CSE 3341, Core Interpreter Project, Part 1 (Warm-up for Tokenizer)
Autumn 2019, Dr. Heym's DL 305 12:40 and 1:50 sections
Due: 11:59 P.M., Wednesday, September 18, 2019

Note: This is a warm-up for the second part of the *Core* interpreter project. In the second part, you will implement the full *tokenizer*. In this first part, you will practice the technique of **designing and simulating a finite state automaton** as a way of recognizing tokens.

Grade: This part of the project is worth 10 points. The rest of the project will be worth 90 (30 + 60) points. (Total for the interpreter project: 100 points.)

Important Notes:

1. *Whitespace*: Whitespace (one or more bytes with numeric value (ASCII encoding) in the set {9, 10, 13, 32}) 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 positive number of whitespaces is *allowed* between any pair of tokens.

   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. Ideally, to keep our grader's workload reasonably low, your code should run on *stdlinux*. Hence, if you develop it on a different computer, please port it to stdlinux and make sure it runs on that environment before submitting. If, however, you have a strong preference for using something that is available on the lab PCs (such as Eclipse), that is also acceptable. Make sure you specify in detail, in your README file, how your code is supposed to be compiled and run, stating clearly which platform is to be used. Please note that the environment in which your code compiles and runs should be something standard and available for general use on the CSE lab machines, not something you installed on your computer. (The graders have a limited amount of time to grade the labs and cannot spend a lot of time figuring out unusual environments.)

Goal: The goal of this part of the project is to implement a *Tokenizer* for a collection of tokens that is a proper subset of the collection of tokens of the language *Core*. The complete grammar for the language 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:

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

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 useful token is the EOF token (for end-of-file); let us assume that is token number 33. For this part of the project, you should submit your

tokenizer (for the proper subset of tokens) together with a test driver for this tokenizer. The test driver for the tokenizer should read in a stream of legal tokens (ending with the EOF token) which may (or may not!) be a legal Core program, and produce a corresponding stream of token *numbers* as its output. This will tell you whether your tokenizer is identifying all tokens correctly.

Let's now describe the proper subset of Core tokens that this preliminary tokenizer should recognize. Actually, this set of tokens is only close to being a subset; it is not strictly so. The reason for this approximation is that, to simplify matters, we won't properly identify reserved words in this first part of the project. In our simplification, we'll say that any string of lowercase letters is a token. We'll assign token number 1 to any of these strings of lowercase letters. The legal tokens for this first part of the project are:

- Lowercase words
- *Special symbols (4 special symbols)*:
  `;` `=` `||` `==`
- Integers (unsigned, possibly with leading zeros)
- Identifiers: start with uppercase letter, followed by zero or more uppercase letters and ending with zero or more digits.

The token number for `;` is 12, for = is 14, for `||` is 19, and for == is 26. Also, we will use here token numbers 31 for integer, and 32 for identifier. While it is not specified as part of the project, you may wish to use token number 34 (or some higher number of your choosing) for any non-token (i.e., error token) string of characters.

For example, given input that begins with "`program int X; begin X===328;XY74||`", the test driver for the tokenizer should produce a stream of numbers that begins with:

1 1 32 12 1 32 26 14 31 12 32 19

corresponding to the tokens, "`program`", "`int`", "`X`", "`;`", "`begin`", "`X`", "`==`", "`=`", "`328`", "`;`", "`XY74`", and "`||`". If the test driver for the tokenizer comes across an illegal token in the input stream, it should print an appropriate error message and stop.

Details: You may write the tokenizer, its test driver, and, in the forthcoming Lab 3, the interpreter in a language belonging to any of the following language families: *C++, Java, or C#.* You may also use *Ruby or Python*, but please read on for an important restriction. Do not use *Scheme* or *LISP*. If you want to use some other language, talk to me first to make sure it is acceptable; one important consideration is that the grader must be comfortable enough with the language you want to use and be able to use it reasonably on stdlinux (or a CSE lab PC), to be able to grade your lab. Your grader requests that, if you use *Ruby or Python*, you make sure your program runs properly in the stdlinux environment. These language families are especially sensitive to specific version and available libraries.

Your program should read its input from a file whose name will be specified as a *command line argument*. Hence, if your executable is named `TokenizerTest` and the input file is `test01`, you should be able to run the program by saying:

```
> TokenizerTest test01
```

where ">" is the Unix prompt. Your program should output to the standard output stream.

Minimally, your program should contain a `Tokenizer` class and a `main()` function, which is the test driver. This test driver should create a `Tokenizer` object, repeatedly call the appropriate methods of the `Tokenizer` class to get the tokens from the input stream one after the other, and output the returned token

numbers to the 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 (i.e., moving what is meant by the "current" token along to the next token in line); but you might as well do so because you will have to do that for the third part of the project.

What To Submit And When: On or before 11:59 P.M., September 18, you should submit the following:

1. An ASCII 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 name dot number at osu.edu email 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). Executable and object files are entirely optional. If it would not cause problems for the grader if they were eliminated and if, with reasonable effort, you can eliminate them, please do so. Otherwise, don't worry about it.
3. A text file called Runfile containing a single line of text that shows how to run your program from the command line, but *without* the required argument stating the path and name of the input file.

    - For example, if you are using Java and class TokenizerTest contains main, file Runfile should contain the line of text *java TokenizerTest*
    - Or, for example, if your makefile produces an executable file call mytoktest, Runfile contains *mytoktest*

4. A documentation folder or file (either hypertext (HTML) or ASCII text). This folder or 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). The documentation does not have to be as extensive as you did for the projects in CSE 3901, 3902, or 3903, but don't completely forget the lessons you learned in that class.

Submit your lab by creating a Compressed (zipped) Folder and placing this folder in the Lab 1 Carmen Dropbox. (In Windows, right-click on the folder and select Send To.) Be sure to click to the end of the process in Carmen and to double-check that your submission has occurred. Your most recent submission is your official submission.

Correct functioning of the Tokenizer is worth 50% (partial credit in case it works for some cases but not all. Documentation is 20%. Quality of code (how readable it is, how well organized it is, etc.) is 30%. A submission **will earn 0% in the quality of code section** of grading if it does not simulate a finite state automaton in order to recognize tokens. Please see sections 2.2.2 and 2.2.3 of our text by Michael Scott and one or more homework assignments.

The grading system imposes a heavy penalty on late assignments: up to 24 hours late - 10% off the score received; up to 48 hours late - 25% off; up to 72 hours late - 50% off; more than 72 hours late - forget it!

The lab you submit must be your own work. Minor consultation with your classmates is okay (ideally, any such consultation should take place on Piazza so that other students can contribute to and benefit from it) but the lab should essentially be your own work.