There is a notation for derivations; you can see an example in the hw10 examples posted on the schedule page.

The structure of a parse tree reflects the rules in the grammar. Each symbol on the right side of a rule has its own branch. See the hw10 examples posted on the schedule page.

For #1, the symbols in the grammar include $\wedge$, $\vee$, $\rightarrow$, (, and) — these are not part of the production rule syntax and should be treated the same as p, q, and the other letters.

You don't always have to push or pop something in a pushdown automaton. A transition $\sigma, \epsilon/\epsilon$ is valid.

For both reading (determining the language accepted by) and writing (creating) pushdown automata, keep in mind the two elements and the role each plays:

- The state transitions consume the input string, and states are used for tracking specific numbers and sequences of symbols.
- The stack is used for matching — one symbol with another, or a count of symbols with another count.

For #4, start with the states. In (c), the string must start with 0 or more occurrences of ab — the a gets to state q_1 and the b is needed to get back to state q_0, because that's the only way to get to the final state q_2. After the initial abs, there can be 0 or more occurrences of ba — the b gets to state q_3 and it must be followed by an a to get back to the final state. So our starting point is $(ab)^*(ba)^*$. Now what about the stack? For the initial abs, one b is pushed for each ab. Then, for the ending bas, one b is popped for each ba — thus there must be the same number of bas as abs in order to end in q_2 with an empty string and an empty stack. Thus the language is $(ab)^n(ba)^n$ for $n \geq 0$ or, in English, zero or more abs followed by the same number of bas.

Use a similar tactic for #4d. Looking at just the states means that there must be 0 or more aas followed by a b and then 0 or more bbs — $(aa)^*b(bb)^*$. Looking at the stack shows that an a is pushed and then popped for each aa, and a b is pushed and then popped for each bb — so actually the stack isn't really contributing anything here. The resulting language is $(aa)^*b(bb)^*$ or, in English, an even number of as followed by an odd number of bs.

For constructing pushdown automata in #5, keep the same two elements in mind. Start with the states — "multiple of 3" is a specific number and so needs to be handled through states. Start with an NFA that accepts a^nb^m where n and m are a multiple of 3. Then address the "same number of as and bs" part — that's a matching thing. How do you use the stack to match as with bs?

A third principle for constructing pushdown automata is *keep it simple*. For (b), the only elements of the language are matching —) with (and] with [— and consuming

as and bs. (There's no counting needed.) Is matching a stack thing or a state thing? If it's not a state thing, there's no need for more states! This pushdown automata can have just a single state.

For #6, remember the definition of deterministic context free — a language is deterministic context free if there is a deterministic pushdown automaton accepting $L\$$. Start with just constructing a pushdown automaton for $L\$$ (don't worry about the deterministic part yet). Is $n_a(w) > n_b(w)$ a state thing or a stack thing? If it is a stack thing, don't add more states unless there's a sequence-of-symbols something going on — a common mistake was to accept the language $a^n b^m$ where $n > m$ rather than L, which allows the as and bs to be in any order. (There is eventually some kind of sequencing going on with $n_a(w) > n_b(w)$ because the stack has to be empty in order to accept — once \$ has been consumed, the task switches to emptying the stack rather than matching as and bs, which means a new state.)