CS 164, Fall 2011 **CS 164: Homework #4** **P. N. Hilfinger**

Due: Tuesday, 27 September 2011

The homework framework, as usual, is in `~cs164/hw/hw4` on the instructional machines and at `$STAFFREPOS/hw4` in the class repository. Unless the problem specifies otherwise, please put your solutions in a file named `hw4.txt`. Turn in your finished `denull` and `hw4.txt` in your personal repository (not the team repository). Turn in your team's response to exercise 3 below as a team.

1. [From Aho, Sethi, Ullman] A grammar is called ϵ-*free* if there are either no ϵ productions, or exactly one ϵ production of the form $S \rightarrow \epsilon$, where S is the start symbol of the grammar, and does not appear on the right side of any productions. (We write ϵ productions either as 'A :' or 'A : ϵ'; both mean the same thing: there are no terminals or non-terminals to the right of the arrow). The template file `denull` is a skeleton for a Python program to do this. It already provides functions for reading and printing grammars. Fill in the removeEpsilons function to fulfill its comment. Apply your algorithm to the grammar:

`S` $\rightarrow$ `aSbS | bSaS |` ϵ

(This is a case in which your algorithm will need to introduce a new start symbol to fulfill the conditions of the problem).

2. The process of parsing an LL(1) grammar (the class that can be processed by the pure (non-looping) recursive-descent parsers we've worked with) can be encoded as a table-driven program. The table has nonterminals in one dimension, and terminals in the other. Each entry (for nonterminal A and terminal symbol τ) is a grammar rule for producing an A (one branch, in the terminology of the Notes), namely the rule to use to produce an A if the next input symbol is τ. Consider the following ambiguous grammar:

```
1.  prog  →  ϵ
2.  prog  →  expr ';'
3.  expr  →  ID
4.  expr  →  expr '-' expr
5.  expr  →  expr '/' expr
6.  expr  →  expr '?'  expr ':'  expr
7.  expr  →  '(' expr ')'
```

The start symbol is `prog`; `ID` and the quoted characters are the terminals.

a. Produce an (improper) LL(1) parsing table for this grammar. It's improper because, since it is ambiguous, some slots will have more than one production; list all of them. Show the FIRST and FOLLOW sets.

b. Modify the grammar to be LL(1) (unambiguous and with at most one production per table entry) and repeat part a with it.

In this case, we're just interested in recognizing the language, so don't worry about preserving precedence and associativity.

3. Team exercise. Create a substantial set of test cases for Project #1 that:

- Tests that the program correctly handles the lexical rules of our Python subset;
- Tests for correct processing of simple **print** statements (at least those without >> or trailing commas.);
- Tests for the correct handling of some expressions—at least those formed from numerals, strings, parentheses, and the arithmetic operators

  ```
  + - (unary and binary)
  * / // % **
  ```

  Correct handling includes getting precedence and associativity right.

You should test for proper handling of both correct and erroneous programs (proper handling of erroneous programs means that the system being tested detects the errors). Put your work in the `tests/correct` and `tests/error` subdirectories of the framework we supplied in the skeleton, and turn in the entire project with the tests *in your team repository,* as a submission of proj1 (that is, in `$TEAMREPOS/tags/proj1-`N, not in `.../hw4-`N). No, we won't be confused by the fact that this initial submission does not contain a working compiler. We have put a simple script in the skeleton for this homework, `check-tests`, which takes a sequence of test programs, strips off our type extensions to create a valid Python program, and then runs the standard Python interpreter over the files and compares the results with the `.std` files.