Implement lexical analyzer for subset of English language using LEX.

lex.l

```
%{
    // This is a lexical analyzer for a subset of C languages
%}

%%

[\t]+   /*ignores white space*/ ;


is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go {printf("%s: is a verb\n", yytext);}

a |
an |
the {printf("%s: is an article\n", yytext);}

if |
then |
and |
but |
or |
so {printf("%s: this is a conjunction\n", yytext);}

he |
her |
him |
she |
```

```
them |
they {printf("%s: this is a preonoun\n", yytext);}

[a-zA-Z]+ {printf("%s: is not recognized, may be noun\n", yytext);}

.|\n {ECHO; /*normal default*/}
%%

main()
{
    yylex();
}

int yywrap()
{
    return(1);
}
```

Implement lexical analyzer for subset of 'C' language using LEX.

lex.l

```
digit [0-9]
letter [A-Za-z_]
%{
    #include <stdio.h>
    #include <string.h>
    //This is a lexical analyzer for a subset of C language
    int lno = 1;
    char symtab[100][100];
    int symtabidx = 0;
    void handle_symtab(char*);
    void print_symtab();
    char match[100];
%}

%%

[\t]+ ;

int |
float |
double |
String |
char |
if |
for |
else |
do |
```

```
while |
printf |
static |
void |
public {fprintf(yyout,"%s\t\t%d\t\tKeyword\n", yytext, lno);}

; |
, |
\" |
\' |
: |
\{ |
\} |
\( |
\) {fprintf(yyout,"%s\t\t%d\t\tDelimiter\n", yytext, lno);}

\+ |
\- |
\* |
\/ |
\^ {fprintf(yyout,"%s\t\t%d\t\tOperator\n", yytext, lno);}

\= |
\> |
\< |
\|\| |
\&\& |
\>\= |
\<\= |
\! |
\!\= {fprintf(yyout,"%s\t\t%d\t\tLogical Operator\n", yytext, lno);}

{letter}({letter}|{digit})* {fprintf(yyout,"%s\t\t%d\t\tIdentifier\n",
yytext, lno);
                            strcpy(match, yytext);
                            handle_symtab(match);}

[\n] {lno++;}

%%

int main()
{

    extern FILE *yyout;

    yyout = fopen("output.txt", "w");
```

```
    fprintf(yyout, "Token Listing for Subset of C languages\n");
    fprintf(yyout, "Lexeme\tLine\t\tToken\n");

    yylex();

    print_symtab();

    fclose(yyout);

    return 0;
}

void handle_symtab(char *text)
{
    for (int i = 0; i<symtabidx; i++)
    {
        if(strcmp(symtab[i], text) == 0)
            return ;
    }
    strcpy(symtab[symtabidx++], text);
}

void print_symtab()
{
    fprintf(yyout, "\n\nSymbol Table\n");
    fprintf(yyout,"Index\t\tSymbol\n");
    for(int i=0;i<symtabidx;i++)
    {
        fprintf(yyout, "%d\t\t%s\n", (i+1), symtab[i]);
    }
}

int yywrap()
{
    return(1);
}
```

Implement lexical analyzer for subset of english language using LEX. Input filename as command line argument

lex.l

```
%{
    // This is a lexical analyzer for a subset of C languages

    // this is the definition section, important when using header files
%}

%%
```

```
    /*this is the rules section*/

[\t]+   /*ignores white space*/ ;


is |
am |
are |
were |
was |
be |
being |
been |
do |
does |
did |
will |
would |
should |
can |
could |
has |
have |
had |
go {printf("%s: is a verb\n", yytext);} //array yytext contains the text
that the pattern matched

a |
an |
the {printf("%s: is an article\n", yytext);}

if |
then |
and |
but |
or |
so {printf("%s: this is a conjunction\n", yytext);}

he |
her |
him |
she |
them |
they {printf("%s: this is a preonoun\n", yytext);}

[a-zA-Z]+ {printf("%s: is not recognized, may be noun\n", yytext);}
```

```
.|\n {ECHO; /*normal default*/}
%%

// user subroutines section which consists of any legal c code
int main(int argc, char *argv[])
{
    yyin = fopen(argv[1], "rb+");
    yylex();
    return 0;
}
int yywrap()
{
    return(1);
}
```

Implement lexical analyzer for subset of 'C' language using LEX. Input filename as command line argument

lex.l

```
digit [0-9]
letter [A-Za-z_]

%{
    //This is a lexical analyzer for a subset of C language
    #include <stdio.h>
    #include <string.h>
    int lno = 1;
    char symtab[100][100];
    int symtabidx = 0;
    void handle_symtab(char*);
    void print_symtab();
    char match[100];
%}

%%

[\t]+ ;

int |
float |
double |
String |
char |
if |
for |
else |
do |
```

```
while |
printf |
static |
void |
public {fprintf(yyout,"%s\t\t%d\t\tKeyword\n", yytext, lno);}

; |
, |
\" |
\' |
: |
\{ |
\} |
\( |
\) {fprintf(yyout,"%s\t\t%d\t\tDelimiter\n", yytext, lno);}

\+ |
\- |
\* |
\/ |
\^ {fprintf(yyout,"%s\t\t%d\t\tOperator\n", yytext, lno);}

\= |
\> |
\< |
\|\| |
\&\& |
\>\= |
\<\= |
\! |
\!\= {fprintf(yyout,"%s\t\t%d\t\tLogical Operator\n", yytext, lno);}

{letter}({letter}|{digit})* {fprintf(yyout,"%s\t\t%d\t\tIdentifier\n",
yytext, lno);
                            strcpy(match, yytext);
                            handle_symtab(match);}

{digit} |
{digit}.{digit} {fprintf(yyout,"%s\t\t%d\t\tNumeric\n", yytext, lno);}

\"[^"\n]*\" {   strncpy(match, &yytext[1], strlen(yytext) - 2);
                fprintf(yyout, "\"\t\t%d\t\tDelimiter\n", lno);
                fprintf(yyout, "%s\t\t%d\t\tConstant\n", match, lno);
                fprintf(yyout, "\"\t\t%d\t\tDelimiter\n", lno);
}
```

```
[\n] {lno++;}

%%

int main(int argc, char *argv[])
{

    extern FILE *yyout;
    yyin = fopen(argv[1], "rb+");

    yyout = fopen("output.txt", "w");
    fprintf(yyout, "Token Listing for Subset of C languages\n");
    fprintf(yyout, "Lexeme\tLine\t\tToken\n");

    yylex();

    print_symtab();

    fclose(yyout);

    return 0;
}

void handle_symtab(char *text)
{
    for (int i = 0; i<symtabidx; i++)
    {
        if(strcmp(symtab[i], text) == 0)
            return ;
    }
    strcpy(symtab[symtabidx++], text);
}

void print_symtab()
{
    fprintf(yyout, "\n\nSymbol Table\n");
    fprintf(yyout,"Index\t\tSymbol\n");
    for(int i=0;i<symtabidx;i++)
    {
        fprintf(yyout, "%d\t\t%s\n", (i+1), symtab[i]);
    }
}

int yywrap()
{
    return(1);
}
```

Implement word count program using LEX.

lex.l

```
%{
    int count = 0;
    void wordcount();
%}

letter [a-zA-Z]

%%

{letter}{letter}* {wordcount();}

[\t]+ ;

[\n] ;

%%

int main()
{
    yylex();

    printf("Number of words are : %d", count);
    return 0;
}

void wordcount()
{
    count++;
}

int yywrap()
{
    return(1);
}
```

Implement word count program using LEX. Input filename as command line argument.

lex.l

```
%{
    int count = 0;
%}

letter [a-zA-Z]

%%
```

```
{letter}{letter}* {count++;}

[\t]+ ;

[\n] ;

%%

int main(int argc, char *argv[])
{
    yylex();
    yyin = fopen(argv[1], "rb+");
    printf("Number of words are : %d", count);
    return 0;
}

int yywrap()
{
    return(1);
}
```

Implement lexical analyzer for subset of english language using LEX. Build symbol table to dynamically declare and lookup parts of speech.

lex.l

```
%{
    // Word Recognizer with symbol table

enum{
    LOOKUP = 0, //Default - looking rather than defining
    VERB,
    NOUN,
    PREP,
    CONJ
};

int state;

int add_word(int type, char *word);
int lookup_word(char *word);

%}

%%

\n      {state = LOOKUP; }
^verb   { state = VERB; }
```

```
^noun   { state = NOUN; }
^prep   { state = PREP; }
^conj   {state = CONJ; }

[a-zA-Z]+ {
                if(state!=LOOKUP){
                    add_word(state, yytext);
                } else {
                    switch(lookup_word(yytext)){
                        case VERB: printf("%s : verb\n", yytext); break;
                        case NOUN: printf("%s : noun\n", yytext); break;
                        case PREP: printf("%s : preposition\n", yytext);
break;
                        case CONJ: printf("%s : conjunction\n", yytext);
break;
                        default:
                            printf( "%s : doesn't recognize\n", yytext) ;
                            break;
                    }
                }
        }
. ;

%%

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

// define a linked list of words and types

struct word {
    char *word_name;
    int word_type;
    struct word *next;
};

struct word *word_list;
extern void *malloc();

int add_word(int type, char *word)
{
    struct word *wp;

    if(lookup_word(word) != LOOKUP){
```

```
        printf("Word %s Already defined", word);
        return 0;
    }

    // word not there, allocate a new entry and link it onto the list

    wp = (struct word *) malloc(sizeof(struct word));

    wp->next = word_list;

    // have to copy the word itself as well

    wp->word_name = (char *) malloc(strlen(word)+1);
    strcpy(wp->word_name, word);
    wp->word_type = type;
    word_list = wp;
    return 1;
}

int lookup_word (char *word){
        struct word *wp = word_list;
        /* search down the list looking for the word */
        for(; wp; wp = wp->next) {
            if(strcmp(wp->word_name, word) == 0)
                return wp->word_type;
        }
        return LOOKUP; /* not found */
}

int yywrap()
{
    return 1;
}
```

Implement a lexical analyzer to input 'C' program file and a)Count number of comments b) Eliminate comments and c) Store output in another file

lex.l

```
%{
    int count = 0;
%}

%%

"//".*\n            { count++; }
"/*"[^*/]*"*/"      { count++; }
```

```
%%

int main()
{
    yyin=fopen("input.txt","r");
    yyout=fopen("output.txt","w");
    yylex();
    fprintf(yyout, "\n\n NOTE: Number of comments removed : %d", count);
    return 0;
}

int yywrap()
{
    return 1;
}
```

Implement a lexical analyzer to input 'C' program file and a)Count number of simple and compound statements . Input filename as command line argument.

lex.l

```
%{
    int compound = 0;
    int simple = 0;
%}

%%

for\([^"\n]*\)\{[^"\n]*\} |
if\([^"\n]*\)\{[^"\n]*\} |
if\([^"\n]*\)\{[^"\n]*\}\nelse\{[^"\n]*\} |
while\([^"\n]*\)\{[^"\n]*\} |
do\{[^"\n]*\}\nwhile\([^"\n]*\) {compound++;}

. ;

[\n] ;

[\t] ;
%%

int main()
{
    yyin=fopen("input.txt","r");
    yylex();
    printf("NOTE: Number of compound statements are : %d", compound);
    return 0;
}
```

```
int yywrap()
{
    return 1;
}
```

Write a YACC specification to implement arithmetic calculator.

lex.l

```
%{
#include "y.tab.h"
#include<stdio.h>
#include<math.h>
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
{yylval.fval=atof(yytext);return NUMBER;}

[ \t ]  ;  /*ignore whitespace*/
\n  return END;
.   return yytext [0];
%%
```

yacc.y

```
%{
   #include<stdio.h>
   #include<math.h>
%}

%union
{
   float fval;
}


%token<fval>NUMBER
%token END
%left '+' '-'
%left '*' '/'


%type<fval>expression
```

```
%%
statement:
expression END  {printf(" = %.4f \n ", $1); return 0;}
;
expression:
expression '+' expression {$$ = $1 + $3;}
|expression '-' expression {$$ = $1 - $3;}
|expression '*' expression {$$ = $1 * $3;}
|expression '/' expression { if($3==0){printf("Divide by zero not
allowed!"); return 0;}
            else{
            $$ = $1 / $3;}}
|NUMBER {$$=$1;}
%%

int main()
{
   yyparse();
}

int yyerror(char* s)
{
   printf("%s\n",s);
   return 0;
}

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

Write a YACC specification to implement scientific calculator.

lex.l

```
%{
#include "y.tab.h"
#include<stdio.h>
#include<math.h>
%}

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
{yylval.fval=atof(yytext);return NUMBER;}
[ \t ]   ;   /*ignore whitespace*/
sqrt |
SQRT { return SQRT; }
log |
```

```
LOG {return LOG;}
sin |
SIN {return SINE;}
cos |
COS {return COS;}
tan |
TAN {return TAN;}
cosec |
COSEC {return COSEC;}
sec |
SEC {return SEC;}
cot |
COT {return COT;}

\n  return END;
.   return yytext [0];
%%
```

yacc.y

```
%{
   #include<stdio.h>
   #include<math.h>
%}

%union
{
   float fval;
}


%token<fval>NUMBER
%token END
%token SQRT
%token LOG SINE COS TAN COSEC SEC COT
%left '+' '-'
%left '*' '/'
%right '^'
%left SQRT
%nonassoc UMINUS
%left LOG SINE COS TAN COSEC SEC COT


%type<fval>expression



%%
```

```
statement:
expression END  {printf(" = %.4f \n ", $1); return 0;}
;
expression:
expression '+' expression {$$ = $1 + $3;}
|expression '-' expression {$$ = $1 - $3;}
|expression '*' expression {$$ = $1 * $3;}
|expression '/' expression { if($3==0){printf("Divide by zero not
allowed!"); return 0;}
            else{
            $$ = $1 / $3;}}
|expression '^' expression {$$=pow($1,$3);}
|SQRT expression {$$=sqrt($2);}
|'-'expression %prec UMINUS{$$=-$2;}
|LOG expression {$$=log($2)/log(10);}
|SINE expression {$$=sin($2*3.14/180);}
|COS expression {$$=cos($2*3.14/180);}
|TAN expression {$$=tan($2*3.14/180);}
|COSEC expression {$$=1/(sin($2*3.14/180));}
|SEC expression {$$=1/(cos($2*3.14/180));}
|COT expression {$$=1/(tan($2*3.14/180));}
|NUMBER {$$=$1;}
%%

int main()
{
   yyparse();
}

int yyerror(char* s)
{
   printf("%s\n",s);
   return 0;
}

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

Write a YACC specification to check syntax of "for" statement of 'C' language.

lex.l

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

alpha [A-Za-z]
```

```
digit [0-9]

%%
[ \t\n]
for return FOR;
{digit}+ return NUM;
{alpha}({alpha}|{digit})* return ID;
"<=" return LE;
">=" return GE;
"==" return EQ;
"!=" return NE;
"||" return OR;
"&&" return AND;
"++" return INCR;
"--" return DECR;
. return yytext[0];
%%
int yywrap()
{
    return 1;
}
```

yacc.y

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

%token ID NUM FOR LE GE EQ NE OR AND INCR DECR
%right '='
%left AND OR
%left '<' '>' LE GE EQ NE
%left '+''-' INCR DECR
%left '*''/'
%nonassoc UMINUS
%left '!'
%%
S : ST {printf("Input accepted.\n");exit(0);};

ST : FOR '('E3';'E2';'E4')' '{' ST1';''}';

ST1 : ST
| E

;
E : ID'='E
| E'+'E
```

```
| E'-'E
| E'*'E
| E'/'E
| E'<'E
| E'>'E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| ID
| NUM
;
E4 : ID INCR
| ID DECR

E2 : E'<'E
| E'>'E
| E LE E
| E GE E
;

E3 : ID'='NUM
;
%%
int main()
{
printf("Enter the expression : \n");
yyparse();
}
void yyerror()
{
printf("Input rejected");
}
```

Write a YACC specification to check syntax of "switch… case" statement of 'C' language.

lex.l

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

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

%%
[ \t\n]
```

```
do return DO;
while return WHILE;
{digit}+ return NUM;
{alpha}({alpha}|{digit})* return ID;
"<=" return LE;
">=" return GE;
"==" return EQ;
"!=" return NE;
"||" return OR;
"&&" return AND;
"++" return INCR;
"--" return DECR;
. return yytext[0];
%%
int yywrap()
{
    return 1;
}
```

yacc.y

```
%{
    #include <stdio.h>
    #include <stdlib.h>
%}
%token ID NUM DO LE GE EQ NE OR AND WHILE
%right '='
%left AND OR
%left '<' '>' LE GE EQ NE
%left '+''-' INCR DECR
%left '*''/'
%right UMINUS
%left '!'
%%
S : ST {printf("Input accepted.\n");exit(0);};

ST : DO '{' ST1';''}' WHILE '(' E2 ')';
| WHILE'(' E2 ')' '{' ST1 ';''}';

ST1 : ST
| E

;
E : ID'='E
| E'+'E
| E'-'E
| E'*'E
| E'/'E
```

```
| E'<'E
| E'>'E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| ID
| NUM
;

E2 : E'<'E
| E'>'E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| ID
| NUM
;
%%
int main()
{
printf("Enter the expression : \n");
yyparse();
}
void yyerror()
{
printf("Input rejected");
}
```

Write a YACC specification to check the syntax of "if" and "if … else" statements of 'C' language.

lex.l

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

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

%%
[ \t\n]
if return IF;
else return ELSE;
```

```
{digit}+ return NUM;
{alpha}({alpha}|{digit})* return ID;
"<=" return LE;
">=" return GE;
"==" return EQ;
"!=" return NE;
"||" return OR;
"&&" return AND;
. return yytext[0];
%%
int yywrap()
{
    return 1;
}
```

yacc.y

```
%{
    #include <stdio.h>
    #include <stdlib.h>
%}
%token ID NUM IF LE GE EQ NE OR AND ELSE
%right '='
%left AND OR
%left '<' '>' LE GE EQ NE
%left '+''-'
%left '*''/'
%right UMINUS
%left '!'
%%
S : ST {printf("Input accepted.\n");exit(0);};

ST : IF '(' E2 ')' '{' ST1';''}' ELSE '{' ST1';''}';
| IF '(' E2 ')' '{' ST1 ';''}';

ST1 : ST
| E


;
E : ID'='E
| E'+'E
| E'-'E
| E'*'E
| E'/'E
| E'<'E
| E'>'E
| E LE E
| E GE E
```

```
| E EQ E
| E NE E
| E OR E
| E AND E
| ID
| NUM
;

E2 : E'<'E
| E'>'E
| E LE E
| E GE E
| E EQ E
| E NE E
| E OR E
| E AND E
| ID
| NUM
;
%%
int main()
{
printf("Enter the expression : \n");
yyparse();
}
void yyerror()
{
printf("Input rejected");
}
```

Program to count number of scanf and printf statement in a “C” program & replace them with readf and writef statements.

lex.l

```
%{
    #include<stdio.h>
    #include<string.h>

    char replace_printf [] = "writef";
    char replacep [] ="printf";

    char replace_scanf [] = "readf";
    char replaces [] ="scanf";

%}

%%
```

```
[a-zA-Z]+ { if(strcmp(yytext, replacep)==0)
                fprintf(yyout,"%s",replace_printf);
            else if(strcmp(yytext, replaces)==0)
                fprintf(yyout,"%s",replace_scanf);
            else
                    fprintf(yyout, "%s", yytext);
                    }


.   fprintf(yyout, "%s", yytext);

%%

int main(int argc, char *argv[])
{
    yyin = fopen(argv[1], "r");
    yyout = fopen("output.txt", "w");

    yylex();
    return 0;
}

int yywrap()
{
    return 1;
}
```

Program to count number of comment lines in a given C program. Also eliminate them and copy that program into separate file.

lex.l

```
%{

%}
start \/\*
end  \*\/

/*Rule Section*/
%%

\/\/(.*) ;

{start}.*{end} ;

%%

/*Driver function*/
int main(int k,char **argcv)
```

```
{
    yyin=fopen(argcv[1],"r");
    yyout=fopen("out.txt","w");
    int yylex();
    return 1 ;
}

int yywrap()
{
    return 1;
}
```

Program to count number of
(1) Positive and negative integers
(2) Positive and negative fractions
Program to count number of vowels and consonants in a given string.

lex.l

```
digit [0-9]

%{
    //headers
    int neg = 0;
    int pos = 0;
%}

%%

^[-][0-9]+ {neg++;
                    printf("negative number = %s\n",
                                    yytext);}


[0-9]+ {pos++;
                    printf("negative number = %s\n",
                                                    yytext);}

[\t]+ ;

%%

int main(){

    yylex();

    printf("The number of negative integers are %d, and the number of
positive integers are %d", neg, pos);
```

```
    return 0;
}

int yywrap()
{
    return 1;
}
```

Write a YACC specification to implement calculator.Extend to handle variables with single letter namesWrite a YACC specification to implement calculator.Extend to handle variables with single letter names

lex.l

```
%option noyywrap
%{
#include "y.tab.h"
#include <math.h>
extern double vbltable[26];
%}
%%

([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {yylval.dval=atof(yytext);
return NUMBER; }
[a-z] {yylval.vblno = yytext[0] - 'a'; return NAME;}
[\t] ;
\$ return 0;
\n|. return yytext[0];
%%
```

yacc.y

```
%{
#include <stdio.h>
#include <math.h>
double memvar;
double vbltable[26];
%}

%union{
    double dval;
    int vblno;
}

%token <vblno> NAME
%token <dval> NUMBER
%token <dval> MEM
%left '-' '+'
```

```
%left '*' '/' '%'
%right '^'
%nonassoc UMINUS

%type <dval> expression

%%
start: statement'\n' | start statement'\n';

statement: NAME'='expression {vbltable[$1]=$3;} | expression {printf(" =
%g\n",$1);}
;
expression: expression'+'expression{$$=$1+$3;}
| expression'-'expression{$$=$1-$3;}
| expression'*'expression{$$=$1*$3;}
| expression'/'expression{$$=$1/$3;}
| expression'%'expression{$$=fmod($1,$3);}
| expression'^'expression{$$=pow($1,$3);}
| '('expression')'{$$=$2;}
| '-'expression %prec UMINUS{$$=-$2;}
| NUMBER {$$=$1;}
| NAME {$$=vbltable[$1];}
;
%%

int main()
{
printf("Enter a mathematical expression: ");
yyparse();
}
int yyerror(char *error){
printf("%s\n", error);
}
```

Program to count number of identifiers in a given input file

lex.l

```
%{
    int identifier=0;
%}

digit [0-9]
letter [a-zA-Z_]

%%

[ \t]+ ;
```

```
int |
float |
double |
String |
char |
if |
for |
else |
do |
while |
printf |
static |
void |
public ;

; |
, |
\" |
\' |
: |
\{ |
\} |
\( |
\) ;

\+ |
\- |
\* |
\/ |
\^ ;

\= |
\> |
\< |
\|\| |
\&\& |
\>\= |
\<\= |
\! |
\!\= ;

{digit} |
{digit}.{digit} ;

\"[^"\n]*\" ;
```

```
{letter}({letter}|{digit})* {printf("Identifier - %s\n",yytext);
                            identifier++;}

[\n] ;


%%

int main(){


    yyin = fopen("input.txt", "r");
    yylex();
    printf("The number of identifiers are : %d", identifier);
    return 1;

}

int yywrap(){
    return 1;
}
```

Constant Folding and Propagation

python

```
def read_quadruples():
    '''returns list of [op,arg1,arg2,res] objects read from file.'''
    with open(r'Assignment 6/quadruples_2.txt') as file:
        lines = [line.strip() for  line in file.readlines()]
        quadruples = [line.split(',') for line in lines]
    return quadruples

def display_table(table):
    print(f"\n{'INDEX':{10}}{'OP':{10}}{'ARG1':{10}}{'ARG2':{10}}{'RES':{10}}")
    for index,line in enumerate(table):
        print(f"{index:<{10}}{line[0]:<{10}}{line[1]:{10}}{line[2]:{10}}{line[3]:{10}}")

def display_code(quadruples):
    print()
    for quadruple in quadruples:
        if ' ' in quadruple:
            print(f'{quadruple[3]} {quadruple[0]} {quadruple[1]}')
        else:
            print(f'{quadruple[3]} = {quadruple[1]} {quadruple[0]} {quadruple[2]}')

def optimize():
    global quadruples
    values = {}

    for index, entry in enumerate(quadruples):
        if entry[0] == '=' and entry[1].isnumeric():
```

```
            values[entry[3]] = int(entry[1])
        elif entry[0] == '=' and entry[3] in values:
            del values[entry[3]]

        if entry[1] in values:
            entry[1] = str(values[entry[1]])
        if entry[2] in values:
            entry[2] = str(values[entry[2]])
        if entry[2] != ' ':
            if entry[1].isnumeric() and entry[2].isnumeric():
                value = eval(entry[1]+entry[0]+entry[2])
                values[entry[3]] = value
                entry[0] = '='
                entry[1] = str(value)
                entry[2] = ' '


if __name__ == '__main__':
    quadruples = read_quadruples()
    print('INPUT:')
    display_code(quadruples)
    display_table(quadruples)
    optimize()
    print('\n\nOUTPUT:')
    display_code(quadruples)
    display_table(quadruples)
```

Input:

```
=,10, ,x
=,20, ,c
=,x, ,v
+,x,c,t1
=,t1, ,d
*,c,20,t2
=,t2, ,b
=,n, ,d
+,d,5,t3
=,t3, ,y
```

Do while

lex.l

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

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

%%

do {return DO;}

while {return WHILE;}

"==" |
```

```
"<=" |
">=" |
">" |
"<" |
"!" |
"!=" {return RELOP;}

{digit}+ {return NUM;}

{alpha}({alpha}|{digit})* {return ID;}

. {return yytext[0];}

%%

int yywrap(){
    return 1;
}
```

yacc.y

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

%token DO WHILE RELOP NUM ID

%%

statement : expression {printf("Input accepted!");}
;

expression : WHILE '(' condition ')' '{' action ';' '}'
| DO '{' action ';' '}' WHILE '(' condition ')'
;

condition : ID RELOP ID
| ID RELOP NUM
| NUM RELOP ID
;

action : action '+' action
| action '-' action
| action '*' action
| action '=' action
| ID
```

```
| NUM
;

%%

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

int yyerror(){
    printf("Input rejected");
}
```

Lexical analyzer python

#design a lexical analyzer for any java language program

#importing libraries
import re
import pandas as pd

#define keywords
keywords = ['import',
'int',
'null',
'char',
'double',
'float',
'extends',
'case',
'continue',
'do',
'if',
'else',
'for',
'return',
'while',
'enum',
'finally',
'final',
'implements',
'this',
'throws',
'try',
'class',
'public',
'static',
'void']

#define delimiters
delimiters = ['(', ')', '{', '}', '[', ']', ';', """," ", "",",","<",">"]

#define operators
operators = ['>=', '<=', '++', '--', '!=', '&&', '||', '==', "->", '=', '+', '-', '*', '/', ">", "<", '^', "."]

```
#identifiers regex
identifier_regex =re.compile("^([a-zA-Z_$][a-zA-Z\\d_$]*)$")

num = re.compile("^[0-9]+$")

symtab = []
linelist = []
lexemelist = []
tokenlist = []
tokenidlist = []

def check(lno, word):
    global identifier_regex, num, operators, keywords, symtab
    curr_word = ""
    if word =="" :
        return()
    elif re.search(num,word) :
        linelist.append(lno)
        lexemelist.append(word)
        tokenlist.append("Constant")
        tokenidlist.append(f"C,{word}")


    elif word in keywords :
        ind = keywords.index(word)
        linelist.append(lno)
        lexemelist.append(word)
        tokenlist.append("Keyword")
        tokenidlist.append(f"KW,{ind}")


    elif word in operators :
        ind = operators.index(word)
        linelist.append(lno)
        lexemelist.append(word)
        tokenlist.append("Operator")
        tokenidlist.append(f"OPR,{ind}")

    elif bool(re.search(identifier_regex,word)):
        if word not in symtab:
            symtab.append(word)
        ind = symtab.index(word)
        linelist.append(lno)
        lexemelist.append(word)
        tokenlist.append("Identifier")
        tokenidlist.append(f"ID,{ind}")
    else:
        cur = ""
        curr_word=""
        for ch in word:
            if ch in operators:
                if cur == "op" :
                    curr_word+=ch
                else:
                    if curr_word != "" :
                        check(lno, curr_word)
                    cur = "op"
                    curr_word = ch
            elif ch.isalpha() or ch.isnumeric():
                if cur == "Id" :
```

```
                curr_word+=ch
            else:
                if curr_word != "" :
                    check(lno, curr_word)
                cur = "Id"
                curr_word = ch
    if curr_word==word:
        # for i in output:
        #     print(i)
        print("\nERROR AT LINE : ",lno," at : ",word)
        exit(0)
    if curr_word!="" : check(lno,curr_word)


#running out script
file = open("code.java","r")
word=""
IC = ""
flag = 0
line_number=0
quotes = False
for line in file:
    # print(line, line_number)
    IC = ""
    for ch in line.lower().strip("\n").strip("\t"):
        if ch in delimiters:
            if word!="":
                if flag == 1:
                    linelist.append(line_number)
                    lexemelist.append(word)
                    tokenlist.append("Constant")
                    tokenidlist.append(f"C,{word}")
                else:
                    check(line_number,word)
            word=""
            if ch == " ":
                continue
            else:
                ind = delimiters.index(ch)
                linelist.append(line_number)
                lexemelist.append(ch)
                tokenlist.append("Delimiter")
                tokenidlist.append(f"DL,{ind}")
                if ch == delimiters[9] :
                    if flag == 0:
                        flag = 1
                    elif flag == 1:
                        flag = 0
        else:
            word+=ch
    if word!="":
        check(line_number,word)
        word=""
    line_number+= 1

df = pd.DataFrame({'Line Number' : linelist, 'Lexeme' : lexemelist, 'Token' : tokenlist, 'Token ID':
tokenidlist})

from tabulate import tabulate
print(tabulate(df, headers='keys', tablefmt='psql'))
```

```
print("Symbol Table")
print("Index\t\tSymbol")
for i in range(len(symtab)):
    print(i+1,"\t\t" ,symtab[i])
```

COMMANDS:

```
Yacc
yacc -d <name>.y
lex <name>.l
gcc lex.yy.c y.tab.c -c -lm
gcc lex.yy.o y.tab.c -o source
./source
```