CSE 3341, Core Interpreter Project, Part 1 (Tokenizer)
Due: 11:59 pm, Oct. 12, 2020

Note: This is the first part of the *Core* interpreter project. In this part, you have to implement the *Tokenizer*.

Grade: This part of the project is worth 25 points. (The second part of the project will consist of the Parser, PrettyPrinter and Executor. The Parser will be worth 40 points and PrettyPrinter and Executor together, will be worth 35 points.)

Goal: The goal of this part of the project is to implement a *Tokenizer* for the language *Core*. The complete grammar for the language (pages 18 and 19 of the class slides) is the same as the one we have been discussing in class *with the exception that instead of the keyword* "`or`", *you should use* "`||`" (in production (12) of that grammar). Of course, the tokenizer shouldn't be concerned with the full grammar of the language. All it should care about is the set of legal tokens of the language. In other words, as long as the input stream contains only legal tokens, your tokenizer should work without complaining. The legal tokens of the Core language are (as listed –with the exception of "`or`" for "`||`"– on page 24 of the slides):

- *Reserved words (11 reserved words)*:
  `program, begin, end, int, if, then, else, while, loop, read, write`
- *Special symbols (19 special symbols)*:
  `; , = ! [ ] && || ( ) + - * != == < > <= >=`
- Integers (unsigned) (max size of 8 digits).
- Identifiers: start with uppercase letter, followed by zero or more uppercase letters and ending with zero or more digits with a total max of 8 characters in an id. 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.

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); let us assume that is token number 33. The tokenizer should read in a stream of legal tokens (ending with the EOF token), and produce a corresponding stream of token *numbers* as its output. This will tell you whether your tokenizer is identifying all tokens correctly. So given the program on page 20 of the slides, the 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`. If the tokenizer comes across an illegal token in the input stream, it should print an appropriate error message and stop. Note that the tokenizer should *not* worry about whether the input stream is a legal Core program or not; all it should care about is that each token in the stream is a legal token.

Important Notes:

1. *Whitespace*: Whitespace is required between each pair of tokens except if one (or both) of the tokens is a special symbol in which case the whitespace between them is optional. Whitespace is not allowed in the *middle* of any token. Any number of whitespaces is *allowed* between any pair of tokens. "Whitespace" means tab, carriage return, line feed, or blank character.

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".

2. Your Tokenizer should provide the four operations, **getToken(), skipToken(), intVal()** and **idName()** specified on page 25 of the slides.

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.

Details: You may write the tokenizer in *Java* or *Python*. Do not use any other language.

Your Tokenizer program 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 (and, in the second part, to be parsed, printed and executed). 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.

For this part, you do not have to implement two separate methods, one for *getting* the current token and one for *skipping* it; but you might as well do so since you will have to do that for the next part of the project.

What To Submit And When: On or before 11:59 pm, Feb. 12, you should submit, on Carmen, 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, the grader 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).

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.

Correct functioning of the Tokenizer is worth 70% (partial credit in case it works for some cases but not all. Documentation is 15%. Quality of code (how readable it is, how well organized it is, etc.) is 15%.

Late penalty: 4 points for each 24 hours or part thereof that your submission is late. This is somewhat tentative and may change slightly at a later date.

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. You may use the Tokenizer library available in Java or other similar facilities. But if you do so, make sure that is explained clearly in your documentation.