CS 363 – Lab #1 – Top-down Parsing

In today's lab we will practice some of the techniques of top-down parsing we have reviewed in class. Let's experiment with the following grammar for integer variable declarations. Note that "int", "id" and "num" are just tokens.

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

This grammar says that a variable declaration begins with a keyword (for the type). Then there is a variable, followed by more variables, separated by commas. The declaration ends with a semicolon. Note that in this definition, we require all variables to be given an initial value.

To prepare for future steps in the lab, write down the answers to these questions:

1. What are the nonterminals of the grammar? The terminals?

2. How many productions are in the grammar?

3. Determine the first() for each nonterminal in the grammar.

4. Determine the follow() for each grammar symbol.

5. Determine the predict() for each production in the grammar.

6. Fill in the parse table.

nonterminal	key	id	num	,	;	=
list						
tail						
Var						

From the server, download the Java source files for today's lab. You should find 2 files: Driver.java and Declaration.java. You will be making updates to Declaration.java.

7. Look at Driver.java. The overall structure of the program is simple. First we call a constructor that just initializes some data. Then we scan the input, and then we parse it. In today's lab there will be 2 versions of top-down parsing. First, we will do top-down parsing based on the parse table you just filled in. Second, we will re-do the parse using the recursive-descent method.

8. Note that all the code necessary for gathering input and scanning it into tokens has already been done for you. For example, integer values like 5 have been converted to a token string "num" and identifier strings have been converted to the string "id". To implement *parseWithTable()*, open Declaration.java. It turns out that it would take too long for us to initialize the complete data structures for the parse table as well as the productions for the grammar. For the small grammar we are working with, it is sufficient to isolate the 4 cases we need to identify. (4 because there are 4 valid entries in the parse table). Each valid entry in the parse table tells us to replace something on the stack based on the current top-of-stack symbol and input. We could write the code something like this:

   ```
   if (the top-of-stack is "list" and the next input token is "int")
      replace list with int var tail (i.e. replace left side of production with right side)
   else if (top-of-stack is "tail" and next token is ",")
      replace tail with , var tail
   handle the remaining valid cases from parse table
   handle the matching of a token on the stack with a token from the input
      if they do not match – give syntax error message.
   ```

   When you give the syntax error message, tell the user the token number (e.g. #7), what token was expected and what token was found instead. This will make the error message helpful.

   Run your parseWithTable() routine with good and bad input.

9. To implement recursive-descent parsing, it will be necessary to implement the parseRecursiveDescent() function. Besides this function, we have the routines that actually perform the recursive descent, one function per nonterminal – namely *parseList*(), *parseVar*() and *parseTail*(). To help you get started, I have implemented parseList(), and you need to implement the other 2 functions.

   Whenever your program detects a syntax error, you should again give a helpful message, indicating the token number at the point of the error, what token was expected and what was found instead.