# COMPILER DESIGN LABORATORY

( VI semester of B.Tech)

As per the curricullam and syllabus

of

**Bharath Institute of Higher Education & Research**

# COMPILER DESIGN LAB MANUAL

NEW EDITION

Bharath
INSTITUTE OF HIGHER EDUCATION AND RESEARCH
(Declared as Deemed - to - be - University under section 3 of UGC Act 1956)
ACCREDITED WITH 'A' GRADE BY NAAC

**PREPARED BY**

**Mrs.S.Amudha**
**Mrs.J.Ranganayaki**

Bharath

INSTITUTE OF HIGHER EDUCATION AND RESEARCH

(Declared as Deemed-to-be University under section 3 of UGC Act, 1956)

(Vide Notification No. F.9-5/2000 - U.3, Ministry of Human Resource Development, Govt. of India, dated 4th July 2002)

## SCHOOL OF COMPUTING

## DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

# LAB MANUAL

# SUBJECT NAME:COMPILER DESIGN LAB

# SUBJECT CODE:U18PCCS6L1

**R 2018**

**(2020-2021)**

| U18PCCS6L1 | COMPILERDESIGNLABORATORY | L | T | P | C |
|---|---|---|---|---|---|
| | TotalContactHours-45 | 0 | 0 | 3 | 2 |
| | Prerequisite–CompilerDesign,CProgramming | | | | |
| | Lab Manual Designed by–Dept.of Computer Science and Engineering. | | | | |

**OBJECTIVES**

This laboratory course is intended to make the students experiment on the basic techniques of compiler construction and tools that can used to perform syntax-directed translation of a high-level programming language into an executable code. Students will design and implement language processors in C by using tools to automate parts of the implementation process. This will provide deeper insights into the more advanced semantics aspects of programming languages, code generation, machine independent optimizations, dynamic memory allocation, and object orientation.

| COURSEOUTCOMES(COs) | |
|---|---|
| CO1 | Design a lexical Analyzer using C |
| CO2 | Implement lexical analyzer using various generating tools |
| CO3 | Perform Context free grammar conversion and parse tree construction. |
| CO4 | Apply tools and technologies for designing a compiler. |
| CO5 | Develop a program for solving parser problems. |
| CO6 | Generate machine code from abstract syntax tree generated by the parser. |

**MAPPING BETWEEN COURSE OUTCOMES & PROGRAM OUTCOMES(3/2/1INDICATESSTRENGTHOFCORRELATION) 3-High, 2-Medium,1-Low**

| COs | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PO12 | PSO1 | PSO2 | PSO3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **CO1** | 3 | 3 | 3 | | 2 | | | | | | | | 2 | 3 | |
| **CO2** | 3 | 3 | 3 | | 2 | | | | | | | | | 3 | |
| **CO3** | 3 | 3 | 3 | | 2 | | | | | | | | | 3 | |
| **CO4** | 3 | 3 | 3 | | 2 | | | | | | | | 2 | 3 | |
| **CO5** | 3 | 3 | 3 | | 2 | | | | | | | | | 3 | |
| **CO6** | 3 | 3 | 3 | | 2 | | | | | | | | | 3 | |
| Category | Professionalcore(PC) | | | | | | | | | | | | | | |
| Approval | 47th AcademicCouncilMeetingheldinAug, 2018 | | | | | | | | | | | | | | |

**COMPILER DESIGN LAB]-[U18PCCS6L1]**

## LIST OF EXPERIMENTS

| | NAME OF THE EXPERIMENT |
|---|---|
| 1 | Design a lexical analyzer for given language and the lexical analyzer |
| 2 | Write a c program to recognize strings under 'a', 'a*b+', 'abb'. |
| 3 | Write a c program to test whether a given identifier is valid or not. |
| 4 | Write a c program to simulate lexical analyzer for validating operators. |
| 5 | Implement the lexical analyzer using jlex, flex or other lexical analyzer generating tools |
| 6 | Write a c program for constructing of ll (1) parsing. |
| 7 | Write a c program to implement lalr parsing |
| 8 | Write a c program to implement operator precedence parsing. |
| 9 | Convert the bnf rules into yacc form and write code to generate abstract syntax tree for the mini language specified in note 1. |
| 10 | Simulation of basic memory mulation of basic memory management schemes |
| 11 | Simulation of virtual memory management schemes |
| 12 | Simulation of file system |

CONTENT

1. **Design a lexical analyser for given language and the lexical analyser 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**.

**AIM**: To write a program to design a lexical analyser for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.

**ALGORITHM:**

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

**PROGRAM:**

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void keyword(char str[10])
{
if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||

strcmp("static",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)
printf("\n%s is a keyword",str);
else
printf("\n%s is an identifier",str);
}

void main()
{
FILE *f1,*f2,*f3;
char c,str[10],st1[10];
int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
printf("\nEnter the c program");/*gets(st1);*/
f1=fopen("input","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF){
if(isdigit(c))
{
```

```
tokenvalue=c-'0';
c=getc(f1);
while(isdigit(c)){
tokenvalue*=10+c-'0';
c=getc(f1);
}
num[i++]=tokenvalue;
ungetc(c,f1);
}
else if(isalpha(c))
{
putc(c,f2);
c=getc(f1);
while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
{
putc(c,f2);
c=getc(f1);
}
putc(' ',f2);
ungetc(c,f1);
}
else if(c==' '||c=='\t')
printf(" ");
else
if(c=='\n')
lineno++;
else
putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\nThe no's in the program are");
for(j=0;j<i;j++)
printf("%d",num[j]);
printf("\n");
f2=fopen("identifier","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))!=EOF)
printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d",lineno);
}
```

**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=99;
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
```

**RESULT:**

It is inferred that from the output it is understood what are the variables, operators, constants, keywords, special symbols and comments and the way a how a lexical analyser works.

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

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

**ALGORITHM :**

1. Read the input string.
2. Check whether the string is starting with „/‟ and check next character is „/‟ or‟‟*‟‟.
3. If condition satisfies print comment.
4. Else not a comment.

**PROGRAM:**

```
#include<stdio.h>
void main()
{
char com[30];
int i=2,a=0;
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();
 }
```

**OUTPUT:**

```
Enter comment: //hello
It is a comment
Enter comment: hello
It is not a comment
```




**RESULT:**

Thus it is understood that above C program identifies whether the given input is comment or not.

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

**AIM:** To Write a C program to recognize strings under 'a', 'a*b+', 'abb'

**ALGORITHM:**

1. By using transition diagram we verify input of the state.
2. If the state recognize the given pattern rule.
3. Then print string is accepted under a*/ a*b+/ abb.
4. Else print string not accepted.

**PROGRAM:**

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
char s[20],c;
int state=0,i=0;
printf("\n Enter a string:");
gets(s);
while(s[i]!='\0')
{
switch(state)
{
case 0: c=s[i++];
if(c=='a')
state=1;
else if(c=='b')
state=2;
else
state=6;
break;
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;
break;
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
state=6;
break;
case 5: c=s[i++];
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);
}
}
if(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();
}
```

**OUTPUT:**

```
Enter a String: aaaabbbbb
aaaabbbbb is accepted under rule 'a*b+'

Enter a string: cdgs
cdgs is not recognized
```

**RESULT:**

It is understood that the above program checks whether the given strings is accepted under the rule ‘a*b+’.

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

**AIM:** To Write a C program to test whether a given identifier is valid or not.

**ALGORITHM:**

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

**PROGRAM:**

```
#include <stdio.h>
#include <conio.h>
void main()
{
int i=0,flag=0;
    char keyw[10][10]={"int","float","break","long","char","for","if","switch","else","while"},a[10];
printf("Enter Identifier : ");
gets(a);
for(i=0;i<10;i++)
    {
if((strcmp(keyw[i],a)==0))
        {
flag=1;
        }
    }
if(flag==1)
    {
printf("\n%s is Keyword.",a);
    }
else
    {
flag=0;

if((a[0]=='_')||(isalpha(a[0])!=0))
        {
for(i=1;a[i]!='\0';i++)
            {
if((isalnum(a[i])==0)&&(a[i]!='_'))
                {
flag=1;
                }
            }
        }
else
        {
flag=1;
        }
    }
```

```
if(flag==0)
    {
printf("\n%s is an Identifier.",a);
    }
else
    {
printf("\n%s is Not an Identifier.",a);
    }
}
```

**OUTPUT:**

```
Enter an identifier: first
Valid identifier

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

**RESULT:**

Thus the program checks whether the given input is identifier or not.

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

**AIM:** To write a C program to simulate lexical analyser for validating operators

**ALGORITHM:**

1. Read the given input.
2. If the given input matches with any operator symbol.
3. Then display in terms of words of the particular symbol.
4. Else print not a operator.

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s[5];
printf("\n Enter any operator:");
gets(s);
switch(s[0])
{
case'>': if(s[1]=='=')
printf("\n Greater than or equal");
else
printf("\n Greater than");
break;
case'<': if(s[1]=='=')
printf("\n Less than or equal");
else
printf("\nLess than");
break;
case'=': if(s[1]=='=')
printf("\nEqual to");
else
printf("\nAssignment");
break;
case'!': if(s[1]=='=')
printf("\nNot Equal");
else
printf("\n Bit Not");
break;
case'&': if(s[1]=='&')
printf("\nLogical AND");
else
printf("\n Bitwise AND");
break;
case'|': if(s[1]=='|')
printf("\nLogical OR");
else
printf("\nBitwise OR");
```

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

    }
```

**OUTPUT:**

```
Enter any operator: *
Multiplication
```

**RESULT:**

It is inferred that the above program works as lexical analyser by identifying the operators**.**

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

**AIM:**

To write a C program to implement the lexical analyzer using JLex, flex or other lexical analyser generating tools

**ALGORITHM:**

1. Store the input in a separate file with the ".c" extension.
2. Start the program
3. Create the module to recognize and display the preprocessor directives.
4. Based on the general structure of the C program , formulate the code for detecting and displaying the Declaration part.
5. Identify the keywords, comment and basic identifiers.
6. To display the expressionsgiven inside the main of the input, display the type of operators given by the user.
7. Stop the program.

**PROGRAM:**

**lex.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;
}
```

**INPUT:**

$vi var.c

```
#include<stdio.h>
main()
{
int a,b;
}
```

**STEPS TO RUN THE PROGRAM:**

```
$lex lex.l
$cc lex.yy.c
$./a.out var.c
```

**OUTPUT**

```
#include is a PREPROCESSOR DIRECTIVE
 FUNCTION
main (
 )
BLOCK BEGINS
int is a KEYWORD
a IDENTIFIER
b IDENTIFIER
BLOCK ENDS
```

**RESULT:**

Thus the Lexical Analyser is implemented using LEX tool and the output is verified.

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

**AIM:** ToWrite a C program for implementing the functionalities of predictive parser for the mini language specified in Note 1.

**ALGORITHM:**

1. Read the input string.
2. By using the FIRST AND FOLLOW values.
3. Verify the FIRST of non-terminal and insert the production in the FIRST value
4. If we have any @ terms in FIRST then insert the productions in FOLLOW values
5. Constructing the predictive parser table

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
#include<string.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;
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(first[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(follow[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();
}
```

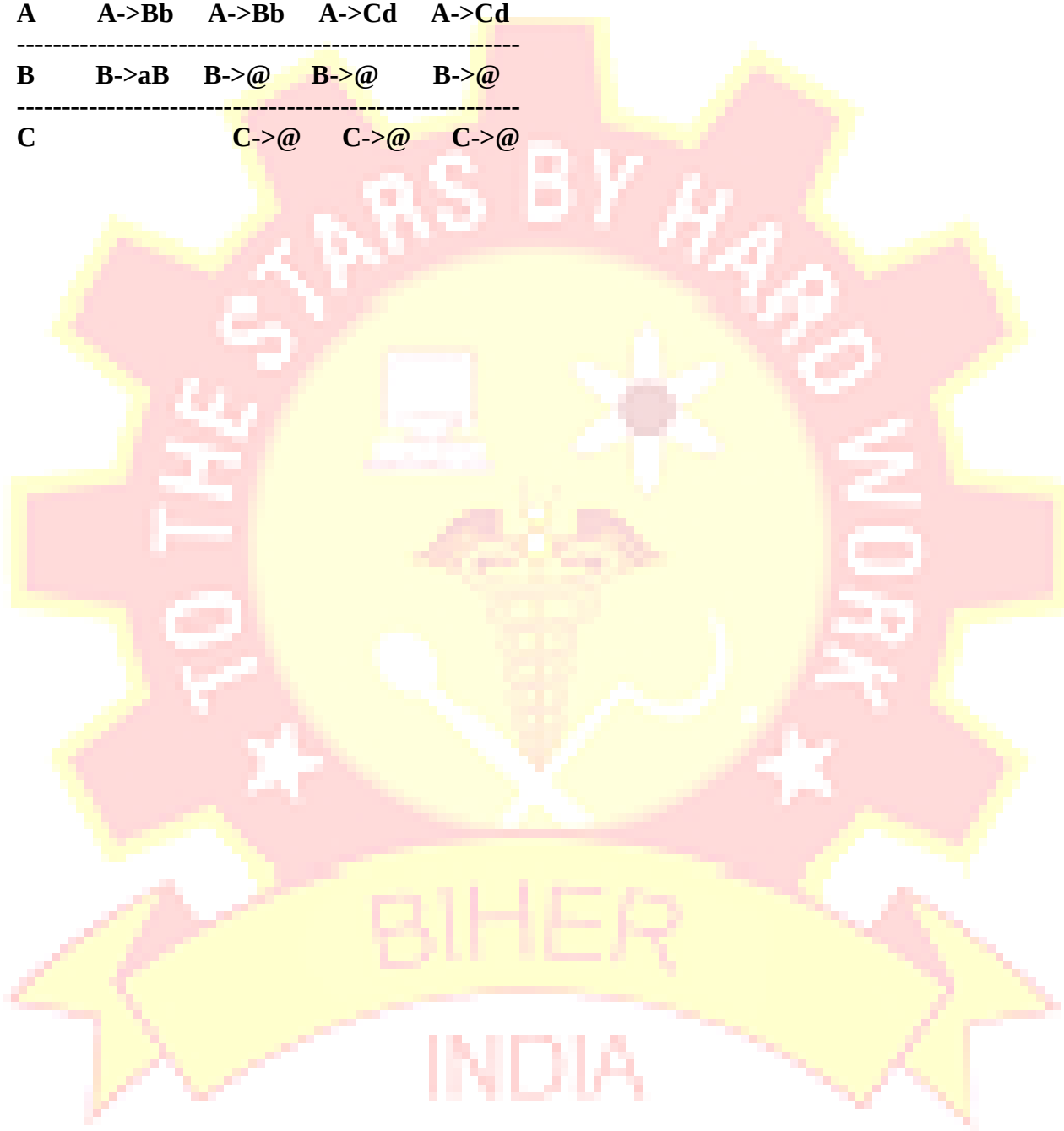
**OUTPUT :**

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->Bb | A->Cd | A->Cd | |
| B | B->aB | B->@ | B->@ | B->@ | |
| C | | | C->@ | C->@ | C->@ |




**RESULT:**

Thus the predictive parser table has been generated using predictive parser.

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

**AIM:**To write a C program to construct LL (1) parser.

**ALGORITHM:**

1. Read the input string.
2. Using predictive parsing table parse the given input using stack.
3. If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to $.

**PROGRAM:**

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

```
break;
case 't': str1=2;
break;
case 'c': str1=3;
break;
case 'f': str1=4;
break;
}
switch(s[j])
{
case 'i': str2=0;
break;
case '+': str2=1;
break;
case '*': str2=2;
break;
case '(': str2=3;
break;
case ')': str2=4;
break;
case '$': str2=5;
break;
}
if(m[str1][str2][0]=='\0')
{
printf("\nERROR");
exit(0);
}
else if(m[str1][str2][0]=='n')
i--;
else if(m[str1][str2][0]=='i')
stack[i]='i';
else
{
for(k=size[str1][str2]-1;k>=0;k--)
{
stack[i]=m[str1][str2][k];
i++;
}
i--;
}
for(k=0;k<=i;k++)
printf(" %c",stack[k]);
printf("       ");
for(k=j;k<=n;k++)
```

```
printf("%c",s[k]);
printf(" \n ");
}
printf("\n SUCCESS");
getch();
  }
```

**OUTPUT**

| **Enter the input string:i*i+i** | |
|---|---|
| Stack | INPUT |
| $bt | i*i+i$ |
| $bcf | i*i+i$ |
| $bci | i*i+i$ |
| $bc | *i+i$ |
| $bcf* | *i+i$ |
| $bcf | i+i$ |
| $bci | i+i$ |
| $bc | +i$ |
| $b | +i$ |
| SUCCESS | |

**RESULT:**

Thus the LL(1) parser table is constructed successfully.

**8.b) Write a C program for constructing recursive descent parsing**

**AIM:**To write a C program to construct recursive descent parser.

**ALGORITHM:**

1. Read the input string.
2. Write procedures for the non-terminals
3. Verify the next token equals to non-terminals if it satisfies match the non-terminal.
4. If the input string does not match print error.

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char input[100];
int i,l;
void main()
{
clrscr();
printf("\nRecursive descent parsing for the following grammar\n"); printf("\nE->TE'\nE'-
>+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/ID\n"); printf("\nEnter the string to be checked:");
gets(input);
if(E())
{
if(input[i+1]=='\0')
printf("\nString is accepted");
else
printf("\nString is not accepted");
}
else
printf("\nString not accepted");
getch();
}
E()
{
if(T())
{
if(EP())
return(1);
else
return(0);
}
else
return(0);
```

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

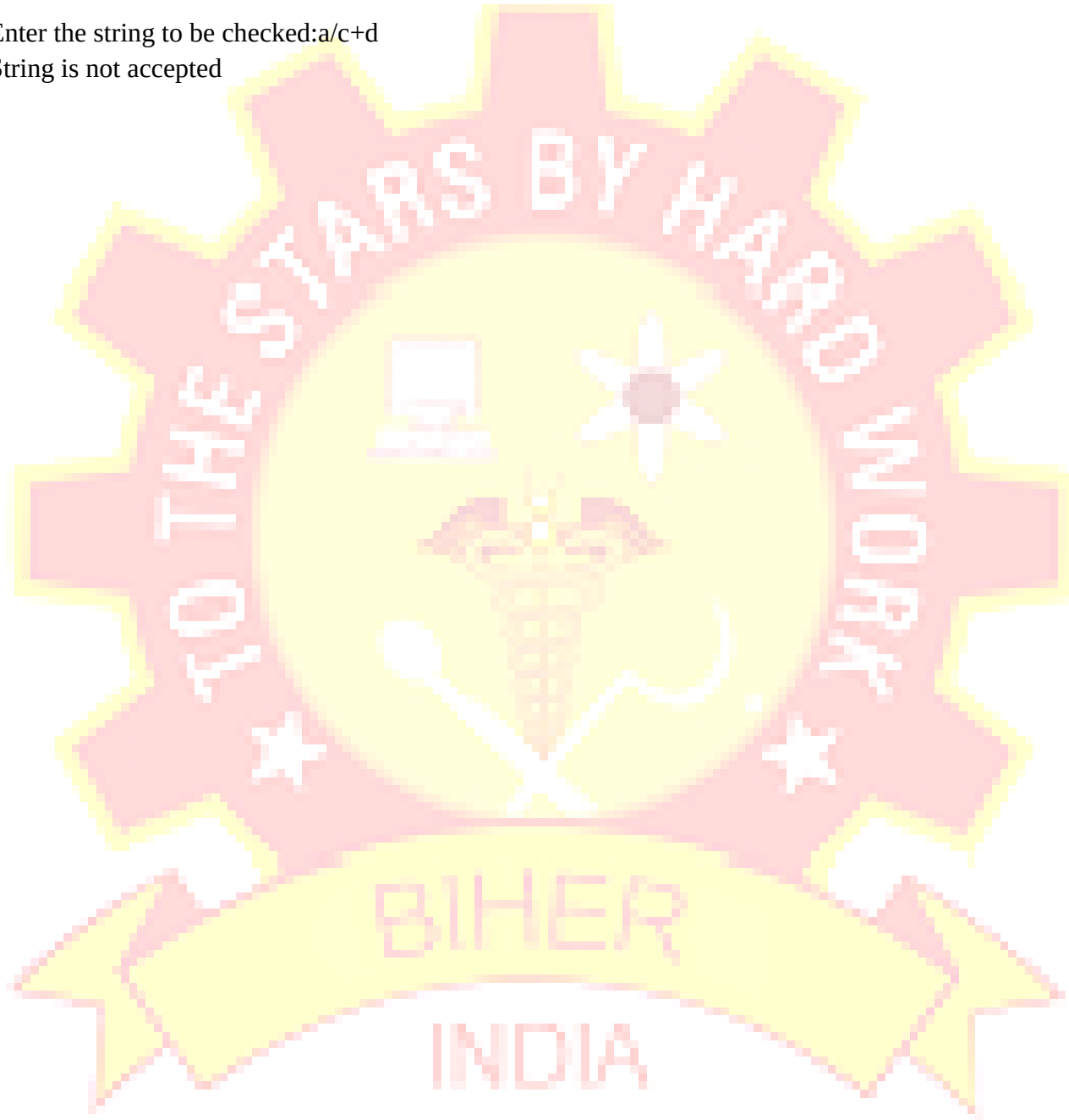
```
}
else
return(1);
}
F()
{
if(input[i]=='(')
{
i++;
if(E())
{
if(input[i]==')')
{
i++;
return(1);
}
else
return(0);
}
else
return(0);
}
else if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')
{
i++;
return(1);
}
else
return(0);
}
```

**OUTPUT:**

Recursive descent parsing for the following grammar
E->TE'
E'->+TE'/@
T->FT'
T'->*FT'/@
F->(E)/ID
Enter the string to be checked:(a+b)*c
String is accepted

Recursive descent parsing for the following grammar
E->TE'
E'->+TE'/@
T->FT'
T'->*FT'/@
F->(E)/

Enter the string to be checked:a/c+d
String is not accepted




**RESULT:**

It is inferred the given grammar is recursive decent parser or not.

## 9. Write a C program to implement LALR parsing

**AIM:** To write a C program to implement LALR parser.

**ALGORITHM:**

1. Read the input string.
2. Push the input symbol with its state symbols in to the stack by referring lookaheads
3. We perform shift and reduce actions to parse the grammar.
4. Parsing is completed when we reach $ symbol**.**

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int ister(char);
int isnter(char);
int isstate(char);
void error();
void isreduce(char,char);
char pop(char *,int *);
void printt(char *,int *,char [],int);
void rep(char [],int);
struct action
{
char row[6][5];
};
const struct action A[12]={
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","emp","acc"},
{"emp","rc","sh","emp","rc","rc"},
{"emp","re","re","emp","re","re"},
{"sf","emp","emp","se","emp","emp"},
{"emp","rg","rg","emp","rg","rg"},
{"sf","emp","emp","se","emp","emp"},
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","sl","emp"},
{"emp","rb","sh","emp","rb","rb"},
{"emp","rb","rd","emp","rd","rd"},
{"emp","rf","rf","emp","rf","rf"}
};
struct gotol
{
char r[3][4];
};
const struct gotol G[12]={
{"b","c","d"},
```

```
{"emp","emp","emp"},
{"emp","emp","emp"},
{"emp","emp","emp"},

{"i","c","d"},
{"emp","emp","emp"},
{"emp","j","d"},
{"emp","emp","k"},
{"emp","emp","emp"},
{"emp","emp","emp"},
};

char ter[6]={'i','+','*',')','(','$'};
char nter[3]={'E','T','F'};
char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};
char stack[100];
int top=-1;
char temp[10];
struct grammar
{
char left;
char right[5];
};
const struct grammar rl[6]={
{'E',"e+T"},
{'E',"T"},
{'T',"T*F"},
{'T',"F"},
{'F',"(E)"},
{'F',"i"},
};

void main()
{
char inp[80],x,p,dl[80],y,bl='a';
int i=0,j,k,l,n,m,c,len;
clrscr();
printf(" Enter the input :");
scanf("%s",inp);
len=strlen(inp);
inp[len]='$';
inp[len+1]='\0';
push(stack,&top,bl);
printf("\n stack \t\t\t input");
printt(stack,&top,inp,i);
do
{
x=inp[i];
p=stacktop(stack);
isproduct(x,p);
if(strcmp(temp,"emp")==0)
```

```
error();
if(strcmp(temp,"acc")==0)
break;
else
{
if(temp[0]=='s')
{
push(stack,&top,inp[i]);
push(stack,&top,temp[1]);
i++;
}
else
{
if(temp[0]=='r')
{
j=isstate(temp[1]);
strcpy(temp,rl[j-2].right);
dl[0]=rl[j-2].left;
dl[1]='\0';
n=strlen(temp);
for(k=0;k<2*n;k++)
pop(stack,&top);
for(m=0;dl[m]!='\0';m++)
push(stack,&top,dl[m]);
l=top;
y=stack[l-1];
isreduce(y,dl[0]);
for(m=0;temp[m]!='\0';m++)
push(stack,&top,temp[m]);
}
}
}
printt(stack,&top,inp,i);
}while(inp[i]!='\0');
if(strcmp(temp,"acc")==0)
printf(" \n accept the input ");
else
printf(" \n do not accept the input ");
getch();
}
void push(char *s,int *sp,char item)
{
if(*sp==100)
printf(" stack is full ");
else
{
*sp=*sp+1;
s[*sp]=item;
}
}
char stacktop(char *s)
```

```
{
char i;
i=s[top];
return i;
}
void isproduct(char x,char p)
{
int k,l;
k=ister(x);
l=isstate(p);
strcpy(temp,A[l-1].row[k-1]);
}
int ister(char x)
{
int i;
for(i=0;i<6;i++)
if(x==ter[i])
return i+1;
return 0;
}
int isnter(char x)
{
int i;
for(i=0;i<3;i++)
if(x==nter[i])
return i+1;
return 0;
}
int isstate(char p)
{
int i;
for(i=0;i<12;i++)
if(p==states[i])
return i+1;
return 0;
}

void error()
{
printf(" error in the input ");
exit(0);
}

void isreduce(char x,char p)
{
int k,l;
k=isstate(x);
l=isnter(p);
strcpy(temp,G[k-1].r[l-1]);
}
```

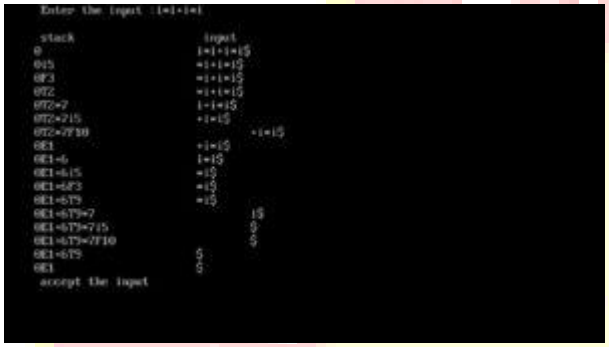
```
char pop(char *s,int *sp)
{
char item;

if(*sp==-1)
printf(" stack is empty ");
else
{
item=s[*sp];
*sp=*sp-1;
}
return item;
}
void printt(char *t,int *p,char inp[],int i)
{
int r;
printf("\n");
for(r=0;r<=*p;r++)
rep(t,r);
printf("\t\t\t");
for(r=i;inp[r]!='\0';r++)
printf("%c",inp[r]);
}

void rep(char t[],int r)
{
char c;
c=t[r];
switch(c)
{
case 'a': printf("0");
break;
case 'b': printf("1");
break;
case 'c': printf("2");
break;
case 'd': printf("3");
break;
case 'e': printf("4");
break;
case 'f': printf("5");
break;
case 'g': printf("6");
break;
case 'h': printf("7");
break;
case 'm': printf("8");
break;
case 'j': printf("9");
break;
case 'k': printf("10");
```

```
break;
case 'l': printf("11");
break;
default :printf("%c",t[r]);
break;
}
}
```

**OUTPUT:**




**RESULT:**

Thus the program to implement LALR Parser has been done successfully.

**10. a) Write a C program to implement operator precedence parsing.**

**AIM:** To write a C program to implement operator precedence parsing.

**ALGORITHM:**

1. Read the input string.
2. Push the input symbol with its state symbols in to the stack by referring lookaheads
3. We perform shift and reduce actions to parse the grammar.
4. Parsing is completed when we reach $ symbol**.**

**PROGRAM:**

**<parser.l>**

```
%{
#include<stdio.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);
}
```

**OUTPUT**

```
$lex parser.l
$yacc –d parser.y
$cc lex.yy.c y.tab.c –ll –lm
$./a.out
2+3
5.0000
```

**RESULT**

Thus the program to perform operator precedence parser is executed successfully.

**10. 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**

**AIM:**To write a C program to implement semantic rules to calculate the expression that that takes an expression with digits, + and *.

**ALGORITHM:**

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

**PROGRAM:**

```
<parser.l>
%{
#include<stdio.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' {
38
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);
}
```

**INPUT & OUTPUT:**

```
$lex parser.l
$yacc –d parser.y
$cc lex.yy.c y.tab.c –ll –lm
$./a.out
2+3
5.0000
```

**RESULT:**

Thus the program to calculate the expressionthat takes an expression with digits, + and * and computes the value has been executed successfully.

**11. Convert the BNF rules into Yacc form and write code to generate abstract syntax tree for the mini language specified in Note 1.**

**AIM:** To Implement the process of conversion from BNF rules to Yacc form and generate Abstract Syntax Tree.

**ALGORITHM:**

1. Reading an input file line by line.
2. Convert it in to abstract syntax tree using three address code.
3. Represent three address code in the form of quadruple tabular form

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

```
Ind=pop();
push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
```

```
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------""\n\t\t Pos Operator 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
```

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






**RESULT:**

It is inferred that the conversion from BNF rules to Yacc form andgenerate Abstract Syntax Tree has been done successfully.

**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.**

**AIM:** To write a C Program to Generate Machine Code from the Abstract Syntax Tree using the specified machine instruction formats.

**ALGORITHM:**

1. Read input string
2. Consider each input string and convert in to machine code instructions

**PROGRAM:**

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

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

```
fprintf(fp2,"\n\t JGT %s,label#%s",operand2,result);
label[no++]=atoi(result);
break;
case '<': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1); fprintf(fp2,"\n\t
JLT %s,label#%d",operand2,result);
label[no++]=atoi(result);
break;
}
}
fclose(fp2); fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("Error opening the file\n");
exit(0);
}
do
{
ch=fgetc(fp2);
printf("%c",ch);
}while(ch!=EOF);
fclose(fp1);
return 0;
}
int check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
}
return 0;
}
```

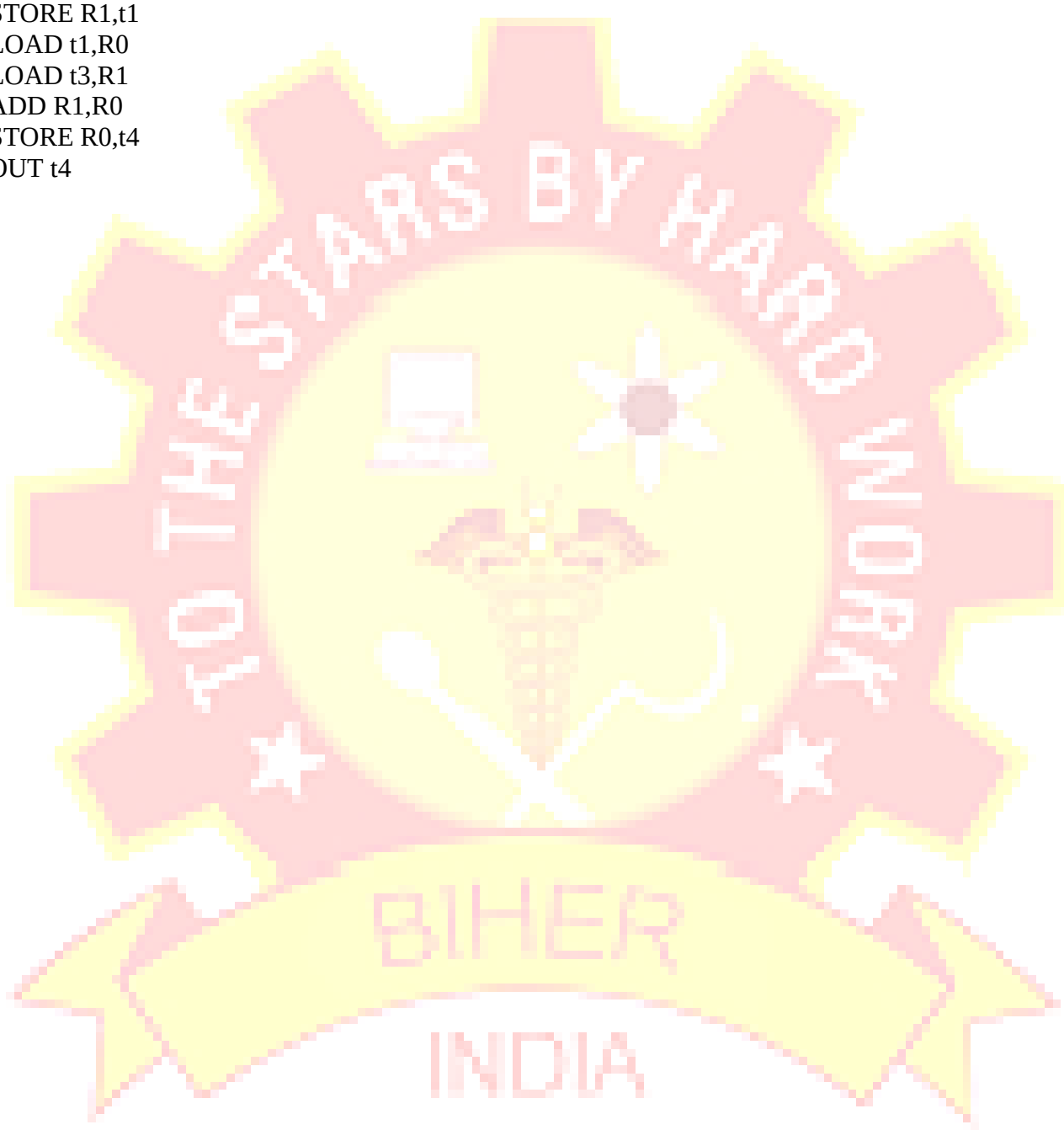
**INPUT:**

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

**OUTPUT:**

```
Enter filename of the intermediate code: int.txt
STORE t1,2
STORE a[0],1
STORE a[1],2
STORE a[2],3
LOAD t1,R0
LOAD 6,R1
ADD R1,R0
STORE R0,t3
LOAD a[2],R0
LOAD t2,R1
ADD R1,R0
STORE R0,t3
LOAD a[t2],R0
LOAD t1,R1
SUB R1,R0
STORE R0,t2
LOAD t3,R0
LOAD t2,R1 21
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
```




**RESULT:**

Thus the program to generate machine code from abstract syntax tree generated by the parser.