CSE 3341, Core Tokenizer Project
Due: 11:59pm on Wednesday, Feb. 23, '22 (50 points)

Grade: This part of the project is worth 50 points. (The second part of the project will consist of the Parser, PrettyPrinter and Executor and will be worth 100 points.)

Goal: The goal of this part of the project is to implement a *Tokenizer* for the *Core* language. The set of tokens of Core are as specified on slide 13 of the file part25Jan.pptx. You should implement your Tokenizer as a DFA as we have discussed in class. As listed on slide 13, the legal tokens of the Core language are:

- *Reserved words* (11 reserved words):
  `program, begin, end, int, if, then, else, while, loop, read, write`
- *Special symbols* (19 special symbols):
  `; , = ! [ ] && || ( ) + - * != == < > <= >=`
- Integers (unsigned)
- Identifiers: start with uppercase letter, followed by zero or more uppercase letters and zero or more digits. Note something like "`ABCend`" is illegal as an id because of the lowercase letters; and it is not two tokens because of lack of whitespace. But `ABC123` and `A1B2C3` are both legal.

For the purposes of this project, we will number these tokens 1 through 11 for the reserved words, 12 through 30 for the special symbols, 31 for integer, and 32 for identifier. One other important "token" is the EOF token (for end-of-file); we number that as token number 33. EOF does not actually appear in the input stream. We are just using EOF to indicate that the tokenizer has reached the end of the input file; and `getToken()` will return 33 when that happens. If `skipToken()` is called, it won't do anything since we are already at the end of the input stream; it will *not* produce an error. (So if `getToken()` is called again, it will again return 33.)

The tokenizer should read in a stream of legal tokens from the given input file and produce a corresponding stream of *numbers* corresponding to the tokens, *one number per line*, as its output. This will tell you (and the graders!) whether your tokenizer is identifying all tokens correctly. Thus, given the Core program:

```
program int X; begin X=25; write X; end
```

(which also appears at the top of slide 3 of part420Jan22.pptx), your Tokenizer should produce the following stream of numbers:

1 4 32 12 2 32 ... 33

corresponding to the tokens, "`program`", "`int`", "`X`", "`;`", "`begin`", "`X`", ..., `EOF`, but with each number being on a separate line.

Note that, although the example above is a legal Core program, the Tokenizer should *not* worry about whether the input stream is a legal Core program or not; that will be the job of the parser. All that your Tokenizer should care about is that each token in the input stream is a legal token. And if the tokenizer comes across an illegal token in the input stream, it should print an appropriate error message and stop.

Important Notes:

1. Greedy tokenizing: Note that something like "===XY" (no whitespaces) *is* legal and will be interpreted as the token "==" followed by the token "=" followed by the token "XY". As we saw in class, that is how "greedy" tokenizing works and that is how your Tokenizer should work as well.

2. As specified in slide 15 of the file "part25Ja22.pptx", your Tokenizer should provide the four operations, `getToken()`, `skipToken()`, `intVal()` and `idName()` that behave in the manner listed on that slide.

3. Your code should run on the CSE lab machines. So if you develop it on a different computer, please make sure that it runs on the lab machines before submitting it. Do NOT wait until the submission deadline to check this.

4. The tokens must be read line-by-line and printed. You should *not* read the entire stream of tokens in one fell-swoop before starting to print them. If there is an error in reading a token, such as a misspelled keyword, all the tokens prior to that token must be printed out before the error message for the bad token is printed. Then the Tokenizer must terminate.

Details:

1. You may write the tokenizer in *Java* or *Python*. Do NOT use any other language.

2. Do NOT use any regular expression package that may be available either as part of the standard libraries or that you might come across online.

3. Your Tokenizer should read its input from a file whose name will be specified as a *command line argument*. This file will contain the Core program to be tokenized. More accurately, the file may not be a legal Core program but simply a string of Core tokens in some random order. And your program should produce, on the *standard output stream*, the corresponding sequence of numbers, each one being between 1 and 33, the last one in the sequence being 33.

4. Your program should consist of the `Tokenizer` class; and the `main()` function which should create a `Tokenizer` object, repeatedly call the methods of the `Tokenizer` class to get the tokens from the input stream, and output the returned token numbers to the *standard* output stream, *one number per line*. Of course, your program should include any additional classes/functions that it needs to operate properly.

What To Submit And When: On or before 11:59 pm, Feb. 23, you should submit the following on Carmen:

1. A *plain text file* named `README` that specifies the names of all the files you are submitting and a brief (1-line) description of each saying what the file contains; plus, instructions to the grader on how to compile your program and how to execute it, and any special points to remember during compilation or execution. If the graders have problems with compiling or executing your program, they will e-mail you *at your OSU e-mail address*; you must respond within 48 hours to resolve the problem. If you do not, they will assume that your program does not, in fact, compile/execute properly.

2. Your source files and makefiles (if any). **DO NOT submit object files**.

3. A documentation file (also a plain text file). This file should include at least the following: A description of the overall design of the tokenizer, in particular, of the `Tokenizer` class; a brief "user manual" that explains how to use the Tokenizer; and a brief description of how you tested the Tokenizer and a list of known remaining bugs (if any) and missing features (if any).

4. Submission of the lab will be on Carmen. But I will not post the details of the lab on Carmen; instead, Carmen will just include a brief description of the project and allow you to submit your project.

The lab you submit must be your own work. Minor consultation with your class mates is ok. Ideally, any such consultation should take place on the Piazza group so that other students can contribute to the discussion and benefit from the discussion) but the lab should essentially be your own work.