MALLA REDDY ENGINEERING COLLEGE (AUTONOMOUS)

(An UGC Autonomous Institution, Affiliated to JNTUH, Accredited 2nd time by NAAC with 'A' Grade & NBA and Recipient of World Bank Assistance under TEQIP--II S.C. 1.1)

Maisammaguda (H), Medchal-Malkajgiri District, Telangana State – 500100

Department of Computer Science and Engineering

80518 & COMPILER DESIGN LAB

III YEAR & II SEMESTER

MR18 REGULATIONS

2018-19 Onwards (MR-18)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. V Semester		
Code: 80518	COMPILER DESIGN LAB	L	T	P
Credits: 2		-	1	2

Prerequisites: NIL

Course Objectives:

This course outlines the major concept areas of language translation and various phases of compiler, extend the knowledge of parser by parsing LL parser and LR parser, analyze the intermediate forms and the role of symbol table, classify code optimization techniques and analyze the data flow and develop machine code generation algorithms.

Software Requirements: C++ Compiler / JDK kit, (LEX, YACC) / UBUNTU

List of Programs:

Consider the following mini Language, a simple procedural high-level language, only operating on integer data, with a syntax looking vaguely like a simple C crossed with Pascal.

The syntax of the language is defined by the following BNF grammar:

<program> ::= <block>
<block> ::= { <variabledefinition> <slist> } | { <slist> }
<variabledefinition> ::= int<vardeflist>;
<vardeflist> ::= <vardec> | <vardec>, <vardeflist>
<vardec> ::= <identifier> | <identifier> [<constant>]
<slist> ::= <statement> | <statement>; <slist>
<statement> ::= <assignment> | <ifstatement> | <whilestatement> | <block> |
<printstatement> | <empty>
<assignment> ::= <identifier> = <expression> | <identifier> [<expression>] =
<expression>
<ifstatement> ::= <bexpression> then <slist> else <slist> endif | if <bexpression> then
<slist> endif
<whilestatement> ::= while <bexpression> do <slist> enddo
<printstatement> ::= print (<expression>)
<expression> ::= <expression> <additionop> <term> | <term> | addingop> <term>
<bexpression> ::= <expression> <relop> <expression>
<relop> ::= < | <= | == | >= | > | !=
<addingop> ::= + | -
<term> ::= <term> <mulitop> <factor> | <factor>
<multop> ::= * | /
<factor> ::= <constant> | <identifier> | <identifier> [<expression>] | (<expression>
)
<constant> ::= <digit> | <digit> <constant>
<identifier> ::= <identifier> <letterordigit> | <letter>
<letterordigit> ::= <letter> | <digit>
<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<empty> has the obvious meaning

Comments (zero or more characters enclosed between the standard C / Java style comment brackets /*...*/) can be inserted. The language has rudimentary support for 1-dimensional arrays. The declaration
int a[3] declares an array of three elements, referenced as a[0], a[1] and a[2] Note also that you should worry about the scoping of names.
A simple program written in this language is:

```
{
  int a[3], t1, t2;
  t1 = 2;
  a[0] = 1; a[1] = 2; a[t1] = 3;
  t2 = -(a[2] + t1 * 6)/ a[2] - t1);
  if t2 > 5 then
  print(t2);
  else
  {
    int t3;
    t3 = 99;
    t2 = -25;
    print(-t1 + t2 * t3); /* this is a comment on 2 lines */
  }
  endif
}
```

1. Design a Lexical analyzer for the above language. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.
2. Implement the lexical analyzer using JLex, flex or lex or other lexical analyzer generating tools.
3. Write a C program to recognize strings under 'a*', 'a*b+', 'abb'
4. Write a C program to test whether a given identifier is valid or not
5. Write a Program for Implementation of recursive descent Parser.
6. Design Predictive parser for the given language.
7. Write a program to calculate first function for the given grammar.
8. Write a Program for Implementation of Shift Reduce parsing
9. Write a program to Design predictive parser or LL(1) parser for the given grammar.
10. Design LALR bottom up parser for the above language.
11. Convert the BNF rules into Yacc form and write code to generate abstract syntax tree.
12. Write program to generate machine code from the abstract syntax tree generated by the parser.

The following instruction set may be considered as target code.
The following is a simple register-based machine, supporting a total of 17 instructions. It has three distinct internal storage areas. The first is the set of 8 registers, used by the individual instructions as detailed below, the second is an area used for the storage of variables and the third is an area used for the storage of program. The instructions can be preceded by a label. This consists of an integer in

the range 1 to 9999 and the label is followed by a colon to seperate it from the rest of the instruction. The numerical label can be used as the argument to a jump instruction, as detailed below.
In the description of the individual instructions below, instruction argument types are specified as follows:
R specifies a register in the form R0, R1, R2, R3, R4, R5, R6 or R7 (or r0, r1, etc). L specifies a numerical label (in the garbage 1 tp 9999).
V specifies a "variable location" (a variable number, or a variable location pointed to by a register - see below).
A specifies a constant value, a variable location, a register or a variable location pointed to by a register (an indirect address). Constant values are specified as an integer value, optionally preceded by a minus sign, preceded by a # symbol. An indirect address is specified by an @ followed by a register.
So, for example an A-type argument could have the form 4 (variable number 4), #4 (the constant value 4), r4 (register 4) or @r4 (the contents of register 4 identifies the variable location to be accessed).
The instruction set is defined as follows:
LOAD A, R
loads the integer value specified by A into register R. STORE R,
V

stores the value in register R to variable V. OUT
R
outputs the value in register R. NEG
R
negates the value in register R. ADD
A, R
adds the value specified by A to register R, leaving the result in register R. SUB
A, R
subtracts the value specified by A from register R, leaving the result in register R. MUL
A, R
multiplies the value specified by A by register R, leaving the result in register R. DIV
A, R
divides register R by the value specified by A, leaving the result in register R. JMP
L
causes an unconditional jump to the instruction with the label L. JEQ
R, L
jumps to the instruction with the label L if the value in register R is zero. JNE
R, L
jumps to the instruction with the label L if the value in register R is not zero. JGE
R, L
jumps to the instruction with the label L if the value in register R is greater than or equal to zero.
JGT R, L
jumps to the instruction with the label L if the value in register R is greater than zero.
JLE R, L
jumps to the instruction with the label L if the value in register R is less than or equal to zero.
JLT R, L
jumps to the instruction with the label L if the value in register R is less than zero.
NOP
is an instruction with no effect. It can be tagged by a label. STOP
stops execution of the machine. All programs should terminate by executing a STOP instruction.

TEXT BOOKS:

1. A.V. Aho .J.D.Ullman ,"**Principles of compiler design**" ,Pearson Education.
2. Andrew N. Appel, "**Modern Compiler Implementation in C**", Cambridge University Press.

3. D.M Dhamdhere, "**Systems programming and operating systems**" ,2nd edition,tata McGraw hill publishing comp pvtLtd.

REFERENCES:

1. John R. Levine, Tony Mason, Doug Brown," **Lex&yacc**", O'reilly
2. Dick Grune, Henry E. Bal, Cariel T. H. Jacobs," **Modern Compiler Design**", Wiley dreamtech.
3. Cooper & Linda," **Engineering a Compiler**", Elsevier.
4. Louden," **Compiler Construction**", Thomson.

Course Outcomes:

At the end of the course, students will be able to

1. **Apply** the knowledge of lex tool & yacc tool to develop a scanner & parser.
2. **Develop** program for solving parser problems.
3. **Create** program for intermediate code generation.
4. **Write** code to generate abstract syntax tree and to convert BNF to YACC.
5. **Implement** target code from the abstract syntax tree.

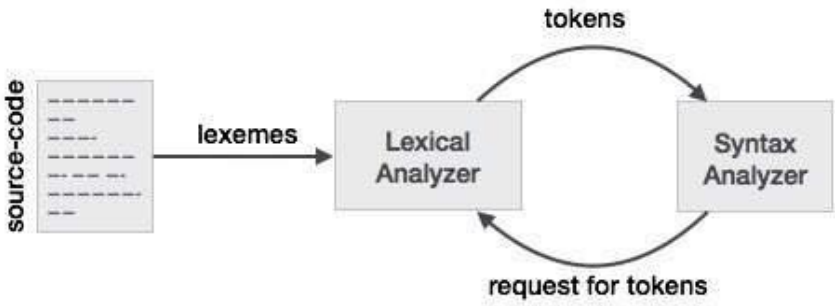
CO- PO, PSO Mapping
(3/2/1 indicates strength of correlation) 3-Strong, 2-Medium, 1-Weak

COs	Programme Outcomes(POs)												PSOs		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2										2		2	
CO2	3	2										2		2	
CO3	2	2										2		2	
CO4	3	2						2				2		2	
CO5	3	2										2		2	

Compiler Design

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



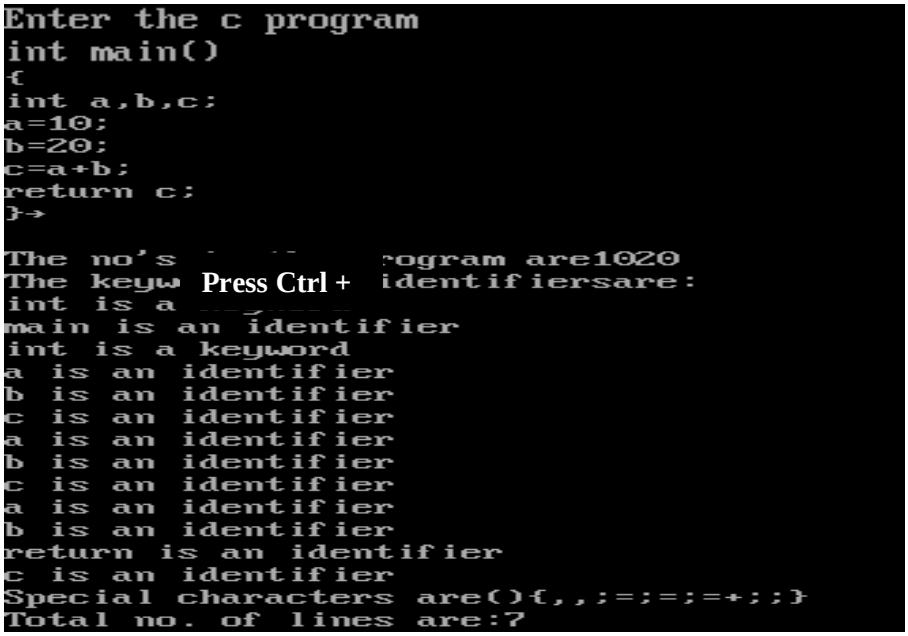
1) A Program to Design Lexical Analyzer .

```
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void keyword(char str[10])
{
if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||
strcmp("double",str)==0||strcmp("static",str)==0||strcmp("switch",str)==0||
strcmp("case",str)==0)
printf("\n%s is a keyword",str);
else
printf("\n%s is an identifier",str);
}
main()
{
FILE *f1,*f2,*f3;
char c,str[10],st1[10];
int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
printf("\nEnter the c program");/*gets(st1);*/
f1=fopen("input","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF)
```

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

```
{
str[k]='\0';
keyword(str);
k=0;
}
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\nSpecial characters are");
while((c=getc(f3))!=EOF)
printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d",lineno);
}
```

Output:



Lexical Analyzer Using Lex Tool :The lex is used in the manner depicted.A specification of the lexical analyzer is preferred by creating a program lex.l in the lex language.

Then lex,l is run through the lex compiler to produce a ‘c’program lex.yy.c·The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.l together with a standard routine that uses the table of recognize leximes. Lex.yy.c is run through the ‘c’ compiler to produce as object program a.out,which is the lexical analyzer that transform as input stream into sequence of tokens.Creating a lexical analyzer with lex is shown in below.

2)Implement the Lexical Analyzer Using Lex Tool.

```
/* program name is lexp.l */
%{
/* program to recognize a c program */
int COMMENT=0;
int cnt=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {printf("\n\t%s is a KEYWORD",yytext);}
"/*" {COMMENT = 1;}
"*/" {COMMENT = 0; cnt++;}

{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n BLOCK ENDS");}
```

```
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}
\)(\;)? {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc,char **argv)
{
if (argc > 1)
{
FILE *file;
file = fopen(argv[1],"r");
if(!file)
{
printf("could not open %s \n",argv[1]);
exit(0);
}
yyin = file;
}
yylex();
printf("\n\n Total No.Of comments are %d",cnt);
return 0;
}
int yywrap()
{
return 1;
}
```

Execution process:

```
lex lex_prog.l
cc lex_prog.yy.c
```

Input:

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

Output:

```
#include<stdio.h>

#include<stdio.h> is a PREPROCESSOR DIRECTIVE
main(){


FUNCTION
        main(
        )

 BLOCK BEGINS
int a,b;}

        int is a KEYWORD
 a IDENTIFIER,
 b IDENTIFIER;
 BLOCK ENDS
/* program ends */

→


 Total No.Of comments are 1
```

Recognizer: Means the first character inputted must be letter 'a' and the next character can have zero or more, the asterisk (*) symbol indicates the Kleene/ Star Closure, + symbol indicates that Positive Closure

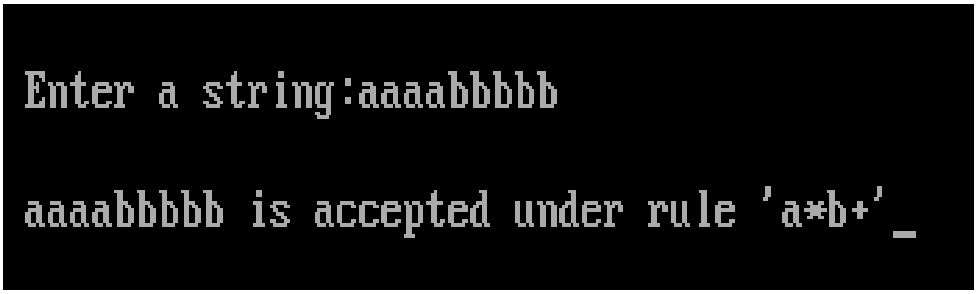
3.Write a C program to recognize strings under 'a*', 'a*b+', 'abb'.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
Void main()
{
char s[20],c;
int state=0,i=0; clrscr();
printf("\n Enter a string:");
gets(s);
while(s[i]!='\0')
{
switch(state)
{
case 0:
c=s[i++];
if(c=='a')
state=1;
else if(c=='b')
state=2;
else
state=6;
break; case 1: c=s[i++];
if(c=='a')
state=3;
else if(c=='b')
state=4;
else
state=6;
break; case 2: c=s[i++];
if(c=='a')
state=6; else if(c=='b')
state=2;
else
state=6;
break;
case 3: c=s[i++];
if(c=='a')
state=3; else if(c=='b')
state=2;
else
state=6;
break; case 4: c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
```

```
state=5;
else
state=6;
break; case 5: c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
state=2;
else
state=6; break;
case 6: printf("\n %s is not recognised.",s); exit(0);
}
}
If(state==1)
printf("\n %s is accepted under rule 'a'",s); else if((state==2)||(state==4))
printf("\n %s is accepted under rule 'a*b+'",s); else if(state==5)
printf("\n %s is accepted under rule 'abb'",s);
getch();
}
```

OUTPUT:

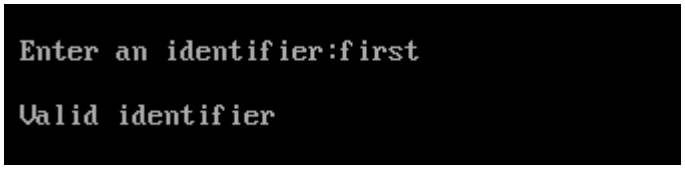


4. Write a C program to test whether a given identifier is valid or not

The first letter must be alphabet(both capital or small i.e. A-Z,a-z) or underscore(_), After first letter it contains sequence of alphabet or digits(0-9) or underscore(_) , and but not contain any special symbol(#,$,%,^,& etc.) and space( ).

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h> void main()
{
char a[10]; int flag, i=1; clrscr();
printf("\n Enter an identifier:");
gets(a);
if(isalpha(a[0]))
flag=1;
else
printf("\n Not a valid identifier");
while(a[i]!='\0')
{
if(!isdigit(a[i])&&!isalpha(a[i]))
{
flag=0; break;
} i++;
}
if(flag==1)
printf("\n Valid identifier");
getch();
}
```

OUTPUT:



Recursive Descent Parser: Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature

5. Write a Program for Implementation of recursive descent Parser.

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
#define SIZE 128
#define NONE -1
#define EOS '\0'
#define NUM 257
#define KEYWORD 258
#define ID 259
#define DONE 260
#define MAX 999
char lexemes[MAX];
char buffer[SIZE];
int lastchar=-1;
int lastentry=0;
int tokenval=DONE;
int lineno=1;
int lookahead;
struct entry
{
char *lexptr;
int token;
}symtable[100];
struct entry
keywords[]={"if",KEYWORD,"else",KEYWORD,"for",KEYWORD,"int",KEYWORD,
"float",KEYWORD,"double",KEYWORD,"char",KEYWORD,"struct",KEYWORD,"ret
urn",KEYWORD,0,0};
void Error_Message(char *m)
{
fprintf(stderr,"line %d, %s \n",lineno,m);
exit(1);
}i
nt look_up(char s[ ])
{
int k;
for(k=lastentry;k>0;k--)
```

```
if(strcmp(symtable[k].lexptr,s)==0)
return k;
return 0;
}i
nt insert(char s[ ],int tok)
{
int len;
len=strlen(s);
if(lastentry+1>=MAX)
Error_Message("Symbpl table is full");
if(lastchar+len+1>=MAX)
Error_Message("Lexemes array is full");
lastentry=lastentry+1;
symtable[lastentry].token=tok;
symtable[lastentry].lexptr=&lexemes[lastchar+1];
lastchar=lastchar+len+1;
strcpy(symtable[lastentry].lexptr,s);
return lastentry;
}/
*void Initialize()
{
struct entry *ptr;
for(ptr=keywords;ptr->token;ptr+1)
insert(ptr->lexptr,ptr->token);
}*/
int lexer()
{
int t;
int val,i=0;
while(1)
{
t=getchar();
if(t==' '||t=='\t');
else if(t=='\n')
lineno=lineno+1;
else if(isdigit(t))
{
ungetc(t,stdin);
scanf("%d",&tokenval);
return NUM;
}
else if(isalpha(t))
{
while(isalnum(t))
{
buffer[i]=t;
t=getchar();
```

```
i=i+1;
if(i>=SIZE)
Error_Message("Compiler error");
}
buffer[i]=EOS;
if(t!=EOF)
ungetc(t,stdin);
val=look_up(buffer);
if(val==0)
val=insert(buffer,ID);
tokenval=val;
return symtable[val].token;
}
else if(t==EOF)
return DONE;
else
{
tokenval=NONE;
return t;
}
}
}
void Match(int t)
{
if(lookahead==t)
lookahead=lexer();
else
Error_Message("Syntax error");
}
void display(int t,int tval)
{
if(t=='+'||t=='-'||t=='*'||t=='/')
printf("\nArithmetic Operator: %c",t);
else if(t==NUM)
printf("\n Number: %d",tval);
else if(t==ID)
printf("\n Identifier: %s",symtable[tval].lexptr);
else
printf("\n Token %d tokenval %d",t,tokenval);
}
void F()
{
//void E();
switch(lookahead)
{
case '(' : Match('(');
E();
```

```
Match(')');
break;
case NUM : display(NUM,tokenval);
Match(NUM);
break;
case ID : display(ID,tokenval);
Match(ID);
break;
default : Error_Message("Syntax error");
}
}
void T()
{
int t;
F();
while(1)
{
switch(lookahead)
{
case '*' : t=lookahead;
Match(lookahead);
F();
display(t,NONE);
continue;
case '/' : t=lookahead;
Match(lookahead);
display(t,NONE);
continue;
default : return;
}
}
}
E()
{
int t;
T();
while(1)
{
switch(lookahead)
{
case '+' : t=lookahead;
Match(lookahead);
T();
display(t,NONE);
continue;
case '-' : t=lookahead;
Match(lookahead);
```

```
T();
display(t,NONE);
continue;
default : return;
}
}
}
void parser()
{
lookahead=lexer();
while(lookahead!=DONE)
{
E();
Match(';');
}
}
main()
{
char ans[10];
printf("\n Program for recursive decent parsing ");
printf("\n Enter the expression ");
printf("And place ; at the end\n");
printf("Press Ctrl-Z to terminate\n");
parser();
}
```

Output:

```
 Program for recursive decent parsing
 Enter the expression And place ; at the end
Press Ctrl-Z to terminate
a+b*c;

 Identifier: a
 Identifier: b
 Identifier: c
Arithmetic Operator: *
Arithmetic Operator: +
2*3;

 Number: 2
 Number: 3
Arithmetic Operator: *
+3;
line 5, Syntax error
```

Predictive Parser: Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking. 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.

6.Write a program to Design predictive parser for the given grammar.

```
#include<stdio.h>
#include<string.h>
# define SIZE 30
char st[100];
int top=-1;
int s1(char),ip(char);
int n1,n2;
char nt[SIZE],t[SIZE];
/*Function to return variable index*/
int s1(char c)
{
int i;
for(i=0;i<n1;i++)
{
if(c==nt[i])
return i;
}
}
/*Function to return terminal index*/
int ip(char c)
{
int i;
for(i=0;i<n2;i++)
{
if(c==t[i])
return i;
}
}
void push(char c)
{
```

```
top++;
st[top]=c;
return;
}
void pop()
{
top--;
return;
}
main()
{
char table[SIZE][SIZE][10],input[SIZE];
int x,f,s,i,j,u;
printf("Enter the number of variables:");
scanf("%d",&n1);
printf("\nUse single capital letters for variables\n");
for(i=0;i<n1;i++)
{
printf("Enter the %d  nonterminal:",i+1);
scanf("%c",&nt[i]);
}
printf("Enter the number of terminals:");
scanf("%d",&n2);
printf("\nUse single small letters for terminals\n");
for(i=0;i<n2;i++)
{
printf("Enter the %d terminal:",i+1);
scanf("%c",&t[i]);
}
/*Reading the parsing table*/
printf("Please enter only right sides of productions\n");
printf("Use symbol n to denote no entry and e to epsilon\n");
for(i=0;i<n1;i++)
{
for(j=0;j<n2;j++)
{
```

```
printf("\nEnter the entry for %c under %c:",nt[i],t[j]);
scanf("%s",table[i][j]);
}
}
/*Printing the parsing taable*/
for(i=0;i<n2;i++)
printf("\t%c",t[i]);
printf("\n----------\n");
for(i=0;i<n1;i++)
{
printf("\n%c|\t",nt[i]);
for(j=0;j<n2;j++)
{
if(!strcmp(table[i][j],"n"))
printf("\t");
else
printf("%s\t",Table[i][j]);
}

printf("\n");
}
printf("Enter the input:");
scanf("%s",input);
/*Initialising the stack*/
top++;
st[top]='$';
top++;
st[top]=nt[0];
printf("STACK content INPUT content PRODUCTION used\n");
printf("----------\n");
i=0;
printf("$%c\t\t\t%s\n\n",st[top],input);
while(st[top]!='$')
{
x=0;
f=s1(st[top]);
```

```
s=ip(input[i]);
if(!strcmp(table[f][s],"n"))
{
printf("'String not accepted");
}
else
if(!strcmp(table[f][s],"e"))
{
pop();
}
else
if(st[top]==input[i])
{
x=1;
pop();
i++;
}
else
{
pop();
for(j=strlen(table[f][s])-1;j>0;j--)
{
{
push(table[f][s][j]);
}
}
for(u=0;u<=top;u++)
printf("%c",st[u]);
printf("\t\t\t");
for(u=i;input[u]!='\0';u++)
printf("%c",input[u]);
printf("\t\t\t");
if(x==0)
printf("%c->%s\n\n",nt[f],table[f][s]);
printf("'\n\n");
}
```

```
printf("\n\nThus string is accepted");
}
}
```

OUTPUT:

```
Enter the number of variables:5
Use single capital letters for the variables
Enter the 1 non terminal:E
Enter the 2 non terminal:A
Enter the 3 non terminal:T
Enter the 4 non terminal:B
Enter the 5 non terminal:F
Enter the number of terminals:6
Use only single small letter for the terminals
Enter the 1 terminal:+
Enter the 2 terminal:*
Enter the 3 terminal:(
Enter the 4 terminal:)
Enter the 5 terminal:i
Enter the 6 terminal:$
Please enter only the right sides of productions.

Use symbol n to denote noentry and e to epsilon
Enter the entry for E under $: n
Enter the entry for E under +: n
Enter the entry for E under *: n
Enter the entry for E under (: TA
Enter the entry for E under ): n
Enter the entry for E under i: TA
Enter the entry for A under +: +TA
Enter the entry for A under *: n
Enter the entry for A under (: n
Enter the entry for A under ): e
Enter the entry for A under i: n
Enter the entry for A under $: e
Enter the entry for T under +: n
Enter the entry for T under *: n
```

Enter the entry for T under (: FB

Enter the entry for T under): n

Enter the entry for T under i: FB

Enter the entry for T under $: n

Enter the entry for B under +: e

Enter the entry for B under *: *FB

Enter the entry for B under (: n

Enter the entry for B under): e

Enter the entry for B under i: n

Enter the entry for B under $: e

Enter the entry for F under +: n

Enter the entry for F under *: n

Enter the entry for F under (: (E)

Enter the entry for F under): n

Enter the entry for F under i: i

Enter the entry for F under $: n

+ * ( ) i $

E| TA TA

A| +TA e e

T| FB FB

B| e *FB e e

F| (E) i

Enter the input: i+i*i$

Stack content Input content Production used

$E i+i*i$

$A i+i*i$ E->TA

$AB i+i*i$ T->FB

$AB i+i*i$ F->i

$A +i*i$

$ +i*i$ B->e

$AT +i*i$ A->+TA

$A i*i$

$AB i*i$ T->FB

$AB i*i$ F->i

$A *i$

$ABF *i$ B->*FB

$AB i$

$AB i$ F->i

$A $

$ $ B->e

$ A->e

Thus string is accepted

First Grammar: We saw the need of backtrack in the previous article of on Introduction to Syntax Analysis, which is really a complex process to implement. There can be easier way to sort out this problem:
If the compiler would have come to know in advance, that what is the "first character of the string produced when a production rule is applied", and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

7 . Write a program to calculate first function for the given grammar.

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[],char );
void addToResultSet(char[],char);
int numOfProductions;
char productionSet[10][10];
main()
{
   int i;
   char choice;
   char c;
   char result[20];
   printf("How many number of productions ? :");
   scanf(" %d",&numOfProductions);
   for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T
   {
      printf("Enter productions Number %d : ",i+1);
      scanf(" %s",productionSet[i]);
   }
   do
   {
      printf("\n Find the FIRST of  :");
      scanf(" %c",&c);
      FIRST(result,c); //Compute FIRST; Get Answer in 'result' array
      printf("\n FIRST(%c)= { ",c);
      for(i=0;result[i]!='\0';i++)
      printf(" %c ",result[i]);       //Display result
      printf("}\n");
       printf("press 'y' to continue : ");
```

```
        scanf(" %c",&choice);
    }
    while(choice=='y'||choice =='Y');
}/*
 *Function FIRST:
 *Compute the elements in FIRST(c) and write them
 *in Result Array.
 */
void FIRST(char* Result,char c)
{
    int i,j,k;
    char subResult[20];
    int foundEpsilon;
    subResult[0]='\0';
   Result[0]='\0';
    //If X is terminal, FIRST(X) = {X}.
    if(!(isupper(c)))
    {
        addToResultSet(Result,c);
                return ;
    }
    //If X is non terminal
    //Read each production
    for(i=0;i<numOfProductions;i++)
    {
//Find production with X as LHS
        if(productionSet[i][0]==c)
        {
//If X ? e is a production, then add e to FIRST(X).
 if(productionSet[i][2]=='$') addToResultSet(Result,'$');
            //If X is a non-terminal, and X ? Y1 Y2 … Yk
            //is a production, then add a to FIRST(X)
            //if for some i, a is in FIRST(Yi),
            //and e is in all of FIRST(Y1), …, FIRST(Yi-1).
      else
            {
```

```
            j=2;
            while(productionSet[i][j]!='\0')
            {
            foundEpsilon=0;
            FIRST(subResult,productionSet[i][j]);
            for(k=0;subResult[k]!='\0';k++)
               addToResultSet(Result,subResult[k]);
             for(k=0;subResult[k]!='\0';k++)
                if(subResult[k]=='$')
                {
                   foundEpsilon=1;
                   break;
                }
             //No e found, no need to check next element
             if(!foundEpsilon)
                break;
             j++;
            }
        }    }
}
   return ;
}
/* addToResultSet adds the computed
 *element to result set.
 *This code avoids multiple inclusion of elements
  */
void addToResultSet(char Result[],char val)
{
   int k;
   for(k=0 ;Result[k]!='\0';k++)
      if(Result[k]==val)
         return;
   Result[k]=val;
   Result[k+1]='\0';
}
```

OUTPUT:

```
 FIRST( )= { }
press 'y' to continue : How many number of productions ? :8
Enter productions Number 1 : E=TD
Enter productions Number 2 : D=+TD
Enter productions Number 3 : D=$
Enter productions Number 4 : T=FS
Enter productions Number 5 : S=FS
Enter productions Number 6 : S=$
Enter productions Number 7 : F=(E)
Enter productions Number 8 : F=a

 Find the FIRST of  :E

 FIRST(E)= {  (  a }
press 'y' to continue : Y

 Find the FIRST of  :D

 FIRST(D)= {  +  $ }
press 'y' to continue : Y

 Find the FIRST of  :S

 FIRST(S)= {  (  a  $ }
press 'y' to continue :
```

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser. This parser requires some data structures i.e.

- A input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Basic Operations:

Shift: This involves moving of symbols from input buffer onto the stack.

Reduce: If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.

Accept: If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.

Error: This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

8.To Write a C Program to implement a Shift reduce parsing.

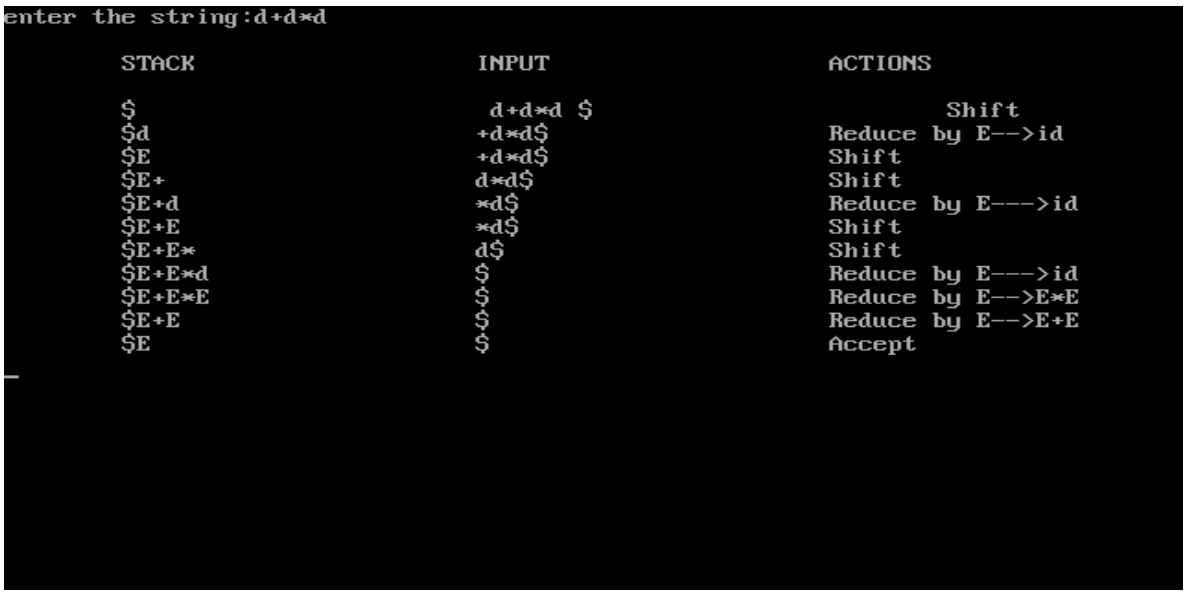
```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
void main()
{
char is[20],isa[20],sip[20],isal[20];
int isl,i,j,k,tl,r,si=0;
char t;
clrscr();
printf("enter the string:");
scanf("%s",&is);
isl=strlen(is);
strcpy(isa,is);
strcpy(isal,isa);
strcpy(sip," ");
printf("\n\tSTACK\t\t\tINPUT\t\t\tACTIONS\n");
printf("\n\t$\t\t\t %s $\t\t\tShift",is);
for(i=0;i<isl;i++)
{
if(i==0)
{
```

```
if((isalpha(is[i]))||(isdigit(is[i])))
{
if(isalpha(is[i+1]))
{
i++;
sip[0]='E';
}
sip[0]='E';
si=0;
printf("\n\t$%c\t\t\t",is[i]);
for(j=i+1;j<isl;j++)
printf("%c",is[j]);
}
printf("$\t\t\tReduce by E-->id\n\t$%c\t\t\t",sip[0]);
for(j=i+1;j<isl;j++)
printf("%c",is[j]);
printf("$\t\t\tShift\n");
}
else
{   if((isalpha(is[i]))||(isdigit(is[i])))
{       if (is[i]==is[i-1])
{
printf("rew\n");
}
else if(isalpha(is[i-1]))
{
printf("ewrw\n");
}
else {
printf("\t\t\t");
for(j=i;j<isl;j++)
printf("%c",is[j]);
printf("$\t\t\tShift\n");
si++;
sip[si]=is[i];
printf("\t$");
```

```
for(j=0;j<=si;j++)
printf("%c",sip[j]);
printf("\t\t\t");
for(j=i+1;j<isl;j++)
printf("%c",is[j]);
printf("$\t\t\tReduce by E--->id\n");
sip[si]='E';
printf("\t$");
for(j=0;j<=si;j++)
printf("%c",sip[j]);
printf("\t\t\t");
for(j=i+1;j<isl;j++)
printf("%c",is[j]);
if(si!=isl-1)
printf("$\t\t\tShift\n");
else
printf("$\t\t\tReduce by E-->E%cE\n",is[i-1]);
if((((i-1)==1)&&((isl-1)==i)))
{
strcpy(sip,"");
sip[0]='E';
si=0;
printf("\t$%c\t\t\t",sip[0]);
for(j=i+1;j<isl;j++)
printf("%c",is[j]);
printf("$\t\t\tAccept\n",is[i-1]);
}
if(i==(isl-1))
{
if(isalpha(sip[si-1])==0)
{
sip[si-2]='E';
si=si-2;
printf("\t$");
for(j=0;j<=si;j++)
printf("%c",sip[j]);
```

```
printf("\t\t\t");
for(j=j+2;j<isl;j++)
printf("%c",is[j]);
printf("$\t\t\tReduce by E-->E%cE\n",sip[si-1]);
}
}
}
}
else {
si++;
sip[si]=is[i];
printf("\t$");
for(j=0;j<=si;j++)
printf("%c",sip[j]);
}
}
}
if(si==2)
printf("\t$E\t\t\t$\t\t\tAccept\n");
getch();
}
```

OUTPUT:



LL(1) Parsing: LL parser (Left-to-right, Leftmost derivation) is a top-down parser for a subset of context-free languages. It parses the input from Left to right, performing Leftmost derivation of the sentence.

An LL parser is called an LL(*k*) parser if it uses *k* tokens of lookahead when parsing a sentence. A grammar is called an LL(*k*) grammar if an LL(*k*) parser can be constructed from it. A formal language is called an LL(k) language if it has an LL(k) grammar.

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

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

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

stack[i]='i'; else
```

```
{
for(k=size[str1][str2]-1;k>=0;k--)
{
stack[i]=m[str1][str2][k];
i++;
}
i--;
}
  for(k=0;k<=i;k++)
printf(" %c",stack[k]);
printf("");
for(k=j;k<=n;k++)
printf("%c",s[k]);
printf(" \n ");
}
printf("\n SUCCESS");
getch(); }
```

OUTPUT:

Enter the input string:i*i+i

Stack	INPUT
$bt	i*i+i$
$bcf	i*i+i$
$bci	i*i+i$
$bc	*i+i$
$bcf*	*i+i$
$bcf	i+i$
$bci	i+i$
$bc	+i$
$b	+i$
$bt+	+i$
$bt	i$
$bcf	i$
$ bci	i$
$bc	$
$b	$
$	$

success

LALR parser

In computer science, an **LALR parser**[a] or **Look-Ahead LR parser** is a simplified version of a canonical LR parser, to parse (separate and analyze) a text according to a set of production rules specified by a formal grammar for a computer language.

10. Design LALR Bottom up Parser.

<parser.l>

```
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+ {yylval.dval=atof(yytext);
return DIGIT;
}
\n|. return yytext[0];
%%
```

<parser.y>

```
%{
/*This YACC specification file generates the LALR parser for the program
considered in experiment 4.*/
#include<stdio.h>
%}
%union
{
double dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
line: expr '\n' {
printf("%g\n",$1);
}
;
expr: expr '+' term {$$=$1 + $3 ;}
| term
;
term: term '*' factor {$$=$1 * $3 ;}
| factor
;
factor: '(' expr ')' {$$=$2 ;}
```

```
| DIGIT
;
%%
int main()
{
yyparse();
}
yyerror(char *s)
{
printf("%s",s);
}
```

Output:

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

Yacc: Yet Another Compiler-Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an ``input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that han- dles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

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

<int.l>

```
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
%}
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
```

```
\n LineNo++;
. return yytext[0];
%%
```

<int.y>

```
%{
#include<string.h>
#include<stdio.h>
struct quad
{
char op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];
struct stack
{
int items[100];
int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
%}
%union
{
char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
```

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

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

```
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------------------------""\n\t\t Pos Operator Arg1 Arg2 Result" "\n\t\t
--------------------");
for(i=0;i<Index;i++)
{
printf("\n\t\t %d\t %s\t %s\t %s\t
%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t ------------------------");
printf("\n\n");
return 0;
}
void push(int data)
{
stk.top++;
if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
{
int data;
if(stk.top==-1)
{
printf("\n Stack underflow\n");
exit(0);
}
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
```

```
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

Input:

```
$vi test.c
main()
{
int a,b,c;
if(a<b)
{
a=a+b;
}
while(a<b)
{
a=a+b;
}
if(a<=b)
{
c=a-b;
}
else
{
c=a+b;
}
}
```

Output:

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

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

Machine Code: A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.

- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

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

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

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

```
break;
case '%': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n \t LOAD %s,R1",operand2);
fprintf(fp2,"\n \t DIV R1,R0");
fprintf(fp2,"\n \t STORE R0,%s",result);
break;
case '=': fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t STORE %s %s",operand1,result);
break;
case '>': j++;
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t JGT %s,label#%s",operand2,result);
label[no++]=atoi(result);
break;
case '<': fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n \t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t JLT %s,label#%d",operand2,result);
label[no++]=atoi(result);
break;
}
}
fclose(fp2);
fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("Error opening the file\n");
exit(0);
}
do
{
ch=fgetc(fp2);
printf("%c",ch);
}while(ch!=EOF);
fclose(fp1);
return 0;
}i
nt check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
}
```

```
return 0;
}
```

Input:

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

Output:

```
Enter filename of  the intermediate code: int.txt
STORE t1,2
STORE a[0],1
STORE a[1],2
STORE a[2],3

LOAD t1,R0
LOAD 6,R1
ADD R1,R0
STORE R0,t3

LOAD a[2],R0
LOAD t2,R1
ADD R1,R0
STORE R0,t3

LOAD a[t2],R0
LOAD t1,R1
SUB R1,R0
STORE R0,t2

LOAD t3,R0
LOAD t2,R1
DIV R1,R0
STORE R0,t2

LOAD t2,R1
STORE R1,t2
LOAD t2,R0
JGT 5,label#11

Label#11: OUT t2
JMP t2,label#13
Label#13: STORE t3,99
LOAD 25,R1
STORE R1,t2

LOAD t2,R0
LOAD t3,R1
MUL R1,R0
STORE R0,t3

LOAD t1,R1
STORE R1,t1

LOAD t1,R0
LOAD t3,R1
ADD R1,R0
STORE R0,t4
OUT t4
```