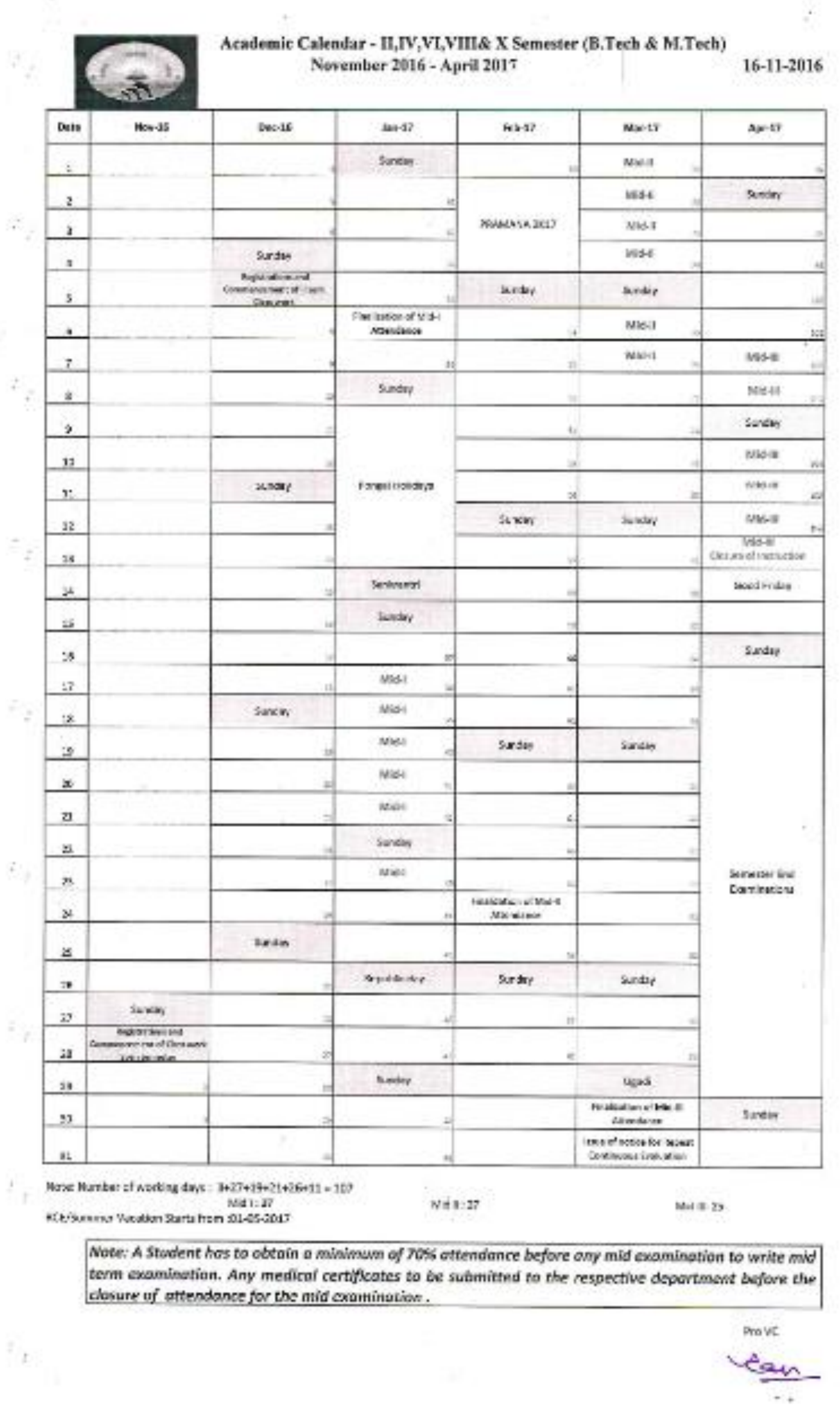
Language Processors Lab Manual

(EURCS 611)

Contents



Academic Calendar - II,IV,VI,VIII& X Semester (B.Tech & M.Tech)
November 2016 - April 2017 **16-11-2016**

Date	Nov-16	Dec-16	Jan-17	Feb-17	Mar-17	Apr-17
1			Sunday		Mid-II	
2				PRAMANA 2017	Mid-II	Sunday
3					Mid-II	
4		Sunday			Mid-II	
5		Registration and Commencement of Class work		Sunday	Sunday	
6			Finalization of Mid-I Attendance		Mid-II	
7					Mid-II	Mid-III
8			Sunday			Mid-III
9			Pongal Holidays			Sunday
10						Mid-III
11		Sunday				Mid-III
12				Sunday	Sunday	Mid-III
13						Mid-III Closure of Instruction
14			Sankranti			Good Friday
15			Sunday			
16						Sunday
17			Mid-I			Semester End Examinations
18		Sunday	Mid-I			
19			Mid-I	Sunday	Sunday	
20			Mid-I			
21			Mid-I			
22			Sunday			
23			Mid-I			
24				Finalization of Mid-II Attendance		
25		Sunday				
26			Republic day	Sunday	Sunday	
27	Sunday					
28	Registration and Commencement of Class work [illegible]					
29			Sunday		Ugadi	
30					Finalization of Mid-III Attendance	Sunday
31					Issue of notice for Subject Continuous Evaluation	

Note: Number of working days : 3+27+19+21+26+11 = 107
Mid I : 27 Mid II : 27 Mid III : 23
#CE/Summer Vacation Starts from 01-05-2017

Note: A Student has to obtain a minimum of 70% attendance before any mid examination to write mid term examination. Any medical certificates to be submitted to the respective department before the closure of attendance for the mid examination .

Pro VC

[illegible]

3. Syllabus & List of cycle wise Experiments

B.Tech. (CSE) Sixth Semester
LANGUAGE PROCESSORS LAB

Code: EURCS 611 Category: CE **Department: CSE**
Credits: 2 Hours: 3 per week

1. Implement transition diagrams for identifying an identifier and constant and classify the identifier as either variable or array or function or structure and constant as integer or real.

2. Consider the following circuit language that used to describe the circuits(parallel,sequential or both). The syntax of the language is defined by following grammar

   <circuit> --->seq<block> | par<block> | <value>
   <block>---><circuit><block>| end
   <value>---><digit><value> | <digit>
   <digit>----->0|1|2|3|4|5|6|7|8|9
   A simple program written in this language is:
   par66 seq par 3672 9 end
   (36, 72 connected in parallel intern connected with 9 in sequential intern connected with 66 in parallel.)

a. Design a Lexical analyser for the above language. The lexical analyser should ignore redundant spaces, tabs and newlines. Although the syntax specification states that value can be arbitrarily long, you may restrict the length to some reasonable value.
b. Implement the lexical analyser using JLex, flex or lex or other lexical analyser generating tools.
c. Implement syntax analyser for the circuit language by using RecursiveDescent parser which takes output of the lexical analyser and checks whether it follows the syntax or not. (Hint: assume that lexical analyser returns val token for integer constant then grammar isC-->seqB|parB|val B-->CB|end and checking for this words in implementation of recursive descent parsing is like next==s and next+1==e and next+2==q for seq)
d. Write Yacc code to get final circuit value for the above simple program output should be 22.

   (Hint: 36, 72 in parallel value is 24. 24, 9 in sequential value is 33. 33,66 in parallel value is 22. Final result is 22.)

3. Write a program to read CFG productions and store into corresponding data structures.

4. Write a program for eliminating Left-Recursion from the given grammar.

5. Write a program for applying left factoring to a given grammar.

6. Implement algorithms for finding First and Follow symbols of a given grammar.

7. Consider the following Expression language that used to describe the Arithmetic

Expressions in Calculator the syntax of the language is defined by following grammar

<Line>--><exp>\n
<exp> ---><exp>+ <exp>| <exp>-<exp>|<exp>*<exp>|<exp>/<exp>
<exp>--><value>
<value>---><digit><value> | <digit>
<digit>----->0|1|2|3|4|5|6|7|8|9
A simple program written in this language is: 2+45*34/17-67+78/6\n
Construct Recursive Decent Parser for the given string.

Program 1:

AIM: Implement transition diagrams for identifying an identifier and constant and classify the identifier as either variable or array or function or structure and constant as integer or real.

Lexical Analyzer: It is the 1st phase of compiler which takes the source program as input and gives a stream of tokens as output.

Algorithm:
Assumptions:

Rules to identify a identifier:
Variable: letter followed by zero/more number of letters or digits. Regular Expression is
l**etter (letter/digit)***
Array: letter followed by zero/more number of letters or digits followed by open square bracket ([ ).
Regular Expression is **letter (letter/digit)*[**
Function: letter followed by zero/more number of letters or digits followed by open and close
Parenthesis. Regular Expression is **letter (letter/digit)*()**

Rules to identify a Constant:

Integer Constant : digit followed by zero/more number of digits. Regular Expression is
digit (digit)*
Real Constant: Regular expression is **digit (digit)*. digit (digit)***

Step 1: Read the input string from the keyboard.
Step 2: Check whether the input is satisfying the rules of identifier (either a normal variable or array or function or structure) or a constant that can be either integer constant or real constant.
Step 3: if the input satisfying the rule of any type display as a token of respective type. Otherwise display as wrong input.

Program:

```
//identify the identifiers and constants and variables
#include<stdio.h>
#include<conio.h>
void main()
{
char str[20];
int state,i=1;
clrscr();
printf("Enter the string:");
gets(str);
if(isalpha(str[0]))
{
state=1;
}
else if(isdigit(str[0]))
{
state=2;
}
else
{
state=10;
}
while(str[i]!='\0')
{
switch(state)
{
case 1:if(isalpha(str[i])||isdigit(str[i]))
       {
       state=1;
       }
       else if(str[i]=='(')
       {
       state=3;
       }
       else if(str[i]=='[')
       {
       state=5;
       }
       else
       {
       state=10;
       }
       break;
case 2:if(isdigit(str[i]))
       {
       state=2;
       }
       else if(str[i]=='.'&&isdigit(str[i+1]))
       {
       state=4;
       }
       else
       {
       state=10;
       }
       break;
case 3:if(str[i]==')'&&str[i+1]=='\0')
       {
       state=6;
       }
       else
       {
       state=10;
       }
       break;
case 4:if(isdigit(str[i]))
       {
       state=4;
       }
       else
       {
       state=10;
       }
       break;
case 5:if(isdigit(str[i]))
       {
       state=5;
       }
       else if(str[i]==']'&&str[i+1]=='\0')
       {
       state=7;
       }
       else
       {
       state=10;
       }
       break;
}
i++;
if(state==10)
{
```

```
break;
}
}
printf("%s",str);
if(state==1)
{
printf("Input  is a variable.");
}
else if(state==2)
{
printf("Input  is a integer constant");
}
else if(state==4)
{
printf("Input is a real constant");
}
else if(state==6)
{
printf("Input  is a function");
}
else if(state==7)
{
printf("Input is a array");
}
else
{
printf("there is an error in the given expression");
}
getch();
}
```

OUTPUT:

```
Enter the string: Amount
Input is a variable.
Enter the string: 10
Input is a integer constant
Enter the string: 10.25
Input is a real constant
Enter the string: add()
Input is a function
Enter the string: a[]
Input is a array
```

Program 2:

AIM : Write a program to read CFG productions and store into corresponding data structures.

Grammar G =(V, T, P, S)
 Where V is a variable
  T is a terminal
  P is a production
  S is a starting production

In case of CFG every production of the form A→α; where A is a single Non Terminal and α belongs to {VUT}*

Algorithm:
Step 1: Read the productions from the keyboard.
Step 2: Display the grammar in the form of Context free grammar.
Step 3: Find out all four quantities of entered grammar.

Program:
//write a program to construct context free grammar

```
#include<stdio.h>
#include<conio.h>
struct str
{
char st[1];
char rule[20];
}g[15];
void main()
{
int i,j,n;
clrscr();
printf("enter the  number of productions:");
scanf("%d",&n);
printf("enter the productions:\n");
for(i=0;i<n;i++)
{
printf("enter the non-terminal:");
scanf("%s",g[i].st);
printf("enter the RHS production:");
scanf("%s",g[i].rule);
}
printf("\nentered productions are:\n");
for(i=0;i<n;i++)
{
printf("%c%c%c",g[i].st[0],'-','>');
printf("%s\n",g[i].rule);
printf("the starting symbol of the production is:");
printf("%c\n",g[i].st[0]);
}
printf("\nthe non-terminals are:");
for(i=0;i<n;i++)
{
printf("%c\n",g[i].st[0]);
}
printf("\nthe terminals are:");
for(i=0;i<n;i++)
{
for(j=0;g[i].rule[j]!='\0';j++)
{
if(!isupper(g[i].rule[j])&&(g[i].rule[j]!='/'))
{
printf("\t%c",g[i].rule[j]);
}
}
}
getch();
}
```

OUTPUT:

```
enter the  number of productions: 3
enter the productions:
enter the non-terminal:E
enter the RHS production:cAd
enter the non-terminal:A
enter the RHS production:bc
enter the non-terminal:B
enter the RHS production:d
entered productions are:
E→cAd
A→ba
B→e
the starting symbol of the production is: E
the non-terminals are: E  A  B
the terminals are: c d b a e
```

Program 3:

AIM : Write a program for eliminating Left-Recursion from the given grammar.

Left Recursion:

If any of the production of the form A→Aα | β is called left recursive production. To remove the left recursion we use the following rules.

A→βA’

A’→αA’ |ε

Ex. E→E+T | T

E→TE’
E’→TE’ |ε

Algorithm:

```
Step 1 : Read the no. of productions.
Step 2: Read the grammar based on the no. of productions.
Step 3: // for eliminating left recursion from the grammar
        Do for all productions
                If left side non terminal is matching with 1st grammar symbol of any alternative.
                        Consider the remain part of that alternative as α and the alternative not
                        having the non terminal as 1st grammar symbol as β and store α and β
                        values in  to arrays.

                        display the productions according to the rules specified.
                else
                        display the production as it is.
```

Program:

```
//left recursive production
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void main()
{
char prod[20],alpha[20],beta[20];
int i,j;
clrscr();
printf("enter the  production:");
gets(prod);
if(prod[0]==prod[3])
{
printf("\nentered production is an left recursive");
i=4;j=0;
while(prod[i]!='/')
{
alpha[j++]=prod[i++];
}
alpha[j]='\0';
i=i+1;
j=0;
while(prod[i]!='\0')
{
beta[j++]=prod[i++];
}
beta[j]='\0';
printf("\nthe production after removing the left productions is:\n");
printf("\n%c->%s%c'",prod[0],beta,prod[0]);
printf("\n%c'->%s%c'/$",prod[0],alpha,prod[0]);
}
else
{
printf("entered production is not left recursive");
}
getch();
}
```

OUTPUT:

1)

```
enter the  production: E□E+T/T

entered production is an left recursive

the production after removing the left productions is:

E□TE’
E’□TE’ |ε
```

2)

```
enter the  production: A□BC/a

entered production is not left recursive
```

//left recursion from grammar

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void main()
{
char prod[20][20],alpha[20][20],beta[20][20];
int i,j,k,n;
clrscr();
printf("enter the number of productions:");
scanf("%d",&n);
printf("enter %d productions:",n);
for(k=0;k<=n;k++)
{
gets(prod[k]);
}
for(k=1;k<=n;k++)
{
if(prod[k][0]==prod[k][3])
{
printf("\nentered production is an left recursive");
i=4;j=0;
while(prod[k][i]!='/')
{
alpha[k][j++]=prod[k][i++];
}
alpha[k][j]='\0';
i=i+1;
j=0;
while(prod[k][i]!='\0')
{
beta[k][j++]=prod[k][i++];
}
beta[k][j]='\0';
printf("\nthe production %d after removing the left productions is:\n",k);
printf("\n%c->%s%c'",prod[k][0],beta[k],prod[k][0]);
printf("\n%c'->%s%c'/$",prod[k][0],alpha[k],prod[k][0]);
}
else
{
printf("entered production is not left recursive");
}
}
getch();
}
```

OUTPUT:

```
enter the number of productions: 3

enter 5 productions:

E□E+T/T

T□T*F/F

F□(e)/id

entered production is an left recursive

the production 1 after removing the left productions is

E□TE’

E’□+TE’/$

entered production is an left recursive

the production 2 after removing the left productions is

T□FT’

T’□*FT’/$

entered production is not left recursive
```

Program 4:

AIM: Write a program for applying left factoring to a given grammar.

Left Factoring:

If any of the production of the form A$\to$αβ_1 | αβ_2 | αβ_3 is called left factored production. To remove the left factoring we use the following rules.

A$\to$αA'

A'$\to$ β_1|β_2|β_3

Program:

//removing left factoring from production

```
#include<stdio.h>
#include<stdlib.h>

void main()
{
char prod[20],alpha[10],beta1[10],beta2[10];
int i,j,k;
clrscr();
printf("enter the production:");
gets(prod);
if(prod[1]=='-'&&prod[2]=='>')
{
i=3;j=0;
while(prod[j]!='/')
{j++;}
j++;
k=0;
while(prod[i]==prod[j])
{
alpha[k++]=prod[i++];
j++;
}
alpha[k]='\0';
k=0;
while(prod[i]!='/')
{
beta1[k++]=prod[i++];
}
beta1[k]='\0';
```

```
k=0;
if(prod[j]=='\0')
{
beta2[k++]='$';
beta2[k]='\0';
}
else
{
while(prod[j]!='\0')
{
beta2[k++]=prod[j++];
}
beta2[k]='\0';
}
printf("\n%c->%s%c'\n",prod[0],alpha,prod[0]);
printf("%c'->%s/%s",prod[0],beta1,beta2);
}
else
{
printf("the production is not the correct format");
}
getch();
}
//removing left factoring from grammer

#include<stdio.h>
#include<stdlib.h>

void main()
{
char prod[20][20],alpha[10][10],beta1[10][10],beta2[10][10];
int i,j,k,x,n;
clrscr();
printf("enter the number of productions:");
scanf("%d",&n);
printf("enter the %d productions:",n);
for(i=0;i<=n;i++)
{
gets(prod[i]);
}
for(x=1;x<=n;x++)
{
if(prod[x][1]=='-'&&prod[x][2]=='>')
{
i=3;j=0;
while(prod[x][j]!='/')
{j++;}
j++;
k=0;
while(prod[x][i]==prod[x][j])
{
alpha[x][k++]=prod[x][i++];
j++;
}
alpha[x][k]='\0';
k=0;
while(prod[x][i]!='/')
{
beta1[x][k++]=prod[x][i++];
}
beta1[x][k]='\0';
k=0;
if(prod[x][j]=='\0')
{
beta2[x][k++]='$';
beta2[x][k]='\0';
}
else
{
while(prod[x][j]!='\0')
{
beta2[x][k++]=prod[x][j++];
}
beta2[x][k]='\0';
}
printf("\n%c->%s%c'\n",prod[x][0],alpha[x],prod[x][0]);
printf("%c'->%s/%s",prod[x][0],beta1[x],beta2[x]);
}
else
{
printf("the production is not the correct format");
}
}
getch();
}
```

Program 5:

AIM: Implement algorithms for finding First and Follow symbols of a given grammar.

Program:
//write a program to fing "First" and "Follow" of a production

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

int n,counter=0;
char arr[20][20];
void First(char *);
void Follow(char *);

void main()
{
int i;
clrscr();
printf("enter the number of productions:");
scanf("%d",&n);
n++;
printf("enter the productions:");
for(i=0;i<n;i++)
{
gets(arr[i]);
}
for(i=1;i<n;i++)
{
printf("\nFirst(%c):",arr[i][0]);
First(&arr[i][0]);
counter=0;
printf("\nFallow(%c):",arr[i][0]);
Follow(&arr[i][0]);
}
getch();
}

void First(char *p)
{
int i,j;
for(i=0;i<n;i++)
{
if(arr[i][0]==*p)
{
if(isupper(arr[i][3]))
{
if(arr[i][3]==arr[i][0])
{}
else
{
First(&arr[i][3]);
}
}
else
{
printf("%c ",arr[i][3]);
}
j=0;
while(arr[i][j]!='\0')
{
if(arr[i][j]=='/')
{
j++;
if(isupper(arr[i][j]))
{
if(arr[i][j]==*p)
{
}
else
{
First(&arr[i][j]);
}
}
else
{
printf("%c ",arr[i][j]);
}
}
j++;
}
}
}
}

void Follow(char *p)
{
int i,j;
if(*p==arr[1][0])
{
printf("$ ");
}
```

```
for(i=0;i<n;i++)
{
j=3;
while(arr[i][j]!='\0')
{
if(arr[i][j]==*p)
{
j++;
if(arr[i][j]=='\0'||arr[i][j]=='/')
{
if(counter==0)
{
Follow(&arr[i][0]);
counter++;
}
}
else if(isupper(arr[i][j]))
{
First(&arr[i][j]);
}
else
{
printf("%c ",arr[i][j]);
}
}
j++;
}
}
}
```

Program 6:
AIM: Write a C program to implement Recursive Decent Parsing for the grammer
E->TE'
E'->+TE'/$
T->FT'
T'->*FT'/$
F->(E)/a

here $ represents "epsilon" production

and check wheather given string a+a*a is accepted or not

```
#include<stdio.h>
#include<conio.h>
int i=0,t=1;
char str[20];
void E();
void EPRIME();
void T();
void TPRIME();
void F();
void error();
void main()
{
clrscr();
printf("the given grammer is:\n");
printf("\nE -> TE'");
printf("\nE' -> +TE'/$");
printf("\nT -> FT'");
printf("\nT' -> *FT'/$");
printf("\nF -> (E)/a");
printf("\nenter the string to be parrsed:");
gets(str);
E();
if(t!=1||str[i]!='\0')
{
printf("\ngiven string is not accepted");
}
else
{
printf("the given string is accepted");
}
getch();
}
```

```
void E()
{
T();
EPRIME();
}

void EPRIME()
{
if(str[i]=='+')
{
i++;
T();
EPRIME();
}
else
{
return;
}
}

void T()
{
F();
TPRIME();
}

void TPRIME()
{
if(str[i]=='*')
{
i++;
F();
TPRIME();
}
else return;
}

void F()
{
if(str[i]=='a')
{
i++;
}
else if(str[i]=='(')
{
i++;
E();
if(str[i]==')')
{
i++;
}
}
else
{error();}
}

void error()
{
t=-1;
}
```

AIM : Consider the following Expression language that used to describe the Arithmetic expressionsin Calculator The syntax of the language is defined by following grammar

<Line>--><exp>\n
<exp> ---><exp>+ <exp>| <exp>-<exp>|<exp>*<exp>|<exp>/<exp>
<exp>--><value>
<value>---><digit><value> | <digit>
<digit>----->0|1|2|3|4|5|6|7|8|9

A simple program written in this language is: 2+45*34/17-67+78/6\n

Write a program to implement recursive decent parsing using parser for the given grammar and pass a string 2+45*34/17-67+78/6

Rename <Line> as L
<exp> as E
<value> as V
<digit> as D
Modified grammar is:
L->E
E->E+E/E-E/ E/E /E*E /V
V->DV/D
D->0/1/2/3/4/5/6/7/8/9

Program :

```
#include<stdio.h>
#include<conio.h>
int i=0,t=1;
char str[40];
void L();
void E();
void EPRIME();
void V();
void VPRIME();
void D();
void error();

void main()
{
clrscr();
printf("\nthe given grammer is:");
printf("\nL -> E");
printf("\nE -> VE'");
printf("\nE' -> *EE'");
printf("\nE' -> -EE'");
printf("\nE' -> +EE'");
printf("\nE' -> /EE'");
printf("\nV -> DV'");
printf("\nV' -> V/$");
printf("\nD -> 0/1/2/3/4/5/6/7/8/9");
printf("\nenter the string :");
gets(str);
L();
if(t!=1||str[i]!='\0')
{printf("entered string is not accepted");}
else
{printf("entered string is accepted");}
getch();
}

void L()
{
E();
}

void E()
{
V();
EPRIME();
}

void EPRIME()
{
if(str[i]=='+'||str[i]=='/'||str[i]=='-'||str[i]=='*')
{
i++;
E();
EPRIME();
}
else return;
}

void V()
{
D();
VPRIME();
}

void VPRIME()
{
if(isdigit(str[i]))
{
```

```
V();
}
else return;
}

void D()
{
if(str[i]<='9'&&str[i]>='0')
{
i++;
}
else error();
}

void error()
{t=-1;}
```

Program 7:
AIM: Write a program to implement operator precedence parsing for the grammar E→E+E|E*E|d and check the string d+d*d is syntactically correct or wrong.

Program:
//write a program for "operator precedence"

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

int i,x,y;
char stack[20];
char arr[4][4]={{'b','>','>','>'},
                {'<','>','<','>'},
                {'<','>','>','>'},
                {'<','<','<','b'}};
char pop();
void push(char *p);
int prec(char p);
char top();

void main()
{
int j,flag=0;
char str[30];
clrscr();
i=0;
stack[i++]='$';
printf("enter the string:");
gets(str);
j=0;
while(str[j]!='$')
{
y=prec(str[j]);
x=prec(top());
if(arr[x][y]=='<')
{
push(&arr[x][y]);
push(&str[j++]);
}
else if(arr[x][y]=='>')
{
while(top()!='<')
{
pop();
}
pop();
}
else if(arr[x][y]=='b')
{
flag=1;
break;
}
}
if(flag==0)
{
while(top()!='$')
{
y=prec(str[j]);
x=prec(top());
if(arr[x][y]=='>')
{
while(top()!='<')
{
pop();
}
pop();
}
else if(arr[x][y]=='b')
{
flag=1;
break;
}
}
}
if(flag==0)
{
printf("the entered string is acceped");
}
else
{
printf("the entered string is not accepted");
}
getch();
}

char pop()
{
return stack[i--];
}

void push(char *p)
```

```
{
stack[i++]=*p;
}

char top()
{
return stack[i-1];
}

int prec(char p)
{
if(p=='d')
{
return 0;
}
else if(p=='+')
{
return 1;
}
else if(p=='*')
{
return 2;
}
else if(p=='$')
{
return 3;
}
}
```

Lex and YACC programs

Lex program to identify the identifier, String and a digit

```
%%
[a-zA-Z]+ printf("It is a String\n");
[a-z]([0-9])* printf("It is Identifier\n");
[0-9]+ printf("It is a digit\n");
. printf("Invalid input\n");
%%
```

Output:

```
tanav@ubuntu:~/lex$ lex f.l
tanav@ubuntu:~/lex$ cc lex.yy.c -ll
tanav@ubuntu:~/lex$ ./a.out
Hai
It is a String
13
It is a digit
a12
It is Identifier
$
Invalid input
```

Lex program to recognize C program

```
%{
#include <stdio.h> /* printf() */
#include <stdlib.h> /* atof() */
%}
DIGIT [0-9]
ID [a-zA-Z][a-zA-Z0-9]*
%%
{DIGIT}+ { printf( "INTEGER: %s (%d)\n", yytext, atoi( yytext ) ); }
{DIGIT}+"."{DIGIT}* { printf( "DOUBLE: %s (%g)\n", yytext,atof(yytext)); }
float |
int |
double |
for |
if { printf( "KEYWORD: %s\n", yytext ); }
{ID} printf( "IDENTIFIER: %s\n", yytext );
":"|"{"|"}"|"("|")"|";"|"," printf( "PUNCTUATION: %s\n", yytext);
"=" printf( "OPERATOR: %s\n", yytext );
[ \t\n]+ /* eat up whitespace */
. printf( "Unrecognized character: %s\n", yytext);
%%
```

```
tanav@ubuntu:~/lex$ lex srg.l
tanav@ubuntu:~/lex$ cc lex.yy.c -ll
tanav@ubuntu:~/lex$ ./a.out
//display hello
#include<stdio.h>
main()
{
int x;
printf("hello\n");
}Unrecognized character: /
Unrecognized character: /
IDENTIFIER: display
IDENTIFIER: hello
Unrecognized character: #
IDENTIFIER: include
Unrecognized character: <
IDENTIFIER: stdio
Unrecognized character: .
IDENTIFIER: h
Unrecognized character: >
IDENTIFIER: main
PUNCTUATION: (
PUNCTUATION: )
PUNCTUATION: {
KEYWORD: int
IDENTIFIER: x
PUNCTUATION: ;
IDENTIFIER: printf
PUNCTUATION: (
Unrecognized character: "
IDENTIFIER: hello
Unrecognized character: \
IDENTIFIER: n
Unrecognized character: "
PUNCTUATION: )
PUNCTUATION: ;
```

Lex program to recognize and count the number of identifiers in given input file

```
%{
#include<stdio.h>
int count=0;
%}
%%
^("int"|"float"|"char"|"double")" "[_a-z0-9]+ {count++;}
","[a-z0-9]+ {count++;}
. { }
```

```
%%
main()
{
yyin=fopen("s.c","r");
yylex();
printf("No. of identifier are:%d\n",count);
}
```

Output:

Input file:

```
#include<stdio.h>
main()
{
float f=3.2;
double d=10.23;
char c='a';
int i,a[6]={2,3,4,5,6,7};
for(i=0;i<6;i++)
printf("a[%d]=%d\n",i,a[i]);
printf("d=%lf\n",d);
printf("c=%c\n",c);
printf("f=%f\n",f);
}
No. of identifier are:15
```

Program to recognize a valid arithmetic expression that uses operators ‘+’,’-‘,’*’,’/’.

lex:l3.l

```
%{
#include"y.tab.h"
%}
%%
[0-9]+ {return NUMBER;}
[a-zA-z][0-9a-zA-Z]* {return ID;}
[ \t]+ {;}
. {return yytext[0];}
\n {return 0;}
%%
```

yacc:l3.y

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token NUMBER ID
%left '+' '-'
```

```
%left '*' '/'
%right UMINUS
%%
stmt :expr ;
expr :expr'+'expr
|expr'-'expr
|expr'*'expr
|expr'/'expr
|'('expr')'
|NUMBER
|ID
|'-'expr %prec UMINUS ;
%%
int main()
{
printf("Enter the Arithmetic Expression : ");
yyparse();
printf("The Entered Arithmetic Expression is Valid\n\n");
return 1;
}
int yyerror()
{
printf("Invalid Expression\n");
exit(0);
}
```

Output:

```
Enter the Arithmetic Expression : a-b
The Entered Arithmetic Expression is Valid
tanav@ubuntu:~/exam$ ./a.out
Enter the Arithmetic Expression : (a+b))
Invalid Expression
```

Program to evaluate an arithmetic expression involving operators +, -, *,/

l1.l

```
%{
#include<stdio.h>
#include"y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
yylval=atoi(yytext);
return NUM;
}
[\t] ;
```

```
\n return 0;
. return yytext[0];
%%
l1.y
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token NUM
%left '+' '-'
%left '*' '/'
%left '(' ')'
%%
expr: e{
printf("result:%d\n",$$);
return 0;
}
e:e'+'e {$$=$1+$3;}
|e'-'e {$$=$1-$3;}
|e'*'e {$$=$1*$3;}
|e'/'e {$$=$1/$3;}
|'('e')' {$$=$2;}
| NUM {$$=$1;}
;
%%
main()
{
printf("\n enter the arithmetic expression:\n");
yyparse();
printf("\nvalid expression\n");
}
yyerror()
{
printf("\n invalid expression\n");
exit(0);
}
```

Output:

```
enter the arithmetic expression:
3+4
result:7
valid expression
```

Implement three -address code

ca.l

```
%{
#include<stdio.h>
#include"y.tab.h"
char ch='a';
%}
%%
[0-9]+ {yylval.dval=yytext[0]; return NUM;}
\n {return 0;}
. {return yytext[0];}
%%
void yyerror(char* str)
{
printf("\n%s",str);
}
char gencode(char ch1,char first,char op,char second)
{
printf("\n%c = %c %c %c\n",ch,first,op,second);
return ch++;
}
main()
{
yyparse();
return 0;
}
```

ca.y

```
%{
#include<stdio.h>
int aaa;
%}
%union {
char dval;
}
%token <dval> NUM
%type <dval> E
%left '+' '-'
%left '*' '/'
%%
statement : E {printf("\nt = %c\n This Three Address Code Generate by AAA...\n",$1);}
;
E : E '+' E {$$=gencode($$,$1,'+',$3);}
| E '-' E {$$=gencode($$,$1,'-',$3);}
| E '*' E {$$=gencode($$,$1,'*',$3);}
| E '/' E {$$=gencode($$,$1,'/',$3);}
| '(' E ')' {$$=$2;}
| NUM {$$=$1;}
;
%%
```

Output:

```
tanav@ubuntu:~/exam$ gcc lex.yy.c y.tab.c -ll -lm
tanav@ubuntu:~/exam$ ./a.out
1+(2*3)+5-2
a = 2 * 3
b = 1 + a
c = b + 5
d = c - 2
t = d
This Three Address Code Generate by AAA.
```

4.Sample Viva Voice questions

01. What is a translator?
02. Differentiate interpreters and compilers.
03. What are loaders and linkers?
04. What is an assembler?
05. What are the phases of a compiler?
06. What is bootstrapping a compiler?
07. What are the two parts of compilation? Explain briefly.
08. Explain in brief linear analysis.
09. Explain in brief hierarchical analysis.
10. Explain in brief semantic analysis.
11. Give the parse tree for the statement a := b + c * 60.
12. What is a symbol table?
13. Briefly explain code optimization.
14. What are rational preprocessors?
15. What is the use of scanner generator?
16. What are the characteristics of a high-level programming language?
17. What do you mean by syntax and semantics of a language?
18. Give the hierarchy of programming elements.
19. Explain in brief bookkeeping in compilation?
20. Give some four compiler-generator tools.
21. Explain Macros.
22. Explain in brief code generation phase.
23. Explain in brief intermediate code generation phase.
24. Mention the cousins of the compiler.
25. What are the functions of a linker?
26. What is an object program?
27. Define a token.
28. What is a preprocessor?
29. Give the format of a macro definition.
30. What is the need of syntax analysis?
31. What are the capabilities of CFG?
32. Define regular expressions.
33. Define a finite state automation.
34. What is a transition diagram? Give example.
35. What is a recognizer?
36. What is CFG? Give example.
37. What is a parse tree?
38. Consider the grammar:

    S --> (L)/a

L --> L,S/S

What are the terminals, non-terminals and start symbols?

39. Differentiate NFA and DFA.
40. What are the types of derivation?
41. What is right-most derivation? Give example.
42. What is left-most derivation? Give example.
43. Explain canonical derivations.
44. What is left-sentential form?
45. What is right-sentential form?
46. What is sentence of a grammar G?
47. What is Context-Free Language?
48. What is regular definition? Give example.
49. What is a non-regular set?
50. What is a Lexical analyzer?
51. Write the algorithm for moving forward pointer in "input buffering" scheme.
52. What is the use of sentinels?
53. Distinguish tokens, patterns and lexemes.
54. What are the lexical-error recovery actions?
55. What is the need for input buffering?
56. Explain prefix and suffix of the string with example.
57. Explain proper prefix, proper suffix and proper substring with example.
58. What is subsequence of a sting?
59. Draw the transition diagram for identifiers.
60. What is the use of lexeme- beginning pointer and forward pointer?
61. What is a parser?
62. Write the role of parser.
63. Mention the error recovery strategies of parser.
64. What is ambiguity? Write the ways to eliminate ambiguity.
65. What is Left recursion? Write the rules to eliminate left recursion.
66. What is Left factoring? How do you eliminate it?
67. What is parsing table?
68. What is the use of LEX?
69. What is a handle?
70. What is handle pruning?
71. What is a transition diagram? Give example?
72. What is the need for separating the parser from scanner?
73. Translate the arithmetic expression a*-(b+c) into a syntax tree.
74. Define a Quadruple. How it is different from Triples.
75. What are Synthesized & Inherited Attributes?
76. What do you mean by back patching? Why we are going for back patching.

77. What are the three storage allocation strategies? Explain each in detail.
78. Need for code optimization.
79. What do you mean by Peep-hole optimization?
80. Construct DAG for the following exp (a+b)-(e-(c+d)).
81. Write the mnemonic code for (a+b)-(e-(c+d)).
82. What do you mean by symbol table.Explain.
83. What informations are stored in symbol table?
84. Explain about static & stack allocation.
85. What are the code optimization techniques?
86. Explain about common sub exp elimination & renaming of temp variables.

5.Lab Internal Exam question papers

Internal 1 Question Paper:

I) Write a C Program to Remove Left Factoring from the grammar.
(Consider all the productions of the form A$\alpha\beta_1/\alpha\beta_2/\alpha\beta_3$)

II) Consider the following Expression language that used to describe the Arithmetic expressions in Calculator The syntax of the language is defined by following grammar

<Line>--><exp>
<exp> ---><exp>+ <exp>| <exp>-<exp>|<exp>*<exp>|<exp>/<exp>
<exp>--><value>
<value>---><digit><value> | <digit>
<digit>----->0|1|2|3|4|5|6|7|8|9

Write a C program to Implement Recursive Decent parser. And parse the string 45+32/54*33.

III) Write a C Program to Remove Left Recursion from the grammar.
(Consider all the productions of the form A$A\alpha_1$| $A\alpha_2$|β)

IV) Write a C Program to identifying an identifier, constant and keyword. And classifying identifier as either variable or array or function and constant as integer or real. Consider the keywords for, do, while.

Internal II Question Paper:

1. Write a C Program to identifying an identifier, constant and keyword. And classifying identifier as either variable or array or function and constant as integer or real. Consider the keywords for, do, while.

2. Write a C program to implement Operator precedence parsing for the grammar.
   EE+E|E*E|E-E|E/E|d and parse the strings i) d*d-d$ ii) d/d*d$

3. Write a C Program to Remove Left Recursion from the grammar.
   (Consider all the productions of the form A$A\alpha|\beta_1|\beta_2$)

4. Consider the following Expression language that used to describe the Arithmetic expressions in Calculator The syntax of the language is defined by following grammar
   <Line>--><exp>
   <exp> ---><exp>+ <exp>| <exp>-<exp>|<exp>*<exp>|<exp>/<exp>
   <exp>--><value>
   <value>---><digit><value> | <digit>
   <digit>----->0|1|2|3|4|5|6|7|8|9

   Write a C program to Implement Recursive Decent parser. And parse the string 45+32/54*33.

5. Write a C program to Implement FIRST set of the given grammar.

6. Write a C Program to Remove Left Factoring from the grammar.
   (Consider all the productions of the form A$\alpha\beta_1/\alpha\beta_2/\alpha\beta_3$)

7. Write a C program to implement Operator precedence parsing for the grammar.
   S(L)|a
   LL,S|S and parse the strings i) (a,a)$ ii) (a,(a,a)$
   Precedence relation table is:

	a	(	)	,	$
A			>	>	>
(	<	<	=	<	
)			>	>	>
,	<	<	>	>	
$	<	<			

8. Consider the following circuit language that used to describe the circuits(parallel, sequential or both). The syntax of the language is defined by following grammar
   <circuit> --->seq<block> | par<block> | <value>
   <block>---><circuit><block>| end
   <value>---><digit><value> | <digit>
   <digit>----->0|1|2|3|4|5|6|7|8|9
   Write a C program to Implement Recursive Decent parser. And parse the string :
   par6seqpar379end