CSE 3341, Core Interpreter Project, Part 2 (Parser, Printer, Executor)
Due: 11:59 pm, Fri., Nov. 14, '25; 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 use the same language, Java or Python, that you used for writing your Tokenizer/Scanner.

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. If your Scanner works only if there is white-space between each pair of tokens, specify that clearly so that the grader is aware of it. You will not be penalized since that was part of the first part of the project.

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 interpreted. The second will be the name of the file that contains the data for the Core program.

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; multiple declaration of the same variable; uninitialized variables; and another kind of error described in the next item. If the Core program violates the BNF grammar or if the same variable is declared more than once 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.

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. If during execution, the interpreter tries to access the current value of an identifier which has not yet been initialized, your interpreter should terminate with a suitable error message.

7. All of the output from your interpreter should go to the standard output stream.

8. If there are no errors related to the requirements of the BNF grammar and no variable is declared more than once and no undeclared variables appear in the `<stmt seq>`, 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. (The following site seems to do a reasonably good job of describing best practices in Python regarding formatting:
`https://go.osu.edu/python_pretty_printing_best_practices`

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
    ```

11. **Important**: Your code *must* follow the principles of *encapsulation* and *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. In other words, do *not* use the `PT[]` array explicitly in your parse, print and execute methods. Instead, use either the `ParseTree` class approach or the approach using a separate class corresponding to each non-terminal in the grammar (the "object-oriented" approach). If you violate this guideline, your lab will be penalized heavily even if it is otherwise correct.

12. **Structure of your main() function**: Your main() function should create a new *Scanner* object and pass the the first command-line argument it received to the Scanner's constructor. Following that, main() should create a *ParseTree* object or a *Prog* object depending on whether you are using the *ParseTree* class approach or the approach using a separate class corresponding to each non-terminal in the Core grammar. Then it should proceed to call the *Parse()* procedure/method on that object, then the *Print()* procedure/method and, finally, the *Exec()* procedure/method.

13. **Test data**: Use the Core programs I posted for the Scanner part of the project as test data to test your interpreter. Revise those programs to introduce various mistakes and check if your interpreter is able to deal with them appropriately. If you come up with your own test data, please share on Piazza.

14. Zip all your files into one archive and submit to Carmen. (If Carmen doesn't accept the Zip file, upload to your "My Files" site on Carmen and submit from there.)

15. If you have any questions or notice any mistakes in the description above, please post on Piazza.

What To Submit And When: On or before 11:59 pm on the due date, 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, they 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.