RashtreeyaSikshanaSamithi Trust
RV COLLEGE OF ENGINEERING
(Autonomous Institution Affiliated to VTU, Belagvi)

Department Of Computer Science & Engineering
Bengaluru – 560059

COMPILER DESIGN

SUB CODE: 16CS62

VI SEMESTER B.E

LAB INSTRUCTORS MANUAL

2018-2019

SOLUTIONS TO LAB PROGRAMS WITH EXPECTED INPUT AND OUTPUT

LEX PROGRAMS

Steps to compile and execute LEX programs:

- **vi program_name.l**
- **lex program_name.l**
- **cc lex.yy.c -ll**
- **./a.out**

Steps to compile and execute Yacc programs:

- **lex prg.l**
- **yacc -d prg.y**
- **cc lex.yy.c y.tab.c -ll**
- **./a.out**

1. a) Write a LEX program to count number of words, lines, whitespaces and characters.

```
%{
int l=0,w=0,s=0,c=0;
%}
%%

[.] l++;
[ ] s++;
[\t] s=s+3;
[a-zA-Z]+ {w++; c=c+yyleng;}
. ;
\n return 0;
%%

int main()
{
printf("Enter the string\n");
yylex();
printf("no. of lines=%d\n\twords=%d\n\tcharacters=%d\n\tspaces=%d\n",l,w,c,s);
}
```

----------OUTPUT----------

```
1. Enter the string
R V College Of Engineering
no. of lines=0
        words=5
```

characters=22
spaces=4

2. Enter the string
R V College of engineering. Department of CSE.
no. of lines=2
words=8
characters=37
spaces=7

1. b. Write a YACC program to recognize strings of the form $a^n b^{n+m} c^m$, n,m>=0.

Lex Program

```
%{
#include "y.tab.h"
%}
%%

"a" { return 'a';}
"b" { return 'b';}
"c" { return 'c';}
. return yytext[0];
\n return 0;
%%
```

Yacc Program

```
%{
#include<stdio.h>
#include<stdio.h>
#include<string.h>
%}
%%
S:B C
;
B:'a' B 'b'
|
;
C:'b' C 'c'
|
;
%%
```

```
int main()
{
yyparse();
printf("\n Valid string\n");
}

int yyerror()
{
printf("INVALID!!!\n");
exit(0);
}
```

----------OUTPUT----------

```
aaaabbc
INVALID!!!

aabbbc
Valid string
```

2 a) Write a LEX program to count number of Positive & negative integers and Positive & negative fractions

```
%{
#include<stdio.h>
int p=0,n=0,pf=0,nf=0;
%}
%%

[+]?[0-9]+ {p++;}
[-][0-9]+ {n++;}
[0-9]*[.][0-9]+ {pf++;}
[-][0-9]*[.][0-9]+ {nf++;}
[-][0-9]*[.]*[0-9]+[/][-][0-9]*[.]*[0-9]+ {pf++;}
[0-9]*[.]*[0-9]+[/][0-9]*[.]*[0-9]+ {pf++;}
[0-9]*[.]*[0-9]+[/][-][0-9]*[.]*[0-9]+ {nf++;}
[-][0-9]*[.]*[0-9]+[/][0-9]*[.]*[0-9]+ {nf++;}
\n return 0;
%%

int main()
{
printf("Enter the no.'s\n");
yylex();
printf("Number of positive integers=%d\n\tnegative integers=%d\n\tpositive
```

```
fractions=%d\n\tnegative fractions=%d\n",p,n,pf,nf);
}
```

----------OUTPUT----------

```
Enter the no.'s
1234
Number of positive integers=1
        negative integers=0
        positive fractions=0
        negative fractions=0

Enter the no.'s
123 -12 1.9 -6.78 0.34
   Number of positive integers=1
        negative integers=1
        positive fractions=2
        negative fractions=1
```

2. b) Write a YACC program to validate and evaluate a simple expression involving operators +,- , * and /.

LEX Program

```
%{
#include "y.tab.h"
extern yylval;
%}
%%

[0-9]+ {yylval=atoi(yytext); return NUM;}
[-] return '-';
[+] return '+';
[*] return '*';
[/] return '/';
. return yytext[0];
\n return 0;
%%
```

Yacc Program

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token NUM
```

```
%left '+' '-'
%left '/' '*'
%%

S:I {printf("result is %d\n",$$);}
;
I:I'+'I {$$=$1+$3;}
|I'-'I {$$=$1-$3;}
|I'*'I {$$=$1*$3;}
|I'/'I {if($3==0){ yyerror(); }
          else $$=$1 / $3;}
|'('I')' {$$=$2;}
|NUM
;
%%

int main()
{
yyparse();
printf("Valid\n");
//printf("result is %d\n",yylval);
}

int yyerror()
{
printf("INVALID!!!!\n");
exit(0);
}
```

----------OUTPUT----------

```
a+b-c(2+3)
INVALID!!!!

2+4*(2+1-8)
result is -18
Valid
```

3. a) Write a LEX program to count the number of comment lines in a given C program. Also eliminate them and copy that program into a separate file.

```
%{
#include<stdio.h>
int flag=0;
int c=0;
int flg=0;
%}
```

```
%%

"//".*              { if(flag==1){fprintf(yyout," ");flg--;}else{c++; fprintf(yyout," ");flg++;}}
"/*".*\n?"*/"?  { if(flg==1){fprintf(yyout," ");} else {flag++; fprintf(yyout," ");c++;}}
.*"*/"              {if(flag==1){ fprintf(yyout," "); c++;flag--;}}


%%

main()
{
        yyin= fopen("v.txt","r");
        yyout = fopen("v1.txt","w");
        yylex();
        printf("Number of comment lines=%d",c);
}
```

input file v.txt

```
//jsdhg/*fjkjhgjghj
fghgfhghg*/
//jhgdjfhgj
/*mndbfjkhk /*m,jhkdf*/ljdfghlk
*/
 /*jhgds//jhgdfjgds
hfdkjkg */
kjsdfhkjhkfjh
```

```
---------------------------------OUTPUT-----------------------------------------------------
no. of comment lines=4
```

ouput file v1.txt

```
kjsdfhkjhkfjh
```

3 b) Write a YACC program to recognize a nested (minimum 3 levels) FOR loop statement for C language.

LEX Program

```
%{
#include "y.tab.h"
%}
%%
```

```
"for" return FOR;
[(] return LPAREN;
[)] return RPAREN;
[{] return LF;
[}] return RF;
[=] return '=';
[-] return '-';
[+] return '+';
[>] return '>';
[<] return '<';
[;] return ';';
[a-zA-Z]* return EXP;
[0-9]+ return NUM;
[ ] return SPACE;
\n return 0;
%%
```

Yacc Program

```
%{
#include<stdio.h>
#include<stdlib.h>
int count=0;
%}
%token FOR LPAREN RPAREN LF RF EXP SPACE NUM
%%
S:I
;
I:FOR A B {count++;}
;
A:LPAREN E';'E';'E RPAREN
;
E:EXP Z NUM
|EXP Z EXP
|EXP U
|SPACE
;
Z:'='|'>'|'<'|'<''='|'>''='|'=''+'|'=''-'
;
U:'+''+'|'-''-'
;
B:LF B RF
|I
|EXP
|EXP SPACE I
|
;
```

```
%%
int main()
{
yyparse();
printf("no. of nested FOR's are: %d\n",count);
}

int yyerror()
{
printf("ERROR!!!\n");
exit(0);
}
```

----------OUTPUT----------

```
for(i=0;i<4;i++)
no. of nested FOR's are: 1

for(i=0;i<3;i++){ for(i=2;i<n;i++)}}
ERROR!!!

for(i=0;i<n;i++){for(j=0;jN7;j++)}
ERROR!!!

for(i=0;i<n;i++){for(j=0;j<8;j++)}
no. of nested FOR's are: 2
```

4 a) Write a LEX program to recognize and count the number of identifiers, operators and keywords in a given input file.

```
%{
#include<stdio.h>
int i=0,k=0,op=0;
%}
%%

auto|break|case|char|continue|do|default|const|double|else|enum|extern|for|if|goto|float|int|long|reg
ister|return|signed|static|sizeof|short|struct|switch|typedef|union|void|while|volatile|unsigned {  }
("/"[^\"]*"/") { k++;}
("_"|[a-z]|[A-Z])("_"|[a-z]|[A-Z]|[0-9])* {i++;}
"#include".*  ;
"#"[a-zA-Z]+.*  ;
[;] ;
```

```
[ ] ;
[,] ;
[+*%/-] {op++;}
[\n] ;

%%

void main()
{
yyin=fopen("d.c","r");
yylex();
printf("No. of identifiers=%d\n, keywords=%d,operators=%d",i,k,op);
}
```

input file d.c

```
#include<stdio.h>
#define max 10
int a,b,gfg;
float vbg;//int b;
/*int a*/
char gfhjk,kjhg;
```

----------OUTPUT----------

```
No. of identifiers=6
```

4. b) Write a YACC program to recognize nested IF control statements(C language) and display the number of levels of nesting.

Lex Program

```
%{
#include "y.tab.h"
%}
%%
"if" return IF;
"else" return ELSE;
[(] return LPAREN;
[)] return RPAREN;
[{] return LF;
[}] return RF;
[a-z]* return EXP;
[ ] return SPACE;
\n return 0;
```

%%

Yacc Program

```
%{
#include<stdio.h>
#include<stdlib.h>
int count=0;
%}
%token IF ELSE LPAREN RPAREN LF RF EXP SPACE
%%

S:I
;
I:IF E B {count++;}
;
E:LPAREN EXP RPAREN
;
B:LF B RF
|I
|EXP
|EXP SPACE I
|
;
%%
int main()
{
yyparse();
printf("no. of nested IF's are: %d\n",count);
}

int yyerror()
{
printf("ERROR!!!\n");
exit(0);
}
```

----------OUTPUT----------

```
if(abc)
no. of nested IF's are: 1

if(abc){if(abc)}
no. of nested IF's are: 2

if(ab){}
no. of nested IF's are: 1
```

if(abc){if(abc)}}
ERROR!!!

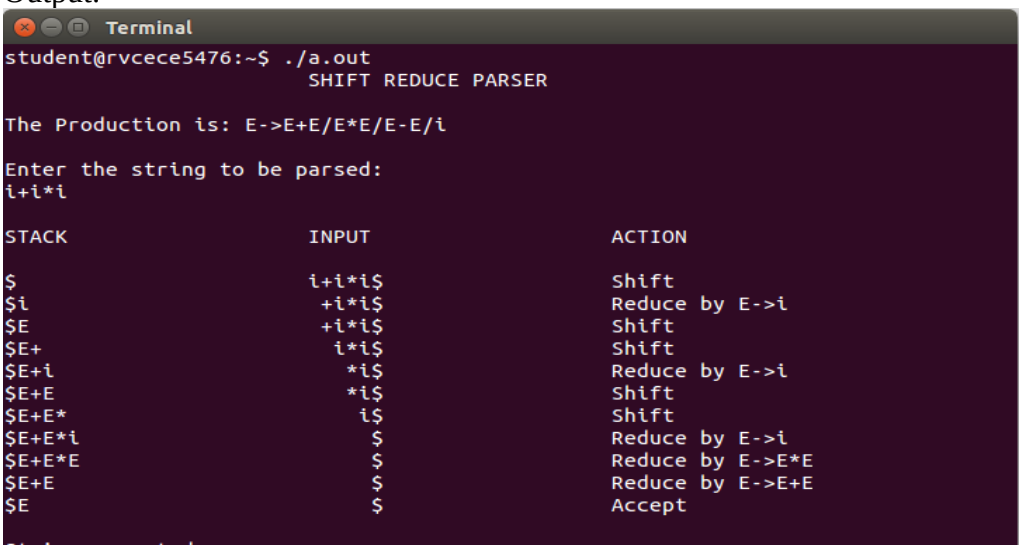
5. Write a C program to implement a Shift Reduce parser for a given grammar and generate the parsing table by parsing the given string.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char exp1[30],stack[30],arr[30],temp[30];
int i,k=0,j,l,r,s;
void push(char exp[])
{
arr[i]=exp1[k];
i++;
}
void dispinp()
{
printf("\t\t\t");
for(k=0;k<strlen(exp1);k++)
printf("%c",exp1[k]); printf("$");
}
void dispstk()
{
printf("\n");
for(k=0;k<strlen(stack);k++)
printf("%c",stack[k]);
}
void assign()
{
stack[++j]=arr[i];
exp1[i]=' ';
dispstk();
dispinp();
}
void main()
{
printf("\t\t\tSHIFT REDUCE PARSER\n");
printf("\nThe Production is: E->E+E/E*E/E-E/i\n");
printf("\nEnter the string to be parsed:\n");
gets(exp1);
printf("\nSTACK\t\t\tINPUT\t\t\tACTION\n");
printf("\n$");
dispinp();
```

```
printf("\t\t\tShift");
for(k=0;k<strlen(exp1);k++)
push(exp1);
l=strlen(exp1);
stack[0]='$';
for(i=0;i<l;i++)
{
switch(arr[i])
{
case 'i': assign();
printf("\t\t\tReduce by E->i");
stack[j]='E';
dispstk();
dispinp();
if(arr[i+1]!='\0')
printf("\t\t\tShift");
break;
case '+': assign();
printf("\t\t\tShift");
break;
case '*': assign();
printf("\t\t\tShift");
break;
case '-': assign();
printf("\t\t\tShift");
break;
default: printf("\nError:String not accepted\n");
goto label;
}
}
l=strlen(stack);
while(l>2)
{
r=0;
for(i=l-1;i>=l-3;i--)
{
temp[r]=stack[i];r++;
}
temp[r]=NULL;
if((strcmp(temp,"E+E")==0)||(strcmp(temp,"E*E")==0)||(strcmp(temp,"E-E")==0))
{
for(i=l;i>l-3;i--)
stack[i]=' ';
stack[l-3]='E';
printf("\t\t\tReduce by E->");
for(i=0;i<strlen(temp);i++)
```

```
printf("%c",temp[i]);
dispstk();
dispinp(); l=l-2;
}
else
{
printf("\nError:String not accepted\n"); goto label;
}
}
printf("\t\t\tAccept"); printf("\n\nString accepted\n");
label: exit(0);
}
```

Output:



6. YACC program that reads the C statements from an input file and converts them into quadruple three address intermediate code.

LEX FILE:

```
%{
#include"y.tab.h"
extern char yyval;
%}

NUMBER [0-9]+
LETTER [a-zA-Z]+

%%
{NUMBER} {yylval.sym=(char)yytext[0]; return NUMBER;}
{LETTER} {yylval.sym=(char)yytext[0];return LETTER;}
```

```
\n {return 0;}
. {return yytext[0];}
%%
```

YACC FILE

```
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void ThreeAddressCode();
void triple();
void qudraple();
char AddToTable(char ,char, char);
int ind=0;
char temp='A';
struct incod
{
char opd1;
char opd2;
char opr;
};
%}
%union
{
char sym;
}
%token <sym> LETTER NUMBER
%type <sym> expr
%left '-' '+'
%right '*' '/'
%%
```

```
statement: LETTER '=' expr ';' {AddToTable((char)$1,(char)$3,'=');}
| expr ';'
;
expr: expr '+' expr {$$ = AddToTable((char)$1,(char)$3,'+');}
| expr '-' expr {$$ = AddToTable((char)$1,(char)$3,'-');}
| expr '*' expr {$$ = AddToTable((char)$1,(char)$3,'*');}
| expr '/' expr {$$ = AddToTable((char)$1,(char)$3,'/');}
| '(' expr ')' {$$ = (char)$2;}
| NUMBER {$$ = (char)$1;}
| LETTER {$$ = (char)$1;}
;
%%
yyerror(char *s)
{
printf("%s",s);
exit(0);
}
struct incod code[20];
int id=0;
char AddToTable(char opd1,char opd2,char opr)
{
code[ind].opd1=opd1;
code[ind].opd2=opd2;
code[ind].opr=opr;ind++;
temp++;
return temp;
}
void ThreeAddressCode()
{
int cnt=0;
```

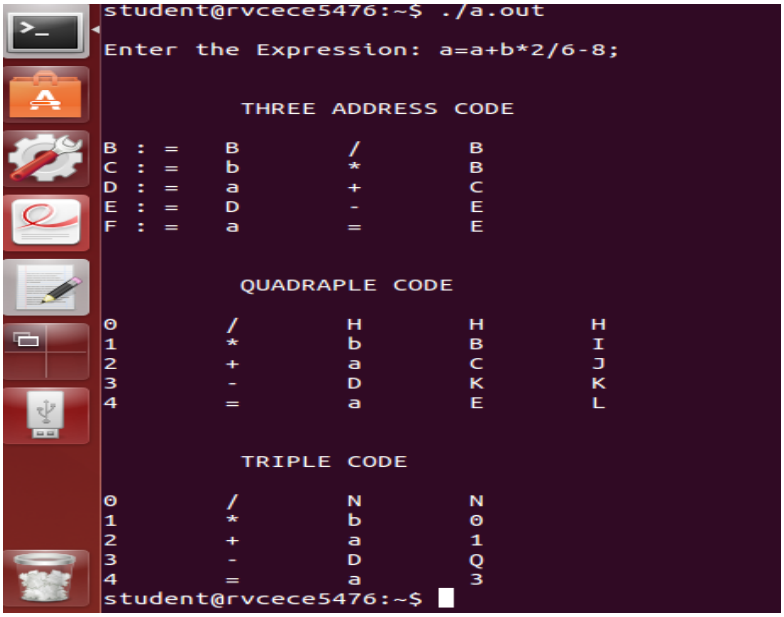
```
temp++;
printf("\n\n\t THREE ADDRESS CODE\n\n");
while(cnt<ind)
{
printf("%c : = \t",temp);
if(isalpha(code[cnt].opd1))
printf("%c\t",code[cnt].opd1);
else
{printf("%c\t",temp);}
printf("%c\t",code[cnt].opr);
if(isalpha(code[cnt].opd2))
printf("%c\t",code[cnt].opd2);
else
{printf("%c\t",temp);}
printf("\n");
cnt++;
temp++;
}
}
void quadraple()
{
int cnt=0;
temp++;
printf("\n\n\t QUADRAPLE CODE\n\n");
while(cnt<ind)
{
//printf("%c : = \t",temp);printf("%d",id);
printf("\t");
printf("%c",code[cnt].opr);
printf("\t");
```

```
if(isalpha(code[cnt].opd1))
printf("%c\t",code[cnt].opd1);
else
{printf("%c\t",temp);}
//printf("%c\t",code[cnt].opr);
if(isalpha(code[cnt].opd2))
printf("%c\t",code[cnt].opd2);
else
{printf("%c\t",temp);}
printf("%c",temp);
printf("\n");
cnt++;
temp++;
id++;
}
}
void triple()
{
int cnt=0,cnt1,id1=0;
temp++;
printf("\n\n\t TRIPLE CODE\n\n");
while(cnt<ind)
{
//printf("%c : = \t",temp);
if(id1==0)
{
printf("%d",id1);
printf("\t");
printf("%c",code[cnt].opr);
printf("\t");if(isalpha(code[cnt].opd1))
```

```
printf("%c\t",code[cnt].opd1);
else
{printf("%c\t",temp);}
//printf(“%c\t”,code[cnt].opr);
cnt1=cnt-1;
if(isalpha(code[cnt].opd2))
printf("%c",code[cnt].opd2);
else
{printf("%c\t",temp);}
}
else
{
printf("%d",id1);
printf("\t");
printf("%c",code[cnt].opr);
printf("\t");
if(isalpha(code[cnt].opd1))
printf("%c\t",code[cnt].opd1);
else
{printf("%c\t",temp);}
//printf(“%c\t”,code[cnt].opr);
cnt1=cnt-1;
if(isalpha(code[cnt].opd2))
printf("%d",id1-1);
else
{printf("%c\t",temp);}
}
printf("\n");
cnt++;
temp++;
```

```
id1++;
}
}main()
{
printf("\nEnter the Expression: ");
yyparse();
temp='A';
ThreeAddressCode();
quadraple();
triple();
}
yywrap()
{
return 1;
}
```

Output:



7. Write a YACC program that identifies Function Definition of C language.

LEX FILE:

```
alpha [a-zA-Z]
digit [0-9]

%%
[ \t]      ;
[ \n]   { yylineno = yylineno + 1;}
int return INT;
float return FLOAT;
char return CHAR;
void return VOID;
double return DOUBLE;
for     return FOR;
while   return WHILE;
if  return IF;
else    return ELSE;
printf  return PRINTF;
struct  return STRUCT;
^"#include ".+ ;
{digit}+       return NUM;
{alpha}({alpha}|{digit})* return ID;
"<="    return LE;
">="    return GE;
"=="    return EQ;
"!="    return NE;
">" return GT;
"<" return LT;
"."     return DOT;
\/\/.* ;
\/\*(.*\n)*.*\*\/ ;
```

```
.       return yytext[0];
%%
YACC FILE
%{
#include <stdio.h>
#include <stdlib.h>

extern FILE *fp;

%}

%token INT FLOAT CHAR DOUBLE VOID
%token FOR WHILE
%token IF ELSE PRINTF
%token STRUCT
%token NUM ID
%token INCLUDE
%token DOT

%right '='
%left AND OR
%left '<' '>' LE GE EQ NE LT GT
%%

start:  Function
    | Declaration
    ;

/* Declaration block */
Declaration: Type Assignment ';'
```

```
    | Assignment ';'
    | FunctionCall ';'
    | ArrayUsage ';'
    | Type ArrayUsage ';'
    | StructStmt ';'
    | error
    ;

/* Assignment block */
Assignment: ID '=' Assignment
    | ID '=' FunctionCall
    | ID '=' ArrayUsage
    | ArrayUsage '=' Assignment
    | ID ',' Assignment
    | NUM ',' Assignment
    | ID '+' Assignment
    | ID '-' Assignment
    | ID '*' Assignment
    | ID '/' Assignment
    | NUM '+' Assignment
    | NUM '-' Assignment
    | NUM '*' Assignment
    | NUM '/' Assignment
    | '\'' Assignment '\''
    | '(' Assignment ')'
    | '-' '(' Assignment ')'
    | '-' NUM
    | '-' ID
    |   NUM
    |   ID
```

```
    ;

/* Function Call Block */
FunctionCall : ID'('')'
    | ID'('Assignment')'
    ;

/* Array Usage */
ArrayUsage : ID'['Assignment']'
    ;

/* Function block */
Function: Type ID '(' ArgListOpt ')' CompoundStmt
    ;
ArgListOpt: ArgList
    |
    ;
ArgList:  ArgList ',' Arg
    | Arg
    ;
Arg:    Type ID
    ;
CompoundStmt:   '{' StmtList '}'
    ;
StmtList:   StmtList Stmt
    |
    ;
Stmt:   WhileStmt
    | Declaration
    | ForStmt
```

```
    | IfStmt
    | PrintFunc
    | ';'
    ;

/* Type Identifier block */
Type:   INT
    | FLOAT
    | CHAR
    | DOUBLE
    | VOID
    ;

/* Loop Blocks */
WhileStmt: WHILE '(' Expr ')' Stmt
    | WHILE '(' Expr ')' CompoundStmt
    ;

/* For Block */
ForStmt: FOR '(' Expr ';' Expr ';' Expr ')' Stmt
       | FOR '(' Expr ';' Expr ';' Expr ')' CompoundStmt
       | FOR '(' Expr ')' Stmt
       | FOR '(' Expr ')' CompoundStmt
    ;

/* IfStmt Block */
IfStmt : IF '(' Expr ')'
        Stmt
    ;
```

```
/* Struct Statement */
StructStmt : STRUCT ID '{' Type Assignment '}'
    ;

/* Print Function */
PrintFunc : PRINTF '(' Expr ')' ';'
    ;

/*Expression Block*/
Expr:
    | Expr LE Expr
    | Expr GE Expr
    | Expr NE Expr
    | Expr EQ Expr
    | Expr GT Expr
    | Expr LT Expr
    | Assignment
    | ArrayUsage
    ;
%%
#include"lex.yy.c"
#include<ctype.h>
int count=0;

int main(int argc, char *argv[])
{
    yyin = fopen(argv[1], "r");

   if(!yyparse())
        printf("\nParsing complete\n");
```

```
    else
        printf("\nParsing failed\n");

    fclose(yyin);
    return 0;
}

yyerror(char *s) {
    printf("%d : %s %s\n", yylineno, s, yytext );
}
```

Input:

```
int main (int c, int b) {
    int a;
    while ( 1 ) {
    int d;}}
```

Output:

Parsing Complete

8. Write a YACC program that generates Intermediate Code for valid Arithmetic Expression.

LEX FILE

```
ALPHA [A-Za-z]
DIGIT [0-9]
%%

{ALPHA}({ALPHA}|{DIGIT})* return ID;
{DIGIT}+ {yylval=atoi(yytext); return NUM;}
[\n\t] yyterminate();
. return yytext[0];
%%
```

(Yacc Program : intar.y)

```
%token ID NUM
%right '='
%left '+' '-'
%left '*' '/'
%left UMINUS
%%

S : ID{push();} '='{push();} E{codegen_assign();}
   ;
E : E '+'{push();} T{codegen();}
   | E '-'{push();} T{codegen();}
   | T
   ;
T : T '*'{push();} F{codegen();}
   | T '/'{push();} F{codegen();}
   | F
   ;
F : '(' E ')'
   | '-'{push();} F{codegen_umin();} %prec UMINUS
   | ID{push();}
   | NUM{push();}
   ;
%%

#include "lex.yy.c"
#include<ctype.h>
char st[100][10];
int top=0;
char i_[2]="0";
char temp[2]="t";

main()
 {
 printf("Enter the expression : ");
 yyparse();
 }

push()
{
  strcpy(st[++top],yytext);
 }

codegen()
 {
 strcpy(temp,"t");
```

```
 strcat(temp,i_);
  printf("%s = %s %s %s\n",temp,st[top-2],st[top-1],st[top]);
  top-=2;
 strcpy(st[top],temp);
 i_[0]++;
 }

codegen_umin()
 {
 strcpy(temp,"t");
 strcat(temp,i_);
 printf("%s = -%s\n",temp,st[top]);
 top--;
 strcpy(st[top],temp);
 i_[0]++;
 }

codegen_assign()
 {
 printf("%s = %s\n",st[top-2],st[top]);
 top-=2;
 }
```

OUTPUT:

```
Enter the expression: a=(k+8)*(c-s)
t0 = k + 8
t1 = c - s
t2 = t0 * t1
a = t2
```

9. Write a YACC program that generates Intermediate Code for FOR Loop.

// Lex file: im4.l

```
alpha [A-Za-z]
digit [0-9]

%%

[\t \n]
for            return FOR;
{digit}+    return NUM;
{alpha}({alpha}|{digit})* return ID;
"<="         return LE;
">="         return GE;
```

```
"=="         return EQ;
"!="         return NE;
"||"         return OR;
"&&"          return AND;
.            return yytext[0];

%%

// Yacc file: im4.y

%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM FOR LE GE EQ NE OR AND
%right "="
%left OR AND
%left '>' '<' LE GE EQ NE
%left '+' '-'
%left '*' '/'
%right UMINUS
%left '!'

%%

S      : FOR '(' E ';'{lab1();} E {lab2();}';' E {lab3();}')' E';'{lab4(); exit(0);}
       ;
E      : V '='{push();} E{codegen_assign();}
       | E '+'{push();} E{codegen();}
       | E '-'{push();} E{codegen();}
       | E '*'{push();} E{codegen();}
       | E '/'{push();} E{codegen();}
       | '(' E ')'
       | '-'{push();} E{codegen_umin();} %prec UMINUS
       | V
       | NUM{push();}
       ;
V      : ID {push();}
       ;

%%

#include "lex.yy.c"
#include<ctype.h>
char st[100][10];
int label[20];
```

```
int top=0;
char i_[2]="0";
char temp[2]="t";

int lno=0,ltop=0;
int start=1;

main()
{
    printf("Enter the expression:\n");
    yyparse();
}

push()
{
   strcpy(st[++top],yytext);
}

codegen()
{
    strcpy(temp,"t");
    strcat(temp,i_);
    printf("%s = %s %s %s\n",temp,st[top-2],st[top-1],st[top]);
    top-=2;
    strcpy(st[top],temp);
    i_[0]++;
}

codegen_umin()
{
    strcpy(temp,"t");
    strcat(temp,i_);
    printf("%s = -%s\n",temp,st[top]);
    top--;
    strcpy(st[top],temp);
    i_[0]++;
}

codegen_assign()
{
    printf("%s = %s\n",st[top-2],st[top]);
    top-=2;
}
lab1()
{
    printf("L%d: \n",lno++);
```

```
}
lab2()
{
   strcpy(temp,"t");
   strcat(temp,i_);
   printf("%s = not %s\n",temp,st[top]);
   printf("if %s goto L%d\n",temp,lno);
   i_[0]++;
   label[++ltop]=lno;
   lno++;
   printf("goto L%d\n",lno);
   label[++ltop]=lno;
   printf("L%d: \n",++lno);
 }
lab3()
{
   int x;
   x=label[ltop--];
   printf("goto L%d \n",start);
   printf("L%d: \n",x);

}

lab4()
{
   int x;
   x=label[ltop--];
   printf("goto L%d \n",lno);
   printf("L%d: \n",x);
}
```

Output:

```
Enter the expression:
for(i=0;i=b;i=i+1) a=a+b;
i = 0
L0:
i = b
t0 = not i
if t0 goto L1
goto L2
L3:
t1 = i + 1
i = t1
goto L0
L2:
t2 = a + b
```

```
a = t2
goto L3
L1:
```

10. Write a YACC program that accepts a regular expression as input and produces its parse tree as output.(Either left or right tree)

LEX FILE

```
%{
#include"y.tab.h"
%}

%%
[a-zA-Z] return ALPHABET;
. return yytext[0];
%%
```

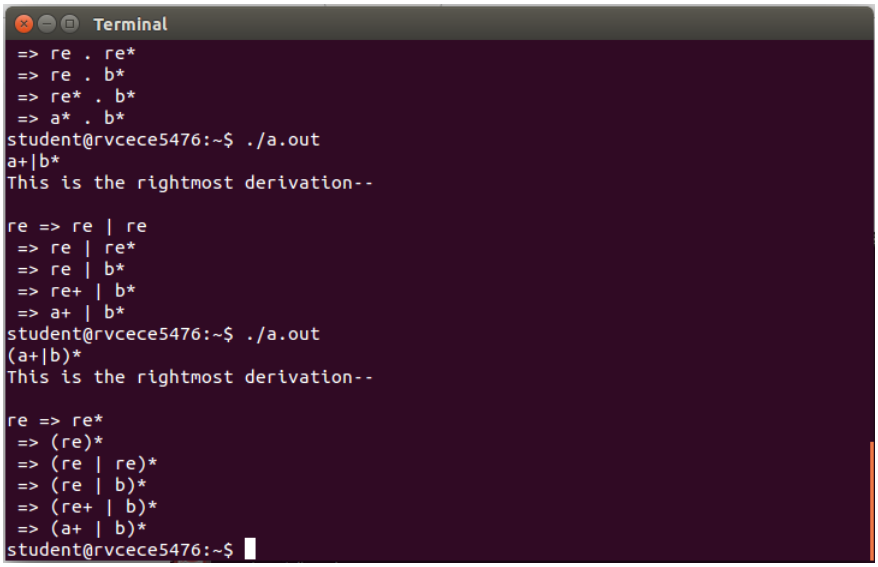
YACC FILE

```
%{/*declaration part*/
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#define MAX 100 /*to store productions*/
int getREindex ( const char* );
signed char productions[MAX][MAX];
int count = 0 , i , j;
char temp[200] , temp2[200];
%}
%token ALPHABET
%left '|'
%left '.'
%nonassoc '*' '+'
%%/*rules section*/
S : re '\n' {
printf ( "This is the rightmost derivation--\n" );
for ( i = count - 1 ; i >= 0 ; --i ) {
   if ( i == count - 1 ) {
      printf ( "\nre => " );
      strcpy ( temp , productions[i] );
      printf ( "%s" , productions[i] );
   }
   else {
      printf ( "\n => " );
```

```
        j = getREindex ( temp );
        temp[j] = '\0';
        sprintf ( temp2 , "%s%s%s" , temp , productions[i] , (temp + j + 2) );
        printf ( "%s" , temp2 );
        strcpy ( temp , temp2 );
    }
}
printf ( "\n" );
exit ( 0 );
}
re : ALPHABET {
temp[0] = yylval; temp[1] = '\0';
strcpy ( productions[count++] , temp );/*copy the input to the prodcution array*/
}/*only conditions defined here will be valid, this is the structure*/
| '(' re ')' /*adds the (expression) to the production array*/
{ strcpy ( productions[count++] , "(re)" ); }
| re '*'
{ strcpy ( productions[count++] , "re*" ); }
| re '+' /*adds expression+ type to the production array*/
{ strcpy ( productions[count++] , "re+" ); }
| re '|' re /*adds the expression|expression to the production array*/
{strcpy ( productions[count++] , "re | re" );}
| re '.' re/*adds the expression.expression to the production array*/
{strcpy ( productions[count++] , "re . re" );}
;
%%
int main ( int argc , char **argv )
{
/* Parse and output the rightmost derivation, from which we can get the parse tree*/
    yyparse();/*calls the parser*/
    return 0;
}
yylex() /*calls lex and takes each character as input and feeds ALPHABET to check for the
structure*/
{
    signed char ch = getchar();
    yylval = ch;
    if ( isalpha ( ch ) )
        return ALPHABET;
    return ch;
}
yyerror() /*Function to alert user of invalid regular expressions*/
{
    fprintf(stderr , "Invalid Regular Expression!!\n");
    exit ( 1 );
}
```

```
int getREindex ( const char *str )
{
   int i = strlen ( str ) - 1;
   for ( ; i >= 0 ; --i ) {
      if ( str[i] == 'e' && str[i-1] == 'r' )
      return i-1;
   }
}
```

OUTPUT:



VIVA QUESTIONS

1. **What is a compiler**

   A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form object code.

2. **Difference between compilers & interpreters.**
   A compiler first takes in the entire program, checks for errors, compiles it and then executes it. Whereas, an interpreter does this line by line, so it takes one line, checks it for errors and then executes it.

3. **What is a Language processor.**
   a parser which parses a particular language are called language processors

4. **What is Symbol table.**
   a symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type,scope level and sometimes its location.

5. **Explain Lexical analysis**
   This is the initial part of reading and analysing the program text: The text is read and divided into tokens, each of which corresponds to a symbol in the programming language,
   e.g., a variable name, keyword or number.

6. **What is the work of Syntax Analysis?**
   This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the syntax tree) that reflects the structure of the program. This phase is often called parsing

7. **Work of Semantic Analysis**
   Checks for errors.

8. **Work of Intermediate Code generation** - generates machine code.

9. **Work of Code optimization**- looks for ways to make code smaller and more efficient.

10. **What are Tokens, Patterns, Lexemes**
    A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
    Example of tokens:
    • Type token (id, num, real, . . .)

11. **Patterns**
    There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token. Regular expressions are an important notation for specifying patterns.
    For example, the pattern for the Pascal identifier token, id, is: id → letter (letter | digit)*.
    Lexeme

12. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
    For example, (=,),<,>

13. **Define Regular Expressions**
    The regular expressions over alphabet specifies a language according to the following rules. is a regular expression that denotes { }, that is, the set containing the empty string.

14. **What is Regular Definitions**
    A regular definition gives names to certain regular expressions and uses those names in other regular expressions.

15. **Define Deterministic Finite Automata (DFA)**
    A deterministic finite automation is a special case of a non-deterministic finite automation (NFA) in which
    1. no state has an -transition
    2. for each state s and input symbol a, there is at most one edge labeled a leaving s.

16. **Define Nondeterministic Finite Automata (NFA)**
    A nondeterministic finite automation is a mathematical model consists of
    1. a set of states S;
    2. a set of input symbol, ∑, called the input symbols alphabet.
    3. a transition function move that maps state-symbol pairs to sets of states.
    4. a state so called the initial or the start state.
    5. a set of states F called the accepting or final state.

17. **Let S and T be language over ={a,b} represented by the regular expressions (a+b*)* and (a+b)*, respectively. Which of the following is true?**
    (a) ScT (S is a subset of T)
    (b) TcS (T is a subset of S)
    (c) S=T
    (d) SnT=Ø
    Answer: (c).

18. **Let L denotes the language generated by the grammar S – OSO/00. Which of the following is true?**
    (a) L = O
    (b) L is regular but not O
    (c) L is context free but not regular
    (d) L is not context free

19. **Answer:** (b)
    Explanation: Please note that grammar itself is not regular but language L is regular as L can be represented using a regular grammar, for example S -> S00/00.

20. **Given an arbitrary non-deterministic finite automaton (NFA) with N states, the maximum number of states in an equivalent minimized DFA is at least.**
    (a) N^2
    (b) 2^N
    (c) 2N
    (d) N!
    Answer: (b)

21. **Define Synthesized Attributes:**
    An attribute is synthesized if its value at a parent node can be determined from attributes of its children.

22. **What do you mean by Syntax-Directed Definitions:**
    • A syntax-directed definition uses a CFG to specify the syntatic structure of the input.
    • A syntax-directed definition associates a set of attributes with each grammar symbol.
    • A syntax-directed definition associates a set of semantic rules with each production rule.

23. **What is Parsing**
    Parsing is the process of determining if a string of tokens can be generated by a grammar. A parser must be capable of constructing the tree, or else the translation cannot be guaranteed correct. For any language that can be described by CFG, the parsing requires O(n3) time to parse string of n token. However, most programming languages are so simple that a parser requires just O(n) time with a single left-to-right scan over the iput string of n tokens.

24. **What are the types of parsing**
    1. Top-down Parsing (start from start symbol and derive string)
    A Top-down parser builds a parse tree by starting at the root and working down towards the leaves.
    o Easy to generate by hand.
    o Examples are : Recursive-descent, Predictive.
    2. Bottom-up Parsing (start from string and reduce to start symbol)
    A bottom-up parser builds a parser tree by starting at the leaves and working up towards the root.
    o Not easy to handle by hands, usually compiler-generating software generate bottom up parser
    o But handles larger class of grammar
    o Example is LR parser.

25. **What is Predictive Parsing**:
    Recursive-descent parsing is a top-down method of syntax analysis that executes a set of recursive procedure to process the input. A procedure is associated with each nonterminal of a grammar.
    A predictive parsing is a special form of recursive-descent parsing, in which the current input token unambiguously determines the production to be applied at each step.

26. **What is Left Recursion:**
    The production is left-recursive if the leftmost symbol on the right side is the same as the non terminal on the left side. For example,
    expr → expr + term.

27. **What are the Issues in the Design of Code generator**
    Code generator concern with:
    1. Memory management.
    2. Instruction Selection.
    3. Register Utilization (Allocation).
    4. Evaluation order.

28. **What is lex**
    Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). Lex is commonly used with the yacc parser generator.

    Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.
    Lexical error:
    A lexical error is any input that can be rejected by the lexer. This generally results from token recognition falling off the end of the rules you've defined.

29. **What are the Types of parsers**
    Top-down parsers
    _ start at the root of derivation tree and _ll in
    _ picks a production and tries to match the input
    _ may require backtracking
    _ some grammars are backtrack-free (predictive)

    Eg: recursive descent, LL

    Bottom-up parsers
    _ start at the leaves and _ll in
    _ start in a state valid for legal _rst tokens
    _ as input is consumed, change state to encode possibilities
    (recognize valid pre_xes)
    _ use a stack to store both state and sentential forms

    Eg: LR, CYK (look ahead) parser , slr, lalr

30. **What does LL and LR mean**
    LL(1) means – first ‘L’ represents scanning input from left to right. Second ‘L’ represents producing leftmost derivation. (1) one input symbol of lookahead at each step.
    LR(k) means – ‘L’ Left to right scanning, R-rightmost derivation in reverse. K-0 or 1 no. of i/p symbols.

31. **Explain Code Generation**
    code generation is the process by which a compiler's code generator converts some internal representation of source code into a form (e.g., machine code) that can be readily executed by a machine

32. **Explain code optimization**
    Code optimization is the process of modifying the code to make some aspect of software or hardware work more efficiently or use fewer resources or reduce compilation time or use memory efficiently etc.

33. **Define CFG**

    CFG is a grammar which naturally generates a formal language in which clauses can be nested inside clauses arbitrarily deeply, but where grammatical structures are not allowed to overlap