CS 164, Spring 2015 **CS 164: Homework #4** **P. N. Hilfinger**

Due: Wed, 21 February 2018

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. [From Aho, Sethi, and Ullman] Consider the following ambiguous grammar:

`E` → `E '+' E` | `E '*' E` | `'(' E ')'` | **id**

and this parsing table for it (end-of-input, ⊣, is never shifted):

State	**id**	'+'	'*'	'('	')'	⊣	*E*
0	s3			s2			s1
1		s4	s5			acc	
2	s3			s2			s6
3		r4	r4		r4	r4	
4	s3			s2			s7
5	s3			s2			s8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

In this table, `s`n denotes a transition in the state machine ("go to state n on seeing this lookahead") and `r`n means "reduce the last symbols just scanned by the state machine (i.e., on top of the parsing stack) using production n." The productions are numbered left to right from 1; production 1 is $E \rightarrow E$ `'+'` E. Blank entries indicate errors. The start state is 0. Use the table to produce a reverse rightmost derivation of the string `id+id+id*(id+id)`. That is, give the sequence of reductions discovered by the parser.

4. Since the grammar of problem 3 is ambiguous, we had to add information to get a table out of it; otherwise, some of the entries would be unresolved. There are several possible parsing tables for it, depending on how we wish to resolve ambiguities. Show the modifications to the table that are necessary to

a. Give `'+'` and `'*'` equal precedence, and make them left associative. Thus, `x+y*z` would group as `(x+y)*z`, `x*y+z` as `(x*y)+z`, and `x*y*z` as `(x*y)*z`.

b. Give `'+'` higher precedence than `'*'`, and make them left associative.

c. Give `'+'` lower precedence than `'*'`, and make `'+'` right associative (`'*'` stays left associative).

d. Make it illegal to mix different operators without parenthesization. For example, to make the example in Exercise 1, above, illegal.

5. Consider the string `id+id(id)`, which is illegal according to the grammar of the preceding two problems.

a. Show what happens when you try to parse it using the parsing table from problem 3. That is, show all the steps taken by the parser up to the point where the machine finds no valid transition.

b. Now modify the table as follows: for each row that contains at least one reduction (`r`n), replace all the empty (error) entries in the action table for that row (i.e., the part between the vertical lines) with that reduction. For example, all entries in row 9 (except for E) would become `r3`, and all entries except that for `'*'` (and E) in row 7 would become `r1`. Show what happens when you try to parse the illegal string with this revised table. (This optimization—introducing *default rules*—makes tables more compressible; the question is whether it causes the parser to recognize illegal sentences.)

6. Here's another LR parsing table.

STATE	⊣	'/'	':'	'<'	'>'	'i'	'v'	E	F	P	S
0	r1	r1	r1	r1	r1	r1	r1				s1
1	acc						s3		s4		
3			s5								
4	r2	r2	r2	r2	r2	r2	r2				
5						s7	s6	s8			
6	r7	r7	r7	s9	r7	r7	r7			s10	
7	r4	r4	r4	r4	r4	r4	r4				
8		s11									
9						s7	s6	s12			
10	r3	r3	r3	r3	r3	r3	r3				
11	r5	r5	r5	r5	r5	r5	r5				
12					s13						
13	r6	r6	r6	r6	r6	r6	r6				

The reductions here have the following properties:

```
r1: 0 symbols  →  S
r2: 2 symbols  →  S
r3: 2 symbols  →  E
r4: 1 symbols  →  E
r5: 4 symbols  →  F
r6: 3 symbols  →  P
r7: 0 symbols  →  P
```

For example, production #2 has nonterminal `S` on the left-hand side and two symbols on the right-hand side (but I won't tell you what they are).

By considering the parse of the following sentence:

```
v:v<v>/v:i/⊣
```

reconstruct the grammar (that is, determine what symbols appear on the right-hand sides of the seven productions).

7. Team Exercise: When I revised my own solution to Project #1 for this semester, I had a previously constructed test suite of Python programs on which to test it. Debugging and development consisted of alternately editing my code and running `make check`, which told me what still needed work. As a team, construct for yourselves a test suite, aiming to cover as much of the syntax of Python as possible. It is best to make many small tests (and some larger ones), rather than one huge single test case, since the former allows you to isolate trouble spots better. Be sure to include both "correct" tests (not intended to produce errors) and "error" ones. The latter should concentrate on lexical issues, such as unterminated comments or strings.

Submit this assignment from your team repository with the tags `proj1a-`N (where N is a sequence number as usual). We will expect to find your tests in the directories `proj1/tests/correct`

and `proj1/tests/error`, along with the expected `.std` files in the "correct" directory. Any team member may submit for the team.

Don't put this off! We'll count it as the testing part of your team's grade on Project #1.