CS3501 **COMPILER DESIGN** **L T P C**
3 0 2 4

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. Lab Requirements: for a batch of 30 students

TOTAL45 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.

After the completion of this course, students should be able to [Blooms Taxonomy]

C502.1	Understand the techniques in different phases of a compiler.
C502.2	Design a lexical analyser for a sample language and learn to use the LEX tool.
C502.3	Apply different parsing algorithms to develop a parser and learn to use YACC tool
C502.4	Understand semantics rules (SDT), intermediate code generation and run-time environment.
C502.5	Implement code generation and apply code optimization techniques.

Expected Course outcome and Program Outcome Mapping

Contribution 1: Reasonable 2: Significant 3: Strong

CO's-PO's & PSO's MAPPING

CO's	PO's												PSO's		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
1	3	3	3	3	-	-	-	-	3	3	1	3	2	3	2
2	3	3	3	3	3	-	-	-	3	2	3	2	2	1	2
3	3	3	2	2	3	-	-	-	3	1	1	1	2	2	3
4	3	2	2	1	1	-	-	-	2	3	2	3	1	2	1
5	3	3	3	2	1	-	-	-	2	1	1	3	2	1	2
AVg.	3.00	2.80	2.60	2.20	2.00	-	-	-	2.60	2.00	1.60	2.40	1.80	1.80	2.00

1 - low, 2 - medium, 3 - high, '-'- no correlation

h

EX.NO	NAME OF THE EXERCISE	PAGENO.
1.	**DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C AND CREATE SYMBOL TABLE WHILE RECOGNIZING IDENTIFIER**	**1**
2.	**IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL**	**10**
3.	**GENERATE YACC SPECIFICATION FOR A FEW SYNTACTIC CATEGORIES** **a) ARITHMETIC EXPRESSION RECOGNIZER**	**15**
	b) PROGRAM TO RECOGNISE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS	**18**
	c) PROGRAM TO RECOGNISE A VALID CONTROL STRUCTURES SYNTAX OF C LANGUAGE(FOR LOOP, WHILE LOOP, IF-ELSE, IF-ELSE-IF, SWITCH-CASE, ETC.)	**21**
	d) PROGRAM TO IMPLEMENT A CALCULATOR USING LEX AND YACC.	**25**
4.	**GENERATE THREE ADDRESS CODE FOR A SIMPLE PROGRAM USING LEX AND YACC**	**28**
5.	**IMPLEMENTATION OF TYPE CHECKING**	**32**
6.	**IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUES**	**38**
7.	**IMPLEMENT THE BACK END OF THE COMPILER**	**40**

EX. NO:1	**DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C AND CREATE SYMBOL TABLE WHILE RECOGNIZING IDENTIFIERS**
Date:	

AIM:

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

ALGORITHM(PATTERNS IN C):

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 open.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.

ALGORITHM(SYMBOL TABLE):

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(PATTERNS IN 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","");
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();
}
```

PROGRAM(SYMBOL TABLE):

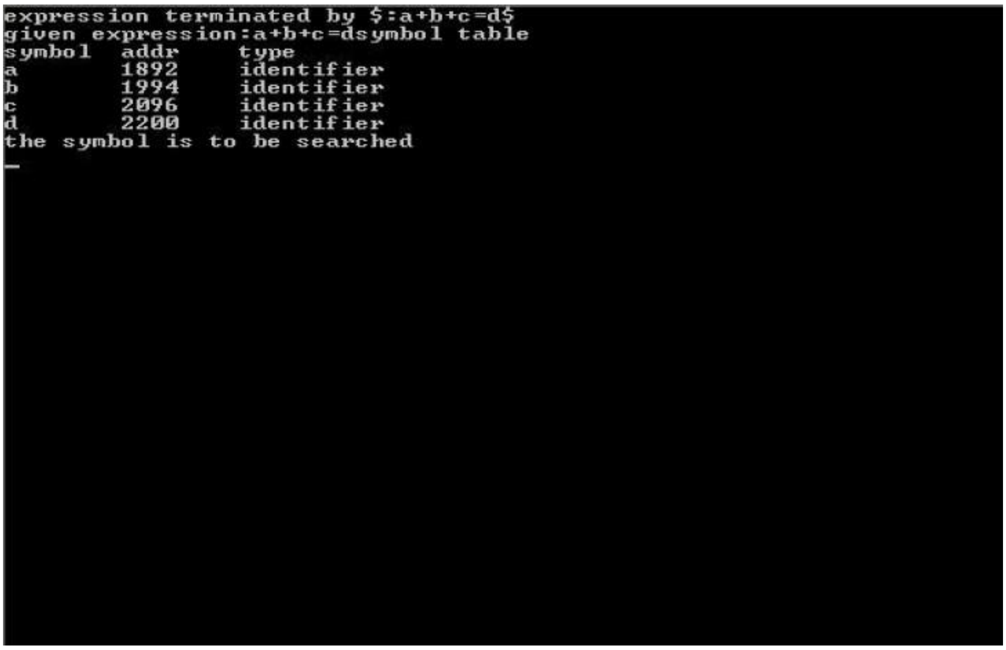
```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<string.h>
#include<math.h>
#include<ctype.h>
void main()
{
int i=0,j=0,x=0,n,flag=0; void *p,*add[15]; char ch,srch,b[15],d[15],c; //clrscr();
printf("expression terminated by $:"); while((c=getchar())!='$') {
b[i]=c; i++;
```

```
}
n=i-1;
printf("given expression:");
i=0;
while(i<=n)
{
printf("%c",b[i]); i++;
}
printf("symbol table\n");
printf("symbol\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("the symbol is to.besearched\n");
srch=getch();
for(i=0;i<=x;i++)
{
if(srch==d[i])
{
printf("symbol found\n");
printf("%c%s%d\n",srch,"@address",add[i]);
flag=1;
}
}
if(flag==0)
printf("symbol not found\n");
getch();
}
```

OUTPUT:

```
Enter the file name : input.c
                LEXICAL ANALYSIS

line : 1
                #        :       preprocessor
                include  :       keyword
                "        :       doublequote
                stdio.h  :       keyword
                "        :       doublequote

line : 2
                #        :       preprocessor
                include  :       keyword
                "        :       doublequote
                conio.h  :       keyword
                "        :       doublequote

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 : 7
                getch    :       identifier
                (        :       openpara
                )        :       closepara
                ;        :       semicolon

line : 8
                }        :       closebrace

line : 9
                $        :       identifier
```



RESULT:

Thus the above program for developing the Lexical analyzer for recognizing the few patterns in C and create a symbol table is executed successfully and the output is verified.

EX. NO:2	**IMPLEMENTATION OF LEXICAL ANALYZER USING LEXTOOL**
Date:	

AIM:

To write a program to implement the Lexical Analyzer using lex tool.

ALGORITHM:

1. Start the program
2. Lex program consists of three parts.
3. Declaration %%
4. Translation rules %%
5. Auxiliary procedure.
6. The declaration section includes declaration of variables, main test, constants and regular
7. Definitions.
8. Translation rule of lex program are statements of the form
9. P1{action}
10. P2{action}
11. .....
12. .....
13. Pn{action}
14. Write program in the vi editor and save it with .1 extension.
15. Compile the lex program with lex compiler to produce output file as lex.yy.c.
16. Eg. $ lex filename.1
17. $gcc lex.yy.c-11
18. Compile that file with C compiler and verify the output.

PROGRAM:

```
#include<stdio.h>

#include<ctype.h>

#include<conio.h>

#include<string.h>
```

```
char vars[100][100];
int vcnt;
char input[1000],c;
char token[50],tlen;
int state=0,pos=0,i=0,id;
char *getAddress(char str[])
{
for(i=0;i<vcnt;i++)
if(strcmp(str,vars[i])==0)
return vars[i];
strcpy(vars[vcnt],str);
return vars[vcnt++];
}
int isrelop(char c)
{
if(c=='+'||c=='-'||c=='*'||c=='/'||c=='%'||c=='^')
return 1;
else
return 0;
}
int main(void)
{
clrscr();
printf("Enter the Input String:");
gets(input);
do
{
c=input[pos];
putchar(c);
switch(state)
{
```

```
case 0:
if(isspace(c))
printf("\b");
if(isalpha(c))
{
token[0]=c;
tlen=1;
state=1;
}
if(isdigit(c))
state=2;
if(isrelop(c))
state=3;
if(c==';')
printf("\t<3,3>\n");
if(c=='=')
printf("\t<4,4>\n");
break;
case 1:
if(!isalnum(c))
{
token[tlen]='\o';
printf("\b\t<1,%p>\n",getAddress(token));
state=0;
pos--;
}
else
token[tlen++]=c;
break;
case 2:
if(!isdigit(c))
```

```
{
printf("\b\t<2,%p>\n",&input[pos]);
state=0;
pos--;
}
break;
case 3:
id=input[pos-1];
if(c=='=')
printf("\t<%d,%d>\n",id*10,id*10);
else{
printf("\b\t<%d,%d>\n",id,id);
pos--;
}state=0;
break;
}
pos++;
}
while(c!=0);
getch();
return 0;
}
```

OUTPUT:

```
Enter the Input String:a+b*c
a       <1,08CE>
+       <43,43>
b       <1,0932>
*       <42,42>
c        <1,0996>
_
```

RESULT:

Thus the program for the exercise on lexical analysis using lex has been successfully executed and output is verified.

EX. NO:3(A)	GENERATE YACC SPECIFICATION FOR A FEW SYNTACTIC CATEGORIES A) ARITHMETIC EXPRESSION RECOGNIZER
Date:	

AIM:

To write a c program to recognize a valid arithmetic expression using YACC.

ALGORITHM :

1. Start the program.
2. Write the code for parser. l in the declaration port.
3. Write the code for the 'y' parser.
4. Also write the code for different arithmetical operations.
5. Write additional code to print the result of computation.
6. Execute and verify it.
7. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
void main()
{ char s[5]; clrscr();
printf("\n Enter any operator:"); gets(s);
switch(s[0])
{
case'>': if(s[1]=='=')
printf("\n Greater than or equal"); else
printf("\n Greater than"); break;
case'<': if(s[1]=='=')
printf("\n Less than or equal");
else
```

```
printf("\nLess than");
break;
case'=':
if(s[1]=='=')
printf("\nEqual to");
else
printf("\nAssignment");
break;
case'!':
if(s[1]=='=')
printf("\nNot Equal");
else
printf("\n Bit Not");
break;
case'&':
if(s[1]=='&')
printf("\nLogical AND");
else
printf("\n Bitwise AND");
break;
case'|':
if(s[1]=='|')
printf("\nLogical OR");
else
printf("\nBitwise OR");
break;
case'+':
printf("\n Addition");
break;
case'-':
printf("\nSubstraction");
```

```
break;
case'*':
printf("\nMultiplication");
break;
case'/':
printf("\nDivision");
break;
case'%': printf("Modulus");
break;
default:
printf("\n Not a operator");
}
getch(); }
```

OUTPUT:

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed successfully and Output is verified.

EX. NO:3(B)	**B) PROGRAM TO RECOGNISE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS**
Date:	

AIM:

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

ALGORITHM:

1. Read the input string, which represents the variable name.

2. Initialize a regular expression pattern for a valid variable:

   - The pattern should start with a letter: [a-zA-Z]

   - Followed by any number of letters (a-zA-Z) or digits (0-9): [a-zA-Z0-9]*

3. Use a regular expression library or function to match the input string against the pattern.

4. If the input string matches the pattern, then it is a valid variable.

5. If there is no match, the input does not represent a valid variable.

6. Report the result, indicating whether the input is a valid variable.

PROGRAM:

variable_test.l

```
%{
/* This LEX program returns the tokens for the Expression */
#include "y.tab.h"
%}
%%
"int " {return INT;}
"float" {return FLOAT;}
"double" {return DOUBLE;}
[a-zA-Z]*[0-9]*{
printf("\nIdentifier is %s",yytext);
return ID;
}
```

```
return yytext[0];
\n return 0;
int yywrap()
{
return 1;
}
```

variable_test.y

```
%{
#include
/* This YACC program is for recognising the Expression*/ %}
%token ID INT FLOAT DOUBLE
%%
D;T L
;
L:L,ID
|ID
;
T:INT
|FLOAT
|DOUBLE
;
%%
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}
yyerror(char*s)
```

{

}

OUTPUT:

```
[root@localhost]#Lex variable_test.I
[root@localhost]#yacc –d variable_test.y
[root@localhost]#gcc lex.yy.c y.tab.c
[root@localhost]#./a.out
int a, b;

Identifier is a
Identifier is b[root@localhost]#
```

RESULT:

Thus the program for the exercise on the syntax using YACC has been executed successfully and Output is verified.

EX. NO:3(C)	**C) PROGRAM TO RECOGNISE A VALID CONTROL STRUCTURES SYNTAX OF C LANGUAGE (FOR LOOP, WHILE LOOP, IF-ELSE, IF-ELSE-IF, SWITCH-CASE, ETC.)**
Date:	

AIM:

To write a program to recognise a valid control structures syntax of C language.

ALGORITHM:

1. Initialize a stack to keep track of control structures.

2. Read the input C program character by character or token by token.

3. For each character or token:

   a. If it's an opening curly brace '{', push it onto the stack.

   b. If it's a closing curly brace '}', pop elements from the stack until you find a matching opening brace. This identifies the end of a control structure.

   c. If it's an 'if' or 'while' or 'for' keyword, push it onto the stack to start a new control structure.

   d. If it's an 'else' keyword, check if the stack contains an 'if' on top. If not, it's an error. Otherwise, pop the 'if' from the stack.

4. After processing the entire input:

   a. If the stack is empty, all control structures are properly nested and balanced.

   b. If the stack is not empty, there are unmatched control structures, indicating an error.

5. Report the result, indicating whether the program has valid control structures.

PROGRAM:

```
#include <stdio.h>

#include <conio.h> // For Turbo C/C++

int main()

{

FILE *file;

char line[1000];

int insideComment = 0;

int insideString = 0;
```

```
int insideCharacter = 0;
clrscr(); // For Turbo C/C++
file = fopen("sample.c", "r");
if (file == NULL) {
printf("Unable to open the file.\n");
return 1;
}
printf("Valid Control Structures:\n");
while (fgets(line, sizeof(line), file) != NULL) {
for (int i = 0; line[i] != '\0'; i++) {
if (line[i] == '/' && line[i + 1] == '*') {
insideComment = 1;
i++;
}
else if (line[i] == '*' && line[i + 1] == '/') {
insideComment = 0;
} else if (line[i] == '/' && line[i + 1] == '/') {
break; // Skip line comments
} else if (line[i] == '"') {
insideString = 1 - insideString;
} else if (line[i] == '\'') {
insideCharacter = 1 - insideCharacter;
} else if (!insideComment && !insideString && !insideCharacter) {
if (strncmp(line + i, "for", 3) == 0) {
printf("For loop: %s\n", line);
break;
} else if (strncmp(line + i, "while", 5) == 0) {
printf("While loop: %s\n", line);
break;
} else if (strncmp(line + i, "if", 2) == 0) {
printf("If statement: %s\n", line);
```

```
break;
} else if (strncmp(line + i, "else if", 7) == 0) {
printf("Else-if statement: %s\n", line);
break;
} else if (strncmp(line + i, "else", 4) == 0) {
printf("Else statement: %s\n", line);
break;
} else if (strncmp(line + i, "switch", 6) == 0) {
printf("Switch statement: %s\n", line);
break;
} else if (strncmp(line + i, "case", 4) == 0) {
printf("Case label: %s\n", line);
break;
} else if (strncmp(line + i, "default", 7) == 0) {
printf("Default label: %s\n", line);
break;
}
}
}
}
fclose(file);
getch(); // For Turbo C/C++
return 0;
}
```

Sample.c:

```
#include <stdio.h>
int main() {
int i;
for (i = 0; i < 10; i++) {
if (i % 2 == 0) {
printf("Even\n");
```

```
}
else{
printf("Odd\n");
}
}
return 0;
}
```

OUTPUT:

Valid Control Structures:

For loop: for (i = 0; i < 10; i++) {

If statement: if (i % 2 == 0) {

Else statement: } else {

RESULT:

Thus the program to recognise a valid control structures syntax of C language was executed and verified successfully.

EX. NO:3(D)	D) PROGRAM TO IMPLEMENT A CALCULATOR USING LEX AND YACC.
Date:	

AIM:

To write a program to implement calculator using LEX and YACC.

ALGORITHM:

1. Initialize a stack to manage the parsing and evaluation of mathematical expressions.

2. Read the mathematical expression or input token by token.

3. a. For each token:

If it's a number or operator, push it onto the stack.

If it's an opening parenthesis '(', push it onto the stack to start a new subexpression.

If it's a closing parenthesis ')':

Pop elements from the stack until you find a matching opening parenthesis '('. This identifies the end of a subexpression.

Evaluate the subexpression and push the result back onto the stack.

If it's an operator (+, -, *, /), ensure proper operator precedence:

Pop operators with higher or equal precedence from the stack and apply them to the operands.

Push the result back onto the stack.

b. Continue processing tokens until the entire expression is parsed.

4. After processing the entire expression, the stack should contain a single result, which is the final calculated value.

5. Report the result as the calculated value of the mathematical expression.

PROGRAM:

```
%{
#include<stdio.h>
int op=0,i;
float a,b;
%}
dig[0-9]+|([0-9]*)"."([0-9]+)
```

```
add "+"
sub  "-"
mul"*"
div "/"
pow "^"
ln \n
%%
{dig}{digi();}
{add}{op=1;}
{sub}{op=2;}
{mul}{op=3;}
{div}{op=4;}
{pow}{op=5;}
{ln}{printf("\n the result:%f\n\n",a);}
%%
digi()
{
if(op==0)
a=atof(yytext);
else
{
b=atof(yytext);
switch(op)
{
case 1:a=a+b;
break;
case 2:a=a-b;
break;
case 3:a=a*b;
break;
case 4:a=a/b;
```

```
break;
case 5:for(i=a;b>1;b--)
a=a*i;
break;
}
op=0;
}
}
main(int argv,char *argc[])
{
yylex();
}
yywrap()
{
return 1;
}
```

OUTPUT:

```
Lex cal.l
Cc lex.yy.c-ll
a.out
4*8
The result=32
```

RESULT:

Thus the program for implementation of calculator using LEX and YACC has been executed successfully and output is verified.

EX. NO:4	GENERATE THREE ADDRESS CODE FOR A SIMPLE PROGRAM USING LEX AND YACC
Date:	

AIM:

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

ALGORITHM:

1. Determine the grammar rules for your programming language. Specify the syntax, including variables, expressions, statements, and control structures.

2. Write a Lex file to tokenize the input source code into meaningful tokens, including keywords, identifiers, operators, and constants.

3. Write a Yacc file to define the syntax rules for your programming language. Specify how to parse expressions, statements, and control structures.

4. Implement a symbol table to keep track of variable names, their types, and their corresponding memory locations.

5. a. Within your Yacc actions, construct a data structure to represent three-address code instructions. Each instruction typically consists of an operator, source operands, and a target variable.

b. As you parse expressions and statements, generate and append three-address code instructions to a list or data structure.

6. Implement a function to print the generated three-address code, which represents the program's logic in a more abstract form.

7. Implement a function to print the generated three-address code, which represents the program's logic in a more abstract form.

PROGRAM:

LEX file:calc.l

```
%{%{

#include "y.tab.h"

%}


%%

int    { return INT; }

return  { return RETURN; }

[ \t]  { /* Ignore whitespace */ }
```

```
\n     { return EOL; }
.      { return yytext[0]; }

%%

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

YACC File:calc.y

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

%token INT
%token RETURN
%token EOL

%%
program:
   INT main EOL statement_list return_statement
   ;

main:
   INT MAIN '(' ')' '{' '}'
   ;

statement_list:
   /* Empty */
   |
   statement statement_list
```

```
  ;

statement:
  INT ID '=' expr EOL { printf("%s = %s %s %s\n", $2, $4, $1, $6); }
  ;

expr:
  ID     { $$ = $1; }
  |
  NUM    { $$ = $1; }
  |
  expr '+' expr  { $$ = $1 + $3; }
  |
  expr '-' expr  { $$ = $1 - $3; }
  ;

return_statement:
  RETURN NUM EOL  { printf("return %s\n", $2); }
  ;

%%

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

C program:

```
int main() {
  int a, b, c;
  a = 5;
  b = 3;
```

```
    c = a + b;
    return 0;
}
```

OUTPUT:

```
a = 5
b = 3
t1 = a + b
return 0
```

RESULT:

Thus the generation of three address code using LEX and YACC was executed and verified successfully.

EX. NO:5	**IMPLEMENTATION OF TYPE CHECKING**
Date:	

AIM:

To write a C program for implementing type checking for 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.

PROGRAM:

```
#include<stdio.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])
{
opstr[j]='<';
j++;
}
else
{
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;

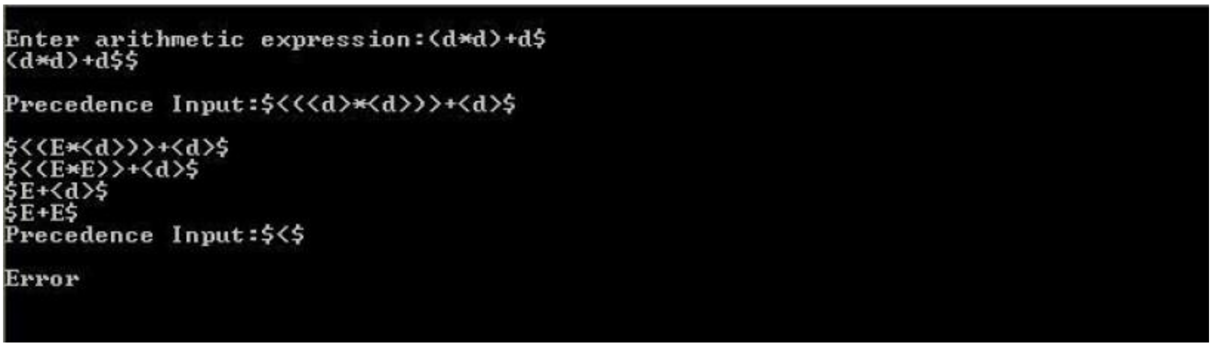
sue:

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 program to implement type checking has been executed successfully and Output is verified.

EX. NO:6	**IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUES**
Date:	

AIM:

To write a C program to implement simple code optimization techniques.

ALGORITHM:

1. Start the program

2. Declare the variables and functions.

3. Enter the expressionand state it in the variable a, b, c.

4. Calculate the variables b & c with ‘temp’ and store it in f1 and f2.

5. If(f1=null && f2=null) then expression could not be optimized.

6. Print the results.

7. Stop the program.

PROGRAM:

Before:

Using for :

```
#include<iostream.h> #include <conio.h>
int main()
{
int  i,  n;
int fact=1;
cout<<"\nEnter a number: "; cin>>n;
for(i=n;i>=1;i--) fact=fact *i;
cout<<"The factoral value is: "<<fact; getch();
return 0;
}
```

After:

Using do-while:

```
#include<iostream.h> #include<conio.h> void main()
{
clrscr(); int n,f; f=1;
```

```
cout<<"Enter the number:\n"; cin>>n;
do
 {
f=f*n; n--;
}while(n>0);
cout<<"The factorial value is:"<<f; getch();
}
```

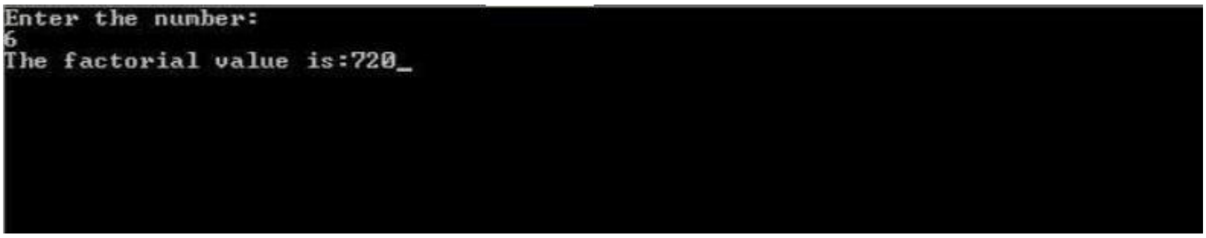
OUTPUT:

Before:

Using for :

After:

Using do-while:



RESULT:

Thus the Simple Code optimization technique is successfully executed.

EX. NO:7	**IMPLEMENT THE BACK END OF THE COMPILER**
Date:	

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. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.

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 program was implemented to the TAC has been successfully executed.