Experiment 1
Tokenization using C

Aim: (Tokenizing) Use C programming to extract tokens from a given source code.

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

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis

Lexical analysis is the first phase of a compiler. It takes 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.

CODE:
C File:

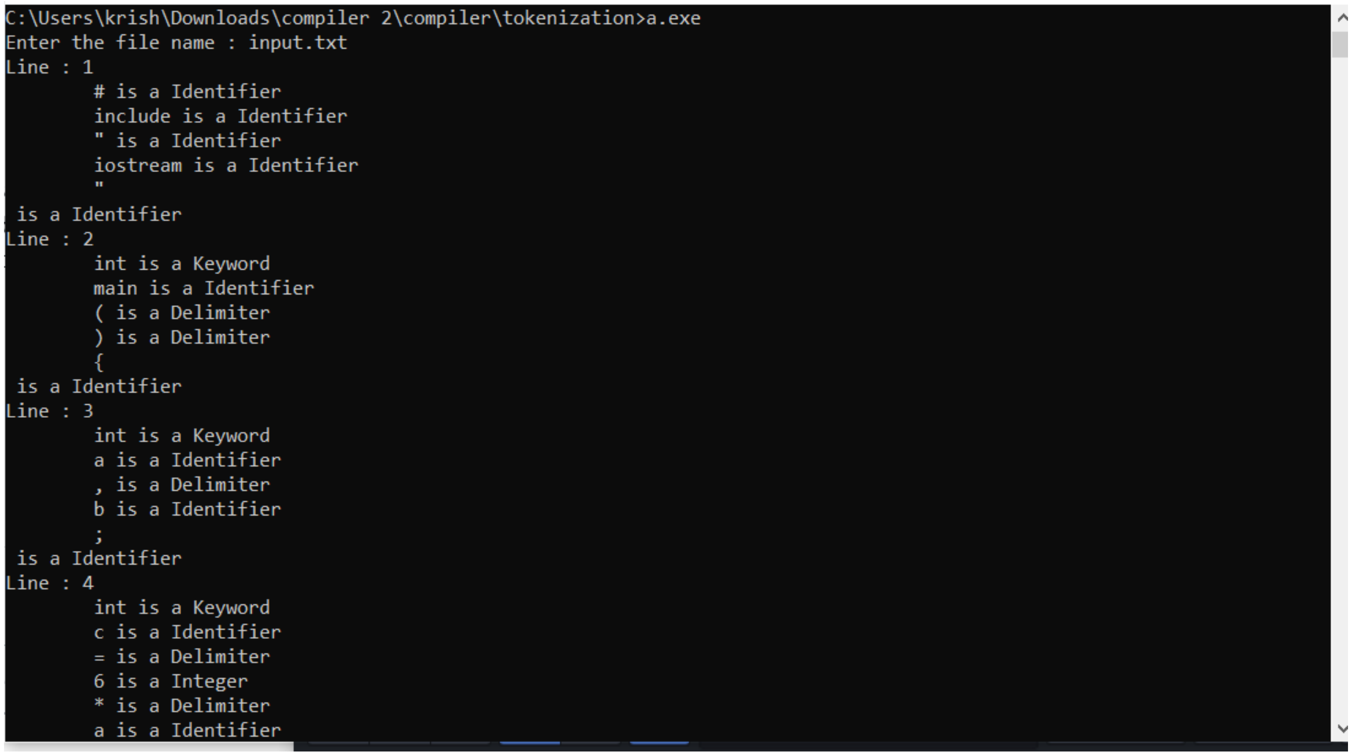
```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

bool delimiter(char* s) {
   if (!strcmp(s," ") || !strcmp(s,"+") || !strcmp(s,"-") || !strcmp(s,"*") || !strcmp(s,"/") || !strcmp(s,",")
|| !strcmp(s,";") || !strcmp(s,">") || !strcmp(s,"<") || !strcmp(s,"=") || !strcmp(s,"(") || !strcmp(s,")") ||
!strcmp(s,"[") || !strcmp(s,"]") || !strcmp(s,"{") || !strcmp(s,"}")) {
       return true;
   }
   return false;
}

bool keyword(char* s) {
   if (!strcmp(s, "if") || !strcmp(s, "else") || !strcmp(s, "while") || !strcmp(s, "do") || !strcmp(s,
"break") || !strcmp(s, "continue") || !strcmp(s, "int") || !strcmp(s, "double") || !strcmp(s, "float") ||
!strcmp(s, "return") || !strcmp(s, "char") || !strcmp(s, "case") || !strcmp(s, "char") || !strcmp(s,
"sizeof") || !strcmp(s, "long") || !strcmp(s, "short") || !strcmp(s, "typedef") || !strcmp(s, "switch") ||
!strcmp(s, "unsigned") || !strcmp(s, "void") || !strcmp(s, "static") || !strcmp(s, "struct") || !strcmp(s,
"goto")) {
       return true;
```

```
    }
    return false;
}

bool operator(char ch) {
    if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '>' || ch == '<' || ch == '=') {
        return true;
    }
    return false;
}

bool identifier(char* s) {
    if (s[0] == '0' || s[0] == '1' || s[0] == '2' || s[0] == '3' || s[0] == '4' || s[0] == '5' || s[0] == '6' || s[0]
== '7' || s[0] == '8' || s[0] == '9') {
        return false;
    }
    return true;
}

bool integer(char* s) {
    int i, len = strlen(s);
    if (len == 0)
        return false;
    for (i = 0; i < len; i++) {
        if (s[i] != '0' && s[i] != '1' && s[i] != '2' && s[i] != '3' && s[i] != '4' && s[i] != '5' && s[i] != '6' &&
s[i] != '7' && s[i] != '8' && s[i] != '9' || (s[i] == '-' && i > 0)) {
            return false;
        }
    }
    return true;
}

int main() {
    printf("Enter the file name : ");
        char filename[30];
    scanf("%[^\n]%*c", filename);
        FILE* myfile = fopen("input.txt", "r");
        if (NULL == myfile) {
                printf("File can't be opened \n");
        }
    char codeline[50];
    int line_number=1;
    while(fgets(codeline, 50, myfile)!=NULL) {
        printf("Line : %d\n",line_number);
        char* s;
        char* rest = codeline;
        while ((s = strtok_r(rest, " ", &rest))) {
        if(s=="") {
            continue;
        }
        printf("\t");
        if(delimiter(s))
            printf("%s is a Delimiter",s);
        else if(keyword(s))
            printf("%s is a Keyword",s);
        else if(operator(s))
            printf("%s is a Operator",s);
        else if(identifier(s))
            printf("%s is a Identifier",s);
        else if(integer(s))
            printf("%s is a Integer",s);
        printf("\n");
        }
        line_number++;
    }
        fclose(myfile);
    printf("\n");
        return 0;
}
```

INPUT:

```
# include " iostream "
int main ( ) {
    int a , b ;
    int c = 6 * a + b ;
    return 0 ;
}
```

OUTPUT:


```
C:\Users\krish\Downloads\compiler 2\compiler\tokenization>a.exe
Enter the file name : input.txt
Line : 1
        # is a Identifier
        include is a Identifier
        " is a Identifier
        iostream is a Identifier
        "
 is a Identifier
Line : 2
        int is a Keyword
        main is a Identifier
        ( is a Delimiter
        ) is a Delimiter
        {
 is a Identifier
Line : 3
        int is a Keyword
        a is a Identifier
        , is a Delimiter
        b is a Identifier
        ;
 is a Identifier
Line : 4
        int is a Keyword
        c is a Identifier
        = is a Delimiter
        6 is a Integer
        * is a Delimiter
        a is a Identifier
```

Experiment 2
Tokenization using Lex Tool

Aim: (Tokenizing) Use LEX to extract tokens from a given source code.
THEORY:

LEXICAL TOOLS

- A language for specifying lexical analyzer.
- There is a wide range of tools for construction of lexical analyzers. The majority of these tools are based on regular expressions.
- One of the traditional tools of that kind is lex.

LEX:

- The lex is used in the manner depicted. A specification of the lexical analyzer is preferred by creating a program lex.1 in the lex language.
- Then lex.1 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.1 together with a standard routine that uses a table of recognized leximes.
- Lex.yy.c is run through the ‘C’ compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into sequence of tokens.

Code:

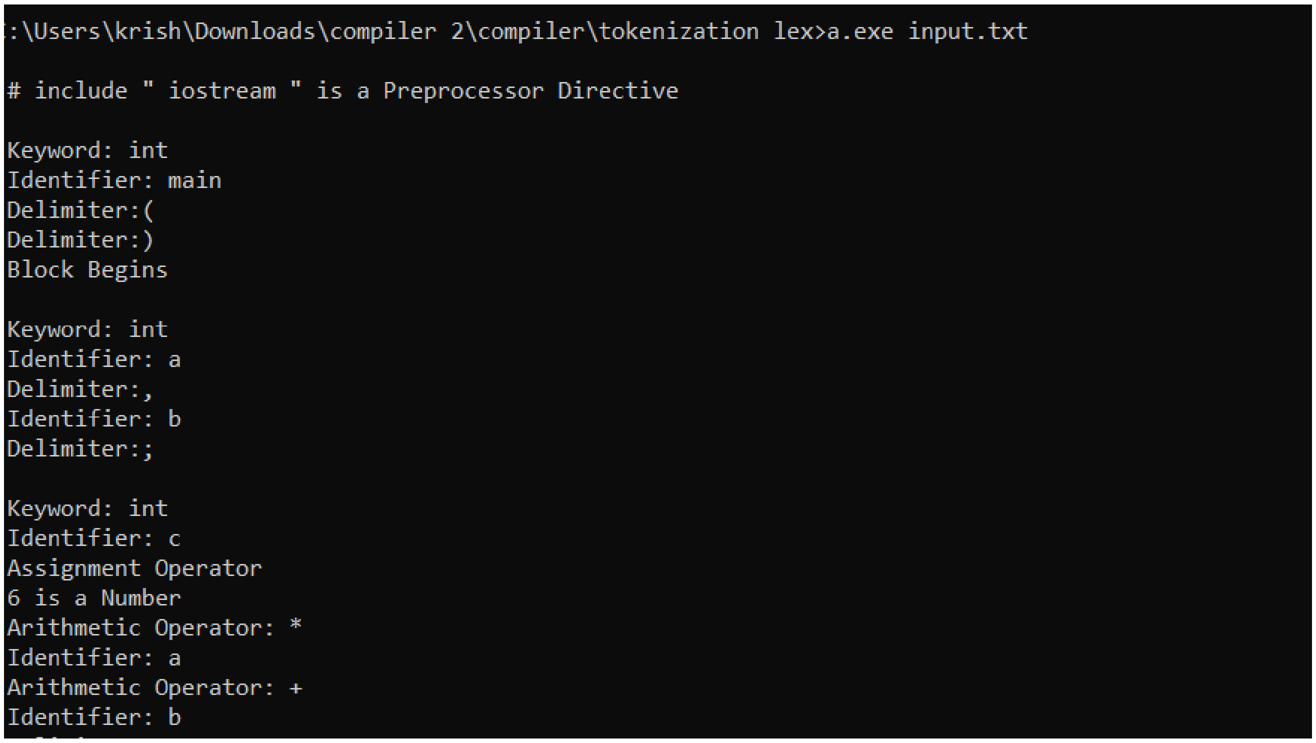
```
%{
  int commentPresent = 0;
%}

%%
#.* {printf("\n %s is a Preprocessor Directive",yytext);}
if|else|while|do|break|continue|int|double|float|return|char|case|long|short|typedef|switch|unsigne
d|void|static|struct|goto|for { printf("\n Keyword: %s ",yytext);}
"/*" {commentPresent = 1;}
"*/" {commentPresent = 0;}
[a-zA-Z][a-zA-Z0-9]*(\[[0-9]*\])? {if(!commentPresent) printf("\n Identifier: %s",yytext);}
"+"|"-"|"*"|"/" {printf("\n Arithmetic Operator: %s ",yytext);}
","|";"|"("|")"|"["|"]" { printf("\n Delimiter:%s",yytext);}
"=" {printf("\n Assignment Operator");}
"{" {if(!commentPresent) printf("\n Block Begins");}
"}" {if(!commentPresent) printf("\n Block Ends");}

\".*\" {if(!commentPresent) printf("\n %s is a String",yytext);}
```

```
[0-9]+ {if(!commentPresent) printf("\n %s is a Number",yytext);}
"<=" | ">="| "<" | "==" {if(!commentPresent) printf("\n %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("\n Could not open the file: %s",argv[1]);
exit(0);
}
yyin=file;
}
yylex();
printf("\n\n");
return 0;
}
int yywrap()
{
return 0;
}
```

INPUT:

```
# include " iostream "
int main ( ) {
   int a , b ;
   int c = 6 * a + b ;
   return 0 ;
}
```

Output:



```
:\Users\krish\Downloads\compiler 2\compiler\tokenization lex>a.exe input.txt

# include " iostream " is a Preprocessor Directive

Keyword: int
Identifier: main
Delimiter:(
Delimiter:)
Block Begins

Keyword: int
Identifier: a
Delimiter:,
Identifier: b
Delimiter:;

Keyword: int
Identifier: c
Assignment Operator
6 is a Number
Arithmetic Operator: *
Identifier: a
Arithmetic Operator: +
Identifier: b
```

Experiment 3
Fortran Do Loop Program

Aim: Implement YACC for Subset of Fortran 'DO loop' program

THEORY:

Syntax Analysis:

- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.
- The output of this phase is a parse tree.

YACC:

- YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator.
- The YACC input file is divided into three parts.

```
/* definitions */
....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

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

- The rules part contains grammar definitions in a modified BNF form. Actions is C code in { } and can be embedded inside (Translation schemes).
- The auxiliary routines part is only C code.It includes function definitions for every function needed in rules part.It can also contain the main() function definition if the parser is going to be run as a program.The main() function must call the function yyparse().

CODE:

Lex File:

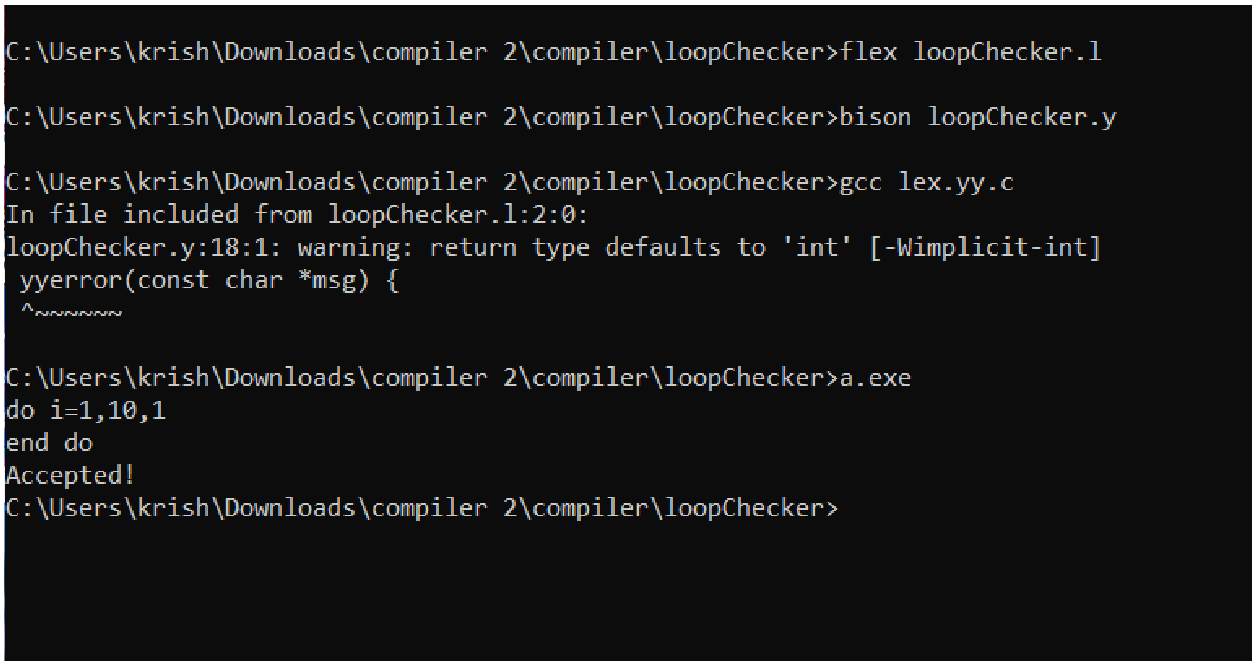
```
%{
   #include "loopChecker.tab.c"
   extern int yylval;
%}

%%
do { return (DO); }
"=" {return (EQU);}
"," {return (SEP);}
"end" {return (END);}
"\n" {return (NEW);}
[a-zA-Z]+ { yylval=yytext[0];return(ID); }
[0-9]+ { yylval=atoi(yytext);return(NUM); }
[\s\S]* {return (STAT);}
%%

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

YACC FILE:

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

```
    int yyerror();
%}

%token DO EQU SEP ID NUM END NEW STAT WS;

%%
S:  DO WS ID EQU E1 SEP E1 SEP E1 NEW END WS DO {printf("Accepted!"); flag=1;}
E1: ID | NUM ;
%%
int main() {
    yyparse();
    return 1;
}
yyerror(const char *msg) {
    if(flag==0) {
    printf("%s",yytext);
        printf("Not Accepted\n");
    }
}
```

OUTPUT:



```
C:\Users\krish\Downloads\compiler 2\compiler\loopChecker>flex loopChecker.l

C:\Users\krish\Downloads\compiler 2\compiler\loopChecker>bison loopChecker.y

C:\Users\krish\Downloads\compiler 2\compiler\loopChecker>gcc lex.yy.c
In file included from loopChecker.l:2:0:
loopChecker.y:18:1: warning: return type defaults to 'int' [-Wimplicit-int]
 yyerror(const char *msg) {
 ^~~~~~~

C:\Users\krish\Downloads\compiler 2\compiler\loopChecker>a.exe
do i=1,10,1
end do
Accepted!
C:\Users\krish\Downloads\compiler 2\compiler\loopChecker>
```

Experiment 4
Calculator using LEX and YACC

AIM: Create a digital Calculator using LEX and YACC tools

CODE:

LEX FILE:

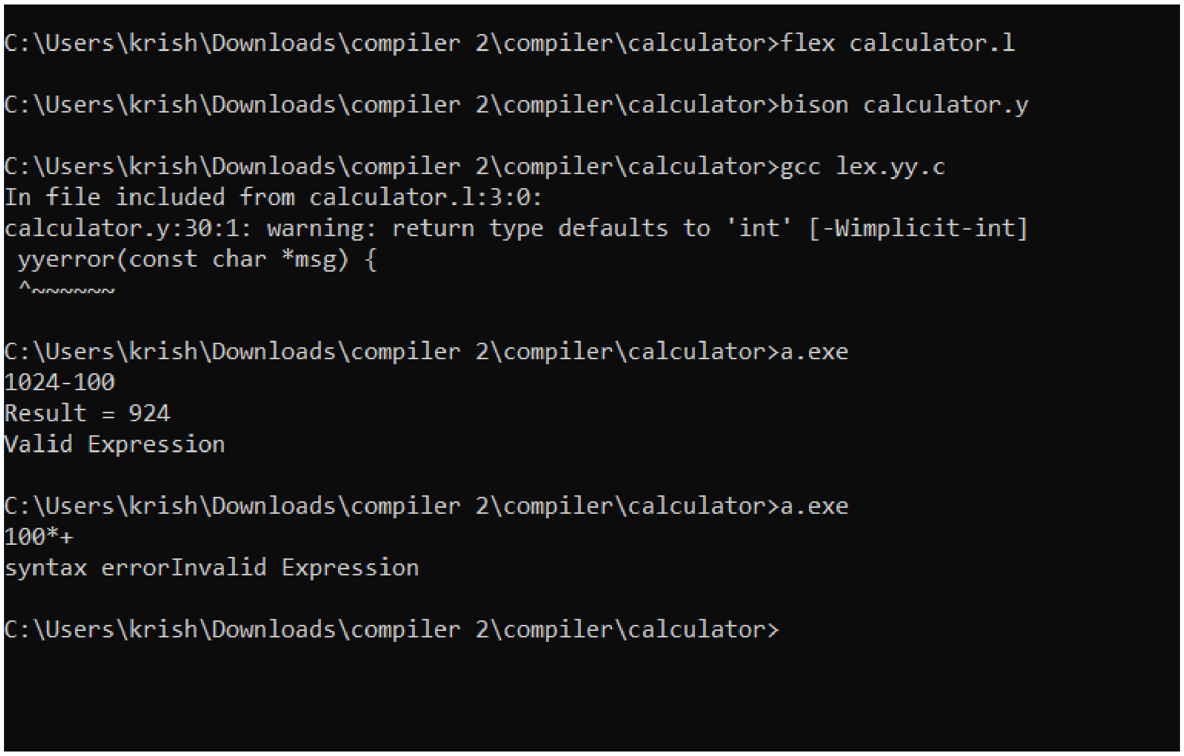
```
%{
   #include<stdio.h>
   #include "y.tab.c"
   extern int yylval;
%}

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

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

YACC FILE:

```
%{
   #include <stdio.h>
   int flag=0;
   int yylex();
   int yyerror();
%}

%token NUM
%left '+' '-'
%left '*' '/'
%left '(' ')'

%%
```

```
S:  E{printf("Result = %d\n", $$); flag=0; return 0;};
E:  E '+' E {$$=$1+$3;}
   |E '-' E {$$=$1-$3;}
   |E '*' E {$$=$1*$3;}
   |E '/' E {$$=$1/$3;}
   |'(' E ')' {$$=$2;}
   |NUM {$$=$1;}
%%

int main() {
    yyparse();
    if(flag==0)
        printf("Valid Expression \n");
    return 1;
}

yyerror(const char *msg) {
    printf(msg);
    printf("Invalid Expression \n");
    flag=1;
}
```

OUTPUT:



```
C:\Users\krish\Downloads\compiler 2\compiler\calculator>flex calculator.l

C:\Users\krish\Downloads\compiler 2\compiler\calculator>bison calculator.y

C:\Users\krish\Downloads\compiler 2\compiler\calculator>gcc lex.yy.c
In file included from calculator.l:3:0:
calculator.y:30:1: warning: return type defaults to 'int' [-Wimplicit-int]
 yyerror(const char *msg) {
 ^~~~~~~

C:\Users\krish\Downloads\compiler 2\compiler\calculator>a.exe
1024-100
Result = 924
Valid Expression

C:\Users\krish\Downloads\compiler 2\compiler\calculator>a.exe
100*+
syntax errorInvalid Expression

C:\Users\krish\Downloads\compiler 2\compiler\calculator>
```