$3);
        $$=test;
    }
  | E '%' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'%',$3);
        $$=test;
    }
  | E '*' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'*',$3);
        $$=test;
    }
  | E '/' E
    {
        char word[]="t";
        char *test=gencode(word,$1,'/',$3);
        $$=test;
    }
  | '(' E ')'
```

```
    {
        $$=$2;
    }
  | NUM
    {
        $$=$1;
    }
  ;
%%
```

Output:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Flex Windows\Lex\bin>code.exe
1+2-3*3/1+4*5
t1 = 1 + 2
t2 = 3 * 3
t3 = K / 1
t4 = Q - I
t5 = 4 * 5
t6 = O + K

t = Q

C:\Flex Windows\Lex\bin>
```

Result:

Thus the three address codes for a simple program using LEX and YACC was generated successfully.

Ex No:5

IMPLEMENT SIMPLE CODE OPTIMIZATION TECHNIQUES

AIM:

To write a program for implementation of Code Optimization Technique.

ALGORITHM:

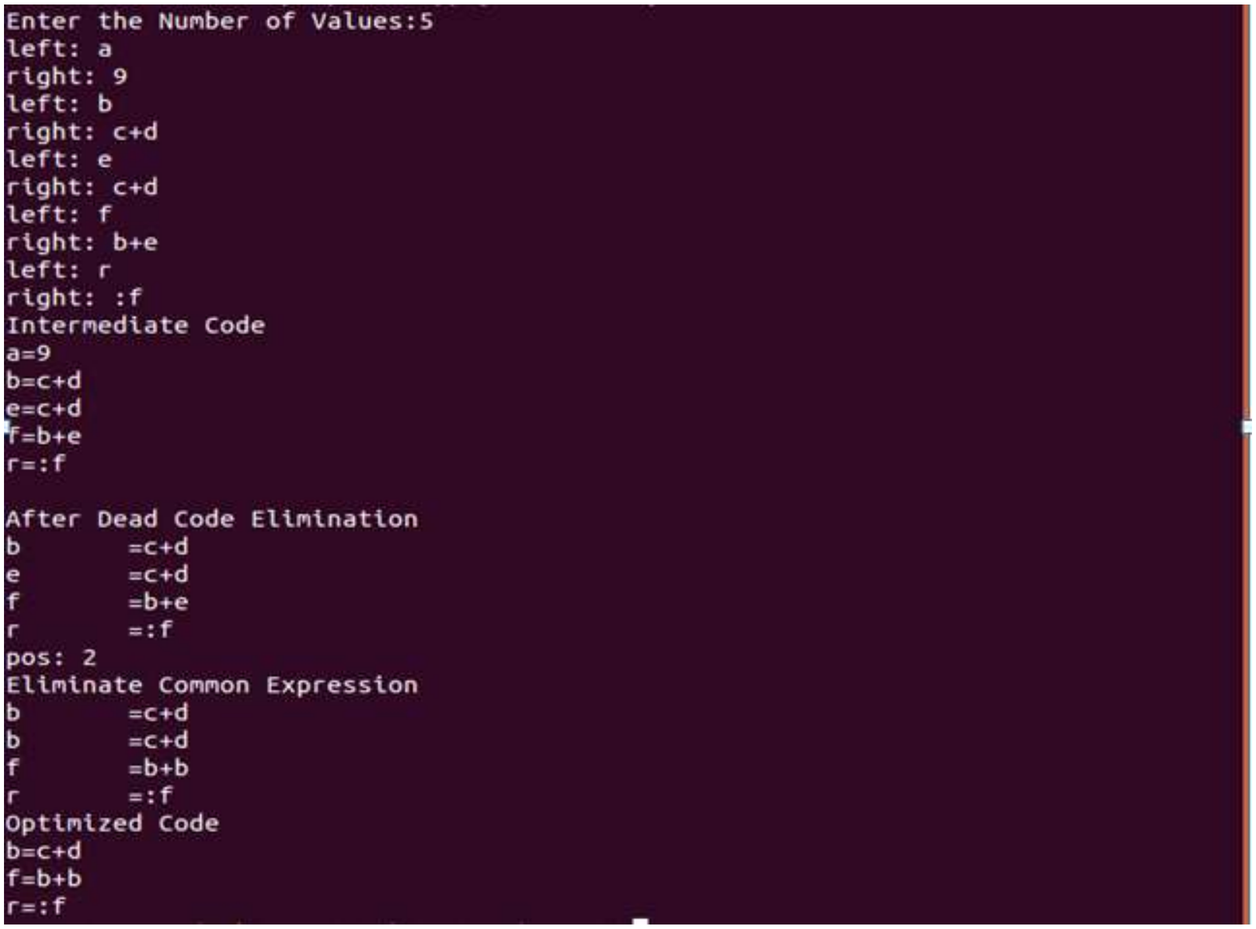
1: Generate the program for factorial program using for and do-while loop to specify optimization technique.
2: In for loop variable initialization is activated first and the condition is checked next. If the condition is true the corresponding statements are executed and specified increment / decrement 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 dowhile is best because first the statement execution is done then only the condition is checked. So, during the statement execution itself we can find the inconvenience of the result and no need to wait for the specified condition result.
6: Finally when considering Code Optimization in loop do-while best with is respect to performance.

PROGRAM :

```
//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:



```
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
```

RESULT

Thus the program for implementation of Code Optimization Technique was executed and verified successfully.

EX.NO.6

IMPLEMENT THE BACK END OF THE COMPILER

AIM:

To 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 a 8086 assembler.

ALGORITHM:

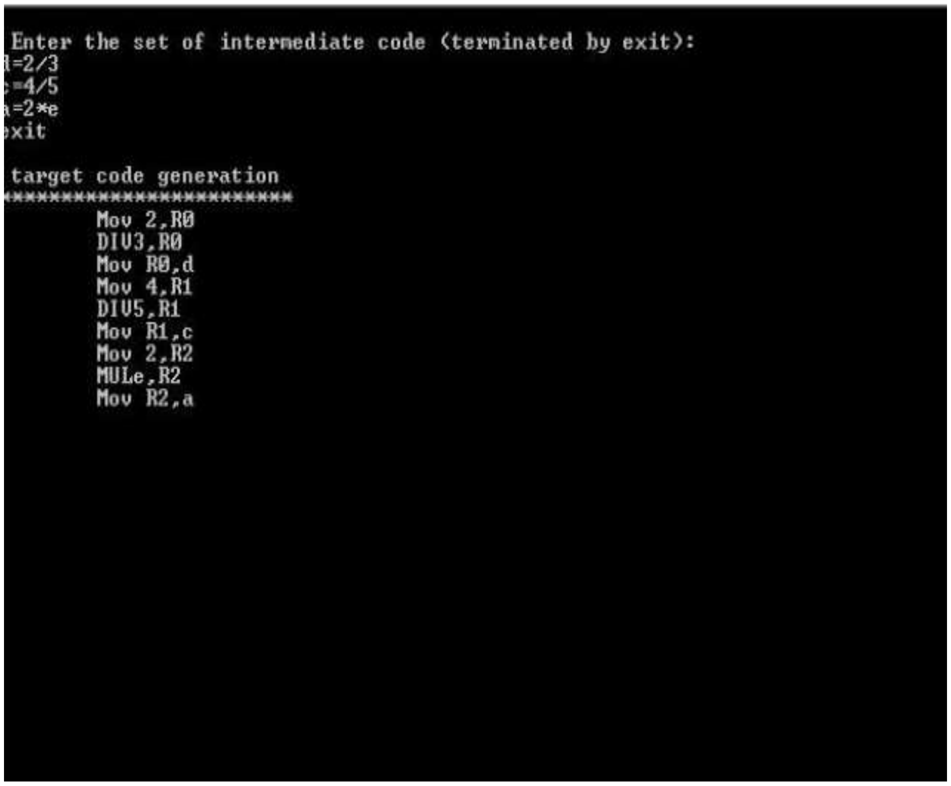
1. Start the program
2. Open the source file and store the contents as quadruples.
3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.
4. Write the generated code into output definition of the file in outp.c
5. Print the output.
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");
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);
//getch();
}
```

Output:



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

target code generation
***********************
        Mov 2,R0
        DIV3,R0
        Mov R0,d
        Mov 4,R1
        DIV5,R1
        Mov R1,c
        Mov 2,R2
        MULe,R2
        Mov R2,a
```

Result:

Thus 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 a 8086 assembler was implemented successfully.