Bottom-up parser worksheet

Compiling a computer program usually entails these 5 phases:

1. Scanning – read the source file(s) and break up the text into tokens, such as identifiers, keywords and constants. Discard the comments, blanks, tabs and new-line characters. This process can be done using an FA because rules for defining tokens are regular.
2. Parsing – This is the heart of compiling. From the stream of tokens, try to understand the structure of the program, and detect syntax errors.
3. Semantic analysis – build the symbol table and verify that all names have been declared correctly
4. Generating code – the output of the compiler is usually assembly or machine code to be run in the CPU (or a virtual machine in the case of Java).
5. Optimize – modify the code so that it performs the same tasks using less time and/or less memory, less power.

Thus, the parsing represents the core of compilation. We can construct a "parse machine" as follows:

1. Start with a context-free grammar for your language.
2. Determine the states of the parse machine. This is called finding the sets of items. An item is simply a grammar rule with a cursor. These sets are traditionally called I_0, I_1, I_2, …. In practice, you'll find that the start state will be I_0 and the accept state will be I_1.
3. Looking at your sets of items, determine the transitions. You will need to recognize that there are 2 kinds of transitions: "goto" and "reduce." A goto action is a normal transition to the next state. A reduce operation means we need to backtrack.
4. Reduce operations need special handling, because it's not immediately clear on what input we need to reduce. For this reason, we need to compute the first and follow for each variable in the grammar. In the following description, we'll assume that there are no ε rules. Handling ε rules would make this process a little more complex.

   first – The first operation creates a set of terminals that can appear at the beginning of a string derived from a variable. Let's say we want to calculate first(A). Here's how to do it:
   a. If any of A's rules begin with a terminal, add that terminal to first(A).
   b. If any of A's rules begin with a variable (e.g. B), compute first(B), and add the resulting set to first(A).

   follow – The follow operation creates a set of terminals that could follow a particular variable when parsing a string in the language. For example, let's say that the input string is abcde and that the substring abc reduces to A. Our parser can now view the input string as Ade. For this string to be in the language, the parse would expect the letter 'd' to follow A. To compute the follow we generally look at all the rules where A appears on the right-hand side:
   a. If A is the start symbol, add a special symbol $ to follow(A) to signify end-of-input.
   b. If there is a rule like Q → Ac…, where c is a terminal, add c to follow(A).
   c. If there is a rule like Q → AB, where B is a variable, add everything in first(B) to follow(A). Now you see why we need to calculate the "first"!
   d. If A appears at the end of a rule, like Q → …A, add everything in follow(Q) to follow(A). In other words, whatever can follow Q can also follow A.

5. Knowing the states and transitions, we can draw the parse machine. If there are many states, this can be a cumbersome picture, so usually we simply write a parse table.
6. Finally, we can run the parse machine on our input to see if it's accepted or has a syntax error.

Example: Let's design a parse machine for this grammar:

S → 0A0
A → 1 | 1A

Step 1. We augment the grammar by including a new rule S′ → S.
Step 2. Time to create the sets of items. For I_0, we always start with S′ → ● S. And because we have a dot before a variable, we need to expand on that variable. So, we include the item S → ● 0A0. So now I_0 is:

I_0: S′ → ● S goto 1
S → ● 0A0 goto 2

The reason why we have "goto" transitions is that when we look at where the cursor is in our items, we see that the next token is either an S or a 0. Next, we consider item set I_1. We start by simply advancing the cursor from S′ → ● S to S′ → S ●. So here is I_1:

I_1: S′ → S ● acc

We don't need to expand any variable because the cursor does not appear before a variable. In fact, since the cursor appears at the end of the rule, we have a "reduce" operation. And S′ → S● is the special case where we accept. On to item set 2. We continue by letting the cursor pass by the first 0. In other words, we go from S → ● 0A0 to S → 0 ● A0.

I_2: S → 0 ● A0 goto 3
A → ● 1 goto 4
A → ● 1A goto 4

Note that we had to expand the variable A. After writing these 3 items, we determine that they are all gotos. The first item will be a goto when the next token is an A. The second and third items will goto when the next input token is a 1. In our transition notation, we could have written this as δ(2, A) = 3 and δ(2, 1) = 4.

Can you figure out the rest?

I_3: I_5:

I_4: I_6:

We're almost done… but we need to calculate the first and follow of our variables S′, S and A. Look at the grammar again:

S′ → S first(S′) = _____ follow(S′) = _____
S → 0A0 first(S) = _____ follow(S) = _____
A → 1 | 1A first(A) = _____ follow(A) = _____

Finally, we can fill in the parse table:

State/item set	Input 0	Input 1	Input $	Input S	Input A
0					
1					
2					
3					
4					
5					
6					