

IT6612 COMPILER LABORATORY**L T P C
0 0 3 2****OBJECTIVES: The student should be made to:**

- Be exposed to compiler writing tools.
- Learn to implement the different Phases of compiler
- Be familiar with control flow and data flow analysis
- Learn simple optimization techniques

LIST OF EXPERIMENTS:

1. Implementation of Symbol Table
2. Develop a lexical analyzer to recognize a few patterns in C.
(Ex. identifiers, constants, comments, operators etc.)
3. Implementation of Lexical Analyzer using Lex Tool
4. Generate YACC specification for a few syntactic categories.
 - a) Program to recognize a valid arithmetic expression that uses operator +, -, *, and /.
 - b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
 - c) Implementation of Calculator using LEX and YACC
5. Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree.
6. Implement type checking
7. Implement control flow analysis and Data flow Analysis
8. Implement any one storage allocation strategies (Heap, Stack, Static)
9. Construction of DAG
10. Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.
11. Implementation of Simple Code Optimization Techniques (Constant Folding, etc.)

TOTAL: 45 PERIODS**OUTCOMES: At the end of the course, the student should be able to**

- Implement the different Phases of compiler using tools
- Analyze the control flow and data flow of a typical program
- Optimize a given program
- Generate an assembly language program equivalent to a source language program

LIST OF EQUIPMENT FOR A BATCH OF 30 STUDENTS:

Standalone desktops with C / C++ compiler and Compiler writing tools 30 Nos. (or) Server with C / C++ compiler and Compiler writing tools supporting 30 terminals or more. LEX and YACC

INDEX

EX.NO	EXPERIMENT NAME
1	Implementation of Symbol Table
2	Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.)
3	Implementation of Lexical Analyzer using Lex Tool
4	Generate YACC specification for a few syntactic categories
4a	Program to recognize a valid arithmetic expression that uses operator +, -, *
4b	Program to recognize a valid variable which starts with a letter followed by any number of letters or digits
4c	Implementation of Calculator using LEX and YACC
5	Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree
6	Implement type checking
7	Implement control flow analysis and Data flow Analysis
8	Implement any one storage allocation strategies (Heap, Stack, Static)
9	Construction of DAG
10	Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used
11	Implementation of Simple Code Optimization Techniques (Constant Folding., etc.)
Beyond the syllabus	
12	Implementation Of Shift-Reduced Parsing Algorithms
13	Construction Of LR -Parsing Table

Ex.no:1**Implementation of Symbol Table****Date:****AIM:**

To write a program in C for implementing Symbol Table.

ALGORITHM

1. Start the program.
2. Declare the variables. Get the character and check it using while Loop.
3. If n value is less than or equal to l then print the symbol address and type. Again check the n value with j.
4. The C value is changed to ASCII and check it using if statement. Store the C value in P and print the identifier.
- Else check the character is equal to +, -, *, /, (,) using if statement. Store the C value in P and print the operator.
5. Enter any symbol to find in the Symbol Table.
6. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<ctype.h>
void main()
{
    int j=0,i=0,x=0,n;
    int flag=0;
    int *p,*add[10];
    char c,ch='y',srch,b[10],d[10];
    clrscr();
    printf("\n Enter an expression and it is terminated by $");

    while((c=getchar())!='$')
    {
        b[j]=c;
        i++;
    }
    n=i-1;
    i=0;
    printf("Symbol Table \n");
    printf(" \n Symbol\t\t Address\t\t type");
    while(j<=n)
    {
        c=b[j];
        if(isalpha(toascii(c)))
        {
            p=malloc(c);
            add[x]=p;
            d[x]=c;
            printf(" \n \t %c\t\t %d\t\t identifier",c,p);
        }
        if(c=='+'||c=='-'||c=='*'||c=='/'||c=='='||c=='')
        {
            p=malloc(c);
            add[x]=p;
```

```
        d[x]=c;
        printf(" \n\t%c\t\t\t%d\t\tOperator",c,p);
    }
    x++;
    j++;
}
while(ch=='y')
{
    flag=0;
    printf(" \nEnter the symbol to search");
    fflush(stdin);
    srch=getchar();
    for(i=0;i<=n;i++)
    {
        if(srch==d[i])
        {
            printf(" \nSymbol found\t");
            printf("%c \t%s%\n",srch,"@address",add[i]);
            flag=1;
        }
    }
    if(flag==0)
    {
        printf("Symbol not found");
    }
    printf("Do you want to continue (y/n): ");
    fflush(stdin);
    ch=getchar();
}
}
```

OUTPUT

```

Turbo C++ IDE
Enter an expression and it is terminated by $a+b+-c$
Symbol Table
Symbol      Address      type      identifier
    a              1934      identifier
    +              2036      Operator
    b              2084      identifier
    +              2186      Operator
    -              2234      Operator
    c              2284      identifier
Enter the symbol to searcha
Symbol found   a      @address1934
Do you want to continue (y/n): y
Enter the symbol to search+
Symbol found   +      @address2036
Symbol found   +      @address2186
Do you want to continue (y/n): y
Enter the symbol to searchy
Symbol not foundDo you want to continue (y/n): n

```

Viva voce

What are the various data structures used for implementing the symbol table?

1.Linear list 2.binary tree 3. hash table

Ex. No: 2 Develop a lexical analyzer to recognize a few patterns in C.
(Ex. identifiers, constants, comments, operators etc)

Date:

Aim:

To write a C program to develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc).

Algorithm:

1. Start the program.
2. Initialize the symbol table with keywords.
3. Read token by token from the input string.
4. Using finite automation check for keywords, identifiers, constants & then Operators successively.
5. If nothing matches print an error message.
6. Until all tokens are over, repeat above three steps.
7. Print token information.
8. Stop.

Procedure to run the program:

1. Start the program.
2. Lex program consists of three parts. a. Declaration %% b. Translation rules %% c. Auxiliary procedure.
3. The declaration section includes declaration of variables, maintest, constants and regular definitions.
4. Translation rule of lex program are statements of the form a. P1 {action} b. P2 {action} c. ... d. ... e. Pn {action}
5. Write a program in the vi editor and save it with .l extension.
6. Compile the lex program with lex compiler to produce output file aslex.yy.c.
eg \$ lex filename.l
\$ cc lex.yy.c -ll
7. Compile that file with C compiler and verify the output.

Program:

Lexical.C:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
void main()
{
    FILE *fi,*fo,*fop,*fk;
    int flag=0,i=1;
    char c,t,a[15],ch[15],file[20];
    clrscr();
    printf("\n Enter the File Name:");
    scanf("%s",&file);
    fi=fopen(file,"r");
    fo=fopen("inter.c","w");
    fop=fopen("oper.c","r");
    fk=fopen("key.c","r");
    c=getc(fi);
    while(!feof(fi))
    {
        if(isalpha(c)||isdigit(c)||((c=='['||c==']'||c=='.'==1))
```

```

        fputc(c,fo);
    else
    {
        if(c=='\n')
            fprintf(fo,"\t$\t");
        else
            fprintf(fo,"\t%c\t",c);
    }
    c=getc(fi);
}
fclose(fi);
fclose(fo);
fi=fopen("inter.c","r");
printf("\n Lexical Analysis");
fscanf(fi,"%s",a);
printf("\n Line: %d\n",i++);
while(!feof(fi))
{
    if(strcmp(a,"$")==0)
    {
        printf("\n Line: %d \n",i++);
        fscanf(fi,"%s",a);
    }
    fscanf(fop,"%s",ch);
    while(!feof(fop))
    {
        if(strcmp(ch,a)==0)
        {
            fscanf(fop,"%s",ch);
            printf("\t\t%s\t\t\t%s\n",a,ch);
            flag=1;
        }
        fscanf(fop,"%s",ch);
    }
    rewind(fop);
    fscanf(fk,"%s",ch);
    while(!feof(fk))
    {
        if(strcmp(ch,a)==0)
        {
            fscanf(fk,"%k",ch);
            printf("\t\t%s\t\t\tKeyword\n",a);
            flag=1;
        }
        fscanf(fk,"%s",ch);
    }
    rewind(fk);
    if(flag==0)
    {
        if(isdigit(a[0]))
            printf("\t\t%s\t\t\tConstant\n",a);
        else
            printf("\t\t%s\t\t\tIdentifier\n",a);
    }
    flag=0;
    fscanf(fi,"%s",a);
}

```

```

}
getch();
}

```

Key.C:

```

int
void
main
char
if
for
while
else
printf
scanf
FILE
include
stdio.h
conio.h
iostream.h

```

Oper.C:

```

( open para
) closepara
{ openbrace
} closebrace
<lesser
>greater
" doublequote
' singlequote
: colon
; semicolon
# preprocessor
= equal
== assign
% percentage
^ bitwise
& reference
* star
+ add
- sub
\ backslash
/ slash

```

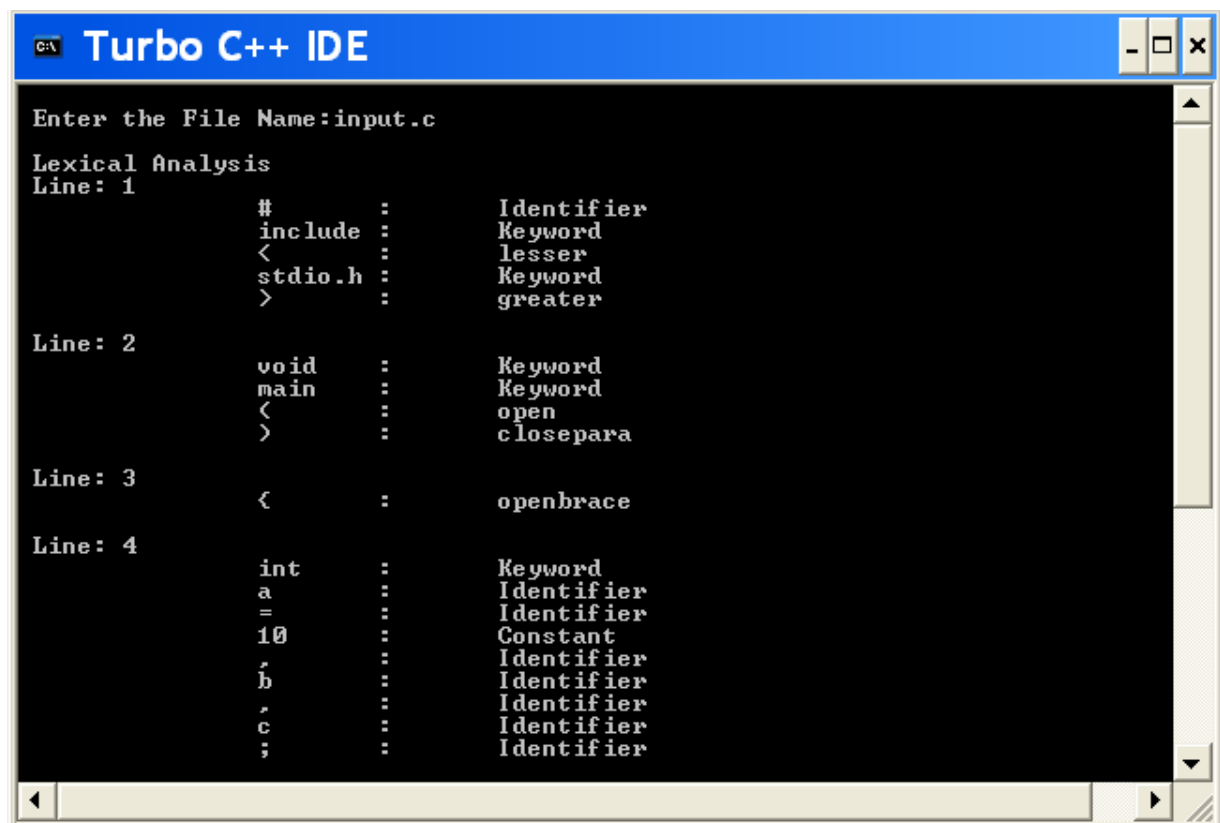
Input.C:

```

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

```

OUTPUT



The screenshot shows the Turbo C++ IDE window with a blue title bar. The main text area displays the output of a lexical analysis performed on a file named 'input.c'. The output is organized by line numbers and lists tokens with their corresponding categories.

```
Enter the File Name:input.c
Lexical Analysis
Line: 1
      #      :      Identifier
      include :      Keyword
      <      :      lesser
      stdio.h :      Keyword
      >      :      greater

Line: 2
      void    :      Keyword
      main    :      Keyword
      <      :      open
      >      :      closepara

Line: 3
      {      :      openbrace

Line: 4
      int     :      Keyword
      a      :      Identifier
      =      :      Identifier
      10     :      Constant
      ,      :      Identifier
      b      :      Identifier
      ,      :      Identifier
      c      :      Identifier
      ;      :      Identifier
```

Result:

The above C program was successfully executed and verified

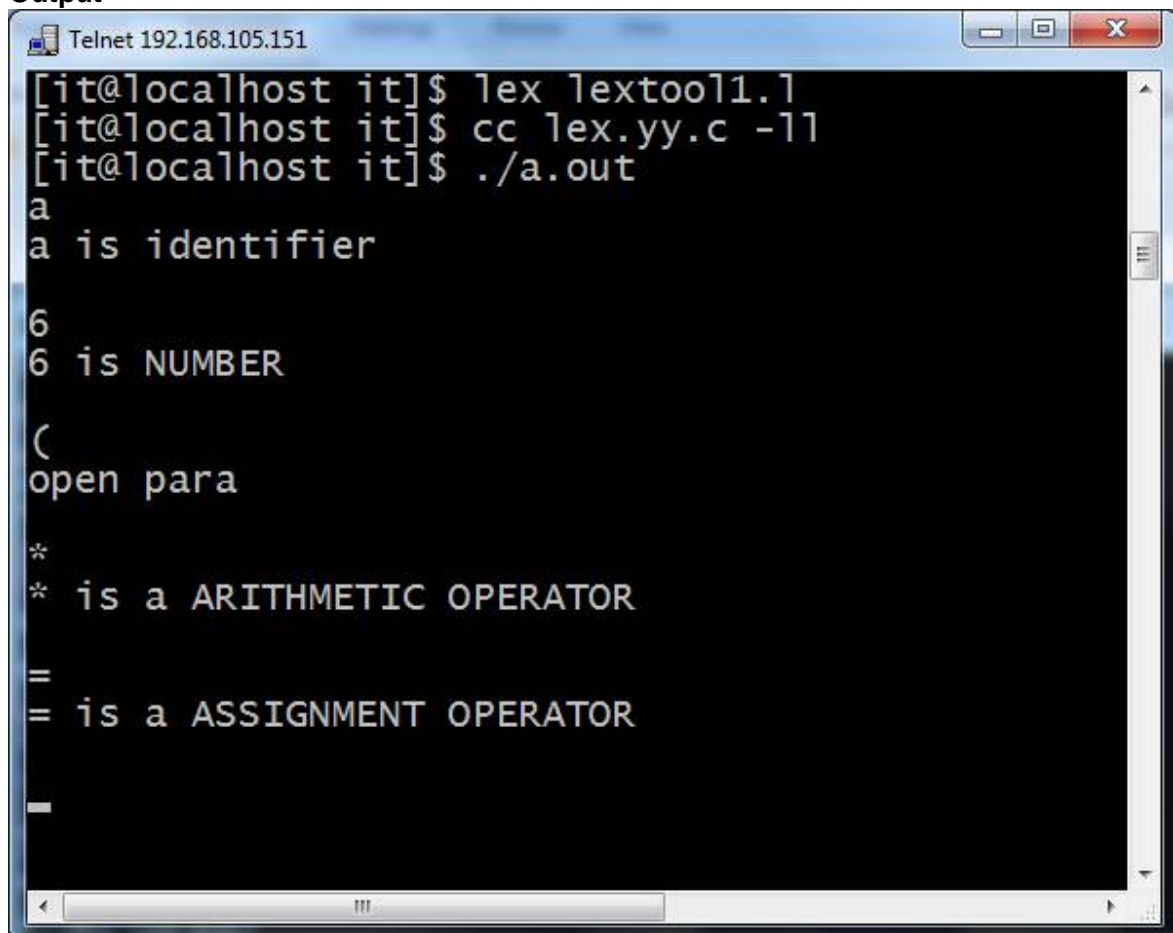
Ex. No:3**Implementation of Lexical Analyzer using Lex Tool****Date:****Aim:** To implement Lexical Analyzer using Lex Tool.**Algorithm:**

1. Start the program
2. Open a file file.c in read and include the yylex() tool for input scanning
3. Define the alphabets and numbers.
4. Print the preprocessor, function, keyword using yytext.lex tool.
5. Print the relational, assignment and all the operator using yytext() tool.
6. Also scan and print where the loop ends and begins.
7. Use yywrap() to enter an error.
8. Stop the program.

Program:**Lextool.l**

```
%{
%}
identifier[a-zA-Z][a-zA-Z0-9]*
%%
#.*    printf("\n%s is PREPROCESSOR DIRECTIVE\n", yytext);
int |
float |
double |
char |
for |
if printf("%s is a keyword\n", yytext);
{identifier} \ ( printf("\n\n FUNCTION CALL\n %s", yytext);
\ {
printf("BLOCK BEGINS\n");
\ }
printf("BLOCK ENDS\n");
{identifier} \ ([0-9]*\ )? printf("%s is identifier\n", yytext);
= printf("%s is a ASSIGNMENT OPERATOR\n", yytext);
[0-9]+ printf("%s is NUMBER\n", yytext);
\< |
\> |
\== |
\>= |
\<= printf("%s is a RELATIONAL OPERATOR\n", yytext);
\ ( printf("open para\n");
\ ) printf("close para\n");
\+ |
\ - |
\ * printf("%s is a ARITHMETIC OPERATOR\n", yytext);
\++ printf("%s is a INCREMENTAL OPERATOR\n", yytext);
\; { ECHO; printf("\n"); }
%%
main()
{
yylex();
}
```

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

OutputA screenshot of a Telnet window titled 'Telnet 192.168.105.151'. The window has standard Windows-style window controls (minimize, maximize, close) in the top right corner. The main area is a black terminal with white text. The text shows a series of commands and their outputs: '[it@localhost it]\$ lex lextool1.1', '[it@localhost it]\$ cc lex.yy.c -ll', and '[it@localhost it]\$./a.out'. Following these commands, the output shows the Lex tool identifying tokens: 'a' is an identifier, '6' is a number, '(' is an opening parenthesis, '*' is an arithmetic operator, and '=' is an assignment operator. The text is as follows:

```
Telnet 192.168.105.151
[it@localhost it]$ lex lextool1.1
[it@localhost it]$ cc lex.yy.c -ll
[it@localhost it]$ ./a.out
a
a is identifier

6
6 is NUMBER

(
open para

*
* is a ARITHMETIC OPERATOR

=
= is a ASSIGNMENT OPERATOR

_
```

Ex.no 4a Program to recognize a valid arithmetic expression that**Date: Uses operator +, -, * and /.**

Aim : To write a yacc and lex program to recognize a valid arithmetic expression that uses operator +, -, * and /.

Algorithm:

Input: Programming language arithmetic expression

Output: A sequence of tokens.

Tokens have to be identified and its respective attributes have to be printed.

Lex:

1. {Declaration and regular definition}

Define header files to include first section

2. [translation rule]

Tokens generated are used in yacc files

[a-z A-Z] alphabets are returned

0-9 one or more combinations of integers

Yacc:

1. Accept token generated in lex part as input

2. Specify the order of procedure

3. Define rules with end points

4. Parse input string from standard input by calling yyparse() main function.

5. Print the result of any rules defined matches as arithmetic expression as valid

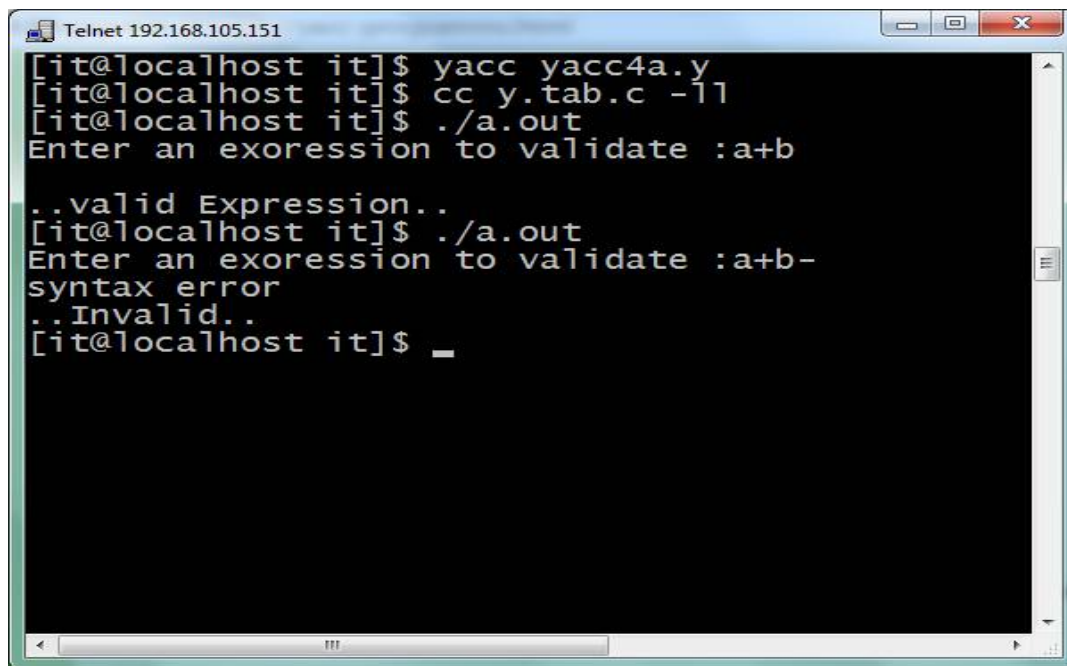
6. If none of the rule defined matches print arithmetic expression is invalid.

Program:**Yacc4a.y(without lex only yacc program)**

```
%{
    #include<stdio.h>
    #include<ctype.h>
    #include<stdlib.h>
}%
%token num let
%left '+' '-'
%left '*' '/'
%%
stmt: stmt '\n'    {printf("\n..valid Expression..\n"); exit(0);}
| expr
|
| error '\n'    {printf("\n..Invalid..\n"); exit(0);}
;
expr: num
| let
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr ')'
%%
main()
{
```

```
printf("Enter an exoression to validate :");
yyvsparse();
}
yylex()
{
int ch;
while((ch=getchar())!=' ');
if(isdigit(ch))
return num;
if(isalpha(ch))
return let;
return ch;
}
yyerror(char *s)
{
printf("%s",s);
}
```

Output :



The screenshot shows a Telnet window titled 'Telnet 192.168.105.151'. The terminal displays the following commands and output:

```
[it@localhost it]$ yacc yacc4a.y
[it@localhost it]$ cc y.tab.c -ll
[it@localhost it]$ ./a.out
Enter an exoression to validate :a+b
..valid Expression..
[it@localhost it]$ ./a.out
Enter an exoression to validate :a+b-
syntax error
..Invalid..
[it@localhost it]$ _
```

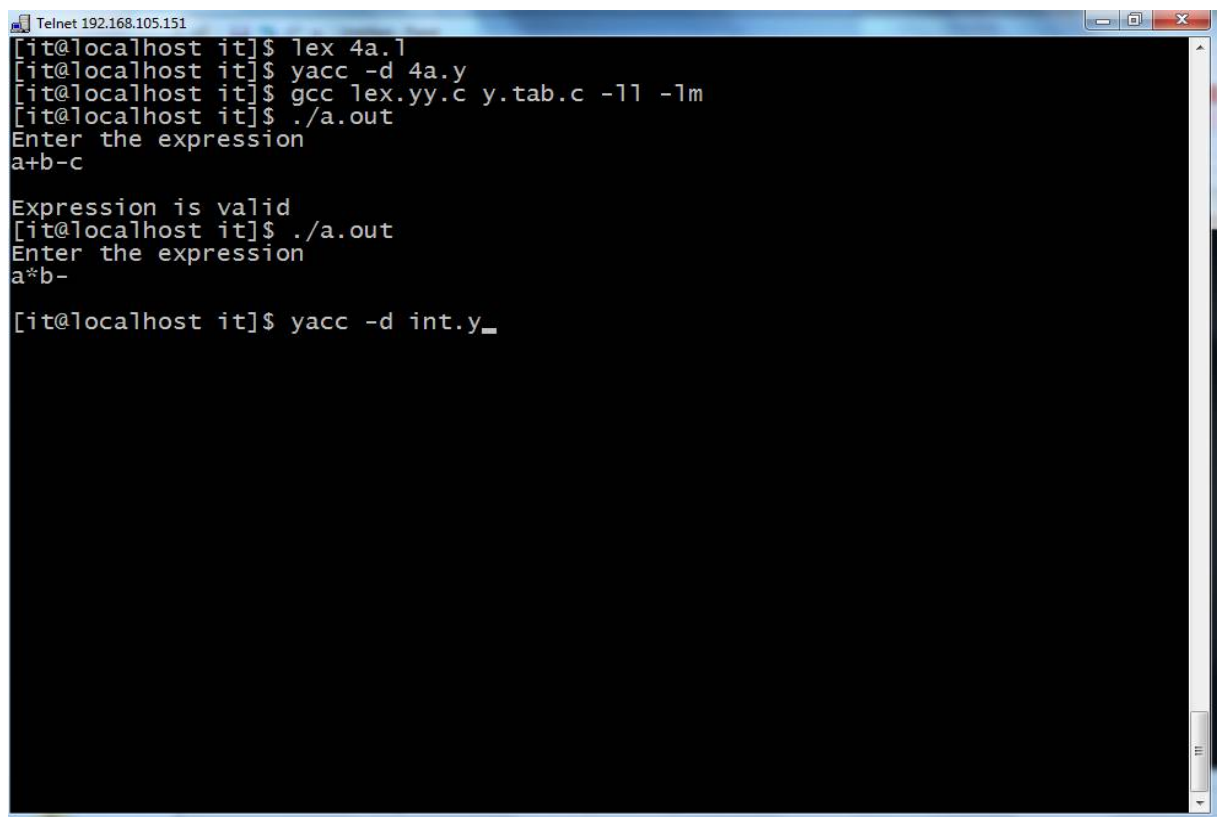
Lex program (with lex and yacc)

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

Yacc program

```
%{
#include <stdio.h>
}%
%token NUMBER ID
%left '+' '-'
%left '*' '/'
%%
expr:
expr '+' expr
|expr '-' expr
|expr '*' expr
|expr '/' expr
|'NUMBER'-'ID'
|'('expr')'
|NUMBER
|ID
;
%%
main()
{
printf("Enter the expression\n");
yyparse();
printf("\nExpression is valid\n");
exit(0);
}
int yyerror(char *s)
{
printf("\nExpression is invalid");
exit(0);
}
```

Output



```
Telnet 192.168.105.151
[it@localhost it]$ lex 4a.l
[it@localhost it]$ yacc -d 4a.y
[it@localhost it]$ gcc lex.yy.c y.tab.c -ll -lm
[it@localhost it]$ ./a.out
Enter the expression
a+b-c

Expression is valid
[it@localhost it]$ ./a.out
Enter the expression
a*b-

[it@localhost it]$ yacc -d int.y_
```

Ex.no :4b Program to recognize a valid variable which starts with a letter followed by any number of letters or digits

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

Algorithm:

Input: Programming language 'if' statement

Output: A sequence of tokens.

Tokens have to be identified and its respective attributes have to be printed

1. Include header file y.tab.h
2. Define tokens
3. Declare [a-z] as L [0-9] as D
4. Declare variable L D P
5. Call function yyerror()
6. If error exists then print "invalid" and exit
7. In main() call yyparse(), if yyparse() print "valid variable"

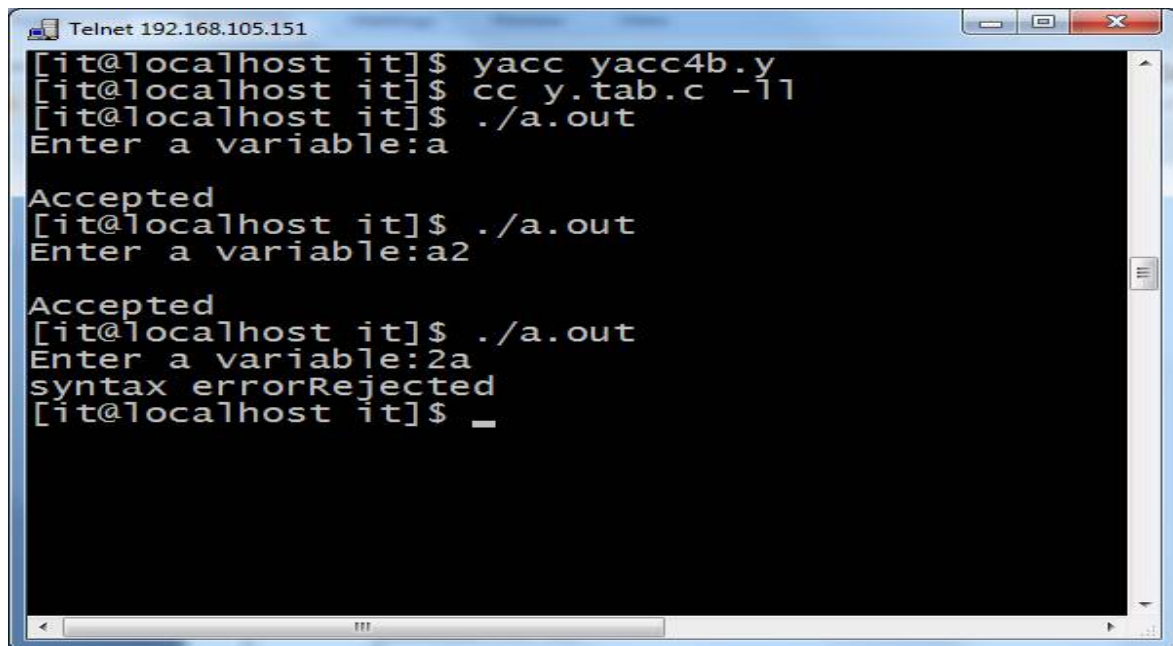
Program: (without lex)

```
%{
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
%
}
%token let dig
%%
TERM: XTERM '\n'
{
printf("\nAccepted\n");
exit(0);
}
|error
{
yyerror ("Rejected\n"); exit(0);
};
XTERM: XTERM let
| XTERM dig
| let
;
%%
main()
{
printf("Enter a variable:");
yyparse();
}
yylex()
{
char ch;
while((ch=getchar())!=' ');
if(isalpha(ch))
return let;
if(isdigit(ch))
return dig;
```



```
return ch;
}
yyerror(char *s)
{
printf("%s",s);
}
```

Output



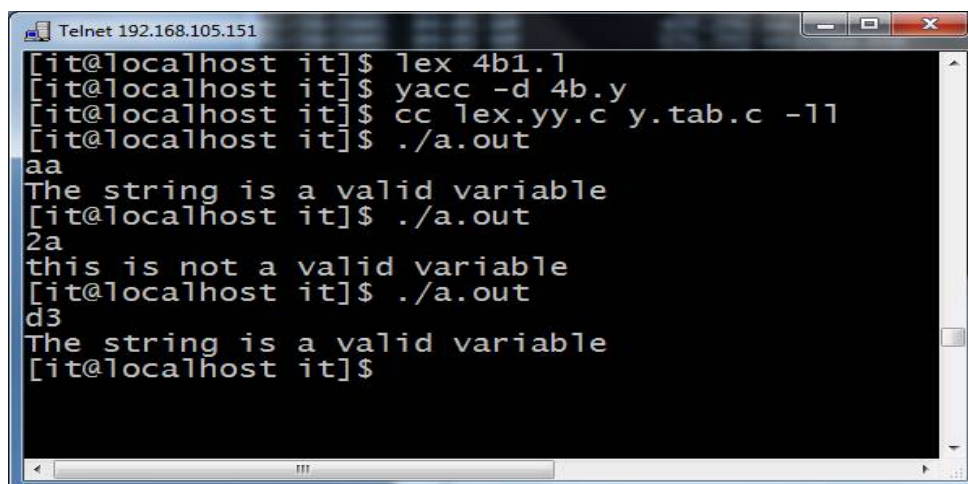
```
Telnet 192.168.105.151
[it@localhost it]$ yacc yacc4b.y
[it@localhost it]$ cc y.tab.c -ll
[it@localhost it]$ ./a.out
Enter a variable:a
Accepted
[it@localhost it]$ ./a.out
Enter a variable:a2
Accepted
[it@localhost it]$ ./a.out
Enter a variable:2a
syntax errorRejected
[it@localhost it]$ _
```

Program: with lex and yacc**Yacc 4b.l**

```
%{
#include "y.tab.h"
extern yylval;
}%
%%
[0-9]+ {yylval=atoi(yytext); return DIGIT;}
[a-zA-Z]+ {return LETTER;}
[\t] ;
\n return 0;
. {return yytext[0];}
%%
```

Yacc 4b.y

```
%{
#include <stdio.h>
}%
%token LETTER DIGIT
%%
variable: LETTER|LETTER rest
;
rest: LETTER rest
|DIGIT rest
|LETTER|DIGIT;
%%
main()
{
  yyparse();
  printf("The string is a valid variable\n");
}
int yyerror(char *s)
{
  printf("this is not a valid variable\n");
  exit(0);
}
```

Output


```
Telnet 192.168.105.151
[it@localhost it]$ lex 4b1.l
[it@localhost it]$ yacc -d 4b.y
[it@localhost it]$ cc lex.yy.c y.tab.c -ll
[it@localhost it]$ ./a.out
aa
The string is a valid variable
[it@localhost it]$ ./a.out
2a
this is not a valid variable
[it@localhost it]$ ./a.out
d3
The string is a valid variable
[it@localhost it]$
```

Ex.no :4c: Implementation of Calculator using LEX and YACC**Date :****Aim:** To write a lex and yacc program to implement Calculator.**Algorithm:**

1. Start the program.
2. Perform the calculation using both the lex and yacc.
3. In the lex tool, if the given expression contains numbers and letters then they are displayed.
4. In the same way, the digits, letters and uminus are identified and displayed using yacc tool.
5. The calculation is performed and the result is displayed.
6. Stop the program

Program: Calci.l

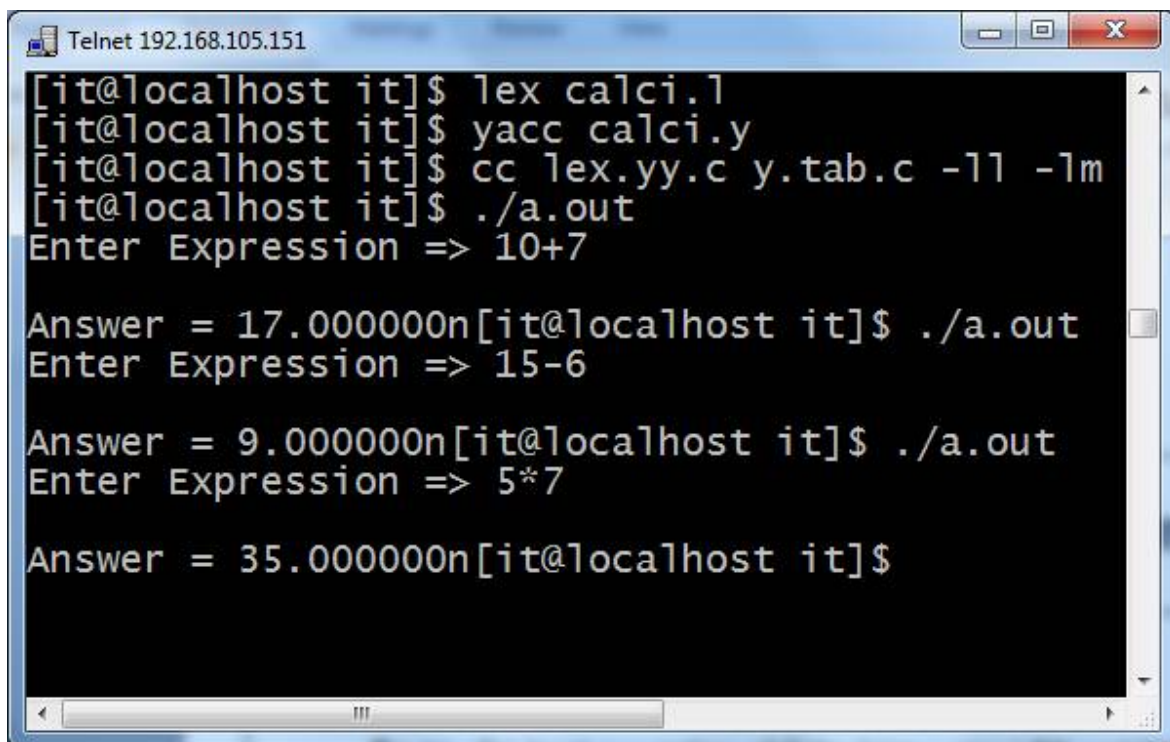
```
%{
#include<stdio.h>
#include<math.h>
#include "y.tab.h"
}%
%%
[0-9]+ {
    yylval.dval=atoi(yytext);
    return NUMBER;
}
[t];
n return 0;
. {return yytext[0];}
%%
void yyerror(char *str)
{
    printf("n Invalid Character...");
}
int main()
{
    printf("Enter Expression => ");
    yyparse();
    return(0);
}
```

Program:calci.y

```
%{
#include<stdio.h>
int yylex(void);
}%
%union
{
    float dval;
}
%token <dval> NUMBER
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%type <dval> exp
```

```
%%  
state : exp {printf("Answer = %fn", $1);} ;  
exp : NUMBER  
    | exp '+' exp {$$=$1+$3;}  
    | exp '-' exp {$$=$1-$3;}  
    | exp '*' exp {$$=$1*$3;}  
    | exp '/' exp {$$=$1/$3;}  
    | '(' exp ')' {$$=$2;}  
    | '-' exp %prec UMINUS {$$=-$2;}  
    ;  
%%
```

Output



```
Telnet 192.168.105.151  
[it@localhost it]$ lex calci.l  
[it@localhost it]$ yacc calci.y  
[it@localhost it]$ cc lex.yy.c y.tab.c -ll -lm  
[it@localhost it]$ ./a.out  
Enter Expression => 10+7  
  
Answer = 17.000000n[it@localhost it]$ ./a.out  
Enter Expression => 15-6  
  
Answer = 9.000000n[it@localhost it]$ ./a.out  
Enter Expression => 5*7  
  
Answer = 35.000000n[it@localhost it]$
```

**EX.NO:5 Convert the BNF rules into Yacc form and write code to
DATE:Generate abstract syntax tree.**

Aim:To convert The BNF rules into Yacc form and write code to generate abstract syntax tree

Algorithm:

Step1: Start the program.

Step2: Declare the declarations as a header file {include}

Step3: Declare the token digit.

Step4: Define the translations rules like line, expr, term, factor

Line: exp '\n' {print("\n %d \n", \$1)}

Expr: expr '+' term (\$\$=\$1=\$3)

Term: term '+' factor(\$\$=\$1*\$3)

Factor

Factor: '(' enter' '\$\$=\$2) % %

Step5: define the supporting C routines

Step6: Stop

Program:

<int.l>

```
%{
#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];
%%
```

<int.y>

```
%{
#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,"",$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,$2);
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 Arg1 Arg2 Result" "\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:
$vi test.c
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:
$lex int.l
$yacc -d int.y
$gcc lex.yy.c y.tab.c -ll -lm
$./a.out test.c

```

```
Telnet 192.168.105.151
[it@localhost it]$ lex int.l
[it@localhost it]$ yacc -d int.y
[it@localhost it]$ cc lex.yy.c y.tab.c -ll
[it@localhost it]$ ./a.out test.c

-----
Pos Operator Arg1 Arg2 Result
-----
0      <      a      b      t0
1      ==     t0     FALSE 5
2      +      a      b      t1
3      =      t1     a      5
4      GOTO
5      <      a      b      t2
6      ==     t2     FALSE 10
7      +      a      b      t3
8      =      t3     a      5
9      GOTO
10     <=     a      b      t4
11     ==     t4     FALSE 15
12     -      a      b      t5
13     =      t5     c      17
14     GOTO
15     +      a      b      t6
16     =      t6     c

-----

[it@localhost it]$ _
```

Ex.no :6**Implement type checking****Date:****Aim:**To implement type checking.**Algorithm:****AIM:**

To develop a C program to test whether a given identifier is valid or not.

ALGORITHM:

- Read the given input string.
- Check the initial character of the string is numerical or any special character except '_' then print it is not a valid identifier.
- Otherwise print it as valid identifier if remaining characters of string doesn't contains any special characters except '_'

PROCEDURE: Go to debug -> run or press CTRL + F9 to run the program.

PROGRAM

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
char a[10];
int flag, i=1;
clrscr();
printf("\n Enter an identifier:");
gets(a);
if(isalpha(a[0]))
flag=1;
else
printf("\n Not a valid identifier");
while(a[i]!='\0')
```

```
{
if(!isdigit(a[i])&&!isalpha(a[i]))
{
flag=0;
break;
}
i++;
}
if(flag==1)
printf("
\
n Valid identifier");
getch();
}
```

OUTPUT:

Input

:

Enter an identifier: first

Output:

Valid

identifier

Enter an identifier: laqw

Not a valid identifier

Ex.no:7 Implement control flow analysis and Data flow Analysis

Aim:To implement control flow analysis and Data flow Analysis

Algorithm:

Step1:

Program:

```
# include<stdio.h>
# include<conio.h>
#include<alloc.h>
#include<string.h>
struct Listnode
{
    char data[50];
    int leader,block,u_goto,c_goto;
    struct Listnode *next;
    char label[10],target[10];
}*temp,*cur,*first=NULL,*last=NULL,*cur1;

FILE *fpr;
void createnode(char code[50])
{
    temp=(struct Listnode *)malloc(sizeof(struct Listnode));
    strcpy(temp->data,code);
    strcpy(temp->label,'\0');
    strcpy(temp->target,'\0');

    temp->leader=0;
    temp->block=0;
    temp->u_goto=0;
    temp->c_goto=0;
    temp->next=NULL;

    if(first==NULL)
    {
        first=temp;
        last=temp;
    }
    else
    {
        last->next=temp;
        last=temp;
    }
}

void printlist()
{
    cur=first;
    printf("\nMIR code is \n\n");
    while(cur!=NULL)
```

```

        {
            printf("\ncode:%s",cur->data);
            printf("\nleader:%d\t",cur->leader);
            printf("block:%d\t",cur->block);
            printf("u_goto:%d\t",cur->u_goto);
            printf("c_goto:%d\t",cur->c_goto);
            printf("label:%s\t",cur->label);
            printf("target:%s\n",cur->target);

            cur=cur->next;
        }
    }

void main()
{
    char codeline[50];
    char c,dup[50],target[10];
    char *substring,*token;
    int i=0,block,block1;
    int j=0;
    fpr= fopen("input.txt","r");
    clrscr();
    while((c=getc(fpr))!=EOF)
    {
        if(c!='\n')
        {
            codeline[i]=c;
            i++;
        }
        else
        {
            codeline[i]='\0';
            createnode(codeline);
            i=0;
        }
    }
    //create last node
    codeline[i]='\0';
    createnode(codeline);
    fclose(fpr);
    // printlist();

    // find out leaders,conditional stmts
    cur=first;
    cur->leader=1;
    while(cur!=NULL)
    {
        substring=strstr((cur->data),"if");
        if(substring==NULL)
        {
            if((strstr((cur->data),"goto"))!=NULL)
            {
                cur->u_goto=1;
                (cur->next)->leader=1;
            }
        }
        else
        {

```

```

        cur->c_goto=1;
        (cur->next)->leader=1;
    }
    substring=strstr((cur->data),":");
    if(substring!=NULL)
    {
        cur->leader=1;
    }
    substring=strstr((cur->data),"call");
    if(substring!=NULL)
    {
        cur->leader=1;
    }
    if(strstr(cur->data,"return")!=NULL)
    {
        cur->leader=1;
        (cur->next)->leader=1;
    }
    cur=cur->next;
}

//to find labels and targets
cur=first;
while(cur!=NULL)
{
    if((cur->u_goto==1)||(cur->c_goto==1))
    {
        substring=strstr(cur->data,":");
        if(substring!=NULL)
        {
            token=strstr(substring,"L ");
            if(token!=NULL)
                strcpy(cur->target,token);
        }
        else
        {
            substring=strstr(cur->data,"L");
            if(substring!=NULL)
                strcpy(cur->target,substring);
        }
    }

    if(strstr(cur->data,":")!=NULL)
    {
        strcpy(dup,cur->data);
        token=strtok(dup,":");
        // printf("\ntoken:%s",token);
        if(token!=NULL)
            strcpy(cur->label,token);
    }
    cur=cur->next;
}

// printlist();
//to identify blocks
cur=first;
while(cur!= NULL)

```

```

{
cur=cur->next;
if((cur->leader)==1)
{
j++;
cur->block=j;
}
else
cur->block=j;
}
// printlist();

// print basic blocks
printf("\n\n.....Basic Blocks.....\n");
cur=first;
j=0;
printf("\nBlock %d:",j);
while(cur!=NULL)
{

if ((cur->block)==j)
{

printf("%s",cur->data);
printf("\n\t");
cur=cur->next;
}
else
{
j++;
printf("\nBlock %d:",j);
}}
//to output the control flow from each block
printf ("\t\t.....Control Flow.....\n\n");
cur=first;
i=0;
while(cur!=NULL)
{
if((cur->block)!=((cur->next)->block))
{
block=cur->block;
if(cur->u_goto==1)
{
strcpy(target,cur->target);
cur1=first;
while(cur1!=NULL)
{
if(strcmp(cur1->label,target)==0)
{
block1=cur1->block;
printf("\t\tBlock%d----->Block%d\n",block,block1);
}
cur1=cur1->next;
}
}
}
else if(cur->c_goto==1)

```



```

    {
strcpy(target,cur->target);
    cur1=first;
while(cur1!=NULL)
    {
if(strcmp(cur1->label,target)==0)
    {
        block1=cur1->block;
        printf("\t\tBlock%d---TRUE--->Block%d---FALSE--->Block%d\n",block,block1,(block+1));
    }
    cur1=cur1->next;
    }
}
else if(strstr(cur->data,"return")==NULL)
    {
        printf("\t\tBlock%d----->Block%d\n",block,(block+1));
    }
else
printf("\t\tBlock%d----->NULL\n",block);
    }

cur=cur->next;
}
cur=last;
block= cur->block;
printf("\t\tBlock%d----->NULL",block);
getch();
}

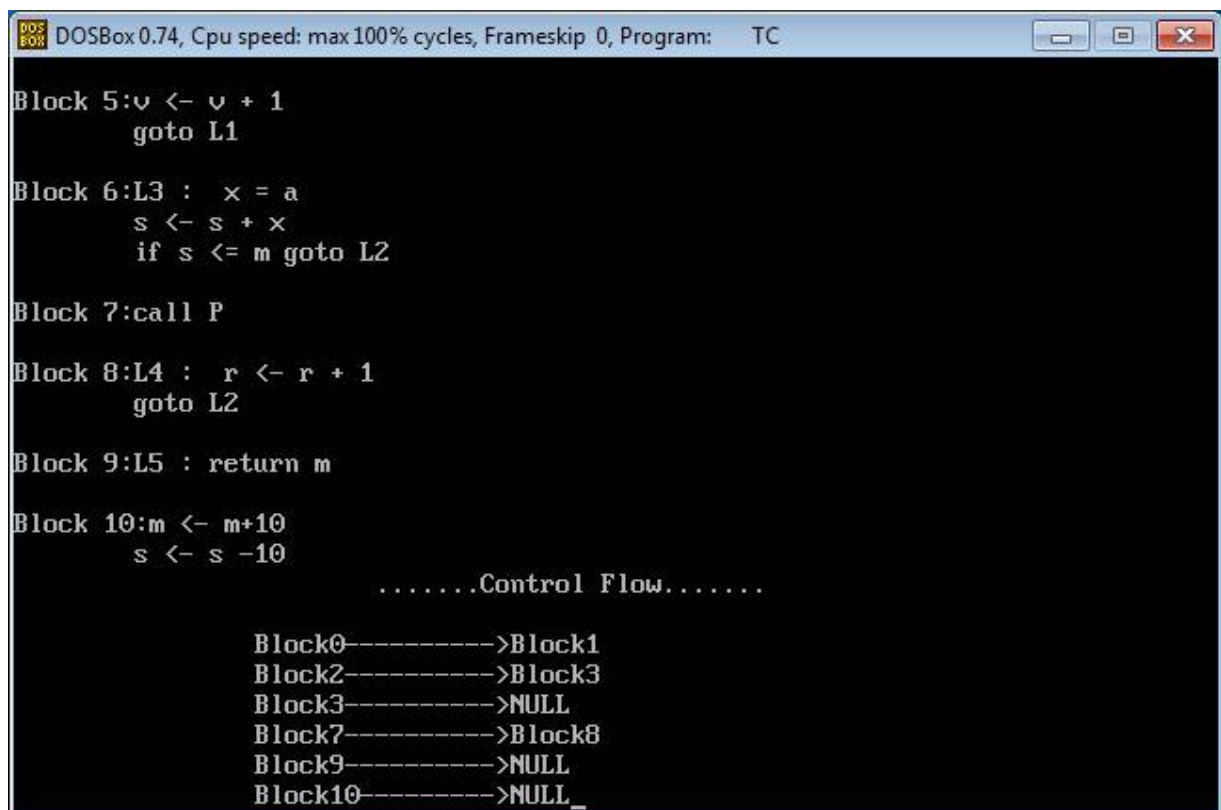
```

Input file: Input.txt

```

m <- 0
v <- 0
L1 : if v < n goto L5
r <- v
s <- 0
return
L2 : if r >= n goto L3
v <- v + 1
goto L1
L3 : x = a
s <- s + x
if s <= m goto L2
call P
L4 : r <- r + 1
goto L2
L5 : return m
m <- m+10
s <- s -10

```

Output:

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Block 5: v <- v + 1  
        goto L1  
  
Block 6: L3 : x = a  
        s <- s + x  
        if s <= m goto L2  
  
Block 7: call P  
  
Block 8: L4 : r <- r + 1  
        goto L2  
  
Block 9: L5 : return m  
  
Block 10: m <- m+10  
        s <- s -10  
  
        .....Control Flow.....  
  
        Block0----->Block1  
        Block2----->Block3  
        Block3----->NULL  
        Block7----->Block8  
        Block9----->NULL  
        Block10----->NULL_
```

Ex.no :8 Implement any one storage allocation strategies(Heap,Stack,Static)

Date:

Aim:To implement storage allocation strategies using Static.

Algorithm:

Step1: Start the program

Step2: Define the pre-processor MAXNUM as 3

Step3: define the sum_up(void) function.

Step4: Inside main function declare count and initialize it to 0.

Step5: Iterate the loop till count <MAXNUM and invoke sum_up().

Step6: Inside sum_up() function declare sum as static variable and get the the number and sum it.

Step7: Print the sum value.

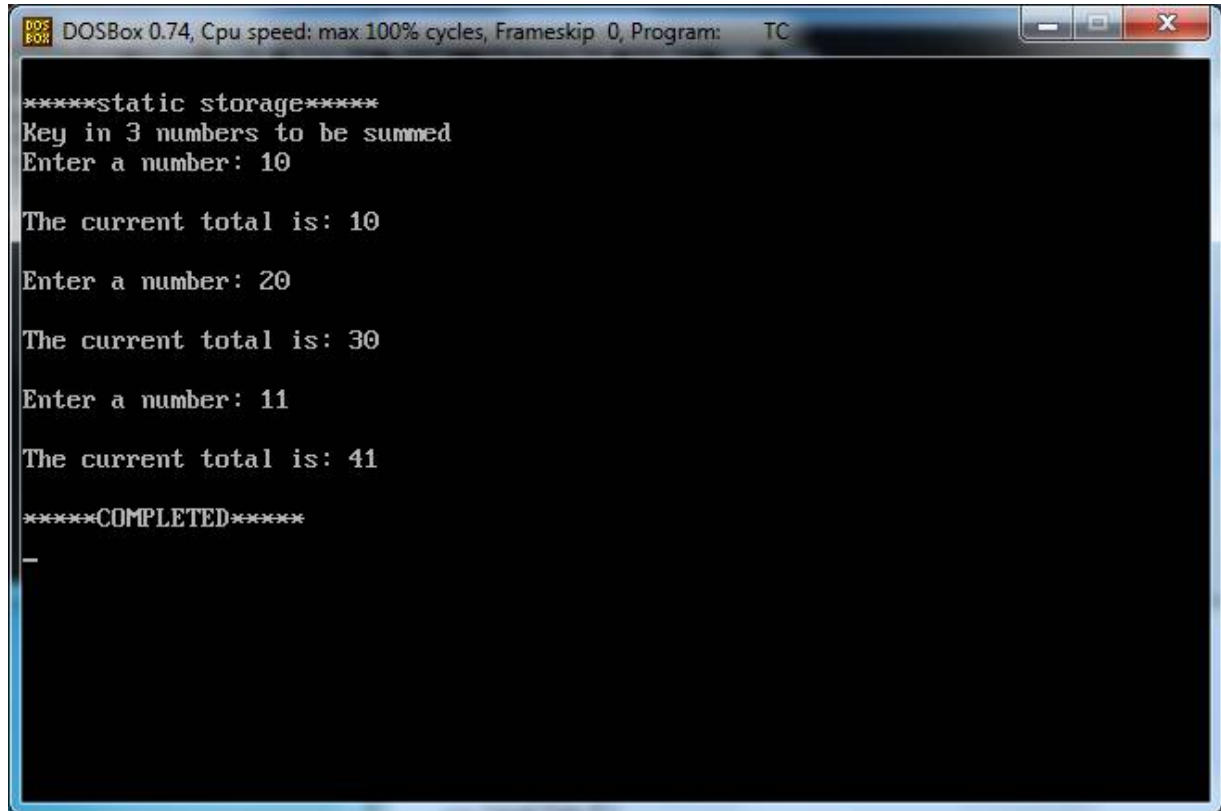
Step8: Stop the program.

Program:

```
#include <stdio.h>
#define MAXNUM 3
void sum_up(void);
int main()
{
    int count;
    printf("\n*****static storage*****\n");
    printf("Key in 3 numbers to be summed ");
    for(count = 0; count < MAXNUM; count++)
        sum_up();
    printf("\n*****COMPLETED*****\n");
    return 0;
}
void sum_up(void)
{
    /* at compile time, sum is initialized to 0 */
    static int sum = 0;
    int num;
    printf("\nEnter a number: ");
    scanf("%d", &num);
```

```
sum += num;  
printf("\nThe current total is: %d\n", sum);  
}
```

Output:



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip: 0, Program: TC  
*****static storage*****  
Key in 3 numbers to be summed  
Enter a number: 10  
The current total is: 10  
Enter a number: 20  
The current total is: 30  
Enter a number: 11  
The current total is: 41  
*****COMPLETED*****  
_
```

Ex.no: 9**Construction of DAG****Date:****Aim:**To construct DAG for the given expression.**Algorithm for labeling the nodes of tree(DAG):****1. For Leaf Nodes**

Assign label 1 to left child node and label 0 to right child node.

2. For Interior Nodes

Case 1: If Node's children's labels are different, then

Node's Label = Maximum among Children's Labels.

Case 2: If Node's children's labels are same, then

Node's Label = (Left Child's Label OR Right Child's Label) + 1

Algorithm for Generating Assembly Code:

(Say, R is a Stack consists of registers)

void gencode(Node)

{

if Node is intermediate node of tree(DAG)

{

Case 1: if Node's left child's label == 1 && Node's right child's label == 0 && Node's left child is leaf node && Node's right child is leaf node

{

print "MOV Node's left child's data,R[top]"

print "op Node's right child's data,R[top]"

}

Case 2: else if Node's left child's label >= 1 && Node's right child's label == 0

{

gencode(Node's left child);

print "op Node's right child's data,R[top]"

}

Case 3: else if Node's left child's label < Node's right child's label

{

int temp;

Swap Register Stack's top and second top element;

gencode(Node's right child);

temp=pop();

gencode(Node's left child);

push(temp);

Swap Register Stack's top and second top element;

print "op R[top-1],R[top]"

}

Case 4: else if Node's left child's label >= Node's right child's label

{

int temp;

gencode(Node's left child);

temp=pop();

gencode(Node's right child);

push(temp);

```

        print "op R[top-1],R[top]"
    }
}

else if Node is leaf node and it is left child of it's immediate parent
{
    print "MOV Node's data,R[top]"
}

}

```

Program:

```

#include<stdlib.h>
#include<iostream>
using namespace std;
/* We will implement DAG as Strictly Binary Tree where each node has zero or two children */
struct bin_tree
{
    char data;
    int label;
    struct bin_tree *right, *left;
};
typedef bin_tree node;

class dag
{
private:
    /* R is stack for storing registers */
    int R[10];
    int top;
    /* op will be used for opcode name w.r.t. arithmetic operator e.g. ADD for + */
    char *op;
public:
    void initializestack(node *root)
    {
        /* value of top = index of topmost element of stack R = label of Root of tree(DAG) minus one */
        top=root->label - 1;
    }

    /* Allocating Stack Registers */
    int temp=top;
    for(int i=0;i<=top;i++)

```

```

    {
        R[i]=temp;
    temp--;
    }
}
/* insertnode() and insert() functions are for adding nodes to tree(DAG) */
void insertnode(node **tree,char val)
{
    node *temp = NULL;
    if(!(*tree))
    {
        temp = (node *)malloc(sizeof(node));
        temp->left = temp->right = NULL;
        temp->data = val;
        temp->label=-1;
        *tree = temp;
    }
}
void insert(node **tree,char val)
{
    char l,r;
    int numofchildren;
    insertnode(tree, val);
    cout<< "\nEnter number of children of " << val << " :";
    cin>> numofchildren;
    if(numofchildren==2)
    {
        cout<< "\nEnter Left Child of " << val << " :";
        cin>> l;
        insertnode(&(*tree)->left,l);

        cout<< "\nEnter Right Child of " << val << " :";
        cin>> r;
        insertnode(&(*tree)->right,r);

        insert(&(*tree)->left,l);
        insert(&(*tree)->right,r);
    }
}

```

```

    }
}
/* findleafnodelabel() will find out the label of leaf nodes of tree(DAG) */
void findleafnodelabel(node *tree,int val)
{
    if(tree->left != NULL && tree->right !=NULL)
    {
        findleafnodelabel(tree->left,1);
        findleafnodelabel(tree->right,0);
    }
    else
    {
        tree->label=val;
    }
}

/* findinteriornodelabel() will find out the label of interior nodes of tree(DAG) */
void findinteriornodelabel(node *tree)
{
    if(tree->left->label== -1)
    {
        findinteriornodelabel(tree->left);
    }
    else if(tree->right->label== -1)
    {
        findinteriornodelabel(tree->right);
    }
    else
    {
        if(tree->left != NULL && tree->right !=NULL)
        {
            if(tree->left->label == tree->right->label)
            {
                tree->label=(tree->left->label)+1;
            }
            else
            {

```



```

if(tree->left->label > tree->right->label)
{
tree->label=tree->left->label;
}
else
{
tree->label=tree->right->label;
}
}
}
}
}
}

```

/* function print_inorder() will print inorder of nodes. Here we are also printing label of each node of tree(DAG) */

```

void print_inorder(node * tree)
{
if (tree)
{
print_inorder(tree->left);
cout<< tree->data <<" with Label " << tree->label << "\n";
print_inorder(tree->right);
}
}

```

/* function swap() will swap the top and second top elements of Register stack R */

```

void swap()
{
int temp;
temp=R[0];
R[0]=R[1];
R[1]=temp;
}

```

/* function pop() will remove and return topmost element of stack */

```

int pop()
{
int temp=R[top];
top--;
}

```

```

return temp;
}
/* function push() will increment top by one and will insert element at top position of
Register stack */
void push(int temp)
{
top++;
R[top]=temp;
}
/* nameofoperation() will return opcode w.r.t. arithmetic operator */
void nameofoperation(char temp)
{
switch(temp)
{
case '+': op =(char *)"ADD"; break;
case '-': op =(char *)"SUB"; break;
case '*': op =(char *)"MUL"; break;
case '/': op =(char *)"DIV"; break;
}
}
/* gencode() will generate Assembly code w.r.t. labels of tree(DAG) */
void gencode(node * tree)
{
if(tree->left != NULL && tree->right != NULL)
{
if(tree->left->label == 1 && tree->right->label == 0 && tree->left->left==NULL && tree->left-
>right==NULL && tree->right->left==NULL && tree->right->right==NULL)
{
cout << "MOV " << tree->left->data << ", " << "R[" << R[top] << "]\n";
nameofoperation(tree->data);
cout << op << " " << tree->right->data << ",R[" << R[top] << "]\n";
}

else if(tree->left->label >= 1 && tree->right->label == 0)
{
gencode(tree->left);
nameofoperation(tree->data);
}
}
}

```

```

cout << op << " " << tree->right->data << ",R[" << R[top] << "]\n";
}

else if(tree->left->label < tree->right->label)
{
int temp;
swap();
gencode(tree->right);
temp=pop();
gencode(tree->left);
push(temp);
swap();
nameofoperation(tree->data);
cout<< op << " " << "R[" << R[top-1] << "],R[" << R[top] << "]\n";
}

else if(tree->left->label >= tree->right->label)
{
int temp;
gencode(tree->left);
temp=pop();
gencode(tree->right);
push(temp);
nameofoperation(tree->data);
cout<< op << " " << "R[" << R[top-1] << "],R[" << R[top] << "]\n";
}

}

else if(tree->left == NULL && tree->right == NULL && tree->label == 1)
{
cout << "MOV " << tree->data << ",R[" << R[top] << "]\n";
}

}

/* deltree() will free the memory allocated for tree(DAG) */

```

```

void deltree(node * tree)
{
if (tree)
    {
deltree(tree->left);
deltree(tree->right);
free(tree);
    }
}

};

```

/* Program execution will start from main() function */

```

int main()
{
node *root;
root = NULL;
node *tmp;
char val;
int i,temp;

dag d;

    /* Inserting nodes into tree(DAG) */

cout<< "\nEnter root of tree:";
cin>> val;

d.insert(&root,val);

    /* Finding Labels of Leaf nodes */
d.findleafnodelabel(root,1);

    /* Finding Labels of Interior nodes */
while(root->label == -1)

```

```

d.findinteriornodelabel(root);
    /* Initializing Stack contents and top variable */
d.initializestack(root);

    /* Printing inorder of nodes of tree(DAG) */
cout<< "\nInorder Display:\n";
    d.print_inorder(root);

    /* Printing assembly code w.r.t. labels of tree(DAG) */
cout<< "\nAssembly Code:\n";
d.gencode(root);

    /* Deleting all nodes of tree */
d.deltree(root);

return 0;
}

```

Output1:

```

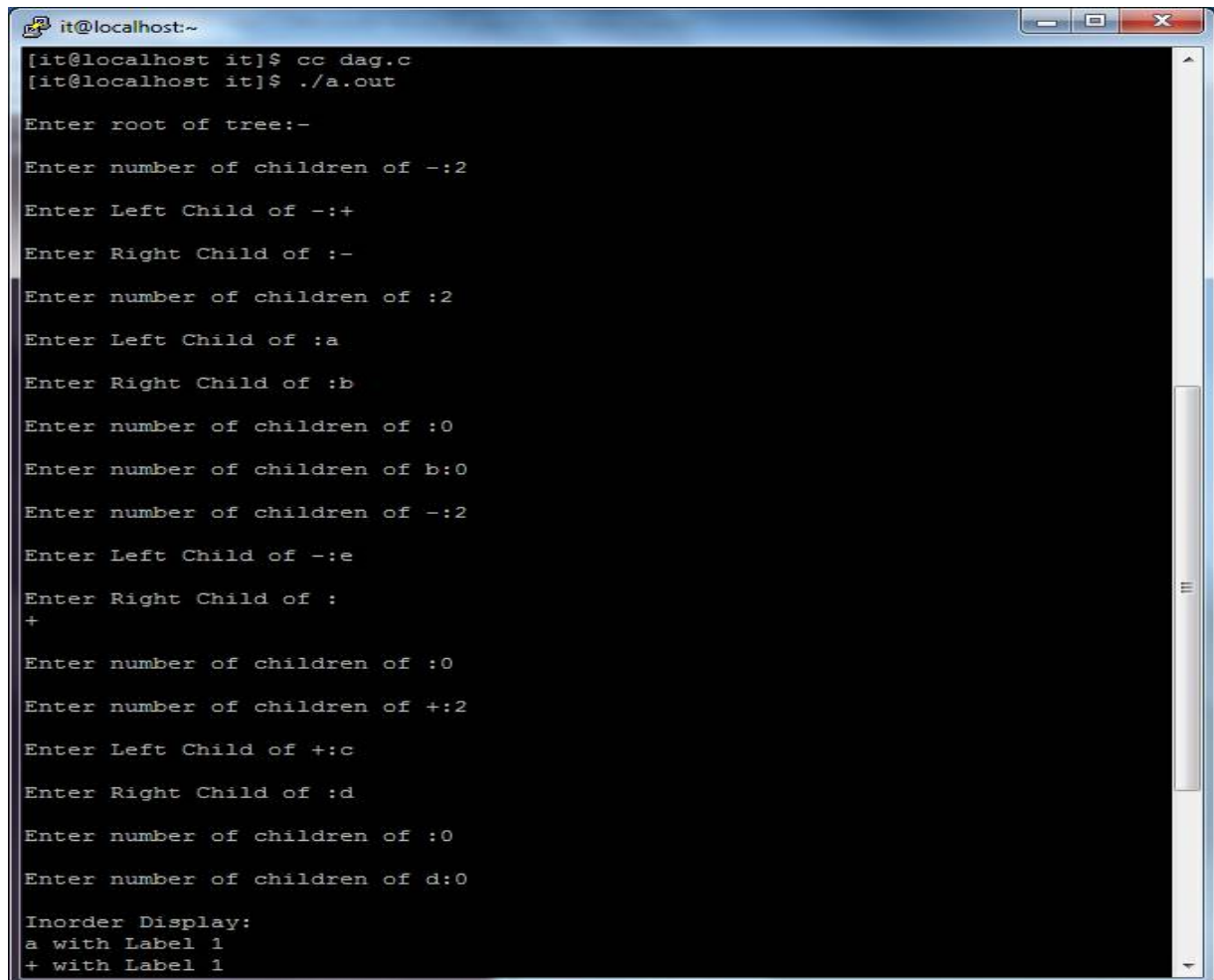
it@localhost:~
Expect frequent system downtime.
[it@localhost it]$ ls
4a.l  4b.l  calci.l  dag.c  lextool1.l  test.c  yacc4b.y
4a.y  4b.y  calci.y  int.l  lextool1.l  var.c   y.tab.c
4b1.l  a.out  dag1.c  int.y  lex.yy.c   yacc4a.y y.tab.h
[it@localhost it]$ cc dag.c
[it@localhost it]$ ./a.out

Enter root of tree: +
Enter number of children of +: 2
Enter Left Child of +: a
Enter Right Child of : b
Enter number of children of : 0
Enter number of children of b: 0

Inorder Display:
a with Label 1
+ with Label 1
b with Label 0

Assembly Code:
MOV a,R[0]
ADD b,R[0]
[it@localhost it]$

```

OUTPUT2:

```
it@localhost:~  
[it@localhost it]$ cc dag.c  
[it@localhost it]$ ./a.out  
Enter root of tree:-  
Enter number of children of -:2  
Enter Left Child of -: +  
Enter Right Child of :-  
Enter number of children of :2  
Enter Left Child of :a  
Enter Right Child of :b  
Enter number of children of :0  
Enter number of children of b:0  
Enter number of children of -:2  
Enter Left Child of -:e  
Enter Right Child of :  
+  
Enter number of children of :0  
Enter number of children of +:2  
Enter Left Child of +:c  
Enter Right Child of :d  
Enter number of children of :0  
Enter number of children of d:0  
Inorder Display:  
a with Label 1  
+ with Label 1
```

```
Inorder Display:
```

```
a with Label 1  
+ with Label 1  
b with Label 0  
- with Label 2  
e with Label 1  
- with Label 2  
c with Label 1  
+ with Label 1  
d with Label 0
```

```
Assembly Code:
```

```
MOV e,R[1]  
MOV c,R[0]  
ADD d,R[0]  
SUB R[0],R[1]  
MOV a,R[0]  
ADD b,R[0]  
SUB R[1],R[0]  
[it@localhost it]$
```

Ex.no: 10 **Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.**

Aim: To write a C program to implement the code generation algorithm.

Algorithm:

Input: Set of three address code sequence.

Output: Assembly code sequence for three address codes (opd1=opd2, op, opd3).

Method:

- 1- Start the program
- 2- Get address code sequence.
- 3- Determine current location of 3 using address (for 1st operand).
- 4- If current location not already exist generate move (B,O).
- 5- Update address of A(for 2nd operand).
- 6- If current value of B and () is null, exist.
- 7- If they generate operator () A,3 ADPR.
- 8- Store the move instruction in memory.
- 9- Stop.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#include<graphics.h>
typedef struct
{
char var[10];
int alive;
}
regist;
regist preg[10];
void substring(char exp[],int st,int end)
{
int i,j=0;
char dup[10]="";
```



```

for(i=st;i<end;i++)
dup[j++]=exp[i];
dup[j]='0';
strcpy(exp,dup);
}
int getregister(char var[])
{
int i;
for(i=0;i<10;i++) {
if(preg[i].alive==0) {
strcpy(preg[i].var,var);
break;
}}
return(i);
}
void getvar(char exp[],char v[])
{
int i,j=0;
char var[10]="";
for(i=0;exp[i]!='\0';i++)
if(isalpha(exp[i]))
var[j++]=exp[i];
else
break;
strcpy(v,var);
}
void main()
{
char basic[10][10],var[10][10],fstr[10],op;
int i,j,k,reg,vc,flag=0;
clrscr();
printf("\nEnter the Three Address Code:\n");
for(i=0;;i++)
{
gets(basic[i]);
if(strcmp(basic[i],"exit")==0)
break;

```

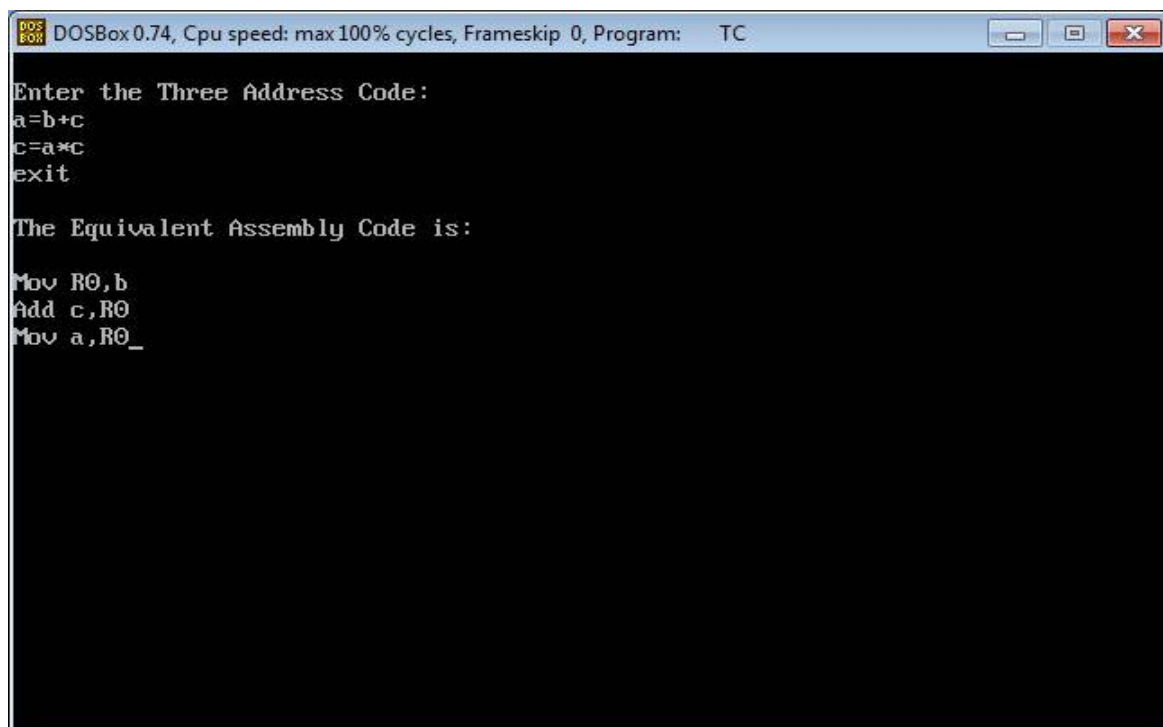
```

}
printf("\nThe Equivalent Assembly Code is:\n");
for(j=0;j<i;j++)
{
getvar(basic[j],var[vc++]);
strcpy(fstr,var[vc-1]);
substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
getvar(basic[j],var[vc++]);
reg=getregister(var[vc-1]);
if(preg[reg].alive==0)
{
printf("\nMov R%d,%s",reg,var[vc-1]);
preg[reg].alive=1;
}
op=basic[j][strlen(var[vc-1])];
substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
getvar(basic[j],var[vc++]);
switch(op)
{
case '+': printf("\nAdd"); break;
case '-': printf("\nSub"); break;
case '*': printf("\nMul"); break;
case '/': printf("\nDiv"); break;
}
flag=1;
for(k=0;k<=reg;k++)
{
if(strcmp(preg[k].var,var[vc-1])==0)
{
printf("R%d, R%d",k,reg);
preg[k].alive=0;
flag=0;
break;
}
}
if(flag)
{

```

```
printf(" %s,R%d",var[vc-1],reg);  
printf("\nMov %s,R%d",fstr,reg);  
}  
strcpy(preg[reg].var,var[vc-3]);  
getch();  
}  
}
```

Output



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip: 0, Program: TC  
Enter the Three Address Code:  
a=b+c  
c=a*c  
exit  
The Equivalent Assembly Code is:  
Mov R0,b  
Add c,R0  
Mov a,R0_
```

EX.NO:11**IMPLEMENTATION OF CODE OPTIMIZATION TECHNIQUES****AIM:**

To write a C program to implement the code optimization algorithm.

ALGORITHM:

The code generation algorithm takes as input a sequence of three – address statements constituting a basic block. For each three – address statement of the form $x := y \text{ op } z$ we perform the following actions:

1. Invoke a function `getreg` to determine the location L where the result of the computation $y \text{ op } z$ should be stored. L will usually be a register, but it could also be a memory location.
2. We shall describe `getreg` shortly, L to place a copy of y in L . If the value of y is currently both in memory and a register. If the value of y is not already in L , generate the instruction `MOV y',` (one of) the current location(s) of y . prefer the register for y .
3. Consult the address descriptor for y to determine y is a current location of z . Again, prefer a register to a memory location if z is in both.
4. Update the address descriptor of x to indicate that x is in location L . If L is a register, update its descriptor to indicate that it contains the value of x , and remove x from all other register descriptors. L where z .
5. Generate the instruction `OP z`
6. If the current values of y and/or z have no next users, are not live on exit from the block, and are in register descriptor to indicate that, after execution of $x := y \text{ op } z$, those registers no longer will contain y and/or z , respectively.

Input: Set of 'L' values with corresponding 'R' values.

Output: Intermediate code & Optimized code after eliminating common expressions.

PROGRAM:

```
#include<stdio.h>
#include<conio.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;
clrscr();
printf("enter no of values");
scanf("%d",&n);
```

```

for(i=0;i<n;i++)
{
printf("left\t");
op[i].l=getche();
printf("right:\t");
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++)
{

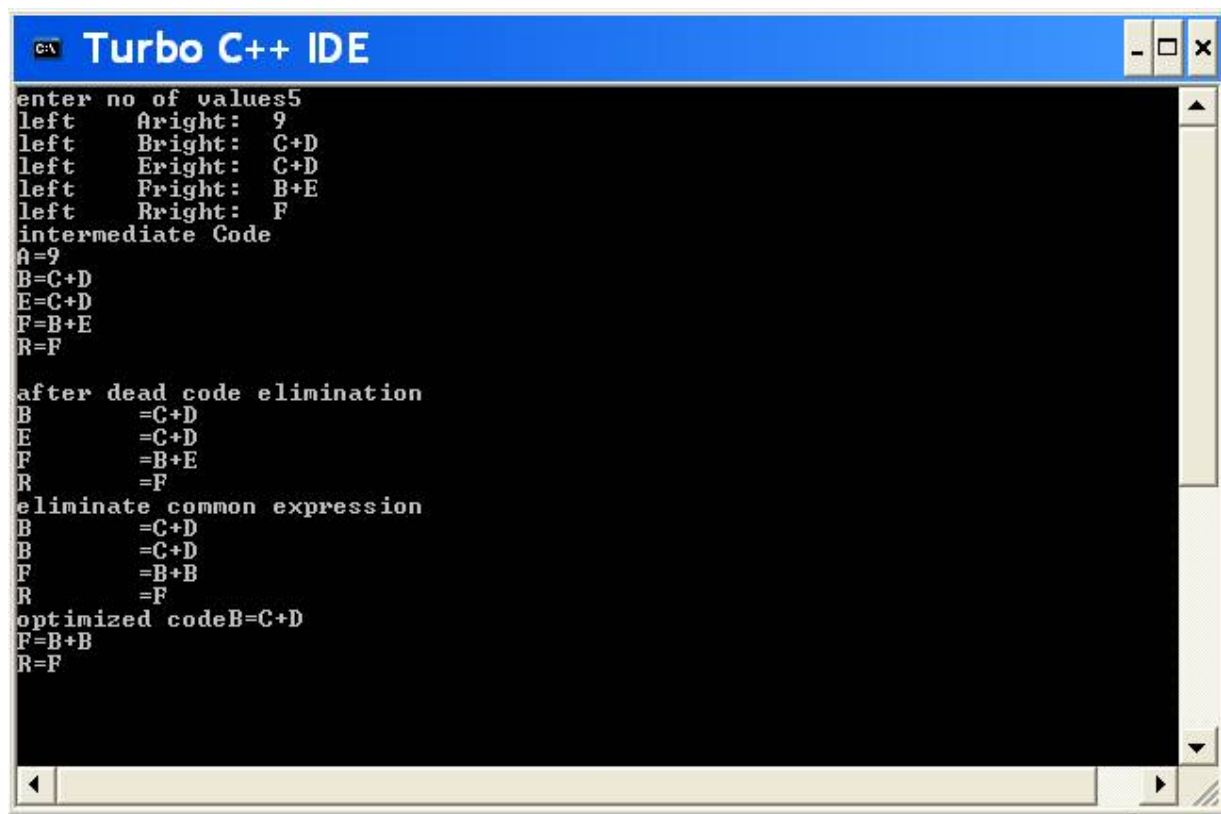
printf("%c\t=",pr[k].l);
printf("%s\n",pr[k].r);
}

//sub expression elimination
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",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);  
}  
// duplicate production elimination  
  
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';  
            strcpy(pr[i].r,'\0');  
        }  
    }  
    printf("optimized code");  
    for(i=0;i<z;i++)  
    {  
        if(pr[i].l!='\0')  
        {  
            printf("%c=",pr[i].l);  
            printf("%s\n",pr[i].r);  
        }  
    }  
    getch();  
}
```

OUTPUT:



```
C:\ Turbo C++ IDE
enter no of values5
left   Aright:  9
left   Bright:  C+D
left   Eright:  C+D
left   Fright:  B+E
left   Rright:  F
intermediate Code
A=9
B=C+D
E=C+D
F=B+E
R=F

after dead code elimination
B      =C+D
E      =C+D
F      =B+E
R      =F
eliminate common expression
B      =C+D
B      =C+D
F      =B+B
R      =F
optimized codeB=C+D
F=B+B
R=F
```

Beyond the syllabus**Ex.No: 12 IMPLEMENTATION OF SHIFT-REDUCED PARSING ALGORITHMS****AIM:**

To write a program for implementing Shift Reduce Parsing using C.

ALGORITHM:

1. Get the input expression and store it in the input buffer.
2. Read the data from the input buffer one at the time.
3. Using stack and push & pop operation shift and reduce symbols with respect to production rules available.
4. Continue the process till symbol shift and production rule reduce reaches the start symbol.
5. Display the Stack Implementation table with corresponding Stack actions with input symbols.

PROGRAM:

```
#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

#include<string.h>

char ip_sym[15],stack[15];

int ip_ptr=0,st_ptr=0,len,i;

char temp[2],temp2[2];

char act[15];

void check();

void main()

{

    clrscr();

    printf("\n\t\tSHIFT REDUCE PARSER\n");

    printf("\n\t\tGRAMMER\n");

    printf("\n\t\tE->E+E\n\t\tE->E/E");
```



```

printf("\n E->E*E\n E->a/b");

printf("\n enter the input symbol:\t");

gets(ip_sym);

printf("\n\t stack implementation table");

printf("\n stack \t\t input symbol\t\t action");

printf("\n_____ \t\t_____ \t\t_____ \n");

printf("\n $\t\t%s$\t\t\t--",ip_sym);

strcpy(act,"shift");

temp[0]=ip_sym[ip_ptr];

temp[1]='\0';

strcat(act,temp);

len=strlen(ip_sym);

for(i=0;i<=len-1;i++)

{

stack[st_ptr]=ip_sym[ip_ptr];

    stack[st_ptr+1]='\0';

    ip_sym[ip_ptr]=' ';

    ip_ptr++;

    printf("\n $%s\t\t%s$\t\t\t%s",stack,ip_sym,act);

    strcpy(act,"shift");

    temp[0]=ip_sym[ip_ptr];

    temp[1]='\0';

    strcat(act,temp);

    check();

    st_ptr++;

}

st_ptr++;

```

```

        check(); }

void check()
{int flag=0;

    temp2[0]=stack[st_ptr];

    temp2[1]='\0';

    if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))
    {

        stack[st_ptr]='E';

        if(!strcmpi(temp2,"a"))

            printf("\n $%s\t\t%s$\t\t\tE->a",stack,ip_sym);

        else

            printf("\n $%s\t\t%s$\t\t\tE->b",stack,ip_sym);

        flag=1;

    }

    if((!strcmpi(temp2,"+"))||(strcmpi(temp2,"*"))||(strcmpi(temp2,"/")))

        {flag=1;

        }

    if((!strcmpi(stack,"E+E"))||(strcmpi(stack,"E\E"))||(strcmpi(stack,"E*E")))

    {        strcpy(stack,"E");

        st_ptr=0;

        if(!strcmpi(stack,"E+E"))

            printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip_sym);

        else

            if(!strcmpi(stack,"E\E"))

                printf("\n $%s\t\t%s$\t\t\tE->E\E",stack,ip_sym);

            else

                if(!strcmpi(stack,"E*E"))

```

```

        printf("\n %s\t\t%s$\t\t\tE->E*E",stack,ip_sym);

        else

        printf("\n %s\t\t%s$\t\t\tE->E+E",stack,ip_sym);

        flag=1;

    } if(!strcmpi(stack,"E")&&ip_ptr==len)

    {

        printf("\n %s\t\t%s$\t\t\tACCEPT",stack,ip_sym);

        getch();

        exit(0);

    }

    if(flag==0)

    {

        printf("\n%s\t\t\t%s$\t\t\t reject",stack,ip_sym);

        exit(0); }

    return;

}

```

OUTPUT:

SHIFT REDUCE PARSER

GRAMMER

E->E+E

E->E/E

E->E*E

E->a/b

Enter the input symbol: a+b

Stack Implementation Table

Stack	Input Symbol	Action
-----	-----	-----
\$	a+b\$	--
\$a	+b\$	shift a
\$E	+b\$	E->a
\$E+	b\$	shift +
\$E+b	\$	shift b

\$E+E	\$	E->b
\$E	\$	E->E+E
\$E	\$	ACCEPT

RESULT: Thus the program for implementation of Shift Reduce parsing algorithm is executed and verified

Ex.No:13 CONSTRUCTION OF LR-PARSING TABLE**AIM:**

To write a program for construction of LR Parsing table using C.

ALGORITHM:

1. Get the input expression and store it in the input buffer.
2. Read the data from the input buffer one at the time and convert in to corresponding Non Terminal using production rules available.
3. Perform push & pop operation for LR parsing table construction.
4. Display the result with conversion of corresponding input symbols to production and production reduction to start symbol. No operation performed on the operator.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
char stack[30];
int top=-1;
void push(char c)
{
    top++;
    stack[top]=c;
}
char pop()
{
    char c;
    if(top!=-1)
    {
        c=stack[top];
        top--;
        return c;
    }
}
```

```

    }
    return 'x';
}

void printstat()
{
    int i;
    printf("\n\t\t $");
    for(i=0; i<=top; i++)
        printf("%c", stack[i]);
}

void main()
{
    int i, j, k, l;
    char s1[20], s2[20], ch1, ch2, ch3;
    clrscr();
    printf("\n\n\t LR PARSING");
    printf("\n\t ENTER THE EXPRESSION");
    scanf("%s", s1);
    l = strlen(s1);
    j = 0;
    printf("\n\t $");
    for(i=0; i<l; i++)
    {
        if(s1[i] == 'i' && s1[i+1] == 'd')
        {
            s1[i] = ' ';
            s1[i+1] = 'E';
            printstat(); printf("id");
            push('E');
            printstat();
        }
        else if(s1[i] == '+' || s1[i] == '-' || s1[i] == '*' || s1[i] == '/' || s1[i] == 'd')
        {
            push(s1[i]);

```

```
        printstat();
    }
}

printstat();
l=strlen(s2);
while(l)
{
    ch1=pop();
    if(ch1=='x')
    {
        printf("\n\t\t\t $");
        break;
    }
    if(ch1=='+'||ch1=='/'||ch1=='*'||ch1=='-')
    {
        ch3=pop();
        if(ch3!='E')
        {
            printf("error");
            exit();
        }
        else
        {
            push('E');
            printstat();
        }
    }
    ch2=ch1;
}
getch();
}
```

OUTPUT:

LR PARSING

ENTER THE EXPRESSION

id+id*id-id

\$

\$id

\$E

\$E+

\$E+id

\$E+E

\$E+E*

\$E+E*id

\$E+E*E

\$E+E*E-

\$E+E*E-id

\$E+E*E-E

\$E+E*E-E

\$E+E*E

\$E

\$

RESULT:

Thus the program for construction of LR Parsing table is executed and verified.