Additional hints for Lab 1: Backward chaining

The backward chaining process

Here's the general idea of backward chaining:

- Given a hypothesis, you want to see what rules can produce it, by matching the consequents of those rules against your hypothesis. All the consequents that match are possible options, so you'll collect their results together in an OR node. If there are no matches, this statement is a leaf, so output it as a leaf of the goal tree.
- If a consequent matches, keep track of the variables that are bound. Look up the antecedent of that rule, and instantiate those same variables in the antecedent (that is, replace the variables with their values). This instantiated antecedent is a new hypothesis.
- The antecedent may have AND or OR expressions. This means that the goal tree for the antecedent is already partially formed. But you need to check the leaves of that AND-OR tree, and recursively backward chain on them.

Some hints from `production.py`

`match(pattern, datum)` - This attempts to assign values to variables so that *pattern* and *datum* are the same. You can `match(leaf_a, leaf_b)`, and that returns either `None` if `leaf_a` didn't match `leaf_b`, or a set of bindings if it did (even empty bindings: `{}`).

Examples:

- `match("(?x) is a (?y)", "John is a student") => { x: "John", y: "student" }`
- `match("foo", "bar") => None`
- `match("foo", "foo") => {}`

Both arguments to `match` must be strings; you cannot pass a consequent (an object of type THEN) to `match`, but you can index into the THEN (because it's a type of list) and pass each element to `match`.

Note: `{}` and `None` are both `False` expressions in python, so you should explicitly check if match's return value is `None`. If `match` returns `{}`, that means that the expressions match but there are no variables that need to be bound; this does not need to be treated as a special case.

`populate(exp, bindings)` - given an expression with variables in it, look up the values of those variables in *bindings* and replace the variables with their values. You can use the bindings from `match(leaf_a, leaf_b)` with `populate(leaf, bindings)`, which will fill in any free variables using the bindings.

- Example: `populate("(?x) is a (?y)", { x: "John", y: "student" }) => "John is a student"`

`rule.antecedent()`: returns the IF part of a rule, which is either a leaf or a RuleExpression. RuleExpressions act like lists, so you'll need to iterate over them.

`rule.consequent()`: returns the THEN part of a rule, which is either a leaf or a RuleExpression.