Roll Number: 18WH1A1234 **Date:_______________**

Lexical Analysis

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High level input program into a sequence of Tokens.

• Lexical Analysis can be implemented with the Deterministic finite Automata.

• The output is a sequence of tokens that is sent to the parser for syntax analysis.

Token

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, number, real, . . .)

• Punctuation tokens (IF, void, return, . . .)

• Alphabetic tokens (keywords)

• Keywords; Examples-for, while, if etc.

• Identifier; Examples-Variable name, function name, etc.

• Operators; Examples '+', '++', '-' etc.

• Separators; Examples ',' ';' etc

• Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

How Lexical Analyzer functions

1. Tokenization i.e. dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error messages by providing row numbers and column numbers. **LEX**

• Lex is a program that generates lexical analyzer. It is used with a YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

• Firstly, a lexical analyzer creates a program lex.1 in the Lex language. Then the Lex compiler runs the lex.1 program and produces a C program lex.yy.c.

• Finally C compiler runs the lex.yy.c program and produces an object program a.out.

• a.out is a lexical analyzer that transforms an input stream into a sequence of tokens.

Roll Number: 18WH1A1234 **Date:______________**

Lex file format:
A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows: { definitions }
%%
{ rules }
%%
{ user subroutines }
Where
Definitions- include declarations of constant, variable and regular definitions.
Rules- define the statement of form p1 {action1} p2 {action2}....pn {action}.
Where pi describes the regular expression and action1 describes the actions (what action the lexical analyzer should take when pattern pi matches a lexeme.)
User subroutines- are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

Roll Number: 18WH1A1234 **Date:______________**

1. Write a LEX Program to scan reserved word & Identifiers of C Language.

Program:

```
%%
"if" |
"else" |
"while" |
"for" |
"do" |
"switch" |
"goto" |
"break" |
"case" |
"const" |
"float" |
"double" |
"int" |
"long" |
"short" |
"signed" |
"unsigned" |
"register" |
"typedef" |
"return" |
"enum" |
"sizeof" |
"static" |
"struct" |
"union" |
"void" |
"main" |
"continue" |
"default" |
"printf" |
"scanf" {printf("%s is a keyword\n",yytext);}
[a-zA-Z][a-zA-Z0-9_]* {printf("%s is a identifier\n",yytext);}
[0-9]+ {printf("%s is a number",yytext);}
[+/*%\-] {printf("%s is an arithmetic operator\n",yytext);}
[< >][=]? |
[!=][=] {printf("%s is a relational operator\n",yytext);}
[&][&]|[|][|]|[!] {printf("%s is a logical operator\n",yytext);}
[&][~][|][^][<<][>>] {printf("%s is a bitwise operator\n",yytext);}
[,] {printf("%s is a seperator\n",yytext);}
[;] {printf("%s is a terminator\n",yytext);}
[()] {printf("%s is a braces\n",yytext);}
[{}] {printf("%s is a paranthesis\n",yytext);}
[[]] {printf("%s is a squarebrace\n",yytext);}
"%d" |
"%c" |
"%s" |
"%f" {printf("%s is a formatspecifier\n",yytext);}
```

Roll Number: 18WH1A1234 **Date:______________**

```
[/][/][a-zA-Z]* {printf("%s is a line comment\n",yytext);}
[/][*][a-z0-9A-Z #()%&+-<>!~;".,]*[*][/] {printf("%s is a block comment\n",yytext);} ["][a-z0-9A-Z
#()%&+-<>!~;".,]*["] {printf("%s is a string\n",yytext);}
[@#$] {printf("%s is a special character\n",yytext);}
"\\n" {printf("%s is a new line character\n",yytext);}
"#include<stdio.h>" |
"#include<stdbool.h>" |
"#include<string.h>" |
"#include<math.h>" {printf("%s is a header file\n",yytext);}
%%
 main()
{
 yylex();
 }
```

OUTPUT:

compile and run:
Step-1: lex reserved.l
Step-2: gcc lex.yy.c –ll
Step-3: ./a.out
scanf
scanf is a keyword
hello
hello is a identifier
printf
printf is a keyword

Roll Number: 18WH1A1234 **Date:______________**

2. Program to count the number of vowels and consonants in a given string.

Program:

```
%{
#include<stdio.h>
int vowels=0;
int cons=0;
%}
%%
[aeiouAEIOU] {vowels++;}
[a-zA-Z] {cons++;}
%%
int yywrap()
{
return 1;
}
main()
{
printf("Enter the string.. at end press ^d\n");
yylex();
printf("No of vowels=%d\nNo of consonants=%d\n",vowels,cons);
}
```

OUTPUT:

compile and run:
Step-1: lex vowels_cons.l
Step-2: gcc lex.yy.c –ll
Step-3: ./a.out
Enter the string.. at end press ^d
ABCDEabcd
No of vowels=3
No of consonants=6

Roll Number: 18WH1A1234 **Date:_____________**

3. Lex program to count no of words that are greater than 5 and less than 10.

Program:

```
%{
 int len=0, counter=0;
%}
%%
[a-zA-Z]+ { len=strlen(yytext);
if(len<10 &&len>5){
        counter++;} }
%%
int yywrap (void )
 {
 return 1;
 }
int main()
 {
printf("Enter the string:");
yylex();
printf("\n %d", counter);
 return 0;
 }
```

OUTPUT:

compile and run:

Step-1: lex noOfWords.l

Step-2: gcc lex.yy.c –ll

Step-3: ./a.out

Enter the string:welcome to programming in lex using c

Syntax Analysis

Syntax Analysis is a second phase of the compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal grammar. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

Syntax Analysis in the Compiler Design process comes after the Lexical analysis phase. It is also known as the Parse Tree or Syntax Tree. The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyser also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program. Otherwise, it will display error messages.

The tasks perform by the parser in compiler design are

- Helps you to detect all types of Syntax errors
- Find the position at which error has occurred
- Clear & accurate description of the error.
- Recovery from an error to continue and find further errors in the code.
- Should not affect compilation of "correct" programs.
- The parse must reject invalid texts by reporting syntax errors

Parsing Techniques

Parsing techniques are divided into two different groups:

- Top-Down Parsing,
- Bottom-Up Parsing

Top-Down Parsing:

In the top-down parsing construction of the parse tree starts at the root and then proceeds towards the leaves. Two types of Top-down parsing are:

1. **Predictive Parsing:**

Predictive parser can predict which production should be used to replace the specific input string. The predictive parser uses look-ahead point, which points towards next input symbols. Backtracking is not an issue with this parsing technique. It is known as LL(1) Parser

2. **Recursive Descent Parsing:**

This parsing technique recursively parses the input to make a prase tree. It consists of several small functions, one for each nonterminal in the grammar.

Predictive Parser

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar

and accepts only a class of grammar known as LL(k) grammar.
Predictive parsing algorithm.

Input:
string ω
parsing table M for grammar G

Output:
If ω is in L(G) then left-most derivation of ω,
error otherwise.

Initial State : $S on stack (with S being start symbol)
ω$ in the input buffer

```
SET ip to point the first symbol of ω$.
repeat
 let X be the top stack symbol and a the symbol pointed by ip.
 if X Vt or $ ∈
 if X = a
 POP X and advance ip.
 else
 error()
 endif
 else /* X is non-terminal */
 if M[X,a] = X → Y1, Y2,... Yk
 POP X
 PUSH Yk, Yk-1,... Y1 /* Y1 on top */
 Output the production X → Y1, Y2,... Yk
 else
 error()
 endif
 endif
until X = $ /* empty stack */
```

Roll Number: 18WH1A1234 **Date:______________**

4. Implement Predictive parsing algorithm.

Program:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
char table[10][10][10], nter[10], ter[10];
char inp[20], stack[20];
int nut, nun, i = 0, top = 0;
int get_nter(char);
int get_ter(char);
void replace(char, char);
void main() {
 int i, j;
 printf("Enter number of Terminals:\n");
 scanf("%d", &nut);
 printf("Enter number of Non-Terminals:\n");
 scanf("%d", &nun);
 printf("Enter all Non-Terminals:\n");
 scanf("%s", nter);
 printf("Enter all Terminals:\n");
 scanf("%s", ter);
 for (i = 0; i < nut; i++)
printf("%c\t", nter[i]);
printf("\n");
for (j = 0; j < nun; j++)
 printf("%c\t", ter[j]);
 printf("\n");
 for (i = 0; i < nun; i++)
for (j = 0; j < nut; j++) {
 printf("Enter for %c and %c \n", nter[i], ter[j]);
 scanf("%s", table[i][j]);
}
 for (j = 0; j < nut; j++)
printf("\t %c", ter[j]);
 printf("\n");
 for (i = 0; i < nun; i++) {
printf("%c", nter[i]);
for (j = 0; j < nut; j++) {
 printf("\t %s", table[i][j]);
}
printf("\n");
 }
 printf("Enter the string to parse:\n");
 scanf("%s", inp);
 stack[top++] = '$';
 stack[top++] = nter[0];
 i = 0;  while(1) {
```

```
if ((stack[top - 1] == '$') && (inp[i] == '$')) {
 printf("String Accepted\n");
 return;
}
else if (!isupper(stack[top - 1])) {
 if (stack[top - 1] == inp[i]) {
i++;
top--;
 }
 else {
printf("Error not accepted\n");
return;
 }
}
else {
 replace(stack[top - 1], inp[i]);
}
}
}
int get_nter(char x) {
 int a;
 for (a = 0; a < nun; a++)
if (x == nter[a])
 return a;
 return 100;
}
int get_ter(char x) {
 int a;
 for (a = 0; a < nut; a++)
if (x == ter[a])
 return a;
 return 100;
}
void replace (char NT, char T) {
 int in1, it1, len;
 char str[10];
 in1 = get_nter(NT);
 it1 = get_ter(T);
 if ((in1 != 100) && (it1 != 100)) {
strcpy(str, table[in1][it1]);
if(strcmp(str, "#") == 0) {
printf("Error\n");
exit(0);
 }
if (strcmp(str, "@") == 0)
top--;
 else {
top--;
len = strlen(str);
len--;
```

```
do {
 stack[top++] = str[len--];
} while (len >= 0);
 }
}
else {
 printf("Not Valid\n");
}
}
```

OUTPUT:

```
Enter the no. of co-ordinates
2
Enter the productions in a grammar
S->CC
C->eC | d
First pos
FIRS[S] = ed
FIRS[C] = ed
Follow pos
FOLLOW[S] =$
FOLLOW[C] =ed$
M [S , e] =S->CC
M [S , d] =S->CC
M [C , e] =C->eC
M [C , d] =C->d
```

Bottom-up parsing

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.

Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step:** The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.
- **Reduce step:** When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

LR Parser

The LR parser is a non-recursive, shift-reduced, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- LR(1) – LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

Roll Number: 18WH1A1234 **Date:_______________**

SLR (1) Parsing

SLR (1) refers to simple LR Parsing. It is the same as LR(0) parsing. The only difference is in the parsing table.To construct SLR (1) parsing table, we use canonical collection of LR (0) items.

In the SLR (1) parsing, we place the reduce move only in the follow of the left hand side. Various steps involved in the SLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a SLR (1) parsing table

Construction of SLR parsing table –

1. Construct C = { I0, I1, ……. In}, the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow : ◦ If [A -> ?.a?] is in Ii and GOTO(Ii , a) = Ij , then set ACTION[i, a] to "shift j". Here a must be terminal. ◦ If [A -> ?.] is in Ii, then set ACTION[i, a] to "reduce A -> ?" for all a in FOLLOW(A); here A may not be S'.

   ◦ Is [S -> S.] is in Ii, then set action[i, $] to "accept". If any conflicting actions are generated by the above rules we say that the grammar is not SLR.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: if GOTO(Ii , A) = Ij then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.

SLR (1) Table Construction

The steps which use to construct SLR (1) Table is given below:

If a state (Ii) is going to some other state (Ij) on a terminal then it corresponds to a shift move in the action part. If a state (Ii) is going to some other state (Ij) on a variable then it correspond to go to move in the Go to part. If a state (Ii) contains the final item like A → ab• which has no transitions to the next state then the production is known as reduce production. For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers.

Example

S -> •Aa

A->αβ•

Follow(S) = {$}

Roll Number: 18WH1A1234 **Date:______________**

Follow (A) = {a}
SLR(1) Parsing algorithm.
ALGORITHM:
Step1: Start
Step2: Initially the parser has s0 on the stack where s0 is the initial state and w$ is in buffer Step3: Set ip point to the first symbol of w$
Step4: repeat forever, begin
Step5: Let S be the state on top of the stack and a symbol pointed to by ip
Step6: If action [S, a] =shift S then begin
Push S1 on to the top of the stack
Advance ip to next input symbol
Step7: Else if action [S, a], reduce A->B then begin
Pop 2* |B| symbols of the stack
Let S1 be the state now on the top of the stack
Step8: Output the production AB
End
Step9: else if action [S, a]=accepted, then return
Else
Error()
End
Step10: Stop

Roll Number: 18WH1A1234 **Date:______________**

5. Implement SLR(1) Parsing algorithm.

Program:
```
#include<stdio.h>
#include<string.h>
int i,j,k,m,n=0,o,p,ns=0,tn=0,rr=0,ch=0;
char read[15][10],gl[15],gr[15][10],temp,templ[15],tempr[15][10],*ptr,temp2[5],dfa[15][15];
struct states
{
 char lhs[15],rhs[15][10];
 int n;
}I[15];
int compstruct(struct states s1,struct states s2)
{
 int t;
 if(s1.n!=s2.n)
 return 0;
 if( strcmp(s1.lhs,s2.lhs)!=0 )
 return 0;
 for(t=0;t<s1.n;t++)
 if( strcmp(s1.rhs[t],s2.rhs[t])!=0 )
 return 0;
 return 1;
}
void moreprod()
{
 int r,s,t,l1=0,rr1=0;
 char *ptr1,read1[15][10];
 for(r=0;r<I[ns].n;r++)
 {
 ptr1=strchr(I[ns].rhs[l1],'.');
 t=ptr1-I[ns].rhs[l1];
 if( t+1==strlen(I[ns].rhs[l1]) )
 {
 l1++;
 continue;
 }
 temp=I[ns].rhs[l1][t+1];
 l1++;
 for(s=0;s<rr1;s++)
 if( temp==read1[s][0] )
 break;
 if(s==rr1)
 {
 read1[rr1][0]=temp;
 rr1++;
 }
 else
 continue;
 for(s=0;s<n;s++)
```

```
 {
 if(gl[s]==temp)
 {
 I[ns].rhs[I[ns].n][0]='.';
 I[ns].rhs[I[ns].n][1]=NULL;
 strcat(I[ns].rhs[I[ns].n],gr[s]);
 I[ns].lhs[I[ns].n]=gl[s];
 I[ns].lhs[I[ns].n+1]=NULL;
 I[ns].n++;
 }
 }
 }
}
void canonical(int l)
{
 int t1;
 char read1[15][10],rr1=0,*ptr1;
 for(i=0;i<I[l].n;i++)
 {
 temp2[0]='.';
 ptr1=strchr(I[l].rhs[i],'.');
 t1=ptr1-I[l].rhs[i];
 if( t1+1==strlen(I[l].rhs[i]) )
 continue;
 temp2[1]=I[l].rhs[i][t1+1];
 temp2[2]=NULL;
 for(j=0;j<rr1;j++)
 if( strcmp(temp2,read1[j])==0 )
 break;
 if(j==rr1)
 {
 strcpy(read1[rr1],temp2);
 read1[rr1][2]=NULL;
 rr1++;
 }
else
 continue;
 for(j=0;j<I[0].n;j++)
 {
 ptr=strstr(I[l].rhs[j],temp2);
 if( ptr )
 {
 templ[tn]=I[l].lhs[j];
 templ[tn+1]=NULL;
 strcpy(tempr[tn],I[l].rhs[j]);
 tn++;
 }
 }
 for(j=0;j<tn;j++)
 {
```

Roll Number: 18WH1A1234 **Date:______________**

```
ptr=strchr(tempr[j],'.');
p=ptr-tempr[j];
tempr[j][p]=tempr[j][p+1];
tempr[j][p+1]='.';
I[ns].lhs[I[ns].n]=templ[j];
I[ns].lhs[I[ns].n+1]=NULL;
strcpy(I[ns].rhs[I[ns].n],tempr[j]);
I[ns].n++;
}
moreprod();
for(j=0;j<ns;j++)
{
//if ( memcmp(&I[ns],&I[j],sizeof(struct states))==1 )
if( compstruct(I[ns],I[j])==1 )
{
I[ns].lhs[0]=NULL;
for(k=0;k<I[ns].n;k++)
I[ns].rhs[k][0]=NULL;
I[ns].n=0;
dfa[l][j]=temp2[1];
break;
}
}
if(j<ns)
{
tn=0;
for(j=0;j<15;j++)
{
templ[j]=NULL;
tempr[j][0]=NULL;
}
continue;
}
dfa[l][j]=temp2[1];
printf("\n\nI%d :",ns);
for(j=0;j<I[ns].n;j++)
printf("\n\t%c -> %s",I[ns].lhs[j],I[ns].rhs[j]);
getch();
ns++;
tn=0;
for(j=0;j<15;j++)
{
templ[j]=NULL;
tempr[j][0]=NULL;
}
}
}
void main()
{
FILE *f;
```

```
int l;
clrscr();
for(i=0;i<15;i++)
{
I[i].n=0;
I[i].lhs[0]=NULL;
I[i].rhs[0][0]=NULL;
dfa[i][0]=NULL;
}
f=fopen("tab6.txt","r");
while(!feof(f))
{
fscanf(f,"%c",&gl[n]);
fscanf(f,"%s\n",gr[n]);
n++;
}
printf("THE GRAMMAR IS AS FOLLOWS\n");
for(i=0;i<n;i++)
printf("\t\t\t\t%c -> %s\n",gl[i],gr[i]);
I[0].lhs[0]='Z';
strcpy(I[0].rhs[0],".S");
I[0].n++;
l=0;
for(i=0;i<n;i++)
{
temp=I[0].rhs[l][1];
l++;
for(j=0;j<rr;j++)
if( temp==read[j][0] )
break;
if(j==rr)
{
read[rr][0]=temp;
rr++;
}
else
continue;
for(j=0;j<n;j++)
{
if(gl[j]==temp)
{
I[0].rhs[I[0].n][0]='.';
strcat(I[0].rhs[I[0].n],gr[j]);
I[0].lhs[I[0].n]=gl[j];
I[0].n++;
}
}
}
ns++;
```

```
 printf("\nI%d :\n",ns-1);
 for(i=0;i<I[0].n;i++)
 printf("\t%c -> %s\n",I[0].lhs[i],I[0].rhs[i]);
 for(l=0;l<ns;l++)
 canonical(l);
 printf("\n\n\t\tPRESS ANY KEY FOR DFA TABLE");
 getch();
 clrscr();
 printf("\t\t\tDFA TABLE IS AS FOLLOWS\n\n\n");
 for(i=0;i<ns;i++)
 {
 printf("I%d : ",i);
 for(j=0;j<ns;j++)
 if(dfa[i][j]!='\0')
 printf("'%c'->I%d | ",dfa[i][j],j);
 printf("\n\n\n");
 }
 printf("\n\n\n\t\tPRESS ANY KEY TO EXIT");
 getch();
}
```

OUTPUT:

Input File For SLR Parser:

S S+T
S T
T T*F
T F
F (S)
F t

Roll Number: 18WH1A1234 **Date:______________**

Introduction to YACC

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers. YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:

YACC input file is divided into three parts.

```
/* definitions */
....
%%
/* rules */
....
%%
/* auxiliary routines */
....
```

Input File: Definition Part:

The definition part includes information about the tokens used in the syntax

definition: %token NUMBER

%token ID

Yacc automatically assigns numbers for tokens, but it can be overridden by

%token NUMBER 621

• Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should not overlap ASCII codes.

- The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column.
- It can also include the specification of the starting symbol in the grammar:

%start nonterminal

Input File: Rule Part:

- The rules part contains grammar definition in a modified BNF form.

• Actions is C code in { } and can be embedded inside (Translation schemes).
Input File: Auxiliary Routines Part:
• The auxiliary routines part is only C code.
• It includes function definitions for every function needed in rules part.
• It can also contain the main() function definition if the parser is going to be run as a program. • The main() function must call the function yyparse().
Input File:
• If yylex() is not defined in the auxiliary routines sections, then it should be included: #include "lex.yy.c"
• YACC input file generally finishes with:
.y
Output Files:
• The output of YACC is a file named y.tab.c
• If it contains the main() definition, it must be compiled to be executable.
• Otherwise, the code can be an external function definition for the function int yyparse() • If called with the –d option in the command line, Yacc produces as output a header file y.tab.h with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file). • If called with the –v option, Yacc produces as output a file y.output containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts. LALR Parser:
Construction of CLR Parsing table:
Input – augmented grammar G'
1. Construct C = { I0, I1, ……. In} , the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii. The parsing actions for state i are determined as follow : i) If [A -> ?.a?, b] is in Ii and GOTO(Ii , a) = Ij, then set ACTION[i, a] to "shift j". Here a must be terminal. ii) If [A -> ?. , a] is in Ii , A ≠ S, then set ACTION[i, a] to "reduce A -> ?".
iii) Is [S -> S. , $] is in Ii, then set action[i, $] to "accept".
If any conflicting actions are generated by the above rules we say that the grammar is not CLR.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: if GOTO(Ii, A) = Ij then GOTO [i, A] = j.
4. All entries not defined by rules 2 and 3 are made error.
Note – if a state has two reductions and both have same lookahead then it will in multiple entries in parsing table thus a conflict. If a state has one reduction and their is a shift from that state on a terminal same as the lookahead of the reduction then it will lead to multiple entries in parsing table thus a conflict.
LALR Parser:
LALR parser are same as CLR parser with one difference. In CLR parser if two states differ only in lookahead then we combine those states in LALR parser. After minimization if the parsing table has no conflict that the grammar is LALR also.
LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.
In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items
LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Roll Number: 18WH1A1234 **Date:______________**

6. Design LALR bottom up parser for the given language

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;
```

Roll Number: 18WH1A1234 **Date:______________**

```
}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
;
```

Roll Number: 18WH1A1234 **Date:______________**

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

Roll Number: 18WH1A1234 **Date:_______________**

7.Lex, Yacc Program to evaluate an arithmetic expression involving operating +, -,* and

Program: Sample.l-

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+ {
 yylval=atoi(yytext);
 return NUMBER;
 }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

Sample.y-

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

%}
%token NUMBER

%left '+' '-'

%left '*' '/' '%'
```

Roll Number: 18WH1A1234 **Date:______________**

```
%left '(' ')'
%%
ArithmeticExpression: E{
 printf("\nResult=%d\n",$$);
 return 0;
 }
E:E'+'E {$$=$1+$3;}
 |E'-'E {$$=$1-$3;}
 |E'*'E {$$=$1*$3;}
 |E'/'E {$$=$1/$3;}
 |E'%'E {$$=$1%$3;}
 |'('E')' {$$=$2;}
 | NUMBER {$$=$1;}
;
%%
void main()
{
 printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");
 yyparse();
 if(flag==0)
 printf("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror()
{
 printf("\nEntered arithmetic expression is Invalid\n\n");
 flag=1;
}
```

OUTPUT:

```
itlab1@itlab1-ThinkCentre-E73:~/pcclab$ flex sample.l
itlab1@itlab1-ThinkCentre-E73:~/pcclab$ yacc -d sample.y
itlab1@itlab1-ThinkCentre-E73:~/pcclab$ ls
lex.yy.c sample.l sample.y y.tab.c y.tab.h
itlab1@itlab1-ThinkCentre-E73:~/pcclab$ cc lex.yy.c y.tab.c -ll
itlab1@itlab1-ThinkCentre-E73:~/pcclab$ ls
a.out lex.yy.c sample.l sample.y y.tab.c y.tab.h
itlab1@itlab1-ThinkCentre-E73:~/pcclab$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
1+2
Result=3
Entered arithmetic expression is Valid
```

Roll Number: 18WH1A1234 **Date:______________**

8.Yacc Program to evaluate an arithmetic expression involving operating +, -,* and /.

Program:

```
%{
#include<ctype.h>
%}
%token DIGIT
%%
line : expr'\n'{printf("\n%d\n",$1);}
;
exp : expr'+' term{$$ =$1+$3;}
|term
;
term : term '*'factor{$$=$1 *$3;}
|factor
;
factor : '('expr')'{$$=$2;}
| DIGIT
;
%%
yylex()
{
 int c;
 c=getchar();
 if(isdigit(c))
  {
yylval=c-'0';
return DIGIT;
  }
 return c;
}
main()
{
 yyparse();
 return 0;
}
yyerror(const char *msg)
{
 printf("\n%s\n", msg);
}
```

Output:

```
[root@localhost]# lex codegen.l
[root@localhost]# yacc –d codegen.y
[root@localhost]# cc lex.yy.c y.tab.c –ll -lm
[root@localhost]# ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison,
Modulus and Round brackets:
1+2
Result=3
```

Roll Number: 18WH1A1234 **Date:_______________**

Three Address Code

Three Address Code is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

General representation –

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

Implementation of Three Address Code –

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple –

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

• Easy to rearrange code for global optimization.
• One can quickly access value of temporary variables using symbol table.

Disadvantage –

• Contain lot of temporaries.
• Temporary variable creation increases time and space complexity.

Example – Consider expression a = b * – c + b * – c.

The three address code is:

t1 = uminus c
t2 = b * t1
t3 = uminus c
t4 = b * t3
t5 = t2 + t4 a = t5

2. Triples –

This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantages:

• Temporaries are implicit and difficult to rearrange code.
• It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

3. Indirect Triples –

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example: Consider expression a = b * – c + b * – c

Roll Number: 18WH1A1234 **Date:______________**

9. Write a C program to generate three address code.

Program:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void pm();
void plus();
void division();
int i,ch,j,l,addr=100;
char ex[10], exp[10], exp1[10], exp2[10], id1[5],op[5],id2[5];
void main()
{
while(1)
{
printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the expression with assignment operator:");
scanf("%s",exp);
l=strlen(exp);
exp2[0]='\0';
i=0;
while(exp[i]!='=')
{
i++;
}
strncat(exp2,exp,i);
strrev(exp);
exp1[0]='\0'
strncat(exp1,exp,l-(i+1));
strrev(exp1);
printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
break;
case 2:
printf("\nEnter the expression with arithmetic operator:");
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';
for(i=0;i<l;i++)
{
if(exp[i]=='+'||exp[i]=='-')
{
if(exp[i+2]=='/'||exp[i+2]=='*')
{
pm();
break;
```

```
}
else
{
plus();
break;
}
}
else if(exp[i]=='/'||exp[i]=='*')
{
division();
break;
}
}
break;
case 3:
printf("Enter the expression with relational operator");
scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||
(strcmp(op,">=")==0)||(strcmp(op,"==")==0)||(strcmp(op,"!=")==0))==0)
printf("Expression is error");
else
{
printf("\n%d\tif %s%s%sgoto %d",addr,id1,op,id2,addr+3);
addr++;
printf("\n%d\t T:=0",addr);
addr++;
printf("\n%d\t goto %d",addr,addr+2);
addr++;
printf("\n%d\t T:=1",addr);
}
break;
case 4:
exit(0);
}
}
}
void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
printf("Three address code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]); }
```

Roll Number: 18WH1A1234 **Date:______________**

```
void division()
{
strncat(exp1,exp,i+2);
printf("Three addresscode:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]); }
void plus()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]); }
```

OUTPUT:

```
1. assignment
2. arithmetic
3. relational
4. Exit
Enter the choice:1
Enter the expression with assignment operator:
a=b
Three address code:
temp=b
a=temp
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a+b-c
Three address code:
temp=a+b
temp1=temp-c
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a-b/c
Three address code:
temp=b/c
temp1=a-temp
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:
a*b-c
Three address code:
temp=a*b
temp1=temp-c
1.assignment
```

```
2.arithmetic
3.relational
4.Exit
Enter the choice:2
Enter the expression with arithmetic operator:a/b*c
Three address code:
temp=a/b
temp1=temp*c
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:3
Enter the expression with relational operator
a
<=
b
100 if a<=b goto 103
101 T:=0
102 goto 104
103 T:=1
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice:4
```

Roll Number: 18WH1A1234 **Date:______________**

Code Generator

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:

Consider the three address statement x:= y + z. It can have the following sequence of

codes: MOV x, R0

ADD y, R0

Register and Address Descriptors:

• A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.

• An address descriptor is used to store the location where the current value of the name can be found at run time. A code-generation algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form a:= b op c perform the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y is currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.
4. If the current value of y or z has no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

Roll Number: 18WH1A1234 **Date:______________**

10. C Program to generate machine code

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#define MAX 50
void push(char);
char pop(void);
int top=-1;
char stack[MAX];
void main()
{
 int i;
 char str[20],a,b,c;
 clrscr();
 printf("\n\t\t *** CODE GENERATION ***\n\n\t for single register microprocessor");  printf("\n\n\n\n
ENTER THE POSTFIX EXPRESSION:");
 gets(str);
 printf("\n");
 for(i=0;str[i]!='\0';i++)
 {
 if(isalpha(str[i]))
 push(str[i]);
 else
 {
 b=pop();
 a=pop();
 switch(str[i])
 {
 case '+': printf("\n\n\t\t LOAD %c, R",a);
 printf("\n\n\t\t ADD %c, R",b);
 printf("\n\n\t\t MOV temp(R), %c",a);
 break;
 case '-': printf("\n\n\t\t LOAD %c, R",a);
 printf("\n\n\t\t SUB %c, R",b);
 printf("\n\n\t\t MOV temp(R), %c",a);
 break;
 case '*': printf("\n\n\t\t LOAD %c, R",a);
 printf("\n\n\t\t MUL %c, R",b);
 printf("\n\n\t\t MOV temp(R), %c",a);
 break;
 case '/': printf("\n\n\t\t LOAD %c, R",a);
 printf("\n\n\t\t DIV %c, R",b);
 printf("\n\n\t\t MOV temp(R), %c",a);
 break;
 }
 push(a);
 }
 }
```

```
 getch();
}
void push(char item)// push unction
{
 if(top==MAX)
 printf("\n\n STACK OVERFLOW");
 else
 {
 top=top+1;
 stack[top]=item;
 }
}
char pop(void)// pop function
{
 char item;
 if(top==-1)
 {
 printf("\n STACK UNDERFLOW");
 }
 else
 {
 item=stack[top];
 top--;
 }
 return item;
}
```

OUTPUT:

```
*** CODE GENERATION ***
for single register microprocessor
ENTER THE POSTFIX EXPRESSION:xyz*+
LOAD y, R
MUL x, R
MOV temp(R), y
LOAD x, R
ADD y, R
MOV temp(R), x
```