# CS4100/5100 LEXICAL ANALYZER: Compiler Project Part 2
## COMPLETE SPECIFICATIONS
## Spring 2022

The Lexical Analyzer, which is Phase 2 of the Compiler Design Project, is a crucial part of the project, and it is essential that it be built to the following specifications in order to minimize problems during Phases 3 and 4. Like Phase 1, this Phase 2 of the project is to be written in Java.

**Purpose of Part 2**

The Lexical Analyzer class (*Lexical)* is to provide the following functionality. The main program will repeatedly call the *Lexical.GetNextToken* method, to produce these actions, in this exact order, to read through the entire input source code file:

1) Read a single text line from the file
2) If 'echo' is on, display that text line exactly as it came from the file (no changed spacing!)
3) Locate and identify each defined token on the current line, adding each IDENTIFIER, and constant (INTEGER, FLOAT, and STRING) to the symbol table.
   **Reserve words do not go into the symbol table! This seems to be a common misconception.**
4) Return the text *(lexeme)* and the integer code for the token, one token at a time
5) Repeat until the End of File is reached

The main program to be used will be provided on Canvas. It will look like:

```
public class main {

    public static void main(String[] args) {
        String filePath = "d:\\";
        final int numSymbols = 100;
        SymbolTable mySymbols;
        mySymbols = new SymbolTable(numSymbols);

        //Constructor for the Lexical Analyzer
        Lexical myLexer = new Lexical(filePath+"testlines.txt",
                        mySymbols, true);
        // The token class is declared as a public inner class
        // having String lexeme, int code, String mnemonic fields
        Lexical.token currToken;

        // Initial read to set up WHILE loop
        // Returns a null token at EOF
        currToken = myLexer.GetNextToken();
        while (currToken != null) {
          System.out.println(currToken.lexeme+" \t| "+currToken.code+
                " \t| "+currToken.mnemonic);
          currToken = myLexer.GetNextToken();
          }
        mySymbols.print();
}}
```

**For future reference,** ultimately, the Syntax Analyzer will NOT call GetNextToken in a loop, but instead will process each token when it is returned while constructing the parse tree. The loop in *main* is just for testing purposes for Part 2.

**Function and Structure of Lexical**

In order to receive full points for this program, it must conform to the following specifications:

1. The **main** program provided must be used and the *Lexical* class must be compatible.

2. The class is to be called *Lexical*, and must provide the following Java inner class called *token*, (code provided), defined as:

```
public class token {
    public String lexeme;
    public int code;
    public String mnemonic

    token() {
            lexeme = "";
            code = 0;
            mnemonic = ""; //A 5-char student-created ID for
                           //readability of the code integer
    }
}
```

3. Lexical's interface must provide a constructor and method *GetNextToken*. The constructor must have the signature:

```
public Lexical(String filename, SymbolTable symbols, boolean echoOn)
```

where the parameters are ***filename*** (the full path and name of the input file to be read), ***symbols*** (an initailized/constructed SymbolTable which will be populated by *GetNextToken* with the found identifiers and constants), and ***echoOn*** (controls whether lines from the input file are immediately printed ***with a line number***). Note that there will be file reading initialization, a number of variables (like storing the echoOn for later use in the class), a variable to reference the passed SymbolTable, and a ReserveTable that needs to be initialized with all the reserve words (and the other terminal tokens). The variables will be created as fields in the Lexical class. In addition, a separate ReserveTable must be initialized to provide a 5-character student-defined mnemonic string for each of the defined integer token codes (for example, "IDENT" for 51, the Identifier code).

The critical GetNextToken method (provided in skeletal form in the java file) has this interface and algorithm:

```
public token GetNextToken(){
        //Create the object to be returned
        token result = new token();
        char ch;
        //Skip whitespace and comments, returning next char
        ch = skipWhiteSpace();
        // Get correct token category based on ch
        if (isLetter(ch)) { //is ident
```

```
                    result = getIdent(ch);
              }
              else
                     if (isDigit(ch)) { //is numeric
                            result = getNumber(ch);
                     }
                     else
                            if (isStringStart(ch)) { //string literal
                                   result = getString(ch);
                            }
                          else  //default char checks
                            result = getOneTwoChar(ch);

       if ((result.lexeme.equals("")) ||(EOF))
              result = null;
       return result;
}
```

The code above is provided as a guideline because many students have difficulty envisioning the creation of the method based on the DFA diagram that was discussed in class. This code demonstrates using a logical approach and the use of other private methods to compress the GetNextToken code into an easy-reading, understandable method.

4. The methods called to get each token class within *GetNextToken* must be based directly on the DFA state diagram which was discussed in class.

5. Some of the specific tasks which must be done by *GetNextToken* or the methods it calls:

a) Reading new lines of input as needed, and immediately echo printing them as selected by *echoOn.*. **NOTE: Lexical must be line-oriented rather than stream oriented. Reading the entire file into some structure is NOT acceptable. The basic file input unit must be one entire line of source code, which is then scanned and tokenized by code written by the student, not by any library calls to tokenizing functions! This is not difficult, but it requires thought. Code is provided for this.**

b) Determining the corrrect integer *tokenCode* for the new token, which will be referenced by **main**, making calls to lookup *reserve words* as needed.

c) Placing each identifier or constant into the SymbolTable **only once**, making the entries case-insensitive. Identifiers which are reserved words must not be added to the symbol table, but should be identified by finding them in a pre-constructed ReservedWord table. Note that within the lexical analyzer program, there should be virtually NO INTEGER LITERALS in the code; instead, use named symbolic constants declared in a centrally accessible location.

d) *GetNextToken* must utilize well-designed auxiliary procedures to make it more readable and understandable. At a minimum, the ones shown in the sample code must be implemented.

e) *GetNextChar* must always return the next character available, including the start of whitespace, calling functions when needed to get the next line of input from the file (via *GetNextLine* which will handle echoing the line when needed, and will return '\n' when the line read has been exhausted). **Code provided.**

f) *SkipWhiteSpace* must call *GetNextChar* to read each character. It must skip spaces, tabs, newlines, and ALSO recognize and skip over all comments, whether they start with '{' or '(*' until the end of the comment with the matching '}' or '*)'. **Code Provided.**

**Errors detected**

GetNextToken should generate a warning message and take actions for the following **non-fatal errors**:

a) Identifier is longer than 30 characters; print warning message and truncate the token lexeme.

b) Numeric constant is longer than 12 characters; print warning and truncate the token lexeme.

c) Unclosed comment (only happens when end of file is reached before comment termination): print a warning message, 'End of file found before comment terminated.'; no other action needed.

d) Unterminated string (end of line or end of file reached before a " was found): print a warning message, "Unterminated string found."

e) Invalid character in file; returns the character as an undefined token (code 99), with no additional warning message needed.

**Code Provided by the Instructor**

The code for a basic Lexical class will be uploaded to Canvas. **READ THE INITIAL COMMENTS IN THE SOURCE CODE FOR THE LIST OF METHODS YOU MUST PROVIDE/UPDATE.** As 'delivered', the Lexical class you receive will do only the minimal (but important) tasks of initializing the input file, handling white-space, and provide the non-functioning place holder methods where the NFA-derived code must be provided by the student, as laid out in the comments at the top of the code. The student must provide their own SymbolTable and ReserveTable classes. After the provided file initialization takes place, the GetNextToken method provided, when called from the provided main, will simply return each individual character in the input file, with a token code of 0. ***Initially getting this result once the student's needed classes are properly imported, will ensure that the provided code is working and ready to be modified as needed to obtain the desired functionality.***

**Turn In Requirements**

Submit your java code for Lexical, the file dump for the completed SymbolTable, and the output files specified in the Canvas assignment to Canvas.