KNOWLEDGE INSTITUTE OF TECHNOLOGY, SALEM – 637 504

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL
2015-2016

CS6612 – COMPILER LABORATORY

Branch: CSE

Year/Sem: III/VI

Prepared By,

Mrs.K.Saranya, AP/CSE
Ms.T.Vinothini, AP/CSE

KNOWLEDGE INSTITUTE OF TECHNOLOGY

AFFILIATED TO ANNA UNIVERSITY, CHENNAI

KAKAPALAYAM (PO), SALEM – 637 504

RECORD NOTE BOOK

REG.NO

Certified that this is the bonafide record of work done by Selvan/Selvi……of the……. Semester……..……Branch during the year…….in the……..……laboratory.

Faculty – Incharge

Head of the Department

……

Submitted for the University practical Examination on …….

Internal Examiner

External Examiner

CONTE
NTS

S.No	Date	Name of the Experiment	Page No.	Date of Completion	Marks Awarded	Staff Signature	Remarks
1		Implementation of Symbol Table					
2		Develop a lexical analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments, operators etc.)					
3		Implementation of Lexical Analyzer using Lex Tool					
4		Generate YACC specification for a few syntactic categories. a) Program to recognize a valid arithmetic expression that uses operator +, - , * and /. b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits. c)Implementation of Calculator using LEX and YACC					
5		Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree.					
6		Implement type checking					
7		Implement control flow analysis and Data flow Analysis					
8		Implement any one storage allocation strategies(Heap,Stack,Static)					
9		Construction of DAG					
10		Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using a 8086 assembler. The target assembly instructions can be simple move, add, sub, jump. Also simple addressing modes are used.					
11		Implementation of Simple Code Optimization Techniques (Constant Folding., etc.)					

EX.No:1

IMPLEMENTATION OF SYMBOL TABLE

AIM:

To write a C program to implement a symbol table which can create, insert, modify, search and display the character.

ALGORITHM:

Step 1: Print the five choices namely create, insert, modify, search and display.

Step 2: Get the choice value.

Step 3: If choice is 1, then get total numbers of symbol to be added in the symbol table and call Create ( ) to create the symbol table and call the display ( ) to display the symbol table with the following contents: symbol name, location and symbol type.

Step 4: If choice is 2, call insert ( ) get a symbol table detail to insert into the symbol table and Increment the current position the insert the given detail.

Step 5: If choice is 3, then read the symbol to be modified and call the modified ( ) to modify the location of particular symbol.

Step 6: If choice is 4, then read the symbol to be searched and call the search ( ) to display the symbol table for particular symbol.

Step 7: If choice is 5, to display the symbol table.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
struct source
{
char la[25];
char oc[25];
char o[25];
int addr;
}s[100];
struct stable
{
char sym[25];
int val;
}st[20];
int n,m=0;
int locctr=0;
void input();
void create();
void display();
void search(char key[10]);
void modify();
void main()
{
int ch,a;
char k[10];
clrscr();
do
{
printf("\n\tSYMBOL TABLE IMPLEMENTATION");
printf("\n1.CREATE(INSERT)\n\n2.SEARCH\n\n3.MODIFY\n\n4.DISPLAY\n\n5.EXIT\n
\nENTER YOUR CHOICE\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
input();
create();
break;
case 2:
printf("ENTER THE LABEL TO SEARCH\n");
scanf("%s",k);
search(k);
break;
case 3:
```

```
modify();
break;
case 4:
display();
break;
case 5:
exit(0);
}
}while(ch<=6);
getch();
}
void input()
{
int i=0;
printf("ENTER CODE & TO STOP TYPE END \n");
printf("\nLABEL\tOPCODE\tOPERAND");
do
{ scanf("%s",s[i].la);
if(strcmp(s[i].la,"END")==0)
break;
scanf("%s",s[i].oc);
scanf("%s",s[i].o);
i++;
}while(1);
n=i;
}
void create()
{
int i;
if(strcmp(s[0].oc,"START")==0)
{
locctr=atoi(s[0].o);
s[1].addr=locctr;
}
for(i=1;i<n;i++)
{
if(strcmp(s[i].oc,"WORD")==0)
locctr=locctr+3;
else if(strcmp(s[i].oc,"RESW")==0)
locctr=locctr+(3*atoi(s[i].o));
else if(strcmp(s[i].oc,"RESB")==0)
locctr=locctr+(atoi(s[i].o));
else if(strcmp(s[i].oc,"BYTE")==0)
locctr=locctr+(strlen(s[i].o)-3);
else
locctr=locctr+3;
```

```
s[i+1].addr=locctr;
} m=0;
for(i=1;i<n;i++)
{
if(s[i].la[0]!='-')
{
strcpy(st[m].sym,s[i].la);
st[m].val=s[i].addr;
m++;
}
}
printf("\nSymbol table created");
}
void display()
{
int i;
printf("\t\tSYMBOL TABLE \n\n");
printf("\nVALUE\tSYMBOL\n");
for(i=0;i<m;i++)
{ printf("%d",st[i].val);
printf("\t%s\n\n\n",st[i].sym);
}
}
void search(char key[10])
{
int i,a;
for(i=0;i<m;i++)
{
if(strcmp(st[i].sym,key)==0)
{
printf("\nThe label is found at address: %d ",st[i].val);
break;
}
else
printf("\n the label is not found");
}
}
void modify()
{
int a,i;
char key[25];
printf("\n\tSOURCE CODE \t\n");
printf("\nADDRESS\tLABEL\tOPCODE\tOPERAND\n");
for(i=1;i<n;i++)
{
```

```
printf("\n%d",s[i].addr);
printf("\t%s",s[i].la);
printf("\t%s",s[i].oc);
printf("\t%s\n\n",s[i].o);
}
printf("\nENTER THE LABEL TO MODIFY\n");
scanf("%s",key);
for(i=0;i<m;i++)
{
if(strcmp(st[i].sym,key)==0)
{
a=1;
break;
}
}
if(a==1)
{
printf("\nENTER THE NEW LABEL\n");
scanf("%s",key);
strcpy(st[i].sym,key);
}
else
printf("\nTHE LABEL IS NOT FOUND\n");
}
```

OUTPUT:

```
SYMBOL TABLE IMPLEMENTATION
1.CREATE(INSERT)
2.SEARCH
3.MODIFY
4.DISPLAY
5.EXIT

ENTER YOUR CHOICE
1

ENTER CODE & TO STOP TYPE END
LABEL          OPCODE              OPERAND
SAMPLE         START               1000
-              LDA                 ALPHA
ALPHA          RESB                1
END
Symbol table created
```

SYMBOL TABLE IMPLEMENTATION
1.CREATE(INSERT)
2.SEARCH
3.MODIFY
4.DISPLAY
5.EXIT
ENTER YOUR CHOICE
4

SYMBOL TABLE

VALUE SYMBOL
1003 ALPHA
SYMBOL TABLE IMPLEMENTATION
1.CREATE(INSERT)
2.SEARCH
3.MODIFY
4.DISPLAY
5.EXIT

ENTER YOUR CHOICE
2

ENTER THE LABEL TO SEARCH
ALPHA
The label is found at address: 1003

SYMBOL TABLE IMPLEMENTATION
1.CREATE(INSERT)
2.SEARCH
3.MODIFY
4.DISPLAY
5.EXIT
ENTER YOUR CHOICE
3

SOURCE CODE

ADDRESS	LABEL	OPCODE	OPERAND
1000	-	LDA	ALPHA
1003	ALPHA	RESB	1

ENTER THE LABEL TO MODIFY
ALPHA
ENTER THE NEW LABEL
GAMMA

```
SYMBOL TABLE IMPLEMENTATION
1.CREATE(INSERT)
2.SEARCH
3.MODIFY
4.DISPLAY
5.EXIT

ENTER YOUR CHOICE
4

SYMBOL TABLE
------------------
VALUE          SYMBOL
1003           GAMMA
SYMBOL TABLE IMPLEMENTATION
1.CREATE(INSERT)
2.SEARCH
3.MODIFY
4.DISPLAY
5.EXIT

ENTER YOUR CHOICE 5
```

RESULT:

Thus the C program to implement a symbol table has been executed successfully.

EX.No:2

DEVELOP A LEXICAL ANALYZER TO RECOGNIZE THE PATTERNS IN C.
(Ex. identifiers, constants, comments, operators etc.)

AIM:

To develop a lexical analyzer to recognize the patterns (Ex. identifiers, constants, comments, operators etc.) using C program.

ALGORITHM:

1. Start the program.
2. Open the file.
3. Declare the variable and functions.
4. Input: Programming language 'if' statement
   Output: A sequence of tokens.
5. Tokens have to be identified and its respective attributes have to be printed.
6. Execute the program
7. Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int main()
{
FILE *fp;
char op[20]={'+','-','*','/','%'};
int i,j=0,num=0,flag,f;
char file[20],ch,id[30];
printf("\n\t\t Token Seperation \n\n");
printf("Enter the file name:");
scanf("%s",&file);
fp=fopen(file,"r");
while(ch!=EOF)
```

```
{
printf("%c",ch);
ch=getc(fp);
} fclose(fp);
fp=fopen(file,"r");
printf("\n\t Line number \t Token \t\t Type \n");
printf("\n\t ___ _ ");
while(!feof(fp))
{ flag=0;
ch=fgetc(fp);
i=0;
if(isalpha(ch)||isdigit(ch))
{
f=0;
while(isalpha(ch)||isdigit(ch))
{ id[j++]=ch;
ch=fgetc(fp);
}
id[j]='\0';
j=0;
if(f==0)
printf("\n\t %d \t\t %s \t\t Identifier",num,id);
} i=0;
while(i<5&&flag!=1)
{
if(ch==op[i])
{
 printf("\n\t %d \t\t %c \t\t Operator",num,ch);
 flag=1;
}
i++;
```

```
} i=0;
if(ch=='\n')
num++;
}
fclose(fp);
}
```

INPUT FILE:

[compiler@localhost ~]$ vi sat.txt

a=b*c-d/e

OUTPUT:

[compiler@localhost ~]$ cc lexical.c

[compiler@localhost ~]$ ./a.out

Token Seperation

Enter the file name: sat.txt

a=b*c-d/e

Line number	Token	Type
0	a	Identifier
0	b	Identifier
0	*	Operator
0	c	Identifier
0	-	Operator
0	d	Identifier
0	/	Operator

RESULT:

Thus the C program to develop a lexical analyzer to recognize the patterns has been executed successfully.

EX.No:3

IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL

AIM:

To implement a lexical analyzer using LEX tool.

ALGORITHM:

1. Start the program.
2. Lex program consists of three parts.
   - Declaration
   %%
   - Translation rules
   %%
   - Auxilary procedure.
3. The declaration section includes declaration of variables, maintest, constants and regular definitions.
4. Translation rule of lex program are statements of the form
   - P1 {action}
   - P2 {action}
   - Pn {action}
5. Write a program in the vi editor and save it with .l extension.
6. Compile the lex program with lex compiler to produce output file as lex.yy.c. eg $ lex filename.l and $ cc lex.yy.c -ll
7. Compile that file with C compiler and verify the output.

PROGRAM:

```
%{
#include<stdio.h>
%}
identifier [a-z A-Z][a-z A-Z 0-9]*
%%
#.*                              { printf("\n is a preprocessor directive",yytext); }
int|void|printf|scanf|struct     { printf("\n\t %s is a keyword",yytext); }
if[\t]*\(                        { printf("\n\t if is a keyword\n\t"); }
while[\t]*\(                     { printf("\n\t while is a keyword\n\t"); }
{identifier}\(                   { printf("\n\nFunction call/definition\t %s",yytext); }
\{                               { printf("\tBlock begins"); }
\}                               { printf("\tBlock ends"); }
{identifier}(\[[0-9]*\])*        { printf("\n\t %s is an identifier",yytext); }
\".*\"                           { printf("\n\t %s is a string",yytext); }
[0-9]+                           { printf("\n\t %s is a number",yytext); }
\)(\;)?                          { printf("\n\t"); ECHO; printf("\n"); }
\( ECHO;
=                                { printf("\n\t %s is an assignment operator",yytext); }
\<=|\>=|\<|\==|\>                { printf("\n\t %s ia a relational operator",yytext); }
.|\n;
\/\*.*\*\/|\/\/.*                { printf("\n\n\t %s is a comment\n",yytext); }
%%
```

```
int main()
{ yylex();
return 0;
}

int yywrap()
{
return 0;
}
```

OUTPUT:

```
[compiler@localhost ~]$ lex lex.l
[compiler@localhost ~]$ cc lex.yy.c
[compiler@localhost ~]$ ./a.out

int
int is a keyword

a
a is an identifier



9
9 is a number

#include<stdio.h>
is a preprocessor directive

=
= is an assignment operator

{
Block begins

}
Block ends

/*comment*/
/*comment*/ is a comment

<
< is a relational operator


>=
>= is a relational operator

"basco"
"basco" is a string
```

RESULT:

Thus the program to implement a lexical analyzer using LEX tool has been executed .

EX.NO.4

GENERATE YACC SPECIFICATION FOR A FEW SYNTACTIC CATEGORIES.

OVERVIEW:

YET ANOTHER COMPILER-COMPILER (YACC)

Computer program input generally has come structure; in fact, every computer program that does input can be thought of as defininf an "input language" which it accepts. 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 handles 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.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

Basic Specifications

Names refer to either tokens or non-terminal symbols. Yacc requires token names to be declared as such. It is often desirable to include the lexical analyzer as part of the specification file. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

Declarations

%%

rules

%%

programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also. Thus, the smallest legal Yacc specification is

%%

Rules

Blanks, tabs, and new lines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /*…*/, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A: BODY ;

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "-", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or non-terminal symbols.

A literal consists of a character enclosed in single quotes. As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

'\n' newline

'\r' return

'\'' single quote

'\\' backslash

'\t' tab

'\b' backspace

'\f' form feed

'\xxx' "xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

%token name1 name2...

in the declarations section. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

4 a) Program to recognize a valid arithmetic expression that uses operator +, - , * and /.

AIM:

To generate the YACC specification for a few syntactic categories like arithmetic expressions and calculator.

ALGORITHM:

1. Start the program
2. YACC program consists of three parts.
   - Declaration %%
   - Translation rules %%
   - Auxilary procedure.
3. The declaration section includes declaration of variables, maintest, constants and regular definitions.
4. Declare the grammar rules with actions.
5. Use the symbol “dollar sign” “$” to Yacc in this context.
6. Use the loop to generate the code.
7. Input: Programming language arithmetic expression
8. Output: A sequence of tokens.
9. Tokens have to be identified and its respective attributes have to be printed.
10. Execute the program
11. Stop the program

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
        int n,i,j;
        char a[50][50];
        clrscr();
        printf("enter the no: intermediate codes:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("enter the 3 address code:%d:",i+1);
                for(j=0;j<6;j++)
                {
                        scanf("%c",&a[i][j]);
                }
        }
        printf("the generated code is");
        for(i=0;i<n;i++)
        {
                printf("\nMov %c,R%d",a[i][3],i);
                if(a[i][4]=='-')
                {
                        printf("\nSub%c,R%d",a[i][5],i);
                }
```

```
                if(a[i][4]=='+')
                {
                        printf("\nAdd%c,R%d",a[i][5],i);
                }
                if(a[i][4]=='*')
                {
                        printf("\nMul%c,R%d",a[i][5],i);
                }
                if(a[i][4]=='/')
                {
                        printf("\nDiv%c,R%d",a[i][5],i);
                }
                printf("\nMov R%d,%c",i,a[i][1]);
                printf("\n");
        }
        getch();
}
```

OUTPUT:

```
Enter the no:intermediate code: 2
Enter the three address code1: a=b/c
Enter the three address code2: r=a+b
The generated code is
 Mov b,R0
 Div c,R0
 Mov R0,a

 Mov a,R1
 Add b,R1
 Mov R1,r
```

b) Program to recognize a valid variable which starts with a letter followed by any number of letters or digits

AIM:

To generate the YACC specification to recognize a valid variable which starts with a letter followed by any number of letters or digits

ALGORITHM:

1. Start the program
2. YACC program consists of three parts.
   - Declaration %%
   - Translation rules %%
   - Auxilary procedure.
3. The declaration section includes declaration of variables, maintest, constants and regular definitions.
4. Declare the grammar rules with actions.
5. Use the symbol "dollar sign" "$" to Yacc in this context.
6. Tokens have to be identified and its respective attributes have to be printed.
7. Execute the program
8. Stop the program

PROGRAM:

lexyacc.l

```
%{
#include "y.tab.h"
#include<stdio.h>
void yyerror(char *);
%}
%%
[a-z] {return VAR;};
[0-9] {return NUM;}
[-+()=/*\n] {return *yytext;}
[\t];
. {yyerror("Invalid character");}
%%
int yywrap(void)
{
        return 1;
}
```

lexyacc.y

```
%{
#include<string.h>
#include<stdio.h>
%}
%token NUM VAR
%left '-''+'
%left '*''/'
```

```
%{
void yyerror(char *);
int yylex(void);
%}
%%
PROGRAM:PROGRAM STATEMENT '\n' {printf("No error.\n");}
|
;
EXPR:NUM
|VAR
|EXPR'+'EXPR
|EXPR'-'EXPR
|EXPR'*'EXPR
|EXPR'/'EXPR
|'('EXPR')'
;
%%

void yyerror(char *s)
{
        printf("%s\n", s);
        return;
}

int main(void)
{
        yyparse();
        return 0;
}
```

y.tab.h

```
#ifndef YYERRCODE
#define YYERRCODE 256
#endif
#define NUM 257
#define VAR 258
```

OUTPUT:

```
[compiler@ linuxserver ~]$ lex lecyacc.l
[compiler@ linuxserver ~]$ yacc lexyacc.y
[compiler@ linuxserver ~]$ cc y.tab.c
lex.yy.c [compiler@ linuxserver ~]$
./a.out

(a+(b+
c)) No
error.

(1+b)
No error.

(123+5)
Syntax
error

[compiler@ linuxserver ~]$
./a.out ab+c
Syntax error
```

4c)Implementation of Calculator using LEX and YACC

AIM:

To implement a calculator using LEX and YACC specification.

YACC: ALGORITHM:

1. A Yacc source program has three parts as follows:

Declarations

%%

translation rules

%%

supporting C routines

2. Declarations Section:

This section contains entries that: i. Include standard I/O header file. ii. Define global variables.

iii. Define the list rule as the place to start processing. iv. Define the tokens used by the parser.

v. Define the operators and their precedence.

3. Rules Section:

The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.

4. Programs
Section:

The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.

5. Main- The required main program that calls the **yyparse** subroutine to start the program.

6. yyerror(s) -This error-handling subroutine only prints a syntax error message.

7. **yywrap** -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The cal.l contains include statements for standard input and output, as programmar file information if we use the -d flag with the yacc command. The cal.y file contains definitions for the tokens that the parser program uses.

PROGRAM:

cal.l

```
%{
#include<stdlib.h>
#include "y.tab.h"
void yyerror(char
*); extern int
yylval;
%}
%%
[a-z] {yylval=(*yytext)-'a';
return VARIABLE;
}
[0-9]+ {yylval=atoi(yytext);
return INTEGER;
}
[-+()=/*\n] {return
*yytext;} [\t];
. {yyerror("Invalid character.");}
%%
int yywrap(void)
{
        return 1;
}
```

cal.y

```
%token INTEGER VARIABLE
%left '+"-'
```

```
%left '*' '/'
%{
void yyerror(char *);
int yylex(void);
int sym[26];
%}
%%
PROGRAM:PROGRAM STATEMENT '\n'
|
;
STATEMENT:EXPR {printf("%d\n",$1);}
;
EXPR:INTEGER



|VARIABLE
|EXPR'+'EXPR {$$=$1+$3;}
|EXPR'-'EXPR {$$=$1-$3;}
|EXPR'*'EXPR {$$=$1*$3;}
|EXPR'/'EXPR {$$=$1/$3;}
|'('EXPR')' {$$=$2;}
;
%%
void yyerror(char *s)
{
        printf("%s\n",s);
        return;
}

int main(void)
{
        yyparse();
        return 0;
}
```

OUTPUT:

```
[compiler@ linuxserver ~]$ lex cal.l

[compiler@ linuxserver ~]$  yacc –d cal.y

[compiler@ linuxserver ~]$  cc y.tab.c
```

lex.yy.c [compiler@ linuxserver ~]$

./a.out

a+a
0

a+b
1

a+z
25

1+2
3

123*2
246

12%5
Invalid character.
syntax error

RESULT:

Thus the program to implement a calculator using LEX and YACC has been executed successfully.

EX.No:5

Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree.

AIM:

To convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree.

ALGORITHM:

1.Read an input file line by line.
2. Convert it in to abstract syntax tree using three address code.
3.Represent three address code in the form of quadruple tabular form.
4.Go to terminal .
5.Open vi editor ,Lex lex.l , cc lex.yy.c ,
6.Execute the program by ./a.out

PROGRAM:

<int.l>

```
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
%}
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
if return IF;
else return ELSE;
```

```
while return WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
\n LineNo++;
. return yytext[0];
%%
```

<int.y>

```
%{
#include<string.h>
#include<stdio.h>
struct quad
{
        char op[5]; char
        arg1[10]; char
        arg2[10]; char
        result[10];
}QUAD[30];
struct stack
{
        int items[100];
        int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
%}
%union
{
        char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
```

```
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}
| EXPR '-' EXPR {AddQuadruple("-",$1,$3,$$);}
| EXPR '*' EXPR {AddQuadruple("*",$1,$3,$$);}
| EXPR '/' EXPR {AddQuadruple("/",$1,$3,$$);}
| '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{ Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
```

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

```
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------------------------""\n\t\t Pos Operator Arg1 Arg2 Result" "\n\t\t
--------------------");
for(i=0;i<Index;i++)
{
printf("\n\t\t %d\t %s\t %s\t %s\t
%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t ----------------------");
printf("\n\n");
return 0;
}
void push(int data)
{ stk.top++;
if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
```

```
{
int data;
if(stk.top==-1)
{
printf("\n Stack underflow\n");
exit(0);
}
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{ strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

Input:

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

OUTPUT:

$lex int.l

$yacc –d int.y

$gcc lex.yy.c y.tab.c –ll –lm

$./a.out test.c

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

RESULT:

Thus the program to Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree has been implemented sucessufully.