1 Project Description

Consider the grammar of “mini-language”, provided in file “grammar.txt”, and implement a lexical analyzer (a.k.a. scanner, lexer or tokenizer) for it.

The scanner reads a file containing a sequence of characters and outputs the corresponding sequence of tokens (i.e., representations of terminal symbols), while omitting whitespace and comments.

If the scanner reads any character that is not allowed by the grammar, it should generate an appropriate error message (with the position of the character) and stop the computation. For instance, the scanner should reject @*t*, as @ is not included in the alphabet of identifiers.

For simplicity, you can assume that all integers are within range, i.e., you do not have to check for overflow.

Each token should include the following information:

1. The position of the lexeme. The position is a pair consisting of the line number of the lexeme and the position of the first character of the lexeme in that line.
2. The kind of the lexeme. To keep things simple, use strings for representing the kind of lexemes.

   We have five different lexemes. The following is the kinds of the lexemes:

   (a) For identifiers the kind is “ID”. For example, for identifier ’speed’, the kind is “ID”.

   (b) For integers (i.e., numbers), the kind is “NUM”. For example, for the integer 3400, the kind is “NUM”.

   (c) For keywords, the keyword itself is the kind. For example, for the keyword ’false’, the kind is “false”.

   (d) For other symbols, the kind is a string corresponding to the symbol. For example, for the symbol ’:=’ the kind is “:=”.

   (e) There is a special kind “end-of-text”. Upon encountering the end of the input file, the scanner must generate a token whose kind is “end-of- text”.
3. The value of the lexeme, if applicable. **Only identifiers and integers have values.** For example the value of lexeme ’19’ is **integer** 19 and the value of lexeme ’speed’ is string “speed”.

   Keywords and other symbols do not have values. For example, the lexeme ’(’ does not have any value. Similarly, the keyword ’while’ does not have a value.

Your program should provide four procedures (that will be called by the parser in the next project):

1. next(): **reads** the next lexeme in the input file. (This will not return any thing: it will cause the next token to be recognized.)
2. kind(): returns the kind of the lexeme that was just read.
3. value(): returns the value of the lexeme (if it is an “ID” or a “NUM”).
4. position(): returns the position of the lexeme that was just read.

The lexical analyzer is invoked by the syntax analyzer (parser). To simulate this in your first project, your main program should have a loop for reading the input and printing the tokens as follows:

```
next();
print( position(), kind(), value() );
while ( kind() != ‘‘end-of-text’’ ) {
   next();
   print( position(), kind(), value() ); }
```

You are not allowed to use the following:

- Any library or module that supports regular expressions. Some examples are `re` of `Python`, and `regex` of `Java` or `C++`.
- Any existing class or library with built-in methods/functions for tokenizing text files.
  - You *can* use the `Scanner` class in `Java` **only for reading file names**. Any other use of this class is not allowed.

1.1 What to submit?

Please refer to COSC 455 Project 1 and 2 Submission Requirements.txt for the detailed instructions.