CS 363 – Lab #3 – Creating syntax trees

In today's lab we will be working with the postfix calculator grammar that we studied in class. You may want to refer to the handout because it contains the grammar, parse table and other information that will be helpful to reference as you do the lab. I have written a bottom-up parser for you, and you can find it on the server in a directory called "lab 3" consisting of the files Driver.java and Program.java.

You will be making changes to Program.java so that it keeps track of syntax subtrees as they are needed during the parse. As we saw in class, on some parse transitions we need to create new a tree node from scratch, or build a new tree out of smaller ones.

Here are the cases to handle:

1. In order to completely process an assignment statement such as "a = 7 ;" we need to push leaves such as "a" and "7" and the "=" on the stack, so that by the time we reach the semicolon, we can "reduce" the statement into a single tree with "=" as the root, and the identifier and constant as its children.
2. We also need to handle the postfix expression statements. When you encounter individual numbers, identifiers or operation symbols, you can create leaves. When reducing an expression, you can create a new tree out of other subtrees. For example "7 3 +" would be made out of the 3 tiny subtrees (+), (7) and (3). And later if we see "4 –" then the "7 3 +" would become the left child of the minus, and the 4 becomes the right child.
3. At the end of your program, you can dump all the subtrees from the stack onto the output. Note that if you pop the stack until it's empty, this will be the sequence of statements in reverse order.

Hint: You'll find it convenient to do some of the tree-creating in the "goTo()" function instead of the case labels. In most cases, you just need to create a leaf node. But sometimes you need to pop some elements from the tree stack and create a new Tree to push back.

You will need to create a small Tree class, with 2 constructors (one parameter for leaf case or 3 parameters to handle building a new tree out of smaller ones) and a toString() for output.

To show how the subtrees are being created, you can print a statement using the Tree toString(). See the example output below.

Example

On the next page there's an example interactive session with the program.

```
Please enter statements, separated by ';'.  Separate each token by a space.
a = 5 ; b = -1 ; a b 8 + - ;
state 0, next input token 'a'
  creating leaf for a
state 2, next input token '='
  creating leaf for =
state 6, next input token '5'
  creating leaf for 5
state 9, next input token ';'
  created tree:  ((=),(a),(5))
state 15, next input token 'b'
state 0, next input token 'stmt'
state 1, next input token 'b'
  creating leaf for b
state 2, next input token '='
  creating leaf for =
state 6, next input token '-1'
  creating leaf for -1
state 9, next input token ';'
  created tree:  ((=),(b),(-1))
state 15, next input token 'a'
state 1, next input token 'stmt'
state 1, next input token 'a'
  creating leaf for a
state 2, next input token 'b'
state 1, next input token 'expr'
state 3, next input token 'b'
  creating leaf for b
state 2, next input token '8'
state 3, next input token 'expr'
state 8, next input token '8'
  creating leaf for 8
state 4, next input token '+'
state 8, next input token 'expr'
state 8, next input token '+'
  creating leaf for +
state 11, next input token '-'
creating((+),(b),(8))
state 8, next input token 'op'
state 10, next input token '-'
state 3, next input token 'expr'
state 8, next input token '-'
  creating leaf for -
state 12, next input token ';'
creating((-),(a),((+),(b),(8)))
state 8, next input token 'op'
state 10, next input token ';'
state 1, next input token 'expr'
state 3, next input token ';'
state 7, next input token '$'
state 1, next input token 'stmt'
state 1, next input token '$'
Parse successful!
Here is the sequence of subtrees.
((=),(a),(5))
((=),(b),(-1))
((-),(a),((+),(b),(8)))
```