Assignment 4

The goal of this assignment is to create a C program which:
-takes a text file as input
-tokenizes the given input
-Parses the tokenized input to determine if it is grammatically valid

This program will consist of four class files: Givens.c Tokenizer.c Parser.c and Analyzer.c . Each class file has a corresponding header file where all constants, method declarations, struct definitions, include statements, and global variables will be placed. (Header files are named the same as their corresponding class file but with a .h file extension rather than a .c file extension) Tokenizer.h and Parser.h will include Givens.h and Analyzer.h will include both Tokenizer.h and Parser.h .

Givens.c

Givens.c is provided with this assignment. Givens.c includes constants for TRUE and FALSE, an enum containing all token values in the given lexical structure, a constant for the max size of a lexeme, and the definition for a struct named lexics, which consists an enum token property named token and a character array property named lexeme. The lexics struct is used to store both a token and its corresponding lexeme. Givens.c also provides two functions which return a boolean value indicating if the given String matches a specified regular expression.

Tokenizer.c

Tokenizer.c is not provided and needs to be created. Tokenizer.c will read characters from a given FILE variable and convert them into tokens. It will do so using a function defined as follows:

```
_Bool tokenizer(struct lexics *aLex, int *numLex, FILE *inf);
```

Which takes an array of type lexics, an int pointer representing the number of tokens in the input file, and a pointer to a FILE. The tokenizer function will read characters from the given FILE parameter, creating lexemes and the associated tokens. Each time a lexeme is generated, a new lexics struct will be created and the lexeme added. The generated lexeme is then tokenized, the token is added to the generated lexics struct, the lexics struct is then added to the end of the given lexics array. (Note: another option is to generate lexemes first, then tokenize the generated lexemes)

The given lexical structure is free format and the location of tokens in the text file does not affect their meaning. Alphanumeric lexemes will be delimited by both whitespace and by character lexemes. Because character lexemes are used as delimiters, they cannot be constructed one token at a time. Rather the next several tokens in the file will need to be examined to determine which (if any) character lexeme is present.

The use of helper functions in the Tokenizer.c class is highly recommended. Once the tokenization process is complete, the tokenizer function should return TRUE. If there occurs an error in the process, the function should return FALSE.

Parser.c

Parser.c is not provided and needs to be created. Parser.c will implement a recursive decent parser based upon a provided EBNF grammar. It will do so using a function defined as follows:

```
_Bool parser(struct lexics *someLexics, int numberOfLexics);
```

Which takes an array of type lexics and an int pointer representing the number of tokens in the given lexics array. The parser method must take the tokens (given in the array of lexics structs) and determine if they are legal in the language defined by the given grammar. The purpose of our parser is to apply the grammar rules and report any syntax errors. If no syntax errors are identified, parser returns TRUE, otherwise it returns FALSE.

Parser.c must be a recursive decent predictive parser which utilizes single-symbol lookahead. Parsers which utilize multi-symbol lookahead will not be accepted. If given a grammatically valid input, every token given must be parsed. If a syntax error is found, parsing does not need to continue. Parsers which do not consume every given token for a grammatically valid input will not be accepted.

Analyzer.c

Analyzer.c is provided with this assignment. Analyzer.c includes a method to prompt the user for a file path, the initialization of an array of type lexics and an int containing the number of lexics structs in the array (initialized to 0). Analyzer.c makes calls both the tokenizer method and the parser method, passing the initialized int and array to both functions.

All four class files need to be compiled into a single executable file. When run, this executable file is expected to call the int main function defined in Analyzer.c with the tokenization and parsing functions being called from Analyzer.c 's int main. All programs will be graded with Gradescope, where you will submit Parser.c, Tokenizer.c, and their corresponding header files. The -std=c99 flag may be used when testing your code as I will be compiling with said flag when I grade your projects.

Final Notes

A set of text files will be provided for testing. There will be nine files which have syntactically valid inputs and nine which have syntactically invalid inputs. All tests are lexicographically valid. These 18 files are identical to the input data on Gradescope. If your program can successfully parse all 18 files, it should pass all tests on Gradescope.

There is no recommended development environment for this assignment, you are welcome to use whatever application you are most comfortable with. However, it is recommended to develop this program on either a Linux or macOS operating system. This is to ensure compatibility with both Gradescope and some of the helper functions in Givens.c . If you are on a Windows machine, I recommend the following options for development:

- The Windows Subsystem for Linux
- The compile.vcu.edu server
- The provided Docker container

Provided EBNF grammar:

```
function        --> header body
header          --> VARTYPE IDENTIFIER LEFT_PARENTHESIS [arg-decl] RIGHT_PARENTHESIS
arg-decl        --> VARTYPE IDENTIFIER {COMMA VARTYPE IDENTIFIER}
body            --> LEFT_BRACKET [statement-list] RIGHT_BRACKET
statement-list  --> statement {statement}
statement       --> while-loop | return | assignment | body
while-loop      --> WHILE_KEYWORD LEFT_PARENTHESIS expression RIGHT_PARENTHESIS statement
return          --> RETURN_KEYWORD expression EOL
assignment      --> IDENTIFIER EQUAL expression EOL
expression      --> term {BINOP term} | LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
term            --> IDENTIFIER | NUMBER
```

Provided lexical structure:

```
LEFT_PARENTHESIS      --> (
RIGHT_PARENTHESIS     --> )
LEFT_BRACKET          --> {
RIGHT_BRACKET         --> }
WHILE_KEYWORD         --> while
RETURN_KEYWORD        --> return
EQUAL                 --> =
COMMA                 --> ,
EOL                   --> ;
VARTYPE               --> int | void
IDENTIFIER            --> [a-zA-Z][a-zA-Z0-9]*
BINOP                 --> + | * | != | == | %
NUMBER                --> [0-9][0-9]*
```

Grading Rubric:

Category	Points
Functions and Classes properly defined	10
Header files correctly utilized	10
Tokenizer.c produces correct output	32.5
Parser.c produces correct output	32.5
Code is well commented	7.5
Code is well formatted	7.5
Total	100