Ex. No: 1	IMPLEMENTATION OF SYMBOL TABLE MANAGEMENT

AIM:

Write a C program to implement symbol table management.

ALGORITHM:

1. Start the program for performing Create, Search, Modify and Display 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, create the symbol table with Label name, Address, Opcode and Operand.
5. If the the symbol is already present, it displays error message. Otherwise create the label and the corresponding address in the symbol table.
6. If the entered choice is 2, then search the given label is present or not. If it is present to display the corresponding label with address. Else print the Error message.
7. If the entered choice is 3, to perform modification operation in symbol table. That is to enter the address to be modified. Then to modify the label name and operand with their corresponding address.
8. If the entered choice is 4, then to display all the contents present in the symbol table. If the choice is 5, then to exit the process.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void create();
void search();
void modify();
void display();
struct symbol
{
char label[10],address[10],value[5];
}s[10];
struct instructions
{
char address[10],label[10],opcode[10],operand[10];
}t;
int n=0,i,flag=0;
char a[10],ch;
void main()
{
```

```
int opt;
clrscr();
ch='y';
printf("\n Symbol Table Management");
do
{
printf("1.create\n2.search\n3.modify\n4.display\n5.exit\n");
printf("\n Enter your choice\n");
scanf("%d",&opt);
switch(opt)
{
case 1:create();
break;
case 2:search();
break;
case 3:modify();
break;
case 4:display();
break;
case 5:exit(0);
break;
}
}
while(opt<5);
getch();
}
void create()
{
printf("\nEnter the address,label,opcode,operand\n");
scanf("%s%s%s%s",&t.address,&t.label,&t.opcode,&t.operand);
if(strcmp(t.label,"_")!=0)
{
flag=0;
for(i=0;i<=n;i++)
{
if(strcmp(s[i].label,t.label)==0)
{
flag=1;
break;
}
}
if(flag==0)
{
strcpy(s[n].address,t.address);
strcpy(s[n].label,t.label);
strcpy(s[n].value,t.operand);
n++;
}
else
{
```

```
printf("\nThe label is already there");
}
}
}
void search()
{
printf("\nEnter the label");
scanf("%s",&a);
for(i=0;i<=n;i++)
{
if(strcmp(a,s[i].label)==0)
{
flag=1;
break;
}
}
if(flag==1)
{
printf("\t\tlabel\taddress\tvalue\t\n");
printf("\t\t%s\t%s\t%s\n",s[i].label,s[i].address,s[i].value);
}
else
{
printf("Not Found");
}
}
void modify()
{
flag=0;
printf("\nEnter the label\n");
scanf("%s",&a);
for(i=0;i<=n;i++)
{
if(strcmp(a,s[i].label)==0)
{
flag=1;
break;
}
}
if(flag==0)
{
printf("\nError");
}
else
{
printf("\nEnter the address,value\n");
scanf("%s%s",&s[i].address,&s[i].value);
}
}
void display()
```

```
        {
        printf("\t\tlabel\taddress\tvalue\t\t\n");
        for(i=0;i<=n;i++)
        {
        printf("\t\t%s\t%s\t%s\t\t\n",s[i].label,s[i].address,s[i].value);
        }
        }
```

OUTPUT:

```
        Implementation of symbol table managment
1.create
2.search
3.modify
4.display
5.exit

Enter your choice :1

Enter the address,label,opcode,operand
   1000   L1  MOV  a,b

1.create
2.search
3.modify
4.display
5.exit

Enter your choice:1

Enter the address,label,opcode,operand
    1005   L2  LAD  R1,R2

1.create
2.search
3.modify
4.display
5.exit

Enter your choice:2

Enter the Label:L1


                    Label               Address           Value
                     L1                  1000               a,b

1.create
2.search
```

```
3.modify
4.display
5.exit

Enter your choice:3

Enter the label:L1


Enter the address,value

   2000     A,B

1.create
2.search
3.modify
4.display
5.exit

Enter your choice:4

      Label          address            Value
      L1              2000               A,B


1.create
2.search
3.modify
4.display
5.exit

Enter your choice:5
Exit.
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the program for implementing symbol Table Management was executed successfully and the output was verified.

Ex.No: 2	IMPLEMENTATION OF LEXICAL ANALYZER USING C

AIM:

To write a C program to implement the Lexical Analyzer which displays the output like identifiers, keywords, operators, punctuators and Constants.

ALGORITHM:

1. Get the file name and open the file in read mode.
2. Check file not exist, if true print the error message and stop.
3. Check the end of file, if true ends the process.
4. Otherwise read the one line of text.
5. Check the token if keywords print the token and type is Keyword.
6. If identifier print the token and type is Identifier.
7. If constant print the token and type is Constant.
8. Otherwise print the token and type is Punctuator.
9. Repeat the step 3.
10. Stop the process.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
static char op[7]={'+','-','*','<','>','='};
static char h[13][20]={"int","char","float","long","while","main","scanf","printf","if","for","else"};
int iskeyword(char*);
int isoperator(char);
void main()
{
FILE *fp;
char fname[10],str[100],token[100];
char tokenname[100][200],tokentype[100][200];
int i,j,l,k=0;
clrscr();
printf("enter the filename:\t");
scanf("%s",fname);
fp=fopen(fname,"r");
if(fp==NULL)
{
printf("file does not exist");
```

```
exit(0);
}
while(!feof(fp))
{
fgets(str,80,fp);
i=0;
while (str[i]!='\0')
{
j=0;
if(isalpha(str[i]))
{
token[j++]=str[i++];
while ((isalpha(str[i]))||(isdigit(str[i])))
{
token[j++]=str[i++];
}
token[j]='\0';
if(iskeyword(token))
{
strcpy(tokenname[k],token);
strcpy(tokentype[k++],"keyword");
}
else
{
strcpy(tokenname[k],token);
strcpy(tokentype[k++],"identifier");
}
}
else if(isdigit(str[i]))
{
token[j++]=str[i++];
while((isdigit(str[i]))||(isdigit(str[i]=='.')))
{
token[j++]=str[i++];
}
token[j]='\0';
strcpy(tokenname[k],token);
strcpy(tokentype[k++],"constant");
}
else if(isoperator(str[i]))
{
token[j++]=str[i++];
if(isoperator(str[i]))
token[j++]=str[i++];
token[j]='\0';
strcpy(tokenname[k],token);
strcpy(tokentype[k++],"operator");
}
else if(str[i]==' ')
{
```

```
while(str[i]==' ')
i++;
}
else
{
token[j++]=str[i++];
token[j]='\0';
strcpy(tokenname[k],token);
strcpy(tokentype[k++],"punctuator");
}
}
}
printf("TOKENNAME \t TOKENTYPE\n");
for(l=0;l<k;l++)
{
printf("%s\t\t%s",tokenname[l],tokentype[l]);
printf("\n");
}
getch();
}
int iskeyword(char*token)
{
int m;
for(m=0;m<11;m++)
{
if((strcmp(token,h[m])==0))
return 1;
}
return 0;
}
int isoperator(char ch)
{
int n;
for(n=0;n<7;n++)
{
if(ch==op[n])
return 1;
}
return 0;
}
```

INPUT FILE:

Sum.c

```
int max(int a,int b);
```

OUTPUT:

```
Enter the file name:Sum.c

Token name                    Token type

int                           keyword
max                           identifier
(                             punctuator
int                           keyword
a                             identifier
,                             punctuator
int                           keyword
b                             identifier
)                             punctuator
;                             punctuator
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the C Program for implementation of Lexical Analyzer was executed successfully and the output was verified.

Ex. No : 3a	IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL

AIM:

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

ALGORITHM:

1. Define the Regular Expression for relational operators.
2. In Rule section, rule1 defines all the keywords and its corresponding action using yytext () function.
3. In rule 2 the identifiers followed by open parenthesis and define the action is FUNCTION using yytext () function.
4. Define the opening and closing curly braces for the action using yytext () function of block begins and ends respectively.
5. Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.
6. A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called tokenization, and the lexer categorizes them according to a symbol type.
7. Otherwise open the file in read mode, then call the yylex () function for scanning the given input file and to display the tokens in the form of name and type of the tokens.

```
/* LEX Program to recognize tokens */

%{
#define LT 256
 #define LE 257
#define EQ 258
 #define NE 259
 #define GT 260
 #define GE 261
#define RELOP 262
 #define ID 263
 #define NUM 264
#define IF 265
 #define THEN 266
 #define ELSE 267
 int attribute;
%}
 delim [ \ t\n]
 ws {delim}+
 letter [A -Za -z]
digit [0 -9]
 id {letter}({letter}|{digit})*
num {digit}+( \.{digit}+)?(E[+ -]?{digit}+)?
%%
 {ws} {}
 if { return(IF); }
 then { return(THEN); }
else { return(ELSE); }
{id} { return(ID); }
{num} { return(NUM); }
 "<" { attribute=LT;return(RELOP); }
 "<=" { attribute=LE;return(RELOP); }
 "<>" { attribute=NE;return(RELOP); }
 "=" { attribute=EQ;return(RELOP); }
 ">" { attribute=GT;return(RELOP); }
 ">=" { attribute=GE;return(RELOP); }
 %%
int yywrap()
{
 return 1;
}
 int main()
{
int token;
while(token=yylex())
{
printf("<%d,",Token);
switch(token){
case ID:caseNUM:
printf("%s>\n",yytext);
break;
case RELOP:
printf("%d>\n",attribute);
break;
default:printf(")\n");
break;
}
}
return 0;
}
```

OUTPUT:

```
[3cse10@localhost ~]$ cc lex.yy.c -ll
[3cse10@localhost ~]$ ./a.out
if a>b then a else b
<265>
<263,a>
<262,260>
<263,b>
<266>
<263,a>
<267>
<263,b>
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the Program for implementation of Lexical Analyzer using LEX Tool was executed successfully and the output in the form of token name and its attribute was displayed.

Ex. No : 3b	IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL

AIM:

To write a C program to implement the Lexical Analyzer which displays the output like identifiers, keywords, operators, punctuators and Constants. By using LEX tool.

ALGORITHM:

5. Define the Regular Expression for identifiers, constants and white space.

6. In Rule section, rule1 defines all the keywords and its corresponding action using yytext () function.

7. In rule 2 the identifiers followed by open parenthesis and define the action is FUNCTION using yytext () function.

8. Define the opening and closing curly braces for the action using yytext () function of block begins and ends respectively.

9. Define constant, identifiers, strings and their corresponding actions using yytext () function.

6. Define the main function, pre-processor directives and operators like arithmetic, relational and bitwise operators and their corresponding action parts using yytext () function.

7. Define the rules for punctuators and display their corresponding action using yytext () Function.

8. Check if the number of argument is less than one, then print Error message and stop the Process.

9. Otherwise open the file in read mode, then call the yylex () function for scanning the given input file and to display the tokens in the form of name and type of the tokens.

PROGRAM:

```
%{
%}
identifier [a-z A-Z][a-z A-Z 0-9]*
constant [+-]?[0-9][0-9]*
ws[ \t\n]*
%%
#.* {printf("\n\t%s is PREPROCESSOR DIRECTIVE.",yytext);}
int|
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const|
else |
return |
printf|
scanf|
goto {printf("\n\t%s is a KEYWORD",yytext);}
if[\t]*\( {printf("\n\tif is a KEYWORD");}
\/\*.*\*\/ |
\/\/.* {printf("\n\n\t%s is a COMMENT",yytext);}
[a-z]*\(\)|
[a-z]*(\[a-z*\])*  {printf("\n\t%s is a FUNCTION",yytext);}
\{{printf("\n\t%s is a BLOCK BEGINS",yytext);}
\} {printf("\n\t%s is a BLOCK ENDS",yytext);}
[A-Za-z](\[0-9*\])* {printf("\n\t%s is IDENTIFIER",yytext);}
\".*\" {printf("\n\n\t%s is a STRING",yytext);}
[0-9]+ {printf("\n\t%s is NUMBER",yytext);}
\= {printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\<|
\== |
\!= |
\>{printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
\+ |
\- |
\* |
\/ |
\% |
\^ {printf("\n\t%s is a ARITHMETIC OPERATOR",yytext);}
\, |
\. |
\; |
\: {printf("\n\t%s is a PUNCTUATOR",yytext);}
\&&|
\| |
\! {printf("\n\t%s is a LOGICAL OPERATOR",yytext);}
\({printf("\n\t%s is a OPENING PARENTHESIS",yytext);}
\) {printf("\n\t%s is a CLOSING PARENTHESIS",yytext);}
\& |
\$ {printf("\n\t%s is a SPECIAL CHARACTER",yytext);}
\.|\n;
%%

int main(intargc,char **argv)
{
	if(argc>1)
	{
	FILE *file;
		file=fopen(argv[1],"r");
		if(!file)
	{
			printf("\nCould not open the file %s.\n",argv[1]);
			exit(0);
	}
		yyin=file;
	}
	yylex();
	printf("\n\n");
	return 0;
}
int yywrap()
{
	return 0;
}
```

INPUT FILE:

```
[cse13@LinSrvr:~] $ vi input.c
```

INPUT.C

```
/* sum of n numbers */
#include<stdio.h>
#include<conio.h>
void main()
{
int sum(int n)
{
int i, sum=0;
for(i=0; i<=n; i++)
{
sum=sum+i;
}
printf("The sum is %d", sum);
}
```

OUTPUT:

```
[cse13@LinSrvr:~] $ lex lextool.l
```

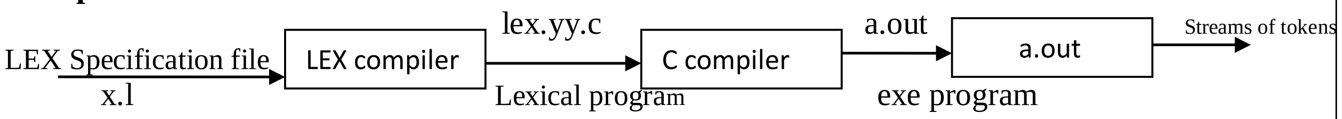
```
[cse13@LinSrvr:~] $ cc lex.yy.c
[cse13@LinSrvr:~] $ ./a.out input.c

/* sum of n numbers */              is a COMMENT
#include<stdio.h>                   is PREPROCESSOR DIRECTIVE
#include<conio.h>                   is PREPROCESSOR DIRECTIVE
void                                is a KEYWORD
main()                              is a FUNCTION
{                                   is a BLOCK BEGINS
int                                 is a KEYWORD
sum                                 is IDENTIFIER
(                                   is a OPENING PARENTHESIS
int                                 is a KEYWORD
n                                   is IDENTIFIER
)                                   is a CLOSE PARENTHESIS
{                                   is a BLOCK BEGINS
int                                 is a KEYWORD
i                                   is IDENTIFIER
,                                   is a PUNCTUATOR
sum                                 is  IDENTIFIER
=                                   is an ASSIGNMENT OPERATOR
0                                   is NUMBER
;                                   is a PUNCTUATOR
for                                 is a KEYWORD
(                                   is a OPEN PARENTHESIS
i                                   is IDENTIFIER
=                                   is an ASSIGNMENT OPERATOR
0                                   is a NUMBER
;                                   is a PUNCTUATOR
i                                   is IDENTIFIER
<=                                  is a RELATIONAL OPERATOR
n                                   is IDENTIFIER
;                                   is a PUNCTUATOR
i                                   is IDENTIFIER
+                                   is an ARITHMETIC OPERATOR
+                                   is an ARITHMETIC OPERATOR
)                                   is a CLOSE PARENTHESIS
{                                   is a BLOCK BEGINS
sum                                 is IDENTIFIER
=                                   is an ASSIGNMENT OPERATOR
sum                                 is IDENTIFIER
+                                   is an ARITHMETIC OPERATOR
i                                   is IDENTIFIER
;                                   is a PUNCTUATOR
```

Description:

Lex :

- Lex is One of the UNIX utility tools which generates the lexical analyzer.
- Lex scans the source program to get the stream of tokens.
- Lex is very much faster in finding the tokens as compared to the handwritten LEX program in C

Lex specification:



Lex file can be created using the extension .l (dot L). In the above diagram, The lex file can be x.l is given to the LEX compiler to produce lex.yy.c. This lex.yy.c file consists of the tabular representation of the transition diagram constructed for the regular expression . f
Finally, lex.yy.c generates an object program a.out which generates streams of tokens.

Terminologies with LEX:

1. Declaration section
2. Rule Section
3. Procedure section

```
%{
    Declaration Section
%}
%%
    Rule Section
%%
Procedure section
```

In **Declaration Section** variable constants are declared
In **Rule section** regular expressions with associate actions are defined
In **procedure section** required procedures and functions are defined

*****	Matches with Zero or more occurrences of preceding expression
.	Matches any single character other than new line character
[a-z]	Matches with any alphabet in lower case
+	Matches with one or more occurrences of preceding expressions

^	Matches the beginning of a line a first character
|	Represents either or

Lex actions:

BEGIN : It indicates the start state.
ECHO: It emits the input as it is.
yytext: The recognized tokens from the input token is stored in yytext().
yylex( ): when scanner encounters yylex ( ) it stats scanning the source program.
yywrap( ): This function is called when the end of the file is encountered.
yyin : Standard input file stores the input source program.
yyleng : It stores the length or number of tokens in the input string.

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the Program for implementation of Lexical Analyzer using LEX Tool was executed successfully and the output in the form of token name and its types was displayed.

GENERATION OF YACC TOOL SPECIFICATIONS FOR A FEW SYNTATIC CATEGORIES.

Ex.No:4a	RECOGNIZING A VALID ARITHMETIC EXPRESSION THAT USES OPERATORS LIKE +, - , * AND /.

AIM:

Write a Program to recognize a valid arithmetic expression that uses operators like +, -, * and / using LEX and YACC Tools.

ALGORITHM:

1. Define the structure of Lex program to perform the validation of arithmetic expression.
2. Declare the function y.tab.h for storing the tokens to be used in YACC program.
3. Define the regular expressions for identifiers and arithmetic operators. Define the rules for those corresponding rules and return the tokens ad ID and KEYWORD.
4. Define YACC program, declare the tokens used in this program by using y.tab.h function used in Lex program.
5. Write the rules / Context Free Grammar for each expression (E) and declare the corresponding actions for each rules.
6. Define main function and get the statement to be parsed and call yyparse () function for matching the tokens with rules.
7. While token does not having sentinel form (eof) to declare Error message. Otherwise to print the result as the given expression is the valid arithmetic expression.

PROGRAM:

LEX PROGRAM:

```
%{
#include "y.tab.h"
%}
%%
printf("Enter your Arithmetic Expression:\n");
"=" {printf("\n %s is an Equal to Operator",yytext);}
"+" {printf("\n %s is an Addition Operator",yytext);}
"-" {printf("\n %s is a Subtraction Operator",yytext);}
"*" {printf("\n %s is a Multiplication Operator",yytext);}
"/" {printf("\n %s is a Division Operator",yytext);}
";" {printf("\n %s is a Puncutator",yytext);}
[a-zA-Z]*[0-9]* {printf("\n %s is an Identifier",yytext);
                return ID;
                }
. return yytext[0];
\n return 0;
%%
int yywrap()
{
return 1;
}
```

YACC PROGRAM:

```
%{
#include<stdio.h>
%}
%token A ID
%%
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:

```
[root@localhost ~]# lex exp4a.l
[root@localhost ~]# yacc -d exp4a.y
[root@localhost ~]# vi y.tab.h
[root@localhost ~]# vi y.tab.c
[root@localhost ~]# cc lex.yy.c y.tab.c -ll -lm
[root@localhost ~]# ./a.out

c=a+b*d/s; (input to be given)

 c is an Identifier
 = is an Equal to Operator
 a is an Identifier
 + is an Addition Operator
 b is an Identifier
 * is a Multiplication Operator
 d is an Identifier
 / is a Division Operator
```

```
s is an Identifier
; is a Puncutator
Valid Arithmetic Expression
```

[root@localhost ~]# ./a.out

c=a<b/d>r; (input to be given)

```
c is an Identifier
= is an Equal to Operator
a is an Identifier
b is an Identifier
/ is a Division Operator
d is an Identifier
r is an Identifier
; is a Puncutator
Valid Arithmetic Expression
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

<u>RESULT:</u>

Thus the program for recognizing valid arithmetic expression that uses operators like +, -, *, = and / was executed successfully and the output was verified.

Ex.No:4b	Recognizing a Variable which starts with a Letter and followed by any number of Letter(s) or digit(s).

AIM:

Write a program to recognize a variable which starts with a letter and followed by any number of letter(s) or digit(s).

ALGORITHM:

1. Define a Lex program and declare the function y.tab.h for storing the tokens to be used in Yacc program.
2. Define the regular expression for keywords and identifiers. Write the rules for Keywords and Identifiers then return the tokens as KEYWORD and ID respectively.
3. Define Yacc program and declare the tokens mentioned in Lex program.
4. Write the production rules for declarative statement which starts with a Letter.
5. Define the production rules for Non-terminals used in declarative statements and define their corresponding action for each Non-terminals present in the rules.
6. Define main () function and get the input statement to be parsed. To call the function yyparse () to match with corresponding rules.
7. While the input statement does not having EOF, to declare Error message. Otherwise to print the result as the given input statement starts with a variable and followed by any number of letter (s) or digit(s).

PROGRAM:

LEX PROGRAM:

```
%{
#include "y.tab.h"
%}
%%
int |
float |
double |
if |
void |
main |
char {printf("\n %s is a Keyword",yytext);}
[a-zA-Z]*[0-9]* {printf("\n %s is an Identifier",yytext);
            return ID;
            }
.             return yytext[0];
\n            return 0;
%%
int yywrap()
{
return 1;
}
```

YACC PROGRAM:

```
%{
#include<stdio.h>
%}
%token ID INT FLOAT DOUBLE CHAR
%%
D:T L
;
L:L,ID
|ID
;
T:INT
|FLOAT
|CHAR
|DOUBLE
;
%%
extern FILE *yyin;
main()
{
do
{
yyparse();
}while(!feof(yyin));
}
yyerror(char *s)
{
}
```

OUTPUT:

```
[root@localhost ~]# lex exp4b.l
[root@localhost ~]# yacc -d exp4b.y
[root@localhost ~]# cc lex.yy.c y.tab.c
[root@localhost ~]# cc lex.yy.c y.tab.c -ll -lm
[root@localhost ~]# ./a.out

int a,s,d,f;

 int is a Keyword
 a is an Identifier
 s is an Identifier
 d is an Identifier
 f is an Identifier


[root@localhost ~]# ./a.out
```

```
Char  a,f=45;

Char  is a Keyword
 a is an Identifier
 f is an Identifier
 45 is an Identifier
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the program for recognizing a variable which starts with a Letter and followed by any number of Letter(s) or digit(s) was executed successfully and the output was verified.

Ex.No:4c	Control Structures Syntax of C Language (for Loop, While Loop, if-else, if-else-if,switch-case,etc.)

AIM:

Develop a Lex Program To Check The Following Control Structures Syntax of C Language Like for Loop, While Loop, if-else, if-else-if, Switch-case Statement.

ALGORITHM:

Step -1: write a context free grammar for the below control structure syntax of c language like for loop, while loop, if-else, if-else-if and switch case statements.
Step-2: write the lex coding to generate tokens of the syntax.
Step – 3: verify the syntaxes of looping and conditional statements, and display the result.

FOR LOOP – LEX PROGRAM:

```
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 PROGRAM:

```
%{
#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:

```
[root@localhost ~]$ lex for.l
[root@localhost ~]$ yacc for.y
conflicts: 25 shift/reduce, 4 reduce/reduce
[root@localhost ~]$ gcc y.tab.c -ll -ly
[root@localhost ~]$ ./a.out
Enter the expression:
for(i=0;i<n;i++)
i=i+1;
Input accepted
[root@localhost ~]$
```

WHILE LOOP – LEX PROGRAM:

```
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];
```

```
%%
```

YACC PROGRAM:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM WHILE LE GE EQ NE OR AND
%right '='
%left AND OR
%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
        | 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:

```
[root@localhost ~]$ lex wh.l
[root@localhost ~]$ yacc wh.y
conflicts: 2 shift/reduce
[root@localhost ~]$ gcc y.tab.c -ll -ly
[root@localhost ~]$ ./a.out
Enter the exp: while(a>1){ b=1;}
Input accepted.
```

SIMPLE IF STATEMENT - LEX PROGRAM

```
ALPHA [A-Za-z]
DIGIT [0-9]
%%
[ \t\n]
if              return IF;
then                return THEN;
{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:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM IF THEN LE GE EQ NE OR AND
%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';'
    ;
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 statement: ");
yyparse();
}
```

Output:

```
[root@localhost ~]$ lex if.l
[root@localhost ~]$  yacc if.y
[root@localhost ~]$  gcc y.tab.c -ll -ly
[root@localhost ~]$  ./a.out
Enter the statement: if(i>) then i=1;
syntax error
[root@localhost ~]$  ./a.out
Enter the statement: if(i>8) then i=1;
Input accepted.
[root@localhost ~]$
```

IF –THEN –ELSE STATEMENT – LEX PROGRAM:

```
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:

```
%{
#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:

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

SWITCH CASE STATEMENT – LEX PROGRAM:

```
alpha [a-zA-Z]
digit [0-9]

%%

[ \n\t]
if          return IF;
then        return THEN;
while       return WHILE;
switch      return SWITCH;
case        return CASE;
default     return DEFAULT;
break       return BREAK;
```

```
{digit}+       return NUM;
{alpha}({alpha}|{digit})* return ID;
"<="           return LE;
">="           return GE;
"=="           return EQ;
"!="           return NE;
"&&"           return AND;
"||"           return OR;
.              return yytext[0];

%%
```

YACC PROGRAM:

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

%token ID NUM SWITCH CASE DEFAULT BREAK LE GE EQ NE AND OR IF THEN
WHILE
%right '='
%left AND OR
%left '<' '>' LE GE EQ NE
%left '+''-'
%left '*''/'
%right UMINUS
%left '!'

%%

S    :    ST{printf("\nInput accepted.\n");exit(0);};
     ;
ST  :    SWITCH'('ID')''{'B'}'
     ;
B    :    C
     |    C D
     ;
C  :    C C
   |    CASE NUM':'ST1 BREAK';'
   ;
D  :    DEFAULT':'ST1 BREAK';'
   |    DEFAULT':'ST1
   ;
```

```
ST1   :   WHILE'('E2')' E';'
    |   IF'('E2')'THEN E';'
    |   ST1 ST1
    |   E';'
    ;
E2   :   E'<'E
    |   E'>'E
    |   E LE E
    |   E GE E
    |   E EQ E
    |   E NE E
    |   E AND E
    |   E OR 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 AND E
    |   E OR E
    |   ID
    |   NUM
    ;

%%

#include"lex.yy.c"

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

Output:

```
[root@localhost ~]$  lex pars.l
[root@localhost ~]$  yacc pars.y
conflicts: 5 shift/reduce
[root@localhost ~]$  gcc y.tab.c -ll -ly
[root@localhost ~]$  ./a.out

Enter the expression: switch(s)
{
case 1:a=b+c;break;
case 2:    if(a<10)
   then a=b*c;
   break;
case 3:    while(a<5)
   b=b+a;
   break;
}

Input accepted.
[root@localhost ~]$
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

Result :

Thus the above Lex And Yacc Programs for Syntax of Looping And Conditional Statements of C Language Was Executed Successfully.

Ex.No:4d	IMPLEMENTATION OF SIMPLE CALCULATOR USING LEX AND YACC Tools

AIM:

Develop a program to implement Simple Calculator using LEX and YACC Tools.

ALGORITHM:

1. Define the Lex program and declare the function y.tab.h for storing the tokens to be used in Yacc program.
2. Define the regular expression for numbers in the form of integer, floating point and exponential then return their corresponding token as NUM.
3. Write the rules for trigonometric functions like SIN, COS, TAN and logarithmic functions like LOG and nLOG to perform trigonometric calculations.
4. If memory is selected for performing their operations and return their token as MEM.
5. Define the precedence of operators and tokens used in the Yacc program.
6. If the given statement is an expression then solve that 'expr' by using temporary variables.
7. Read the expression from user by using getchar () function. If the character is equal to space then check for '.' (or) digit and scan the character in yylval and return the number and character. Otherwise print the Syntax Error.

PROGRAM:

LEX PROGRAM:

```
% {
  int op = 0,i;
  float a, b;
% }

dig [0-9]+|([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul "*"
div "/"
pow "^"
ln \n
%%
```

```
/* digi() is a user defined function */
{dig} {digi();}
{add} {op=1;}
{sub} {op=2;}
{mul} {op=3;}
{div} {op=4;}
{pow} {op=5;}
{ln} {printf("\n The Answer :%f\n\n",a);}

%%
digi()
{
 if(op==0)

/* atof() is used to convert
      - the ASCII input to float */
 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:

```
[root@localhost ~]# lex exp4c.l
[root@localhost ~]# yacc -d exp4c.y
[root@localhost ~]# cc lex.yy.c y.tab.c
[root@localhost ~]# cc lex.yy.c y.tab.c -ll -lm
[root@localhost ~]# ./a.out

Enter the expression: 3*4
Answer = 12
Enter the expression: 2-4
Answer = -2
Enter the expression: 2+-2
Answer = 0
Enter the expression: -2+-2
Answer = -4
Enter the expression: 12/3
Answer = 4
Enter the expression: 3+4*10;
Syntax Error
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the program for simple calculator using LEX and YACC tools was executed successfully and the output was verified.

Ex.No:5	**Generate Three Address Code for simple Program using LEX and YACC**

AIM:

Write a program to generate three address code for simple program using LEX and YACC Tools.

ALGORITHM:

1. Define Lex program, for writing regular expressions for identifiers and constants.
2. Define the rules for identifiers, constants and return the tokens as ID and NUM.
3. Define the rules for relational operators and return the token as RELOP.
4. Define the Yacc program for declaring the structure with member operator, argument 1, argument 2 and result.
5. Declare the stack with item of array and top pointer.
6. Declare the tokens and operators associatively used for the program.
7. Define the production rules for the C statements like declaration statement, conditional statement, execution statement and define the action of each rule.
8. If the statement is IF statement then creates the entry to the intermediate code table and PUSH the current entry number.
9. If the statement is conditional statement then POP the entry.
10. If the statement is else statements then POP the entry.
11. Define the main () function. Enter the input program to be generating the intermediate code. And the yyparse () function to match the input for corresponding rules in BNF form.
12. If the matching is found correct to display the result in Quadruple format. Otherwise to declare the Error message.

PROGRAM:

LEX PROGRAM:

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

%%

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

%%
```

YACC PROGRAM:

```
%{
#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("Errror %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:

```
[root@localhost ~]# yacc -d yacc.y
[root@localhost ~]# lex  lex.l
[root@localhost ~]# gcc y.tab.c lex.yy.c -w
[root@localhost ~]# ./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
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the program for generation of three address code was executed successfully and output was verified.

Ex.No:6a	**IMPLEMENTATION OF TYPE CHECKING**

AIM:

Develop a C program to implement type checking operation for the context free grammar.

ALGORITHM:

1. Start the program and display the available production rules.
2. Read the input expression from the user.
3. Display the input expression and call the function e (). Print the production rule E→TE’ and call t (str), e1(str), t().
4. Print the production rule T→Ft’ and call f(str), t1(str), f() functions for checking the required expressions.
5. Print the production F→id / digit by reading the digit / Character from user and Increment pointer variable by 1 and then call the function t1().
6. Check whether input character matches with ‘*’, If it is true to print the production T’→*FT’ and increment pointer variable by 1.
7. Call the function f(str) and t(str). Otherwise if str[i] matches with ‘/’ then print the rule T’→/TF’ and then increment the pointer variable by 1.
8. Call the functions f(str), t1(str) else print T’→ε. Then call the function e1().
9. If input symbol matches with ‘+’ Operator then print the production E’→+TE’ then increment the pointer variable by 1.
10. Similarly check for ‘-‘, ‘=’ and print the production E’→=TE’ then increment the pointer variable by 1.
11. Call the functions t(str), E1(str). Else print E’→ε. Finally check the string that do not belong to any of this function then print Error message.

PROGRAM:

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

int i=0;
void e(char *);
void t(char *);
void f(char *);
void t1(char *);
void el(char *);

void e(char *str)
{
        printf("\nE→TE'\n");
```

```
		t(str);
		el(str);
	}
	void t(char *str)
	{
		printf("T→FT’ \n");
		f(str);
		t1(str);
	}
	void f(char *str)
	{
		printf("\nF→");
		while((isdigit(str[i]))||(isalpha(str[i])))
		{
			printf("%c",str[i]);
			printf(" ");
			i++;
		}
		printf("\n");
	}
	void t1(char *str)
	{
		if(str[i]=='*')
		{
			printf("T'→*FT'\n");
			i++;
			f(str);
			t1(str);
		}
		else if(str[i]=='/')
		{
			printf("T'→/FT'\n");

			i++;
			f(str);
			t1(str);
		}
		else
			printf("T'→e\n");
	}

	void el(char *str)
	{
		if(str[i]=='+')
		{
			printf("E'→+TE'\n");
			i++;
			t(str);
			el(str);
		}
```

```
        else if(str[i]=='-')
        {
                printf("E'→-TE'\n");
                i++;
                t(str);
                el(str);
        }
        else
                printf("E'→e\n");
}

void main()
{
        int a;
        char *str;
        clrscr();
        printf("available production\n");
        printf("E→TE'\n T→FT'\nF→id\nF→digit\nT'→e\nT'→/FT'\n
                T'→*FT'\nE'→e\nE'→-TE'\nE'→+TE'\n");

        printf("Enter the input symbol\n");
        gets(str);
        printf("The input symbol is: %s\n",str);
        printf("Sequence of production:\n");
        e(str);
        a=strlen(str);

        printf("\n The Length of the given String is: %d",a);

        for(i=0;i<a;i++)
        {
                if(str[i]!='+'&&str[i]!='-'&&str[i]!='/'&&str[i]!='*'&&
                                        (!isalpha(str[i])&&(!isdigit(str[i]))))
                {
                        printf("\nThe input string is not successfully parsed");
                        getch( );
                        exit(0);
                }
        }
        printf("\nThe input string is successfully parsed");
        getch();
}
```

OUTPUT:

```
Available production:

E → TE'
T → FT'
F → id
F → digit
T' → e
T' → /FT'
T' → *FT'
E' → e
E' → -TE'
E' → +TE'

Enter the input symbol
x+23/a

The input symbol is: x+23/a

Sequence of production:

 E → TE'
 T →FT'

 F → x
 T' → e
 E' → +TE'
 T → FT'

 F → 2 3
 T' → /FT'

 F → a
 T' → e
 E' → e

 The Length of the given String is: 6
 The input string is successfully parsed

Available production

E→ TE'
T→ FT'
F→id
F→ digit
T'→ e
T' → /FT'
T'→ *FT'
```

```
E' → e
E' → -TE'
E' → +TE'

Enter the input symbol
x=y+23

The input symbol is: x=y+23

Sequence of production:

 E→ TE'
 T→FT'

 F→ x
 T' → e
 E' → e

 The Length of the given String is: 6
 The input string is not successfully parsed
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the C program for implementation of type checking using recursive descent parser was executed and the output was verified successfully.

Ex.No:6b	IMPLEMENTATION OF TYPE CHECKING

AIM:

To write a C program to implement type checking

ALGORITHM:

Step1: Track the global scope type information (e.g. classes and their members)
Step2: Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.
Step3: If type found correct, do the operation
Step4: Type mismatches, semantic error will be notified

PROGRAM:

```
//To implement type checking
#include<stdio.h>
#include<stdlib.h>
Struct symTable
{
int type:
char var[10];
}sT[50];
int c=0;
void sep(char a[])
{
int len=strlen(a);
int  i,j=0;
char temp[50],tp[50];
for(i=0;i<len;++i)
{
if(a[i]!=32)
tp[i]=a[i];
else
break;
}
tp[i]='\0';
temp[0]='\0';
++i;
for(;i<len;i++)
{
if(a[i]!=','&&a[i]!=32&&a[i]!=';')
temp[j++]=a[i];
else
{
if(strcmp(tp,"int")==0)
sT[c].type=1;
else if(strcmp(tp,”float”)==0)
sT[c].type=2;
strcpy(sT[c++].var,temp);
temp[0]=’\0’;
j=0;
}
}}
int check(char a[])
{
int  len=strlrn(a);
int  i,j=0,key=0,k;
char  temp[50];
for(i=0;i<len;++i)
{
if(a[i]!=32&&a[i]!=’+’&&a[i]!=’=’&&a[i]!=’;’)
temp[j++]=a[i];
else
{
temp[j]=’\0’;
for(k=0;k<c;++k)
{
if(strcmp(sT[k].var,temp)==0)
{
if(key==0)
key=sT[k].type;
else if(sT[k].type!=key)
return 0;
}
}
j=0;
}
}
return 1;
}
void main()
{
int  N,ans,i;
char  s[50];
printf(“\n Enter the total lines of declarations:\n”);
scanf(“%d”,%N);
while(N--)
{
scanf(“%[^\n]”,s);
sep(s);
}
printf(“Enter the expression:\n”);
scanf(“%[^\n]”,s);
if(check(s))
printf(“Correct\n”);
else
printf(“Semantic Error\n”);}
```

OUTPUT:

```
Enter the total lines of declaration:
2
Int a,b;
Float c;
Enter the expression:
C=a+b;
Semantic Error.
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the C program for implementation of type checking for variable declaration was executed and the output was verified successfully.

Ex.No:7	IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUE

AIM:

To write a program for implementation of Code Optimization Technique.

ALGORITHM:

Step1: Generate the program for factorial program using for and do-while loop to specify optimization technique.

Step2: 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.

Step3: The for loop operation is activated till the condition failure.

Step4: In do-while loop the variable is initialized and the statements are executed then the condition checking and increment / decrement operation is performed.

Step5: 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.

Step6: Finally when considering Code Optimization in loop do-while is best with 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);
}
}
}
```

INPUT:

```
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
```

OUTPUT:

```
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

Eliminate Common Expression
b       =c+d
b       =c+d
f       =b+b
r       =f

The Optimized Code is
b=c+d
f=b+b
r=f
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the program for code optimization is implemented and executed successfully, and the output is verified.

Ex.No:8	BACK END OF THE COMPILER

AIM:

To write a C program 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 and jump. Also simple addressing modes are used.

ALGORITHM:

1. Define the structure with operator argument 1, argument 2 and result.
2. Get the input file and open it in read mode.
3. Read the file content and store it in the structure.
4. Compare the operator with '+', if it is equal then call the ADD() function.
5. Compare the operator with '-', if it is equal then call the SUB () function.
6. Compare the operator with '*', if it is equal then call the MUL () function.
7. Compare the operator with '/', if it is equal then call the DIV() function.
8. Compare the operator with '=', if it is equal then call the ASSIGN () function.
9. Otherwise Print the Error message.
10. Define ADD () function. Generate the register number, MOV instruction with register and the argument 1 to be displayed. Then generate ADD instruction with register and argument 2 to be displayed.
11. Define SUB () function. Generate the register number, MOV instruction with register and the argument 1 to be displayed. Then generate SUB instruction with register and argument 2 to be displayed.
12. Define MUL () function. Generate the register number, MOV instruction with register and the argument 1 to be displayed. Then generate MUL instruction with register and argument 2 to be displayed.
13. Define DIV () function. Generate the register number, MOV instruction with register and the argument 1 to be displayed. Then generate DIV instruction with register and argument 2 to be displayed.
14. Define ASSIGN () function. Generate the register number, MOV instruction with result and register. Finally print the result as three address code and produces the 8086 assembly language instructions.
15. The instructions can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, mul, div and assign.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<conio.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 is:\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\t MOV %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:
A=a*b
C=f*h
G=a*h
F=q+w
T=q-j
Exit

The Target code is,
MOV a,R0
MULb,R0
MOV R0.a
MOv f,R1
MUL h,R1
MOv R1,c
MOv a,R2
MUL h,R2
MOV R2,g
MOV q,R3
ADD w,R3
MOV R3,f
MOV q,R4
SUB j,R4
MOV R4,t
```

Description	Max.Marks	Obtained Marks
Observation & Viva Voce	10	
Record	5	
Total	15	

RESULT:

Thus the program for implementing 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 and jump was executed successfully.