CSE 655, Core Interpreter Project, Part 1 (Tokenizer)
Due: 11:59 pm, May 4, 2012

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 40 points. The rest of the project will be worth 60 points.

Important Notes:

1. In production number (12) of the Core grammar (page 19 of the notes), replace "*or*" by "`||`".
2. *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, blank character, etc.

   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`". But you may want to *first* get your tokenizer working correctly under the assumption that there will be one or more whitespaces between each pair of tokens and *then* take care of the more general case.
3. Your code should ideally run on *stdlinux* or *stdsun*. So if you develop it on a different computer, please port it to stdlinux/stdsun and make sure it runs on that environment before submitting. But if you have used something that is available on the lab PCs (such as Eclipse), that is also acceptable. Make sure you specify, in your README file, how your code is supposed to be compiled and run. Please note that the environment in which your code compiles and runs should be something is standard in the CSE lab machines, not something you installed on your computer.

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 19 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)
- 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. 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.

Details: You may write the tokenizer in any of the following languages: *C++, Resolve C++, Java*. 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 the stdsun/stdlinux, to be able to grade your lab.

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

```
> Tokenizer coreProgram
```

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

Your program should consist of the `Tokenizer` class; and the `main()` function which 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; 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, May 4, 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 CSE 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**. (If the `submit` system complains that your submission is too large, chances are you are submitting object files.)
3. A documentation file (also ASCII 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). The documentation does not have to be as extensive as you did for the CSE 560 project, but don't completely forget the lessons you learned in that class.

Submit your lab using the following submit command (assuming your current directory contains *all* (*and only*) the files you want to submit for this lab; note the "." at the end of the line.

```
submit c655ab lab1 .
```

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%.

Late penalty: 4 points for each 24 hours or part thereof. 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 course newsgroup so that other students can contribute to the discussion and benefit from the discussion) but the lab should essentially be your own work.