CSE 655, Core Interpreter Project, Part 2 (Parser, Printer, Executor)
Autumn 2011, 9:30 section
Due: 11:59 P.M., Monday, November 14, 2011

Important notes: This is the second part of the *Core* interpreter project. In this part, you have to implement the *parser, printer,* and *executor*. This part of the project is worth 60 points.

Goal: The goal of this part is to complete 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 in Carmen Content under the title "BNF for Core," 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, with any appropriate modifications you wish to make to it, of course. (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 want to do something different, first talk to me. 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. So if your executable is named `Interpreter`, the Core program is in the file `coreProgram`, and the data for that program is in the file `inputData`, you should be able to run the program by saying:

```
> Interpreter coreProgram inputData
```

where "`>`" is the Unix prompt. Your program should output to the standard output stream. State in your `README` file, discussed below, the command line users should use. For example, where we said "`Interpreter`" above, two words could be substituted for it, e.g., "`java CoreInt`" could be substituted for "`Interpreter`" if that would be appropriate in your case.

What To Submit And When: On or before 11:59 pm, November 14, you should submit the following:

1. An ASCII 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; 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. (If the `submit` system complains that your submission is too large, chances are you are submitting object files.)
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 arguments stating the paths and names of the input files.

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

4. A documentation file (also ASCII text file). This file should include at least the following: A description of the overall design of the interpreter, including the `ParseTree` class; 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). 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 by creating a Compressed (zipped) Folder and placing this folder in the Lab 2 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 interpreter is worth 50% (partial credit in case your interpreter works for some cases but not all, or parses but does not execute, etc.). The documentation is worth 20%. And the quality of your code (how readable it is, how well organized it is, etc.) is worth 30%.

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 class mates is ok (ideally, any such consultation should take place on the course discussion forum so that other students can contribute to the discussion and benefit from the discussion) 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.., ..stmt seq2..` are, respectively, the `<cond>`, `<stmt seq>` in the *then* part, and `<stmt seq>` in the *else* part, of the `if-then-else` 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 which is an instance of the `ParseTree` class; or the approach with a number of classes, one corresponding to each non-terminal. If you use the first approach, the `ParseTree` class 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 implement your interpreter so that parts like the executor are aware of the details of how parse trees are stored, you can expect a severe penalty in your grade. The `ParseTree` class should also maintain the table(s) containing the names of the identifiers and their current values. Recall also that a `ParseTree` has to keep track of the *current node*, and the class must provide operations to move (the current node) up and down the tree. Indeed, we always do things *at* the current node, rather than with the entire parse tree. If you use the second approach with classes such as `Stmt`, `StmtSeq`, etc., there won't, of course, be a `ParseTree` class. But again make sure that you don't violate data abstraction principles.

If you want to assume a pre-defined upper limit on the number of nodes in the (abstract) parse tree, 1000 would be more than enough; you can 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 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 "OO" approach, use the *Singleton* pattern 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 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 will be posted, as necessary, on the course discussion forum. So please make it a point to read that forum frequently.*