CSE 3341, Core Tokenizer Project
Due: 11:59pm on Monday, Oct. 3, '22
(30 points; the other 20 points is for the DFA assignment for the Tokenizer)

Goal: The goal of this part of the project is to implement in Java or Python, a *Tokenizer* for the *Core* language. Your implementation should be based on the DFA for the Core tokens.

The set of tokens of Core are as specified on slide 14 of the file part2.pptx. You should implement your Tokenizer as a DFA as we have discussed in class. As listed on slide 14, 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, as we discussed in class, 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.

How to indicate end-of-file: 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. The only time that the Tokenizer will produce an error is when in runs into a string of characters that do not correspond to a token such as "`ABCend`".

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 grader!) whether your tokenizer is identifying all tokens correctly. Thus, given the Core program:

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

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. Of course, if the tokenizer comes across an illegal token in the input stream, it should print an appropriate error message and *stop*. It should not try to continue beyond that point. The particular details of the error message are up to you but, at a minimum it should indicate the particular sequence of characters that the Tokenizer came across that is not a legal token.

Important Notes:

1. Greedy tokenizing: 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 tokenizing works in all programming languages and that is how your Tokenizer should work as well.
2. As specified in slide 16 of the file "part2.pptx", your Tokenizer should provide the four operations, `getToken(), skipToken(), intVal()` and `idName()` that behave in the manner listed on that slide.
3. In addition, your `Tokenizer` class should have a constructor that takes a single argument which will be the name of input file that your Tokenizer should work on. The constructor should open that file, set up an appropriate input buffer, etc.
4. Your submission should also include a `main()` that receeives the name of the input file as a command-line argument. Your `main()` should call the `Tokenizer` class's constructor and pass that argument to the constructor. The `main()` function should then repeatedly call `getToken()`, print the token (number) it gets and call `skipToken()` until it gets the `EOF` token at which point the loop should terminate, after printing the 33 for the `EOF` token. The output from `main()` should go to the standard output stream.
5. 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.
6. You may write the tokenizer in *Java* or *Python*. Do NOT use any other language. Do *NOT* use the *Tokenizer* libraries of those languages or others that you may find online.
7. 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.

What To Submit And When: On or before 11:59 pm, Oct. 3, you should submit a `.zip` file on Carmen that includes the following:

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 grader has problems with compiling or executing your program, he will e-mail you *at your OSU e-mail address*; you must respond within 48 hours to resolve the problem. If you do not, he will assume that your program does not, in fact, compile/execute properly.

2. Your source files and makefiles (if any). **DO NOT include 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, such consultation should take place on Piazza so that other students can contribute to the discussion and benefit from the discussion) but the lab should essentially be your own work.