CS 164, Fall 2020 **CS 164: Homework #4** **P. N. Hilfinger**

Due: Fri, 16 October 2020

The homework framework, as usual, is in `~cs164/hw/hw4` on the instructional machines and as branch `hw4` in the shared 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).

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  →  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 recursive-descent parsers we've worked with—the one's without the **while**-loop trick) 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. Write a legal Python program that simply prints "`static`" and that would also be legal if Python used dynamic scoping, but would print "`dynamic`" instead.

4. Let's look at a very simplified sketch of scope analysis to give you a chance to work out the logic in a simpler setting than the project. This language has the following syntax:

```
program: /* empty */ | program outer_stmt ;
outer_stmt: stmt | def | class ;
stmt:  ID "::" type "=" expr ";"
    |  ID "=" expr ";"
    ;

stmts: /* empty */
    | stmts stmt
    | stmts def ;

def: "def" ID "{" stmts "}" ;

class: "class" ID "{" stmts "}"

type: ID

expr: /* empty */ | expr ID ;
```

The skeleton file `scoper.py` reads this syntax and produces an AST for it (see the skeleton). Fill in the skeleton to

1. Find all the distinct definitions, according to the rules. The definitions are names defined by **def**, names defined by **class**, and names that are assigned to.
2. Check that definitions are consistent: identifiers may not be multiply defined (no overloading here); multiple assignments to an identifier in the same declarative region result in only one local variable; a name may be defined to be only one kind of thing (local, method, or class) in the same declarative region.
3. Check that each name defined by an 'outer_stmt' (via **def**, **class**, or assignment) is so defined before any uses of it.
4. Make sure that all names appearing in 'exprs' are defined somewhere in an enclosing declarative region (for inner functions and locals, this can be before or after the use, as in Python.)

5. If there are multiple assignments to the same variable (i.e., in the same declarative region) using the `::` syntax, make sure the type name is the same in each case (no check is needed for other assignments).
6. Make sure that all 'type' identifiers refer to classes (and are defined prior to the use of the type).
7. As in Python, members of a class are *not* directly visible inside a method of the class (i.e., without '.', which this problem does not address.)
8. Number each distinct defined local, function, or class, and decorate the identifiers with the appropriate numbers. To make things deterministic, the skeleton has you do this by decorating identifier nodes in the AST with objects (type `Decl`), so that identifiers that refer to the same entity point to the same `Decl`. There is machinery in the skeleton to number these decorations.

The skeleton will print out the resulting program and annotations. The files `eg.scp` and `eg.out` provide a sample input and output.