CSE 3341, Core Interpreter Project, Part 2 (Parser, Printer, Executor)
Due: 11:59 pm, Friday, March 25, '22; 100 points

Notes:

1. This is the second part of the *Core* interpreter project. In this part, you have to implement the *parser, printer*, and *executor*. You should the same language, Java or Python, that you used for writing your Tokenizer.

2. If there are any special considerations for compiling and running your code, make sure you specify, in your README file, how your code is supposed to be compiled and run.

3. Your interpreter should take two command-line arguments. The first will be the name of the file that contains the Core program to be interpretd. The second will be the name of the file that contains the data for the Core program. Note that this is a change from the Tokenizer project. In that project, your Tokenizer read the input Core program from the standard input stream. But now, since there are two input files, the names of these files will be given as command-line arguments.

4. The Core program in the first file will *not* contain any illegal tokens but may contain other kinds of errors, i.e., not meeting the requirements of the BNF grammar of Core; undeclared variables; uninitialized variables; and another kind of error described in the next item.

5. The data in the second file will consist of a sequence of integers (positive or negative), one per line. This data will be read when your interpreter executes the "`read`" statements in the Core program. If this file is empty when the interpreter tries to execute a "`read`" statement, your interpreter should terminate with a suitable error message.

6. The output from your interpreter should go to the standard output stream.

7. If the Core program violates the BNF grammar or if undeclared variables are used in the `<stmt seq>` portion of the Core program, your interpreter, *before execution begins*, should print an appropriate error message and stop.

8. If there are no such errors, the *print*-procedures of your interpreter should *pretty-print* the Core program and then execute the program.

9. *Pretty-printing requirements*: There are no specific requirements about what precisely "pretty-printing" means. Follow your own instincts on what would make the structure of any given program easy to understand and try to implement that. The goal is to make the structure of the code clear by just looking at the pretty-printed version. Python's indenting style is a good model to follow. Our eyes/brain seem naturally wired to group together lines that are aligned (vertically) with each other. So, the Python model is a good one to follow; or come up with your own variation – as long as it makes the code clear by looking at it.

10. During execution, if your interpreter executes an `<out>` statement such as "`write X, Y;`", and the values of `X` and `Y` at that point are 20 and 30, your interpreter should produce the following output:

```
X = 20
Y = 30
```

Zip all your files into one archive and submit to Carmen.

Important: Your code *must* follow the principles of *encapsulation* (also known as "abstraction") that we have talked about, rather than have the details of the representation of the abstract parse tree visible to all parts of your interpreter. If you violate this guideline, your lab will be penalized heavily even if it is otherwise correct. You may use either the approach with a single `ParseTree` class or the approach with a separate class corresponding to each non-terminal in the BNF grammar of Core.

What To Submit And When: On or before 11:59 pm, March 25, you should submit, on Carmen, the .zip file as specified above. DO NOT include object files in your .zip file.

If the grader has problems with compiling or executing your program, he will e-mail you; 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. The project will be graded using the project rubric on Piazza.