CS 363 – Lab #2 – Bottom-Up Parsing

This lab is essentially a continuation of last week's lab. In Lab #1, we experimented with top-down parsing. This week our attention turns to bottom-up parsing.

The idea is that as we parse, we keep track of which state we are in (and keep a stack of earlier states), and we determine our next state based on the input. Each transition is either a "goto" or a "reduce" operation. Goto means we are in the process of reading some syntactic element, so we just consume the token and move on. Reduce means we are finished reading something, and we should backtrack to an earlier state. One of the states of the parse table is an accept state, which means the input is valid. We have a syntax error if we encounter an undefined situation, such as finding an unexpected token.

As before, the type of input we are interested in parsing is a declaration for integer variables with initializations, like this: "int i = 5 , j = 6 ;"

It turns out that for the purpose of bottom-up parsing, we can simplify last week's grammar to become:

decl → int list **;**
list → var **,** list | var
var → **id = num**

where the tokens are the comma, semicolon, id, = and num. If you would rather work with the same grammar from last week, that is okay too.

Step 1: The first thing to do is to convert this grammar into sets of items. I've started the process for you, and you'll need to finish it. (Since the start symbol never appears on the right side of any rule, we don't need to augment the grammar.)

Step 2: Find the follow( ) for each nonterminal in the grammar, so that you will know on what input we should perform the appropriate *reduce* operations.

Step 3: Fill in the parse table.

Step 4: Convert the parse table information into code. Basically, we'll be adding code to the same Declaration.java file we worked on last week. But first, in Driver.java, add a statement to call the function parseBottomUp( ). In Declaration.java, you need to implement this function. Instead of setting up the data structures necessary to store the parse table, we can use the same technique we did last week. We can just write a case for each valid transition in the parse table. I've started parseBottomUp( ), so all you need to do is fill in the remaining cases.

Step 5: Test your code on some valid and invalid input. ☺

Sets of Items for grammar:
decl → int list **;**
list → var **,** list | var
var → **id = num**

I_0: decl → ● int list ; goto 1

I_1: decl → int ● list ; goto 2
list → ● var , list goto 3
list → ● var goto 3
var → ● id = num goto 4

I_2: decl → int list ● ; goto 5

I_3: list → var ● , list goto 6
list → var ● reduce (–1, list)

I_4:

I_5:

I_6:

I_7:

I_8:

I_9:

Fill in parse table here:

State #	int	;	,	id	=	num	$	list	var
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									