JIT

Touching new heights of excellence!

COMPILER DESIGN

LAB MANUAL

ATTAINMENT OF PROGRAM OUTCOMES & PROGRAM SPECIFIC OUTCOMES

Exp. No.	Experiment	Program Outcomes Attained	Program Specific Outcomes Attained
1	Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C language.	PO1, PO2, PO3	PSO1
2	***** Write a C program to identify whether a given line is a comment or not.	PO1	PSO1
3	*Write a C program to recognize strings under 'a', 'a*b+', 'abb'.	PO1, PO2	PSO1, PSO2
4	*Write a C program to test whether a given identifier is valid or not.	PO1	PSO1
5	*Write a C program to simulate lexical analyzer for validating operators.	PO1, PO2, PO3	PSO1, PSO2
6	Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.	PO1, PO2	PSO1
7	Write a C program for implementing the functionalities of predictive parser for the mini language specified in Note 1.	PO1, PO2, PO4, PO5	PSO1
8	a) *Write a C program for constructing of LL (1) parsing. b) *Write a C program for constructing recursive descent parsing.	PO1, PO2, PO3, PO4	PSO1, PSO2
9	Write a C program to implement LALR parsing.	PO1, PO2, PO4	PSO1
10	a) *Write a C program to implement operator precedence parsing. b) *Write a C program to implement Program semantic rules to calculate the expression that takes an expression with digits, + and * and computes the value.	PO1, PO2	PSO1, PSO2
11	Convert the BNF rules into Yacc form and write code to generate abstract syntax tree for the mini language specified in Note 1.	PO1, PO2	PSO1

12	Write a C program to generate machine code from abstract syntax tree generated by the parser. The instruction set specified in Note 2 may be considered as the target code.	PO1, PO2, PO3, PO4, PO5	PSO1

*****Content beyond the University prescribed syllabi

EXPERIMENT- 1

OBJECTIVE:

Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C language.

RESOURCE:

Turbo C ++

PROGRAM LOGIC:

1. Read the input Expression
2. Check whether input is alphabet or digits then store it as identifier
3. If the input is is operator store it as symbol
4. Check the input for keywords

PROCEDURE:

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

PROGRAM:

```
#include<stri
ng.h>
#include<cty
pe.h>
#include<std
io.h>
void keyword(char str[10])
{
 if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int"
,str)==0||str
cmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("static",st
r)==0||strcmp("switch",str
)==0||strcmp("case",str)==0)
         printf("\n%s is a keyword",str);
 else
         printf("\n%s is an identifier",str);
}
main
()
{        FILE
```

```
*            str[10],st1[10];
f            int
1            num[100],lineno=0,tokenvalue=0,i
,            =0,j=0,k=0; printf("\nEnter the c
*            program");/*gets(st1);*/
f            f1=fopen("input","w");
2            while((c=getchar())!=EOF)
,                putc(c,f1);
*            fclose(f1);
f            f1=fopen("input","r"
3            );
;            f2=fopen("identifier
             ","w");
c            f3=fopen("specialch
h            ar","w");
a            while((c=getc(f1))!=
r            EOF) {
                 if(isdigit(c))
c                {
,
                         tokenvalue=c-'0';
```

```
                          c=getc(f1);
                          while(isdigit(c))

                            {
                            tokenvalue*=10+c-
                            '0'; c=getc(f1);
                            }
          }
          els         num[i++]=tokenvalue;
          e           ungetc(c,f1);


                  if(isalpha(c))
                    {
                          putc(c,f
                          2);
                          c=getc(f
                          1);
                          while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
                             {
                                  putc(c,f
                                     2);
                                  c=getc(
                                    f1);
                             }
                          putc(' ',f2);
                          ungetc(c,f1);
                    }
                    els
                    e
                   if(c==' '||c=='\t')
                       printf(" ");
                   else
                          if(c=='\n')
                                  lineno++;
                          fc           (f2); fclose(f3); fclose(f1);
                          lo
}                         se
```

```
     else
                                   putc(c,
                                   f3);
printf("\nThe no's in the
program are");
for(j=0;j<i;j++)
printf("%d",num[j
]); printf("\n");
f2=fopen("identifi
er","r"); k=0;
printf("The keywords and
identifiersare:");
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("\nSpecial
        characters are");
        while((c=getc(f3))!=EO
        F) printf("%c",c);
        printf("\n");
        fclose(f3);
        printf("Total no. of lines are:%d",lineno);
}
```

PRE LAB QUESTIONS

1. What is token?
2. What is lexeme?
3. What is the difference between token and lexeme?
4. Define phase and pass?
5. What is the difference between phase and pass?
6. What is the difference between compiler and interpreter?

LAB ASSIGNMENT

1. Write a program to recognize identifiers.
2. Write a program to recognize constants.
3. Write a program to recognize keywords and identifiers.
4. Write a program to ignore the comments in the given input source program.

POST LAB QUESTIONS

1. What is lexical analyzer?
2. Which compiler is used for lexical analyzer?
3. What is the output of Lexical analyzer?
4. What is LEX source Program?

INPUT & OUTPUT:

Input:

```
Enter Program $ for termination:
 {
  int a[3],t1,t2;
  t1=2; a[0]=1; a[1]=2; a[t1]=3;
  t2=-(a[2]+t1*6)/(a[2]-t1);
  if t2>5
then
print(t2)
; else {
  int
  t3;
  t3=9
  9;
  t2=-
  25;
  print(-t1+t2*t3); /* this is a comment on 2 lines */
  } endif
  }
  $
```

Output:

```
Variables : a[3] t1
t2 t3 Operator : - +
* / > Constants :  2
1 3 6 5 99 -25
Keywords : int if then
else endif Special
Symbols : , ; ( ) { }
Comments : this is a comment on 2 lines
```

EXPERIMENT-2

OBJECTIVE:

***** Write a C program to identify whether a given line is a comment or not.

RESOURCE:

Turbo C++

PROGRAM LOGIC:

Read the input string.

Check whether the string is starting with ‘/’ and check next character is ‘/’ or’*’. If condition satisfies print comment.

Else not a comment.

PROCEDURE:

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

PROGRAM:

```
#include<stdi
o.h>
#include<coni
o.h> void
main()

{
  char
  com[30];
  int
  i=2,a=0;
  clrscr();
  printf("\n Enter
  comment:");
  gets(com);
  if(com[0]=='/') {
                 if(com[1]=='/')
                  printf("\n It is a
                 comment"); else
                 if(com[1]=='*') {
                                for(i=2;i<=30;i++)
```

```
                                        {
                                              if(com[i]=='*'&&com[i+1]=='/')
                                             {
                                                   printf("\n It is a
                                                    comment"); a=1;
                                                    break;}
                                             else
                                             continue
                                             ; }
                                        if(a==0)
                                   }    printf("\n It is not a comment");
                              else
                              printf("\n It is not a comment");
                         }
                 else
                      printf("\n It is not a
                 comment"); getch(); }
```

INPUT & OUTPUT:

Input: Enter comment: //hello
Output: It is a comment **Input:** Enter comment: hello
Output: It is not a comment

EXPERIMENT-3

OBJECTIVE
:

Write a C program to recognize strings under 'a', 'a*b+', 'abb'.

RESOURCE:

Turbo C++

PROGRAM LOGIC:

By using transition diagram we verify

input of the state. If the state recognize

the given pattern rule.

Then print string is accepted under

a*/ a*b+/ abb. Else print string not

accepted.

PROCEDURE:

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

PROGRAM:

```
#include<std
io.h>
#include<co
nio.h>
#include<stri
ng.h>
#include<std
lib.h> void
main()
{
        char s[20],c;
         int
        state=0,i=0
        ; clrscr();
        printf("\n Enter a
        string:"); gets(s);
        while(s[i]!='\0')
                 {
                         switch(st
                         ate)
                                   {  case 0:
                                      c=s[i++];
                                      if(c=='a')             state=1;
                else if(c=='b')
                                state=2;
else
             state=6;
                                             brea
                                   k; case 1:
                                   c=s[i++];
                                           if(c=='a')
                                                   state=3;
```

```
       else if(c=='b')
               state=4;
else
                                       state=6;
                                  break
                             ; case 2:
                             c=s[i++];
                                  if(c=='a')
                                       state
                                  =6; else
                                  if(c=='b')
                                       state=2;
                                  else
                                       state=6;
                                  brea
                                  k;
case 3: c=s[i++];
if(c=='a')
                                       state
                                  =3; else
                                  if(c=='b')
                                        state=2;
else
                                       state=6;
                                  break
                             ; case 4:
                             c=s[i++];
                                  if(c=='a')
                                       state=6;

                                  else if(c=='b')
                                       state=5;
                                  else                                      b
```

```
                                        reak; case
                                        5: c=s[i++];  state=6;
                                             if(c==
                                             'a')


                                                    state=6;
else if(c=='b')
          state=2;
                                             else
                                                    state
                                                    =6;
                                                    break
                                                    ;
                                        case 6: printf("\n %s is not
                                             recognised.",s); exit(0);
                                   }
                    }
```

```
        I       f(state==1)
                        printf("\n %s is accepted under
                rule 'a'",s); else
                if((state==2)||(state==4))
                        printf("\n %s is accepted under
                rule 'a*b+'",s); else if(state==5)
                       printf("\n %s is accepted under rule 'abb'",s);
                getch();
    }
```

INPUT &
OUTPUT:
Input :

```
Enter a String: aaaabbbbb
```

Output:

```
aaaabbbbb is accepted under rule 'a*b+'

Enter a string:

cdgs cdgs is not

recognized
```

EXPERIMENT-4

OBJECTIVE
:

*Write a C program to test whether a given identifier is valid or not

RESOURCE:

Turbo C++

PROGRAM LOGIC:

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<std
io.h>
#include<con
io.h>
#include<cty
pe.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;
                              brea
                              k;
                        }
                        i+
                        +;
                  }
                  if(flag==1)
                  printf("\n Valid
                  identifier");
                  getch();
          }
```

INPUT & OUTPUT:

Input: Enter an identifier: first

Output:

```
Valid identifier
Enter an
identifier:1aqw
Not a valid
identifier
```

EXPERIMENT-5

OBJECTIVE
:

*Write a C program to simulate lexical analyzer for validating operators.

RESOURCE:

Turbo C++

PROGRAM LOGIC :

Read the given input.

If the given input matches with any

operator symbol. Then display in terms

of words of the particular symbol. Else

print not a operator.

PROCEDURE:

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

PROGRAM:

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

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

```
                else
                printf("\nBitwise
                 OR"); break;
         case'+': printf("\n
                Addition");
                break;
          case'-':
                 printf("\nSubstra
                 ction"); break;
         case'*':
                printf("\nMultiplicat
                ion"); break;
         case'/': printf("\nDivision");
                 break;
         case'%': printf("Modulus");
                break;
         default: printf("\n Not a operator");
         }
         getch();
  }
```

INPUT &

OUTPUT:

Input

Enter any

operator: *

Output

Multiplication

EXPERIMENT-6

OBJECTIVE:

Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.

RESOURCE:

Linux using Putty

PROGRAM LOGIC:

Read the input string.
Check whether the string is identifier/ keyword /symbol by using the rules of identifier and keywords using LEX Tool

PROCEDURE:

Go to terminal .Open vi editor ,Lex lex.l , cc lex.yy.c , ./a.out

PROGRAM:

```
/* program name is lexp.l */
%{
/* program to recognize a c
program */ 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 |do |if |break |continue |void |switch |case |long
|struct |const |typedef |return
|else |goto {printf("\n\t%s is a
KEYWORD",yytext);} "/*"
{COMMENT = 1;}
/*{printf("\n\n\t%s is a
COMMENT\n",yytext);}*/ "*/"
{COMMENT = 0;}
/* printf("\n\n\t%s is a COMMENT\n",yytext);}*/
{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
} {if(!COMMENT) printf("\n BLOCK ENDS");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s
IDENTIFIER",yytext);} ".*\" {if(!COMMENT)
printf("\n\t%s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}
 {if(!COMMENT)
```

```
printf("\n\t");ECHO;printf("\n");} (

 ECHO;
{if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}

<= |>= |< |== |> {if(!COMMENT) printf("\n\t%s is a RELATIONAL
OPERATOR",yytext);}
%%
int main(int argc,char **argv)
{
if (argc > 1)
{
FILE *file;
file =
fopen(argv[1],"r"
); if(!file)
{
printf("could not open %s
\n",argv[1]); exit(0);
}
yyin = file;
}
yylex();
printf("\n\
n");
return 0;
} int yywrap()
{
return 0;
}
```

PRE LAB QUESTIONS:

1. List the different sections available in LEX compiler?
2. What is an auxiliary definition?
3. How can we define the translation rules?
4. What is regular expression?
5. What is finite automaton?

LAB ASSIGNMENT:

1. Write a program that defines auxiliary definitions and translation rules of Pascal tokens?
2. Write a program that defines auxiliary definitions and translation rules of C tokens?
3. Write a program that defines auxiliary definitions and translation rules of JAVA tokens

POST LAB QUESTIONS:

1. What is Jlex?
2. What is Flex?
3. What is lexical analyzer generator?
4. What is the input for LEX Compiler?
5. What is the output of LEX compiler?

6.6 INPUT & OUTPUT:

Input

```
$vi var.c
#include<stdio.h
> main()
{
```

```
int a,b;
}
```

Output

```
$lex lex.l
$cc lex.yy.c
$./a.out var.c
#include<stdio.h> is a
PREPROCESSOR DIRECTIVE
FUNCTION
main (
)
BLOCK BEGINS
int is a
KEYWORD
a
IDENTIFIER
b
IDENTIFI
ER
BLOCK
ENDS
```

EXPERIMENT-7

OBJECTIVE:

Write a C program for implementing the functionalities of predictive parser for the mini language specified in Note 1.

RESOURCE:

Turbo C++

PROGRAM LOGIC:

Read the input string.

By using the FIRST AND FOLLOW values.

Verify the FIRST of non terminal and insert the production in the FIRST value

If we have any @ terms in FIRST then insert the

productions in FOLLOW values Constructing the

predictive parser table

PROCEDURE:

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

PROGRAM:

```
#include<std
io.h>
#include<co
nio.h>
#include<stri
ng.h>
char prol[7][10]={"S","A","A","B","B","C","C"};
char pror[7][10]={"A","Bb","Cd","aB","@","Cc","@"};
char prod[7][10]={"S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@"};
char first[7][10]={"abcd","ab","cd","a@","@","c@","@"};
char
follow[7][10]={"$","$","$","a$","b$",
"c$","d$"}; char table[5][6][10];
numr(char c)
 {
```

```
switch(c)
{
        case 'S':
        return 0;
        case 'A':
        return 1;
        case 'B':
        return 2;
        case 'C':
        return 3;
        case 'a':
        return 0;
        case 'b':
        return 1;
       case 'c':
        return 2;
        case 'd':
        return 3;
        case '$':
        return 4;
}
```

```
        return(2);

 }

void main()

 {

         int i,j,k;

        clrscr();

        for(i=0;i<5

        ;i++)

        for(j=0;j<6

        ;j++)
                strcpy(table[i][j]," ");

         printf("\nThe following is the predictive parsing table for the

        following grammar:\n"); for(i=0;i<7;i++)

                printf("%s\n",prod[i])

        ; printf("\nPredictive

        parsing table is\n");

        fflush(stdin);
         for(i=0;i<7;i++)

         {

                 k=strlen(firs

                 t[i]);

                 for(j=0;j<10;
             j++)

             if(first[i][j]!

             ='@')

            strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);

   }

  for(i=0;i<7;i++)

   {

            if(strlen(pror[i])==1)

            {

             if(pror[i][0]=='@')

            {

                      k=strlen(follo

                      w[i]);

                      for(j=0;j<k;j+

                      +)

                       strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i
                       ]);
```

```
                    }

            }

    }
    strcpy(table[0][0]," ");
    strcpy(table[0][1],"a");
    strcpy(table[0][2],"b");
    strcpy(table[0][3],"c");
    strcpy(table[0][4],"d");
    strcpy(table[0][5],"$");
    strcpy(table[1][0],"S");
    strcpy(table[2][0],"A");
    strcpy(table[3][0],"B");
    strcpy(table[4][0],"C");
    printf("\n-
    ------------------------------------------------------------------------\n"
    ); for(i=0;i<5;i++)
     for(j=0;j<6;j++)
            {
           printf("%-
            10s",table[i][j]);
            if(j==5)
            printf("\n-------------------------------------------------------\n");
```

```
          }
  getch();
 }
```

PRE LAB QUESTIONS:

1. What is top-down parsing?
2. What are the disadvantages of brute force method?
3. What is context free grammar?
4. What is parse tree?
5. What is ambiguous grammar?
6. What are the derivation methods to generate a string for the given grammar?
7. What is the output of parse tree?

LAB ASSIGNMENT:

1. Write a program to compute FIRST for the

   following grammar? E→ TE'

   E'→+TE'/î

   T→FT’

   T'→*F

   T'/î

   F→(E)

   /i
2. Write a program to compute FIRST for the

   following grammar? S→iCtSS’

   S’→eS/ î
3. Write a program to construct predictive parsing table for

   the following grammar? S→iCtSS’

   S’→eS/ î

POST LAB QUESTIONS

1. What is Predictive parser?
2. How many types of analysis can we do using Parser?
3. What is Recursive Decent Parser?
4. How many types of Parsers are there?
5. What is LR Parser?

INPUT & OUTPUT:

The following is the predictive parsing table for the following grammar:

S-

>A

A-

>Bb

A-

>Cd

B-

>aB

B-

>@

C-

```
>Cc
C-
>@
Predictive parsing table is
------------------------------------------------------------------------

     a      b       c       d      $

------------------------------------------------------------------------

S   S->A  S->A  S->A   S->A

------------------------------------------------------------------------

A   A->Bb          A->BbA->Cd       A->Cd

------------------------------------------------------------------------

B   B->aB          B->@           B->@          B->@

------------------------------------------------------------------------

C   C->@  C->@ C->@

------------------------------------------------------------------------
```

EXPERIMENT-8(a)

OBJECTIVE

:

*Write a C program for constructing of LL (1) parsing.

RESOURCE:

Turbo C++

PROGRAM LOGIC:

Read the input string.

Using predictive parsing table parse the given input using stack .

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

PROCEDURE:

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

PROGRAM

```
#include<stdio

.h>

#include<coni

o.h>

#include<strin

g.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,str

            2; 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;
                                              brea
                                        k; case 'b':
                                        str1=1;
                                              brea
                                       k; case 't':
                                       str1=2;
                                              brea
                                       k; case 'c':
        }                              str1=3;
                            switch(s          brea
                            [j])
                                       k; case 'f':
        {
                                       str1=4;
                                              break;



                                        case 'i': str2=0;
                                              brea
                                       k; case '+':
                                       str2=1;
                                              brea
                                        k; case '*':
                                        str2=2;
        }                                     brea
```

```
k                          : str2=3;
;                                 brea
                            k; case ')':
c                           str2=4;
a                                 brea
s                           k; case '$':
e                           str2=5;
                                  break;
'
(
'
                    if(m[str1][str2][0]=='\0')
                     {
                            printf("\nERR
                           OR"); 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(); }
```

INPUT & OUTPUT:

```
Enter the input string:i*i+i

Stack              INP
                   UT
$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	$

EXPERIMENT-8(b)

OBJECTIVE:

Construction of recursive descent parsing for the

following grammar E->TE'

E'->+TE/@"@ represents null

character" T->FT'

T`-

>*FT'/

@ F-

>(E)/ID

RESOURCE:

Turbo C++

PROGRAM LOGIC:

Read the input string.

Write procedures for the non terminals

Verify the next token equals to non terminals if it satisfies

match the non terminal. If the input string does not match

print error.

PROCEDURE:

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

PROGRAM:

```
#include<std
io.h>
#include<co
nio.h>
#include<stri
ng.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");
                els
                e       printf("\nString is not accepted");
        }
        els
        e
                printf("\nString not accepted");
```

```
             getch
   }         ();
  E(
  )
   {
             if(T(
             ))
                       if(EP())
             {
                       return(1);


                        return(0);
              else


              }
             else
              return(0);
             }
             EP
             ()
             {
                       if(input[i]=='+')
                       {
                                 i++;
                                if(T(
                                ))
                                 {
                                 if(EP
                                 ())
                                return(1);
                       }
                        els     else
                        e
                                return(0);
                                ()
                  }              {
                  T
              return(0
     }        );

              e
              l
              s
              e
     if(F(
     ))       r
              e
              t
              u
              r
              n
              (
              1
              )
              ;




              {
                              if(TP())
                               return(1)
                              ; else
                              return(0)
                         }    ;
                 else
```

```
                    return(0);
        }
        TP
        ()
        {       if(input[i]=='*')
                {
                        i++;
                        if(F())
                        {
                        if(TP())
                        return(
                        1); else
                        return(
                 }      0);
                els
                e
                return(0);
        }
        els
        e
        return(1);
        }
        F
        ()
        {       if(input[i]=='(')
                {
                        i++;
                        if(E())
                        {
                        if(input[i]==')')
                        {
                                                else


                        }
```

```
i++;
return(1);
```

```
                        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
                        );
        }
```

INPUT & 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
```

EXPERIMENT-9

OBJECTIVE:

Write a program to Design LALR Bottom up Parser.

RESOURCE:

TURBO C++

PROGRAM LOGIC:

Read the input string.

Push the input symbol with its state symbols in to the stack

by referring lookaheads We perform shift and reduce actions

to parse the grammar.

Parsing is completed when we reach $ symbol.

PROCEDURE:

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

PROGRAM:

```
/*LALR
 PARSER E-
 >E+T
        E->T
        T-
        >T*F
        T->F
         F-
        >(E)
        F->i
*/

#include<std
io.h>
#include<co
nio.h>
#include<std
lib.h>
```

```
#include<stri
ng.h>
void push(char *,int
*,char); char
stacktop(char *);
void
isproduct(char,cha
r); int ister(char);
int
isnter(char)
; int
isstate(char
); void
error();
void
isreduce(char,cha
r); 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,in
p,i);
do
{
x=inp[i];
p=stacktop(stack);

isproduct(x,p);
if(strcmp(temp,"emp")==0)
        error();
if(strcmp(temp,"acc")==0)
        break;
els
e
{       if(temp[0]=='s')
        {
                push(stack,&top,in
                p[i]);
                push(stack,&top,tem
                p[1]); i++;
        }
        els
        e
        {   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


             getch();
            }


printf(" \n do not accept the input ");


            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[to
             p];
             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 ");
    els
    e
    {   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;
    }
}
```

EXPERIMENT-10(a)

OBJECTIVE:

*Write a C program to implement operator precedence parsing.

RESOURCE:

Turbo C++

PROGRAM LOGIC:

Read the arithmetic input string.

Verify the precedence between terminals and symbols

Find the handle enclosed in < . > and reduce it to

production symbol. Repeat the process till we reach

the start node.

PROCEDURE:

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

PROGRAM:

```
#include<stdio.
h> char
str[50],opstr[75
];
int f[2][9]={2,3,4,4,4,0,6,6,0,1,1,3,3,5,5,0,5,0};
int
col,col1,col
2; char c;
swt()
{
    switch(c)
```

```
{
        case'+':col=0;
        break; case'-
        ':col=1;break;
        case'*':col=2;
        break;
        case'/':col=3;b
        reak;
        case'^':col=4;
        break;
        case'(':col=5;b
        reak;
        case')':col=6;b
        reak;
        case'd':col=7;
        break;
        case'$':col=8;
        break;
         default:printf("\nTERMINAL
                 MISSMATCH\n"); exit(1);
```

```
                        break;
    }
   // return 0;
}
main()
{
    int
    i=0,j=0,col1,cn,k
    =0; int
    t1=0,foundg=0;
   char
    temp[20];
    clrscr();
   printf("\nEnter arithmetic
    expression:");
    scanf("%s",&str);
    while(str[i]!='\0')
            i++;
   str[i]='$';
    str[++i]='\0';
    printf("%s\n",str
   ); come:
    i=0;
   opstr[0]=
    '$'; j=1;
   c='$';
    swt();
   col1=c
   ol;
```

```
          c=str[i
          ];
          swt();
          col2=c
          ol;
           if(f[1][col1]>f[2][col2])
          {
                    opstr[j]='>'
                    ; j++;
           }
        else if(f[1][col1]<f[2][col2])
        {
                   opstr[j]='<'
                   ; j++;
         }
```

```
els
e
{         opstr[j]='=';j++;

}


while(str[i]!='$')
{
          c=str[i];
          swt();
         col1=co
         l;
         c=str[+
         +i];
         swt();
         col2=co
         l;
          opstr[j]=str[-
          -i]; j++;
          if(f[0][col1]>f[1][col2])
         {
                  opstr[j]='
                  >'; j++;
           }
          else if(f[0][col1]<f[1][col2])
         {
                 opstr[j]='
                  <'; j++;
```

```
               }
              else
              {
              opstr[j]='=';j++;
               }
}              i+
opstr[j]       +;
='$';
opstr[++j]='\0';
printf("\nPrecedence
Input:%s\n",opstr); i=0;
j=0;
while(opstr[i]!='\0')
```

```
{
        foundg=0;
         while(foundg!=1)
         {
                if(opstr[i]=='\0')got
                o redone;
                if(opstr[i]=='>')foun
                dg=1; t1=i;
                 i++;
        }
         if(foundg
        ==1)
        for(i=t1;i>
        0;i--)
         if(opstr[i]=='<')break;
        if(i==0){printf("\nERROR\n
        ");exit(1);} cn=i;
         j=0;
         i=t1+
         1;
        while(opstr[i]!='\0')
        {
                 temp[j]=opstr[i
                ]; j++;i++;
         }
        temp[j]='\0';
        opstr[cn]='E';
         opstr[++cn]='\0
```

```
                '; 
                strcat(opstr,tem
                p);
                printf("\n%s",op
                str); i=1;
            }
            redone:k=0;
            while(opstr[k]!=
            '\0')
              {
                k++;
                if(opstr[k]=='<')
                {
                        Printf("\nError")
                        ; exit(1);
                }
```

```
}
if((opstr[0]=='$')&&(opstr[2]=='
$'))goto sue; i=1
while(opstr[i]!='\0')
{
        c=opstr[i];
         if(c=='+'||c=='*'||c=='/'||c
         =='$')
        {
                temp[j]=c;j
                ++;} i++;
        }
        temp[j]='\0';
        strcpy(str,tem
        p); goto
        come;
        sue:
        printf("\n
        success");
        return 0;
}
```

INPUT & OUTPUT:

Enter the arithmetic

expression (d*d)+d$

Output:

```
(d*d)+d$
Precedence input:$<(<d>*<d>)>+<d>$
```

```
$<(E*<d>)>+<d>$
$<(E*E)>+<E>$
$E+<E>$
$E+E$
Precedence input:$<+>$
$E$
success
```

EXPERIMENT-10(b)

OBJECTIVE:

Program to implement semantic rules to calculate the expression that takes an expression with digits, + and * and computes the value.

RESOURCE:

Linux using putty

PROCEDURE:

Reading an input file
Calculate the sum or multiplication of
given expression. Using expression rule
print the result of the given values.

PROGRAM:

```
<parser.l>


%{
#include<stdi
o.h>
#include
"y.tab.h"
%}
%%
[0-9]+
{yylval.dval=atof(yytext
); return DIGIT;
}
\n|. return yytext[0];
%%
<parser.y>
%{
/*This YACC specification file generates the LALR
```

```
parser for the program considered in experiment 4.*/
#include<stdio.h>
%}
%union
{
double dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
line: expr '\n' {
```

```
printf("%g\n",$1);
}
;
expr: expr '+' term {$$=$1 + $3 ;}
| term
;
term: term '*' factor {$$=$1 * $3 ;}
| factor
;
factor: '(' expr ')' {$$=$2 ;}
| DIGIT
;
%%
int main()
{
yyparse();
}
yyerror(char *s)
{
printf("%s",s);
}
```

10.6 INPUT & OUTPUT:

```
$lex parser.l

$yacc –d parser.y

$cc lex.yy.c y.tab.c –ll –lm

$./a.

out

2+3

5.0000
```

EXPERIMENT-11

OBJECTIVE:

Convert The BNF rules into Yacc form and write code to generate abstract syntax tree.

RESOURCE :

linux using putty

PROGRAM LOGIC:

Reading an input file line by line.

Convert it in to abstract syntax tree using three

address code. Represent three address code in

the form of quadruple tabular form.

PROCEDURE:

Go to terminal .Open vi editor ,Lex lex.l , cc lex.yy.c , ./a.out

PROGRAM

<int.l>

```
%{
#include"y.ta
b.h"
#include<stdi
o.h>
#include<strin
g.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,$$);}
| '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' 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].resul
t,"-1"); push(Index);
```

```
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].re
sult,"-1"); push(Index);
Index++;
};
ELSEST: ELSE{
tInd=po
p();
Ind=po
p();
push(tI
nd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOC
K{
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].resul
t,"-1"); push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].re
sult,"-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==10
0)
{
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",t
Index++);
```

```
strcpy(result,QUAD[Index++].res
ult);
}
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;
}
}
```

PRE-LAB QUESTIONS

1 What are the functions we use to construct a syntax tree?
2 What is Meta data?
3 How list of identifiers are represented using BNF rules?
4 What is three address code?
5 What are the record structures we use to represent three address code?

LAB ASSIGNMENT

1 Write YACC for the desktop calculator?
2 Write BNF rules for the

following grammar? E→E+T/T

T→

T*F/F

F→(E)/

id

POST-LAB QUESTIONS:

1. What is Abstract Syntax tree?
2. What are BNF Rules?
3. What is DAG representation?
4. How LALR (1) states are generates?
5. In which condition the user has to supply more information to YACC?

INPUT & OUTPUT:

```
$lex int.l
$yacc –d int.y
$gcc lex.yy.c y.tab.c –ll –lm$./a.out test.c
```

OUTPUT

Pos	Operator	Arg1	Arg2	Result
0	<	a	b	t0
1	==	t0	**FALSE**	**5**
2	+	**a**	**b**	**t1**

3	==	**t1**		**5**
4	**GOTO**			
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**

EXPERIMENT-12

OBJECTIVE:

Write a C program to generate machine code from abstract syntax tree generated by the parser. The instruction set specified in Note 2 may be considered as the target code.

RESOURSE:

TURBO C++

PROGRAM LOGIC:

Read input string

Consider each input string and convert in to machine code instructions

PROCEDURE:

Go to terminal .Open vi editor ,Lex lex.l , cc lex.yy.c , ./a.out

PROGRAM:

```
#include<std

io.h>

#include<std

lib.h>

#include<stri

ng.h> int

label[20];

int

no=0;

int

main()

{

        FILE *fp1,*fp2;
```

```
char fname[10],op[10],ch;

char

operand1[8],operand2[8],re

sult[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==lab
        el[i])
        return 1;
 }
        return 0;
 }
```

PRE-LAB QUESTIONS

1 What are the different forms of object code?
2 What is mean by relocatable object code?
3 What is the cost of register to register operation?
4 What is address descriptor?
5 What is register descriptor?

LAB ASSIGNMENT

1 Write a program to generate the code for the following three address code statements? A=B+C
W=X-Y
2 Write a program to generate the code for the following three address code statements? W=(A+B)*C

POST-LAB QUESTIONS

1. What is target code?
2. What is machine code?
3. What is Cross compiler?
4. Give the example for cross compiler?
5. What is the difference between syntax & Semantics?

INPUT & OUTPUT:

```
$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
```