CSE 3341, Core Interpreter Project, Part 2 (Parser, Printer, Executor)
Due: 11:59 pm, Wednesday (not Friday), Mar. 4, '20; 75 points.

/share/CSE/web/userdirs/soundarajan.1/courses/3341/interprojP2.pdf **Notes**: This is the second part of the *Core* interpreter project. In this part, you have to implement the *parser, printer*, and *executor*. Your code should ideally run on the CSE lab machines. So if you develop it on a different computer, please make sure it runs on that environment before submitting. Make sure you specify, in your README file, how your code is supposed to be compiled and run. This part of the project is worth 60 points.

Goal: The goal of this part is to complete the the interpreter for the language *Core*. The complete grammar for the language is the same as the one we have been discussing in class, and appears on pages 18 and 19 of the class slides, except for the changes specified in the first part of this lab. Your *parser* should use the *Tokenizer* you have built in the first part of the project. (If you have not completed the first part of the project and don't expect to be able to complete it very soon, send me mail *immediately* so we can discuss your situation.) You should write this part of the interpreter in the same language that you used for part 1. If you find any problems with the project description below, please let me know by e-mail. Thanks.

Your interpreter should read its input from two files whose names will be specified as *command line arguments*, the first being the Core program to be interpreted, the second being the data file for that Core program. Your program should output to the standard output stream.

What To Submit And When: On or before 11:59 pm, Mar. 4, you should submit the following:

1. A plain text file named `README` that specifies the names of all the other 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; 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 plain text). This file should include at least the following: A description of the overall design of the interpreter, including the `ParseTree` class or the classes corresponding to the various non-terminals; a brief user manual that explains how to use the interpreter; and a brief description of how you tested the interpreter 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 interpreter is worth 50% (partial credit as appropriate); documentation worth 20%; quality of code, 30%. Late penalty: 4 pts for each 24 hrs or part thereof. The lab you submit must be your own work. Minor consultation with your class mates is ok (ideally, on Piazza so that others can contribute to the discussion and benefit from it) but the lab should essentially be your own work.

Output: When your interpreter is executed with a command such as

```
> Interpreter coreProgram inputData
```

it should produce two sets of output (to the standard output stream). The first should be a *pretty print* version of the *Core* program in the `coreProgram` file; for example, if there is an `if-then-else` statement in the *Core* program, that may be output as:

```
if ..cond.. then
      ..stmt seq1..
else
      ..stmt seq2..
end;
```

where `..cond..`, `..stmt seq1..`, etc. are the <cond>, <stmt seq> in the *then* part etc. of the given statement. The second set of output is produced when "`write`" statements in the *Core* program are executed. If, for example, the statement "`write X, ABC;`" is executed when the value of `X` is 10 and that of `ABC` is 20, your interpreter should produce two lines of output:

```
   X =  10
 ABC =  20
```

Approach: Use recursive descent for parsing, printing and executing the Core program. If you are tempted to do it any other way, don't! Recursive descent is the easiest way to make this work. If you are not convinced, try your alternate approach *after* finishing the project using the recursive descent approach.

You may use either the approach with a single, monolithic `parseTree` object that is an instance of the `ParseTree` class; or the one with many classes, one corresponding to each non-terminal. If you use the first, the `ParseTree` class, you should implement the parse tree abstraction so the rest of the interpreter doesn't know how parse trees are stored. If you ignore this issue and and parts like the executor are aware of the details of how parse trees are stored, you can expect a severe penalty in the grade. The `ParseTree` class should also maintain the table(s) containing the names of the id's and their values. Recall also that the `ParseTree` has to keep track of the *current node*, and we must have operations to move (the current node) up and down the tree. If you use the second approach with classes such as `Stmt`, `StmtSeq`, etc, there won't, of course, be a `ParseTree` class; and you will have to worry about the symbol table being maintained appropriately. But again make sure that you don't violate abstraction principles.

If you want to assume an upper limit on the number of nodes in the (abstract) parse tree, 2000 would be adequate; you may also assume that there will be no more than 20 distinct identifiers (each of size no more than 10 characters) in any given program that your interpreter has to interpret. In a few days, I will post a couple of sample *Core* programs that you can use as inputs to your interpreter. If you create test programs of your own, as you should, please share them on Piazza.

If you use the `ParseTree` class approach, the two most complex parts of the project, I expect, will be the `ParseTree` class, and the parser. Don't start with the implementation of the `ParseTree` class, instead worry only about the operations it is going to provide. Start with the list of operations we saw in class, and add additional operations that you find the need for as you design the parser, printer, and executor. I doubt that you will find the need for too many new operations beyond the ones listed in the the class notes. One other point: you may find it convenient to wrap the `tokenizer` object into the `ParseTree` class; if you do this, the operations of getting the current token etc. would have to be provided by the `ParseTree` class but they would be implemented by just calling the corresponding operations of the `Tokenizer` class. If you use the other approach with a separate class for each non-terminal, use an appropriate approach to ensure that there is only one *Tokenizer* object.

Some Comments: The `main()` function of your interpreter will do some initialization (perhaps just declare the `ParseTree` object. Then call the `parseProg`, `printProg`, and `execProg` procedures in that order. Or, if you are using the "OO" approach, it would create a `Prog` object and then call `parseProg` etc. on that object. Note also that negative integers cannot appear as literals in the Core program but negative integers may be in the data file (for the Core program). *Additional information about the input format may be posted, as necessary, on Piazza. So please make it a point to read that frequently.*