COMPILER DESIGN LABORATORY
LAB MANUAL

IFETCE R-2023

IFET COLLEGE OF ENGINEERING

[An Autonomous Institution]

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT CODE : 23CS6L01

YEAR/SEM : III/VI

Prepared by:	Verified by:	Prepared by:
Mrs.R.Vanitha AP/CSE Mrs.S.Nithyabharathi AP/CSE Ms.G.Nashiha Sulthana AP/CSE	Mrs.C.Manimegalai AP/CSE	Mr. D. Raghu Raman, HoD/CSE

Vision To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

DEPARTMENT VISION

To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

DEPARTMENT MISSION

- To develop graduates of world-class technical competence in the area of Computer Science and Engineering, with necessary skills to solve real world problems.
- To establish a center of excellence in partnership with industries, research institutions, and other organizations to meet the evolving needs of society.
- To inculcate ethical values and the spirit of entrepreneurship

PROGRAMME OUTCOMES (POs)

- On the completion of the program the graduates are expected to know or develop the abilities defined through the Program Outcomes.
- The Program Outcomes of Computer Science and Engineering program is given below.

PO1 Apply knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the conceptualization of computer science engineering models.

PO2 Identify, formulate, research literature and solve complex engineering problems reaching substantiated conclusions using first principles of mathematics and engineering sciences.

PO3 Design solutions for complex engineering problems and design systems, components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal, and environmental considerations.

PO4 Conduct investigations of complex problems including design of experiments, analysis and interpretation of data, and synthesis of information to provide valid conclusions.

PO5 Create, select and apply appropriate techniques, resources, and modern engineering tools, including prediction and modeling, to complex engineering activities, with an understanding of the limitations.

PO6 Function effectively as an individual, and as member or leader in diverse teams, and in multidisciplinary settings.

PO7 Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO8 Demonstrate understanding of the societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to engineering practice.

PO9 Understand and commit to professional ethics and responsibilities and norms of engineering practice.

PO10 Understand the impact of engineering solutions in a societal context and demonstrate knowledge of and need for sustainable development.

Vision To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

PO11 Demonstrate a knowledge and understanding of management and business practices, such as risk and change management, and understand their limitations.

PO12 Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technologies.

PROGRAMME SPECIFIC OUTCOMES (PSO)

Graduates of Computer Science & Engineering programs must:

PSO 1 Apply the fundamental knowledge of Computer Science & Engineering to design and develop software applications.

PSO 2 Develop competency in software development, software testing, data storage, computing, and business intelligence fields.

PSO 3 Ability to design and develop innovative solutions that meet rising global demands using the latest technologies.

PROGRAM EDUCATIONAL OBJECTIVES

PEO 1 Graduates will be equipped to take on technical and managerial roles, covering areas such as design, development, problem-solving, and production support within the software industry and R&D sectors.

PEO 2 Graduates will be able to successfully pursue higher education at reputed institutions.

PEO 3 Graduates will act as ethical and socially responsible solution providers and entrepreneurs within Computer Science and other engineering disciplines.

Vision To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

SYLLABUS

23CS6L01/ 23IT6L01	COMPILER DESIGN LABORATORY	L	T	P	C
		-	-	3	1.5

LEARNING OBJECTIVES:

The faculty will enhance the skills of the students to

- Demonstrate the working of lexical analysis with various open-source tools.
- Develop front and back end tools of compiler with run time environment.
- Describe different methods of semantic analysis techniques.
- Construct parsing techniques using C Programming.
- Infer the use of optimization techniques in compiler.

LIST OF EXPERIMENTS:

1. Implementation of Lexical Analyzer using Lex Tool
2. Implementation of lexical Analyzer to recognize a few patterns
3. Implementation of Arithmetic Calculator using LEX and YACC
4. Construction of LL (1) parsing.
5. Construction of recursive descent parsing.
6. Implementation of LALR parsing.
7. Implementation of Operator Precedence parsing.
8. Implementation of type checking.
9. Implementation of Symbol Table.
10. Implement Simple code optimization techniques (constant folding, Strength reduction and Algebraic transformation).
11. Implementation of control flow analysis and Data flow Analysis.
12. Implementation of storage allocation strategies. **TOTAL: 45 PERIODS**

Vision To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

COURSE OUTCOMES:

Upon completion of the course, the students will be able to

L303.1 Summarize the ability to install and configure lexical analysis tools in a compiler.

L 303.2 Implement storage management techniques using heap and stack in runtime environments.

L303.3 Develop applications using semantic and lexical analyzers for compilers

L303.4 Use recursive and LL parsing techniques for structured programming languages.

L303.5 Apply data and control flow analysis to optimize code using advanced code optimization techniques

Mapping of Course outcomes (COs) to Program outcomes (POs) & Programme Specific Outcomes (PSOs)

POs & PSOs / COs	MAPPING WITH PROGRAMME OUTCOMES														
	PO 1	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**P0 12**	**PSO 1**	**PSO 2**	**PSO 3**
L303.1	2	2	1	1	1	-	-	-	1	2	1	1	3	2	-
L303.2	3	2	2	1	1	-	-	-	1	2	1	1	3	2	-
L303.3	3	3	3	3	2	-	-	1	1	2	1	2	3	3	-
L303.4	3	2	2	1	2	-	-	-	1	2	1	1	3	2	-
L303.5	3	2	2	1	2	-	-	1	1	2	1	2	2	3	1
L303	**3**	**3**	**2**	**2**	**2**	**-**	**-**	**1**	**1**	**2**	**1**	**1**	**3**	**2**	**1**

Correlation levels: 1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High)

.

INDEX

Vision To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

EX NO 1 IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL

Aim:

To write a program for implementing a Lexical analyzer using LEX tool

Algorithm:

Step 1: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%. The format is as follows: definitions %% rules %% user subroutines

Step 2: In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.

Step 3: In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.

Step 4: Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

Step 5: When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.

Step 6: In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.

Step 7: The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

Program:

//Implementation of Lexical Analyzer using Lex tool

```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
```

```
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
\{  {if(!COMMENT)printf("\n BLOCK BEGINS");}
\}  {if(!COMMENT)printf("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 %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)
{
FILE *file;
```

```
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

Input:

```
//var.c
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

Output:

```
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ lex exp3_lex.l
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ cc lex.yy.c
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ ./a.out

#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

        void is a keyword
FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 1 is a NUMBER ;

 b IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 2 is a NUMBER ;

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

FUNCTION
        printf(
         "Sum:%d" is a STRING,
 c IDENTIFIER
        )
;
BLOCK ENDS
```

Result:

Thus the program for implementation of Lexical Analyzer using Lex tool has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.1	2	2	1	1	1	-	-	-	1	2	1	1	3	2	-

EX NO 2 DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C

Aim:

To develop a lexical analyzer to identify identifiers, constants, comments, operators etc using C program.

Algorithm:

Step 1: Start the program.

Step 2: Declare all the variables and file pointers.

Step 3: Display the input program.

Step 4: Separate the keyword in the program and display it.

Step 5: Display the header files of the input program

Step 6: Separate the operators of the input program and display it.

Step 7: Print the punctuation marks.

Step 8: Print the constant that are present in input program.

Step 9: Print the identifiers of the input program.

Program:

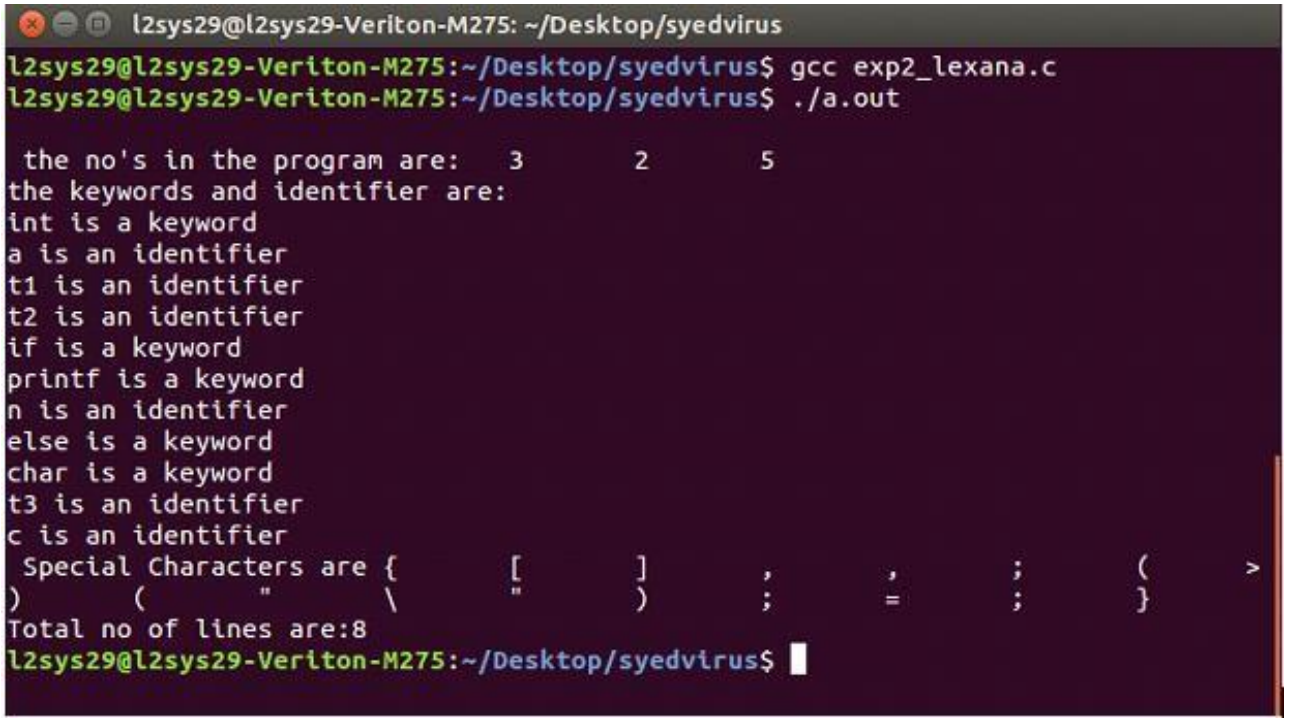
```
//Develop a lexical analyzer to recognize a few patterns in C.
#include<string.h>
#include<ctype.h>
#include<stdio.h>
#include<stdlib.h>
void keyword(char str[10])
{
 if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int",str)==0||strcmp("f
loat",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("printf",str)==0||strcmp("switc
h",str)==0||strcmp("case",str)==0)
  printf("\n%s is a keyword",str);
 else
  printf("\n%s is an identifier",str);
}
void main()
{
 FILE *f1,*f2,*f3;
```

```
char c,str[10],st1[10];
int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF)
{
 if(isdigit(c))
 {
 tokenvalue=c-'0';
 c=getc(f1);
 while(isdigit(c))
 {
  tokenvalue*=10+c-'0';
  c=getc(f1);
 }
 num[i++]=tokenvalue;
 ungetc(c,f1);
 }
 else
 if(isalpha(c))
 {
  putc(c,f2);
  c=getc(f1);
  while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
  {
   putc(c,f2);
   c=getc(f1);
  }
  putc(' ',f2);
  ungetc(c,f1);
 }
 else
```

```
 if(c==' '||c=='\t')
 printf(" ");
 else
 if(c=='\n')
 lineno++;
 else
 putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\n the no's in the program are:");
for(j=0;j<i;j++)
printf("\t%d",num[j]);
printf("\n");
f2=fopen("identifier","r");
k=0;
printf("the keywords and identifier are:");
while((c=getc(f2))!=EOF)
if(c!=' ')
str[k++]=c;
else
{
 str[k]='\0';
 keyword(str);
 k=0;
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\n Special Characters are");
while((c=getc(f3))!=EOF)
printf("\t%c",c);
printf("\n");
fclose(f3);
```

```
printf("Total no of lines are:%d",lineno);
}
```

Output:



Result:

Thus, the program for developing a lexical analyzer to recognize a few patterns in C has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.1	2	2	1	1	1	-	-	-	1	2	1	1	3	2	-

EX NO 3 IMPLEMENT AN ARITHMETIC CALCULATOR USING LEX AND YACC

Aim:

To write a program for implementing an arithmetic calculator for computing the given expression using semantic rules of the LEX and YACC tools.

Algorithm:

Step 1: A Yacc source program has three parts as follows:

Declarations %% translation rules %% supporting C routines

Step 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.

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

Step 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.

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

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

Step 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 programmer 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.

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

Program:

//*Implementation of calculator using LEX and YACC*

Lex Part:

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

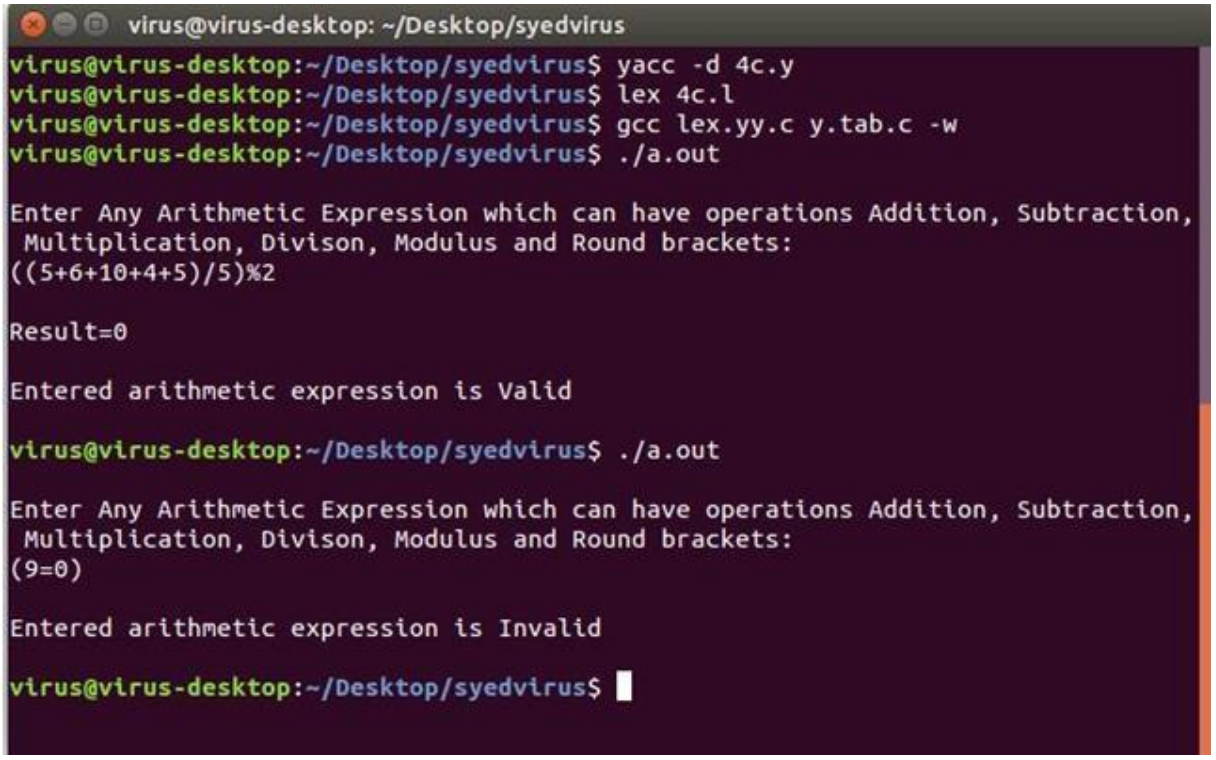
```
extern int yylval;
%}
%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;
      }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

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:



Result:

Thus, the program for implementation of an arithmetic calculator for computing the given expression using semantic rules of the LEX and YACC tools has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.3	3	3	3	3	2	-	-	1	1	2	1	2	3	3	-

EX NO 4 CONSTRUCTION OF LL (1) PARSING

AIM :

To write a program for constructing a LL(1) parser using predictive parsing table.

ALGORITHM:

Step 1: Read the input string.

Step 2: The input string is parsed. Initially, it is assumed that its end is marked with a special symbol $.

Step 3: Using predictive parsing table parse the given input using stack.

Step 4: To construct the parsing table, we have two functions:
In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Step 5: If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to $.

Step 6: The output would be a production rule representing a left-most derivation of the string in the input buffer.

.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char s[20],stack[20];
void main()
{
char m[5][6][3]={"tb"," "," ","tb"," "," "," ","+tb"," "," ","n","n","fc"," "," ","fc"," "," "," ","n","*fc","
a","n","n","i"," "," ","(e)"," "," "};
int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
 int i,j,k,n,str1,str2;
 clrscr();
 printf("\n Enter the input string: ");
 scanf("%s",s);
strcat(s,"$");
```

```
n=strlen(s);
stack[0]='$';
 stack[1]='e';
 i=1;
 j=0;
printf("\nStack Input\n");
printf("_____\n");
 while((stack[i]!='$')&&(s[j]!='$'))
 {
 if(stack[i]==s[j])
 {
 i--;
 j++;
}
switch(stack[i])
 {
 case 'e': str1=0;
break;
 case 'b': str1=1;
break;
 case 't': str1=2;
break;
 case 'c': str1=3;
break;
 case 'f': str1=4;
break;
 }
 switch(s[j])
 {
 case 'i': str2=0;
break;
 case '+': str2=1;
break;
```

```
 case '*': str2=2;
break;
 case '(': str2=3;
break;
 case ')': str2=4;
break;
 case '$': str2=5;
break;
 }
 if(m[str1][str2][0]=='\0')
 {
 printf("\nERROR");
 exit(0);
 }
 else if(m[str1][str2][0]=='n')
 i--;
 else if(m[str1][str2][0]=='i')
stack[i]='i';
else
 {
 for(k=size[str1][str2]-1;k>=0;k--)
 {
 stack[i]=m[str1][str2][k];
 i++;
 }
 i--;
 }
 for(k=0;k<=i;k++)
 printf(" %c",stack[k]);
 printf(" ");
 for(k=j;k<=n;k++)
 printf("%c",s[k]);
 printf(" \n ");
```

```
 }
printf("\n SUCCESS");
 getch(); }
```

Output:

```
Enter the input string:i*i+i
Stack                INPUT
$bt                  i*i+i$
$bcf                 i*i+i$
$bci                 i*i+i$
$bc                   *i+i$
$bcf*                 *i+i$
$bcf                   i+i$
$bci                   i+i$
$bc                    +i$
$b                     +i$
$bt+                   +i$
$bt                     i$
$bcf                    i$
$ bci                   i$
$bc                      $
$b                       $
$                        $
success
```

Result:

Thus the program for constructing a LL(1) parser using predictive parse table has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.4	**3**	**2**	**2**	**1**	**2**	**-**	**-**	**-**	**1**	**2**	**1**	**1**	**3**	**2**	**-**

.

EX NO 5 CONSTRUCTION OF RECURSIVE DESCENT PARSING

AIM :

To write a program for constructing a parser for the given grammar.

E->TE'
E'->+TE/@ "@ represents null character"
T->FT'
T`->*FT'/@
F->(E)/ID

Algorithm:

Step 1: Read the input string. Recursive descent parser will verify recursive descent whether the syntax of the input stream is correct by checking each character from left to right.

Step 2: The input string is parsed. Initially, it is assumed that its end is marked with a special symbol $.

Step 3: Reading characters from the input stream and matching them with terminals from the grammar that describes the syntax of the input.

Step 4: After eliminating Left recursion, we have to simply move from one character to next by checking whether it follow the grammar.

Step 5: Verify the next token equals to non-terminals if it satisfies match the non-terminal.

Step 6: If the input string does not match print error.

Step 7: Print the output.

.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char input[100];
int i,l;
void main()
{
clrscr();
printf("\nRecursive descent parsing for the following grammar\n");
 printf("\nE->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/ID\n");
 printf("\nEnter the string to be checked:");
 gets(input);
```

```
 if(E())
 {
 if(input[i+1]=='\0')
 printf("\nString is accepted");
 else
 printf("\nString is not accepted");
 }
else
 printf("\nString not accepted");
getch();
}
E()
{
 if(T())
 {
 if(EP())
 return(1);
 else
 return(0);
 }
else
 return(0);
}
EP()
{
 if(input[i]=='+')
 {
 i++;
 if(T())
 {
 if(EP())
return(1);
 else
```

```
return(0);
}
else
 return(0);
 }
 else
 return(1);
}
T()
{
if(F())
 {
 if(TP())
 return(1);
 else
 return(0);
 }
Else
return(0);
}
TP()
{
 if(input[i]=='*')
 {
 i++;
 if(F())
 {
 if(TP())
 return(1);
 else
 return(0);
 }
else
```

```
 return(0);
 }
 else
 return(1);
}
F()
{
if(input[i]=='(')
 {
 i++;
 if(E())
 {
 if(input[i]==')')
 {
i++;
return(1);
 }
else
 return(0);
 }
else
 return(0);
 }
 else if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')
{
i++;
return(1);
}
 else
return(0);
}
```

Output:

Recursive descent parsing for the following grammar

E->TE'

E'->+TE'/@

T->FT'

T'->*FT'/@

F->(E)/ID

Enter the string to be checked:(a+b)*c

String is accepted

Recursive descent parsing for the following grammar

E->TE'

E'->+TE'/@

T->FT'

T'->*FT'/@

F->(E)/ID

Enter the string to be checked:a/c+d

String is not accepted

Result:

Thus the program for constructing recursive descent parser has been executed successfully.

POs/ PSOs CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.4	**3**	**2**	**2**	**1**	**2**	**-**	**-**	**-**	**1**	**2**	**1**	**1**	**3**	**2**	**-**

EX NO 6 IMPLEMENTATION OF LALR PARSER IN C

Aim:

To write a program for constructing a LALR parser for the given grammar.

Algorithm:

Step 1: Read the input string.

Step 2: The input string is parsed. Initially, it is assumed that its end is marked with a special symbol $.

Step 3: Devise the augmented grammar for the given string.

Step 4: Find LR(1) collection of items and construct parsing table.

Step 5: Define two functions: goto[list of terminals] and action[list of non-terminals] in the LALR parsing table

Step 6: Verify whether the given string is acceptable or not.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int ister(char);
int  isnter(char);
int isstate(char);
void error();
void isreduce(char,char);
char pop(char *,int *);
void printt(char *,int *,char [],int);
void rep(char [],int);
struct action
{
char row[6][5];
```

```
};
const struct action A[12]={
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","emp","acc"},
{"emp","rc","sh","emp","rc","rc"},
{"emp","re","re","emp","re","re"},
{"sf","emp","emp","se","emp","emp"},
{"emp","rg","rg","emp","rg","rg"},
{"sf","emp","emp","se","emp","emp"},
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","sl","emp"},
{"emp","rb","sh","emp","rb","rb"},
{"emp","rb","rd","emp","rd","rd"},
{"emp","rf","rf","emp","rf","rf"}
};
struct gotol
{
char r[3][4];
};
const struct gotol G[12]={
{"b","c","d"},
{"emp","emp","emp"},
{"emp","emp","emp"},
{"emp","emp","emp"},
{"i","c","d"},
{"emp","emp","emp"},
{"emp","j","d"},
{"emp","emp","k"},
{"emp","emp","emp"},
{"emp","emp","emp"},
};
char ter[6]={'i','+','*',')','(','$'};
char nter[3]={'E','T','F'};
```

```
char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};
char stack[100];
int top=-1;
char temp[10];
struct grammar
{
char left;
char right[5];
};
const struct grammar rl[6]={
{'E',"e+T"},
{'E',"T"},
{'T',"T*F"},
{'T',"F"},
{'F',"(E)"},
{'F',"i"},
};
void main()
{
char inp[80],x,p,dl[80],y,bl='a';
int i=0,j,k,l,n,m,c,len;
clrscr();
printf(" Enter the input :");
scanf("%s",inp);
len=strlen(inp);
inp[len]='$';
inp[len+1]='\0';
push(stack,&top,bl);
printf("\n stack \t\t\t input");
printt(stack,&top,inp,i);
do
{
x=inp[i];
p=stacktop(stack);
```

```
isproduct(x,p);
if(strcmp(temp,"emp")==0)
error();
if(strcmp(temp,"acc")==0)
break;
else
{
if(temp[0]=='s')
{
push(stack,&top,inp[i]);
push(stack,&top,temp[1]);
i++;
}
else
{
if(temp[0]=='r')
{
j=isstate(temp[1]);
strcpy(temp,rl[j-2].right);
dl[0]=rl[j-2].left;
dl[1]='\0';
n=strlen(temp);
for(k=0;k<2*n;k++)
pop(stack,&top);
for(m=0;dl[m]!='\0';m++)
push(stack,&top,dl[m]);
l=top;
y=stack[l-1];
isreduce(y,dl[0]);
for(m=0;temp[m]!='\0';m++)
push(stack,&top,temp[m]);
}
}
```

```
}
printt(stack,&top,inp,i);

}while(inp[i]!='\0');
if(strcmp(temp,"acc")==0)
printf(" \n accept the input ");
else
printf(" \n do not accept the input ");
getch();
}
void push(char *s,int *sp,char item)
{
if(*sp==100)
printf(" stack is full ");
else
{
*sp=*sp+1;
s[*sp]=item;
}
}
char stacktop(char *s)
{
char i;
i=s[top];
return i;
}
void isproduct(char x,char p)
{
int k,l;
k=ister(x);
l=isstate(p);
strcpy(temp,A[l-1].row[k-1]);
}
int ister(char x)
```

```
{
int i;
for(i=0;i<6;i++)
if(x==ter[i])
return i+1;
return 0;
}
int isnter(char x)
{
int i;
for(i=0;i<3;i++)
if(x==nter[i])
return i+1;
return 0;
}
int isstate(char p)
{
int i;
for(i=0;i<12;i++)
if(p==states[i])
return i+1;
return 0;
}
void error()
{
printf(" error in the input ");
exit(0);
}
void isreduce(char x,char p)
{
int k,l;
k=isstate(x);
l=isnter(p);
```

```
strcpy(temp,G[k-1].r[l-1]);
}
char pop(char *s,int *sp)
{
char item;
if(*sp==-1)
printf(" stack is empty ");
else
{
item=s[*sp];
*sp=*sp-1;
}
return item;
}
void printt(char *t,int *p,char inp[],int i)
{
int r;
printf("\n");
for(r=0;r<=*p;r++)
rep(t,r);
printf("\t\t\t");
for(r=i;inp[r]!='\0';r++)
printf("%c",inp[r]);
}
void rep(char t[],int r)
{
char c;
c=t[r];
switch(c)
{
case 'a': printf("0");
break;
case 'b': printf("1");
break;
```

```
case 'c': printf("2");
break;
case 'd': printf("3");
break;
case 'e': printf("4");
break;
case 'f': printf("5");
break;
case 'g': printf("6");
break;
case 'h': printf("7");
break;
case 'm': printf("8");
break;
case 'j': printf("9");
break;
case 'k': printf("10");
break;
case 'l': printf("11");
break;
default :printf("%c",t[r]);
break;
}
}
```

Output:

Stack	input
0	i*i+i$
0i5	*i+i$
0F3	*i+i$
0T2	*i+i$
0T2*7	i+i$
0T2*7i5	+i$
0T2*7i5F10	+i$
0T2	+i$
0E1	+i$
0E1+6	i$
0E1+6i5	$
0E1+6F3	$
0E1+6T9	$
0E1	$

accept the input*/

Result:

Thus, the program for constructing LALR parser for the given grammar has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.4	3	2	2	1	2	-	-	-	1	2	1	1	3	2	-

EX NO 7 IMPLEMENTATION OF OPERATOR PRECEDENCE PARSING

Aim:

To write a program for implementation of Operator Precedence Parser.

Algorithm

Step1. Include the necessary header files.

Step 2. Declare the necessary variables with the operators defined before.

Step 3. Get the input from the user and compare the string for any operators.

Step 4:Find the precedence of the operator in the expression from the predefined operator.

Step 5: Set the operator with the maximum precedence accordingly and give the relational operators them.

Step 6: Parse the given expression with the operators and values.

Step 7: Display the parsed expression.

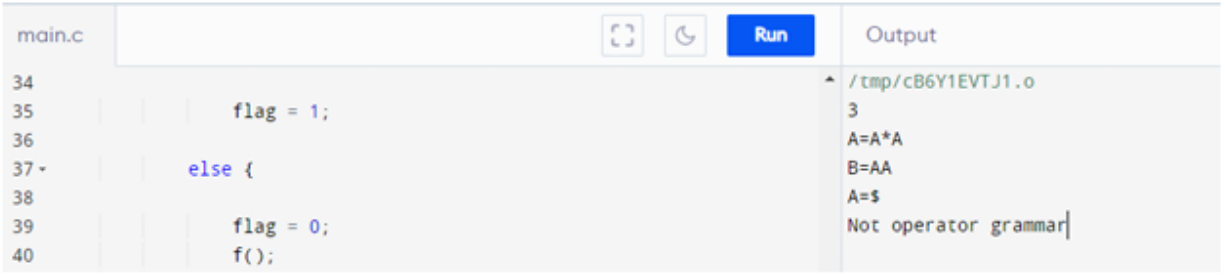
Step 8: Exit the program.

Program Code:

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
// function f to exit from the loop
// if given condition is not true
void f()
{
printf("Not operator grammar");
exit(0);
}
void main()
{
char grm[20][20], c;
// Here using flag variable,
// considering grammar is not operator grammar
int i, n, j = 2, flag = 0;
// taking number of productions from user
scanf("%d", &n);
for (i = 0; i < n; i++)
scanf("%s", grm[i]);
for (i = 0; i < n; i++) {
c = grm[i][2];
while (c != '&#092;&#048;') {
if (grm[i][3] == '+' || grm[i][3] == '-'
|| grm[i][3] == '*' || grm[i][3] == '/')
flag = 1;
```

```
else {

flag = 0;
f();
}

if (c == '$') {
flag = 0;
f();
}

c = grm[i][++j];
}
}

if (flag == 1)
printf("Operator grammar");
}
```

Output:



Result:
Thus, the program for implementation of Operator Precedence Parser has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.4	3	2	2	1	2	-	-	-	1	2	1	1	3	2	-

Vision To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

EX NO 8 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

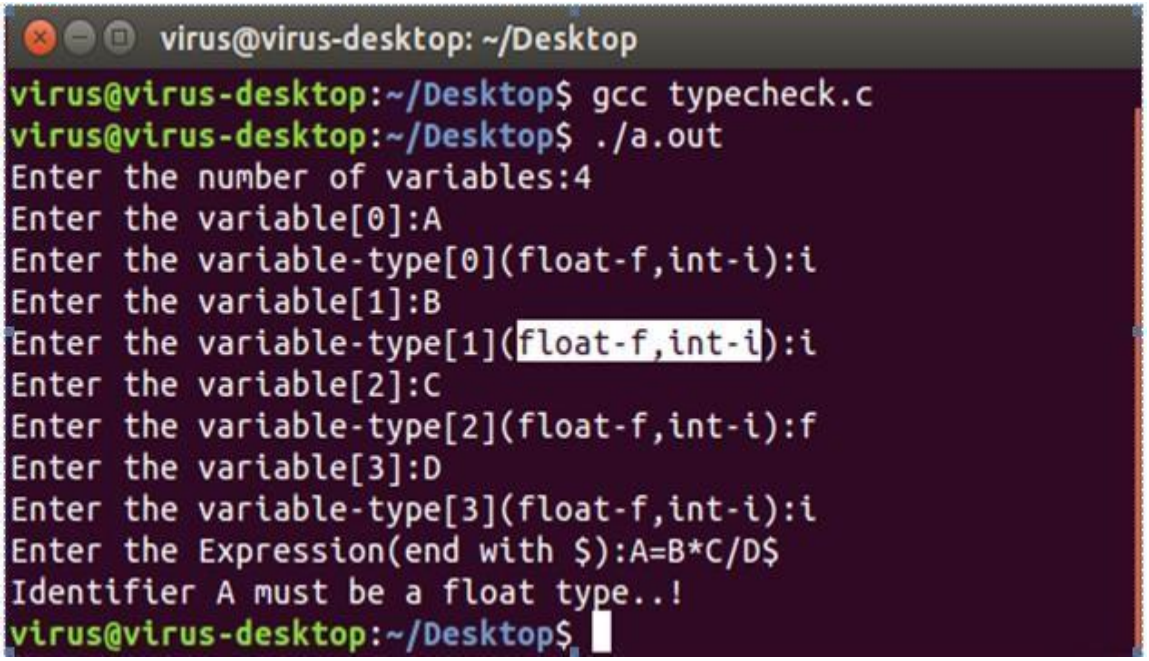
Step4: Type mismatches, semantic error will be notified

Program:

```
//To implement type checking
#include<stdio.h>
#include<stdlib.h>
int main()
{
int n,i,k,flag=0;
char vari[15],typ[15],b[15],c;
printf("Enter the number of variables:");
scanf(" %d",&n);
for(i=0;i<n;i++)
{
printf("Enter the variable[%d]:",i);
scanf(" %c",&vari[i]);
printf("Enter the variable-type[%d](float-f,int-i):",i);
scanf(" %c",&typ[i]);
if(typ[i]=='f')
flag=1;
}
printf("Enter the Expression(end with $):");
i=0;
getchar();
while((c=getchar())!='$')
```

```
{
b[i]=c;
i++;  }
k=i;
for(i=0;i<k;i++)
{
if(b[i]=='/')
{
flag=1;
break;  }  }
for(i=0;i<n;i++)
{
if(b[0]==vari[i])
{
if(flag==1)
{
if(typ[i]=='f')
{  printf("\nthe datatype is correctly defined..!\n");
break;  }
else
{  printf("Identifier %c must be a float type..!\n",vari[i]);
break;  }  }
else
{  printf("\nthe datatype is correctly defined..!\n");
break;  }  }
}
return 0;
}
```

Output:



Result:

Thus, the program for implementation of type checking has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.3	3	3	3	3	2	-	-	1	1	2	1	2	3	3	-

EX NO 9 IMPLEMENTATION OF SYMBOL TABLE

Aim:

To write a program for implementing Symbol Table using C.

Algorithm:

Step 1: Start the program for performing insert, display, delete, search and modify option in symbol table

Step 2: Define the structure of the Symbol Table

Step 3: Enter the choice for performing the operations in the symbol Table

Step 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.

Step 5: If the entered choice is 2, the symbols present in the symbol table are displayed.

Step 6: If the entered choice is 3, the symbol to be deleted is searched in the symbol table.

Step 7: If it is not found in the symbol table it displays "Label Not found". Else, the symbol is deleted.

Step 8: If the entered choice is 5, the symbol to be modified is searched in the symbol table.

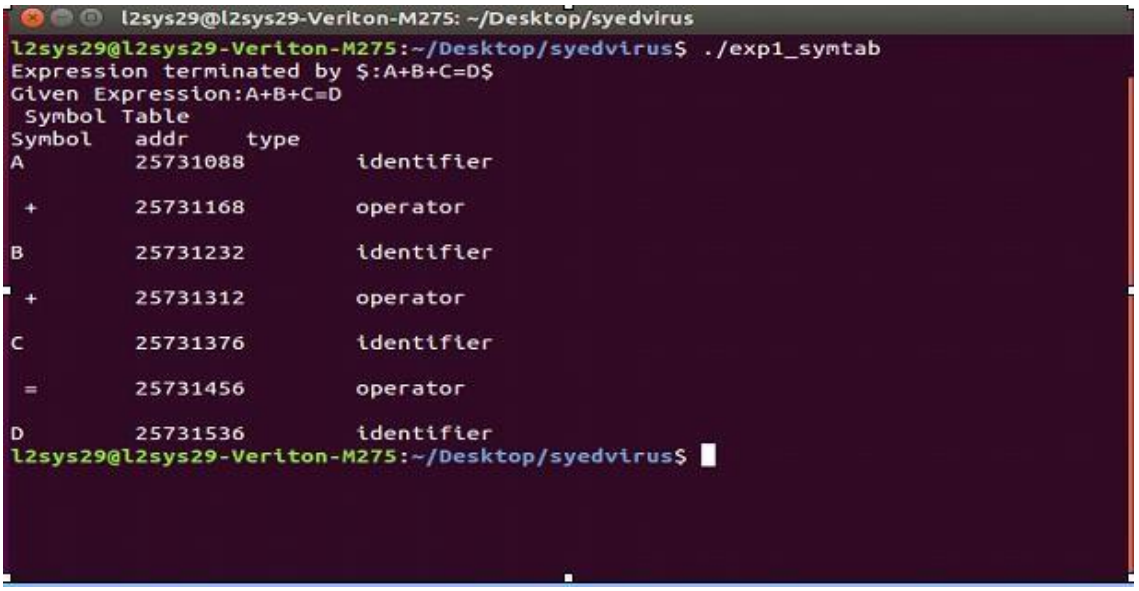
Program:

```
//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:



Result:

Thus, the program for symbol table has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.3	3	3	3	3	2	-	-	1	1	2	1	2	3	3	-

EX NO :10 IMPLEMENT SIMPLE CODE OPTIMIZATION TECHNIQUES (CONSTANT FOLDING, STRENGTH REDUCTION AND ALGEBRAIC TRANSFORMATION).

Aim:

To write a C program to implement simple code optimization techniques such as Constant Folding, Strength Reduction, and Algebraic Transformation.

Algorithm:

Step 1: Start the program.

Step 2: Read input values required for arithmetic expressions.

Step 3: Apply constant folding by evaluating constant expressions at compile time.

Step 4 : Apply strength reduction by replacing costly operations with cheaper ones.

Step 5: Apply algebraic transformation to simplify expressions.

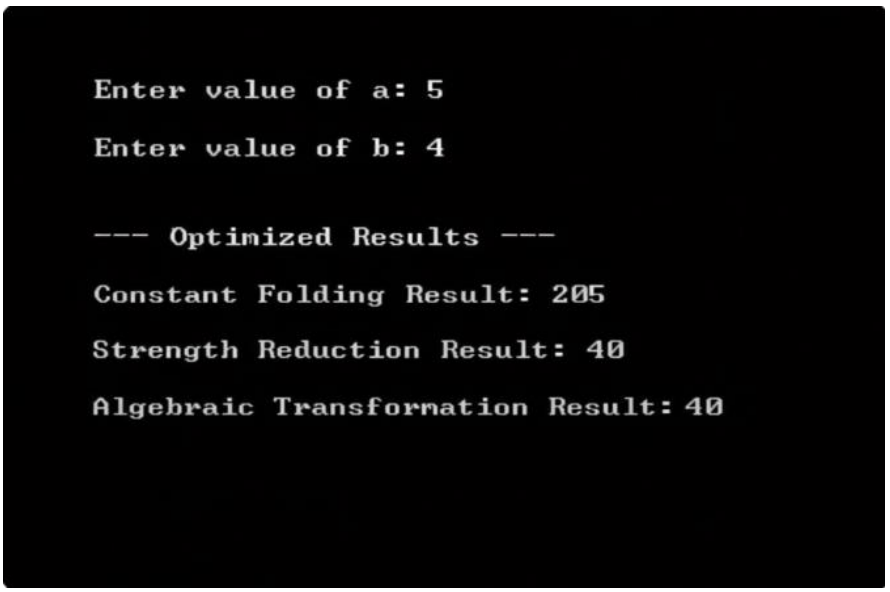
Step 6: Display the optimized results.

Step 7: Stop the program.

Program:

```
#include<stdio.h>
int main()
{
   int a, b, c;
   int result1, result2, result3;
   printf("Enter value of a: ");
   scanf("%d", &a);
   printf("Enter value of b: ");
   scanf("%d", &b);
   /* Constant Folding */
   /* Expression: c = 10 * 20 */
   c = 10 * 20;   // computed at compile time
   result1 = c + a;
   /* Strength Reduction */
   /* Expression: a * 8 replaced by a << 3 */
   result2 = a << 3;
   /* Algebraic Transformation */
   /* Expression: (a * b) + (a * b) simplified to 2 * a * b */
   result3 = 2 * a * b;
   printf("\n--- Optimized Results ---\n");
   printf("Constant Folding Result: %d\n", result1);
   printf("Strength Reduction Result: %d\n", result2);
   printf("Algebraic Transformation Result: %d\n", result3);
   return 0;
}
```

Output:



Result:

Thus, the program for implementing simple code optimization techniques such as constant folding, strength reduction, and algebraic transformation using C has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.5	3	2	2	1	2	-	-	1	1	2	1	2	2	3	1

EX NO 11 IMPLEMENTATION OF CONTROL FLOW ANALYSIS AND DATA FLOW ANALYSIS

Aim:

To write a program for implementing of control flow analysis and data flow analysis

Algorithm:

1. Start the program
2. Declare the necessary variables
3. Get the choice to insert, delete and display the values in stack
4. Perform PUSH() operation
    a. t = newnode()
    b. Enter info to be inserted
    c. Read n
    d. t ->info= n
    e. t ->next=top
    f. top = t
    g. Return
5. Perform POP() operation
    a. If (top=NULL)
    b. Print"underflow"
    c. Return
    d. X=top
    e. Top=top->next
    f. Delnode(x)
    g. Return
6. Display the values
7. Stop the program.

Program:

(DATA FLOW AND CONTROL FLOW ANALYSIS)

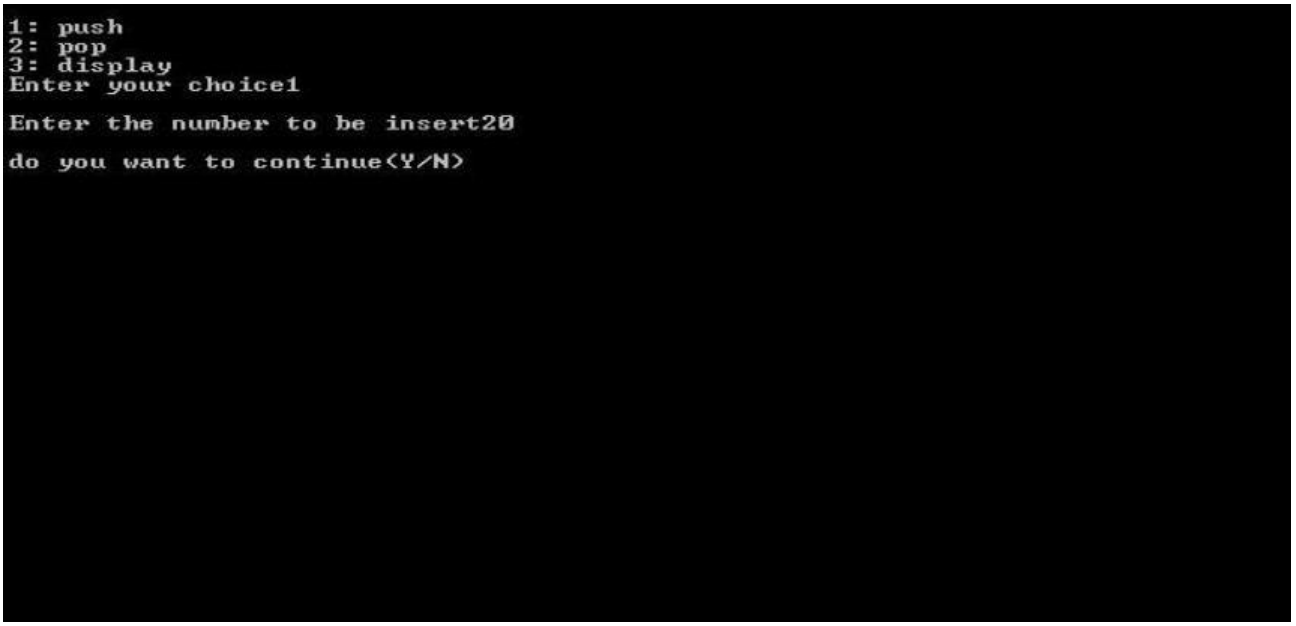
```
#include<conio.h>
struct stack
{
int no;
```

```
struct stack *next;
}
*start=null
typedef struct stack st;
voidpush();
int pop();
voiddisplay();
voidmain()
{
char ch;
int choice, item;
do
{
clrscr();
printf("\n1:push");
printf("\n2:pop");
printf("\n3:display");
printf("\n enter your choice");
scanf("%d",&choice);
switch(choice)
{
case1:push();
break;
case2:item=pop();
printf("the delete element in %d",item);
break;
case3:display();
break;
default: printf("\nwrong choice");
};
printf("\n do you want to continue(y/n");
fflush(stdin);
scanf("%c",&ch);
```

```
}
while(ch==’y’||ch==’y’);
}
voidpush()
{
st*node;
node=(st*)malloc(sizeof(st));
printf(“\n enter the number to be insert”);
scanf(“%d”,&node->no);
node->next=start;
start=node;
}
intpop();
{
st*temp;
temp=start;
if(start==null)
{
printf(“stack is already empty”);
getch();
exit();
}
else
{
start=start->next;
free(temp);
}
return(temp->no);
}
void display()
{
st*temp;
temp=start;
while(temp->next!=null)
```

```
{

printf("\nno=%d",temp->no);
temp=temp->next;
}
printf("\nno=%d",temp->no);
}
```

OUTPUT



Result:

Thus the program for implementing of control flow analysis and data flow analysis has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.5	3	2	2	1	2	-	-	1	1	2	1	2	2	3	1

EX NO 12 IMPLEMENTATION OF STORAGE ALLOCATION STRATEGIES

Aim:

To write a program for implementing storage allocation strategies using C.

Algorithm:

Step1: Initially check whether the stack is empty

Step2: Insert an element into the stack using push operation

Step3: Insert more elements onto the stack until stack becomes full

Step4: Delete an element from the stack using pop operation

Step5: Display the elements in the stack

Step6: Top the stack element will be displayed

Program:

```
//Implementation of heap allocation storage strategies//
#include<stdio.h>
#include<stdlib.h>
#define TRUE 1
#define FALSE 0
typedef struct Heap
{
int data;
struct Heap *next;
}
node;
node *create();
void main()
{
int choice,val;
char ans;
node *head;
void display(node *);
node *search(node *,int);
node *insert(node *);
void dele(node **);
```

```
head=NULL;
do
{
printf("\nprogram to perform various operations on heap using dynamic memory management");
printf("\n1.create");
printf("\n2.display");
printf("\n3.insert an element in a list");
printf("\n4.delete an element from list");
printf("\n5.quit");
printf("\nenter your chioce(1-5)");
scanf("%d",&choice);
switch(choice)
{
case 1:head=create();
break;
case 2:display(head);
break;
case 3:head=insert(head);
break;
case 4:dele(&head);
break;
case 5:exit(0);
default:
printf("invalid choice,try again");
}
}
while(choice!=5);
}
node* create()
{
node *temp,*New,*head;
int val,flag;
char ans='y';
```

```
node *get_node();
temp=NULL;
flag=TRUE;
do
{
printf("\n enter the element:");
scanf("%d",&val);
New=get_node();
if(New==NULL)
printf("\nmemory is not allocated");
New->data=val;
if(flag==TRUE)
{
head=New;
temp=head;
flag=FALSE;
}
else
{
temp->next=New;
temp=New;
}
printf("\ndo you want to enter more elements?(y/n)");
}
while(ans=='y');
printf("\nthe list is created\n");
return head;
}
node *get_node()
{
node *temp;
temp=(node*)malloc(sizeof(node));
temp->next=NULL;
```

```
return temp;
}
void display(node *head)
{
node *temp;
temp=head;
if(temp==NULL)
{
printf("\nthe list is empty\n");
return;
}
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
printf("NULL");
}
node *search(node *head,int key)
{
node *temp;
int found;
temp=head;
if(temp==NULL)
{
printf("the linked list is empty\n");
return NULL;
}
found=FALSE;
while(temp!=NULL && found==FALSE)
{
if(temp->data!=key)
temp=temp->next;
```

```
else
found=TRUE;
}
if(found==TRUE)
{
printf("\nthe element is present in the list\n");
return temp;
}
else
{
printf("the element is not present in the list\n");
return NULL;
}
}
node *insert(node *head)
{
int choice;
node *insert_head(node *);
void insert_after(node *);
void insert_last(node *);
printf("n1.insert a node as a head node");
printf("n2.insert a node as a head node");
printf("n3.insert a node at intermediate position in t6he list");
printf("\nenter your choice for insertion of node:");
scanf("%d",&choice);
switch(choice)
{
case 1:head=insert_head(head);
break;
case 2:insert_last(head);
break;
case 3:insert_after(head);
break;
```

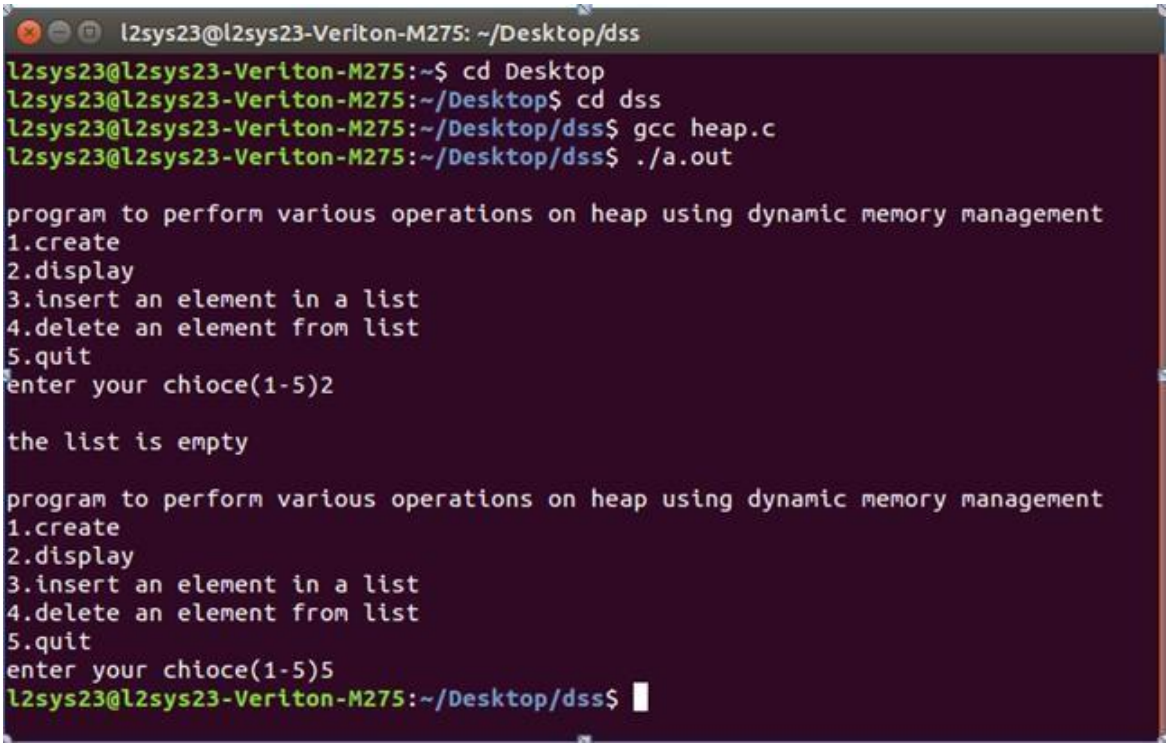
```
}
return head;
}
node *insert_head(node *head)
{
node *New,*temp;
New=get_node();
printf("\nEnter the element which you want to insert");
scanf("%d",&New->data);
if(head==NULL)
head=New;
else
{
temp=head;
New->next=temp;
head=New;
}
return head;
}
void insert_last(node *head)
{
node *New,*temp;
New=get_node();
printf("\nenter the element which you want to insert");
scanf("%d",&New->data);
if(head==NULL)
head=New;
else
{
temp=head;
while(temp->next!=NULL)
temp=temp->next;
temp->next=New;
New->next=NULL;
```

```
}
}
void insert_after(node *head)
{
int key;
node *New,*temp;
New=get_node();
printf("\nenter the elements which you want to insert");
scanf("%d",&New->data);
if(head==NULL)
{
head=New;
}
else
{
printf("\enter the element which you want to insert the node");
scanf("%d",&key);
temp=head;
do
{
if(temp->data==key)
{
New->next-temp->next;
temp->next=New;
return;
}
else
temp=temp->next;
}
while(temp!=NULL);
}
}
node *get_prev(node *head,int val)
{
```

```
node *temp,*prev;
int flag;
temp=head;
if(temp==NULL)
return NULL;
flag=FALSE;
prev=NULL;
while(temp!=NULL && ! flag)
{
if(temp->data!=val)
{
prev=temp;
temp=temp->next;
}
else
flag=TRUE;
}
if(flag)
return prev;
else
return NULL;
}
void dele(node **head)
{
node *temp,*prev;
int key;
temp=*head;
if(temp==NULL)
{
printf("\nthe list is empty\n");
return;
}
printf("\nenter the element you want to delete:");
```

```
scanf("%d",&key);
temp=search(*head,key);
if(temp!=NULL)
{
prev=get_prev(*head,key);
if(prev!=NULL)
{
prev->next=temp->next;
free(temp);
}
else
{
*head=temp->next;
free(temp);
}
printf("\nthe element is deleted\n");
}
}
```

Output:



Result:

Thus, the program for implementing storage allocation strategies using C has been executed successfully.

POs/ PSOs / CO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO 3
L303.2	3	2	2	1	1	-	-	-	1	2	1	1	3	2	-

CONTENT BEYOND SYLLABUS
CONVERT THE BNF RULES INTO YACC FORM AND WRITE CODE TO GENERATE ABSTRACT SYNTAX TREE

Aim:

To write a YACC program to change YACC form into abstract syntax Tree

Algorithm:

Step1: Reading an expression.
Step2: Calculate the value of given expression
Step3: Display the value of the nodes based on the precedence.
Step4: Using expression rule print the result of the given values

Program:

//*Convert the BNF rules into YACC form and*
//*write code to generate Abstract Syntax Tree*

LEX PART:

```
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
%}
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
\n LineNo++;
. return yytext[0];
%%
```

YACC PART:

```
%{
#include<string.h>
#include<stdio.h>
struct quad
{
char op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];
struct stack
{
int items[100];
int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
%}
%union
{
char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
```

```
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}
| EXPR '-' EXPR {AddQuadruple("-",$1,$3,$$);}
| EXPR '*' EXPR {AddQuadruple("*",$1,$3,$$);}
| EXPR '/' EXPR {AddQuadruple("/",$1,$3,$$);}
| '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK { strcpy(QUAD[Index].op,"GOTO"); strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
Ind=pop();
push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
```

```
| NUM;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE('CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------------------------""\n\t\t Pos Operator \tArg1 \tArg2 \tResult" "\n\t\t--------------
----");
for(i=0;i<Index;i++)
{
printf("\n\t\t %d\t %s\t %s\t %s\t%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t ----------------");
printf("\n\n"); return 0; }
```

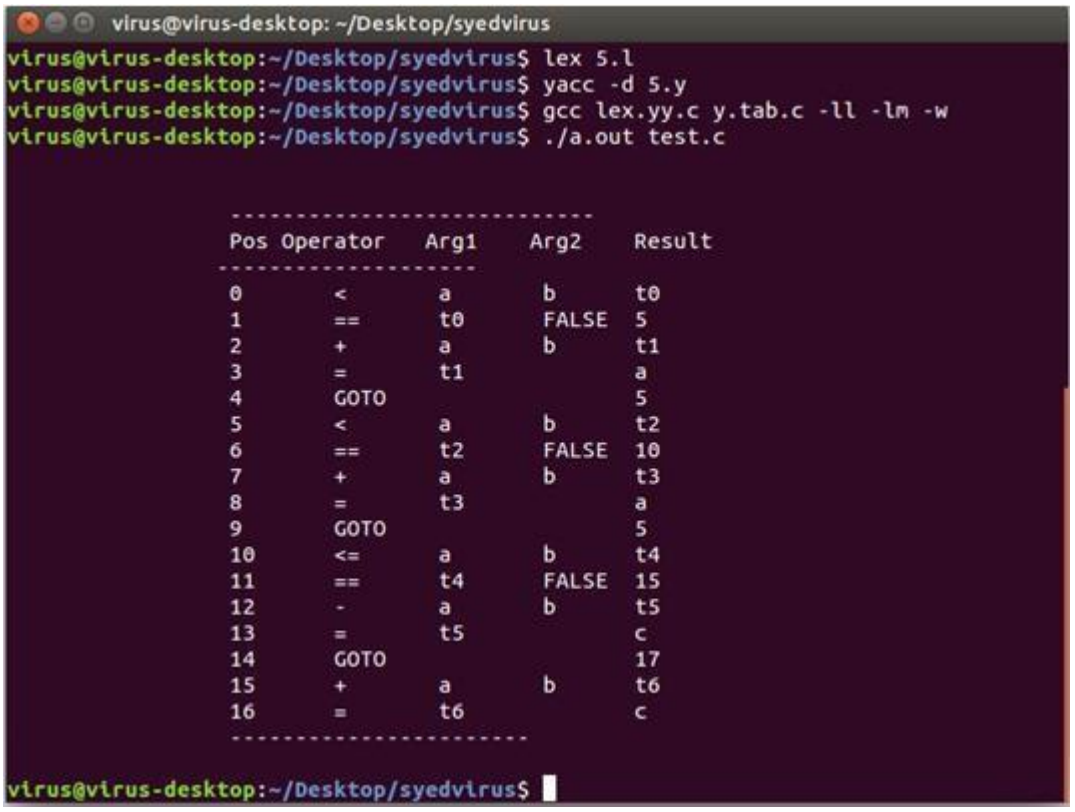
```
void push(int data)
{ stk.top++;
 if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
{
int data;
if(stk.top==-1)
{
printf("\n Stack underflow\n");
exit(0);
}
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

INPUT:

```
main()
{
int a,b,c;
if(a<b)
{
a=a+b;
}
while(a<b)
{
a=a+b;
}
if(a<=b)
{
c=a-b;
}
else
```

```
{
c=a+b;
}
}
```

Output:


virus@virus-desktop: ~/Desktop/syedvirus
virus@virus-desktop:~/Desktop/syedvirus$ lex 5.l
virus@virus-desktop:~/Desktop/syedvirus$ yacc -d 5.y
virus@virus-desktop:~/Desktop/syedvirus$ gcc lex.yy.c y.tab.c -ll -lm -w
virus@virus-desktop:~/Desktop/syedvirus$ ./a.out test.c
Pos Operator Arg1 Arg2 Result
0 < a b t0
1 == t0 FALSE 5
2 + a b t1
3 = t1 a
4 GOTO 5
5 < a b t2
6 == t2 FALSE 10
7 + a b t3
8 = t3 a
9 GOTO 5
10 <= a b t4
11 == t4 FALSE 15
12 - a b t5
13 = t5 c
14 GOTO 17
15 + a b t6
16 = t6 c
virus@virus-desktop:~/Desktop/syedvirus$

WRITE A C PROGRAM TO GENERATE MACHINE CODE FROM ABSTRACT SYNTAX TREE GENERATED BY THE PARSER

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int label[20];
int no=0;
int main()
{
FILE *fp1,*fp2;
char fname[10],op[10],ch;
char operand1[8],operand2[8],result[8];
int i=0,j=0;
printf("\n Enter filename of the intermediate code");
scanf("%s",&fname);
fp1=fopen(fname,"r");
fp2=fopen("target.txt","w");
if(fp1==NULL || fp2==NULL)
{
printf("\n Error opening the file");
exit(0);
}
while(!feof(fp1))
{
fprintf(fp2,"\n"); fscanf(fp1,"%s",op);
i++; if(check_label(i))
fprintf(fp2,"\nlabel#%d",i);
if(strcmp(op,"print")==0)
{
fscanf(fp1,"%s",result);
fprintf(fp2,"\n\t OUT %s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s %s",operand1,operand2);
fprintf(fp2,"\n\t JMP %s,label#%s",operand1,operand2);
label[no++]=atoi(operand2);
}
if(strcmp(op,"[]=")==0)
{
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t STORE %s[%s],%s",operand1,operand2,result);
}
if(strcmp(op,"uminus")==0)
{
fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t LOAD -%s,R1",operand1);
fprintf(fp2,"\n\t STORE R1,%s",result);
```

```
}

switch(op[0])
{
case '*': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t
LOAD",operand1);
fprintf(fp2,"\n \t LOAD
%s,R1",operand2);
fprintf(fp2,"\n \t MUL R1,R0");
fprintf(fp2,"\n \t STORE
R0,%s",result); break;
case '+': fscanf(fp1,"%s %s
%s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t ADD R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '-': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t SUB R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
 case '/': fscanf(fp1,"%s %s s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t DIV R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '%': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t DIV R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '=': fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t STORE %s %s",operand1,result);
break;
case '>': j++;
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t JGT %s,label#%s",operand2,result);
label[no++]=atoi(result);
break;
 case '<': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1); fprintf(fp2,"\n\t
JLT %s,label#%d",operand2,result);
label[no++]=atoi(result);
```

```
 break;
}
}
fclose(fp2); fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("Error opening the file\n");
exit(0);
}
 do
{
ch=fgetc(fp2);
printf("%c",ch);
}while(ch!=EOF);
fclose(fp1);
return 0;
}
int check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
}
return 0;
}
```

Input :

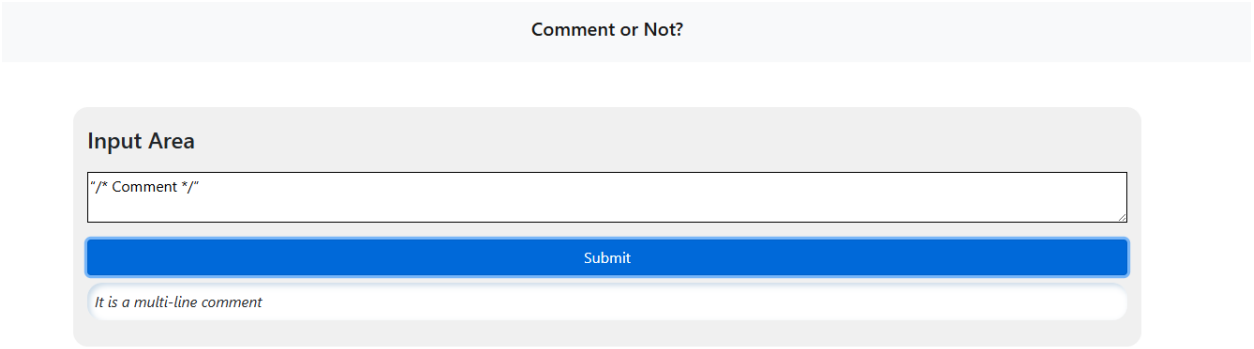
```
$ vi int.txt
= t1 2
[]= a 0 1
[]= a 1 2
[]= a 2 3
*t1 6 t2
+ a[2] t2 t3
- a[2] t1 t2
/ t3 t2 t2
uminus t2 t2
print t2
goto t2 t3
= t3 99
uminus 25 t2
* t2 t3 t3
uminus t1 t1 + t1 t3 t4
print t4
```

Output:

```
Enter filename of the intermediate code: int.txt
STORE t1, 2
STORE a[0], 1
STORE a[1], 2
STORE a[2], 3
LOAD t1, R0
LOAD 6, R1
ADD R1, R0
STORE R0, t3
LOAD a[2], R0
LOAD t2, R1
ADD R1,R0
STORE R0,t3
LOAD a[t2],R0
LOAD t1,R1
SUB R1,R0
STORE R0,t2
LOAD t3,R0
LOAD t2,R1
DIV R1,R0
STORE R0,t2
LOAD t2,R1
STORE R1,t2
LOAD t2,R0
JGT 5, label#11
Label#11: OUT t2
JMP t2, label#13
Label#13: STORE t3, 99
LOAD 25, R1
STORE R1,t2
LOAD t2,R0
LOAD t3,R1
MUL R1,R0
STORE R0,t3
LOAD t1,R1
STORE R1,t1
LOAD t1,R0
LOAD t3,R1
ADD R1,R0
STORE R0,t4
OUT t4
```

VIRTUAL LAB

Ex:No 1 Design and Simulation of a Comment Detection System for Single-Line and Multi-Line Comments



VIRTUAL LAB LINK:

https://gr16vlabiem.netlify.app/

Vision To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.