CS4100/5100 LEXICAL ANALYZER PROJECT
COMPLETE SPECIFICATIONS
Fall 2020

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. The following details the requirements.

Function and Structure of the Lexical Analyzer
In order to receive full points for this program, it must conform to the following specifications:

1. The **main** program must be conceptually structured like:

```
InitializeStructures;
InitializeInputFile(GetFileName);
while NOT END_OF_FILE {
        GetNextToken(echoOn);
        PrintToken(nextToken, tokenCode);
        }
SymbolTable.Print();
Terminate;
```

Only the *GetNextToken* name is critical here, the others are illustrative. The variables (or function returns/setters/getters) *nextToken* and *tokenCode* must be globally accessible/assignable to *GetNextToken* for future ease of use in syntax analysis. Plan ahead for making sure that the next available character and the end of file conditions are properly synched with *GetNextToken*.

2. The Lexical Analyzer must be encapsulated as an independent function called *GetNextToken* (abbreviated here sometimes as GNT). It must take one boolean input argument *which selects whether input* lines are echoed when read, as described below.

3. As apparent from the main pseudo-code above, a SymbolTable instance, and the variables/functions/getters *nextToken* and *tokenCode* must be globally accessible throughout the program, as the syntax analyzer will need them in assignment #3. **Design for this!**

4. Each time it is called, *PrintToken* must neatly print in straight columns (1) the lexeme, (2) the token code, (3) a table-looked-up 4-character mnemonic for that code (must use a ReserveTable instance), and, (4) for identifiers and literals added to the SymbolTable, the SymbolTable location index of the token.

5. *GetNextToken* must be based directly on the DFA state diagram which is to be handed in with the program. The diagram must use the same overall style as that discussed in class.

6. As its routine processing, if its *echoOn* parameter is true, *GetNextToken* must print each **unmodified** input line as it is read from the file, before processing the raw text. The token outputs are printed by the main program, so that the program output appears as:

the exact source line as it exists in the file, printed by GNT when it reads the line
token 1 info, printed by main
token 2 info, printed by main
etc.

the exact source line as it exists in the file, printed by GNT when it reads the line
token 1 info, printed by main
token 2 info, printed by main
etc.

7. Each time it is called, *GetNextToken* is responsible for the following:

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. The basic file input unit must be an 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.**

b) Recognizing the next token in the input, placing the lexeme in the globally accessible string *nextToken.*

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

d) Placing each identifier or numeric 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.

e) Setting a globally accessible boolean variable or function *ENDFILE* set to TRUE when the input file data is exhausted, so that the main program can check this in its loop stop condition.

f) *GetNextToken* must utilize well-designed auxiliary procedures to make it more readable and understandable. At a minimum, *GetNextChar, SkipBlanks, IsLetter, IsDigit, IsWhitespace* functions should be used to make the code clear.

g) *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* probably, which will handle echoing the line when requested).

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.

b) Numeric constant is longer than 30 characters; print warning and truncate.

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; return the character as an undefined token (code 99), with no additional warning message needed.

Turn In Requirements

See the Lexical Analyzer Grade Sheet for details on turn-in expectations and point values for each required element of the project.