Lab 4: Rule-Based Systems

Submit Assignment

Due Thursday by 11:59pm **Points** 5 **Submitting** a file upload
Available Sep 24 at 10:01am - Oct 17 at 10pm 23 days

Lab 4: Rule-Based Systems

To complete Lab 4, download: **lab4.zip**.

Lab 4 is due on **Thursday, October 8th at 11:59pm EDT**. Once you have completed lab4 and it passes the local tests, submit **only** your lab4.py here. All of your answers belogn in the main file lab4.py.

Contents

Preamble

You will probably want to use the Python feature called *list comprehensions* at some point in this lab, to apply a function to everything in a list. You can read about them in the official Python tutorial (http://docs.python.org/tutorial/datastructures.html#list-comprehensions).

Those who think recursively may also benefit from the function `functools.reduce`, documented in the Python library reference (https://docs.python.org/3.0/library/functools.html#functools.reduce). (Did you know: The built-in implementation of `reduce` was removed with the release of Python 3, because "99 percent of the time an explicit for loop is more readable" (https://docs.python.org/3.0/whatsnew/3.0.html#builtins).)

If you're having an issue or a bug that just doesn't make sense, the **FAQ** section below may be helpful. We will update the FAQ section with more questions and answers as new issues come to light.

A note on nomenclature

Wherever possible, we try to be consistent with our use of certain keywords. In particular, the words *clause*, *expression*, and *statement* are often vague and overloaded, but we attempt to use them consistently in this lab:

- IF, THEN, and DELETE commands are **clauses**.
- AND, OR, and NOT commands are **expressions**.
- Simple phrases such as `'opus is a penguin'` or `'sibling (?x) (?y)'` are **statements**.
- AND and OR expressions comprise zero or more **statements** and expressions; NOT expressions comprise a single **statement**, which is semantically negated.

We also often refer to rules themselves as expressions. However, the context of use should make the intended meaning clear.

A Production Rule System

In this section, we explain the basic structure of *production rule system*. There aren't any problems to solve, but please read it carefully anyway.

A production rule system is instantiated with a list of *data* and a list of *rules*:

- A **datum** (plural, **data**) or an **assertion** is simply a string statement declaring a fact or a relation, e.g. `'Jake is a person'`.
- A **rule** is a combination of a predicate (*antecedent*) and a resultant action or set of actions (*consequent*).

The anatomy of a rule

Conceptually, a rule is a mechanism that lets us conclude things, or take some actions, based on what things we already have or know.

Syntactically, a rule is an idiom that contains certain keywords such as IF, AND, OR, NOT, THEN, and DELETE. Importantly, rules can contain variables such as `'(?x)'` or `'(?y)'`. This allows a rule to match more than one possible datum. The consequent of a rule can contain variables that were bound in the antecedent.

Here is an example of a rule:

```
IF( AND( 'parent (?x) (?y)',
         'parent (?x) (?z)' ),
    THEN( 'sibling (?y) (?z)' ))
```

This could be taken to mean:

> If *x* is a parent of *y*, and *x* is a parent of *z*, then *y* is a sibling of *z*.

Given the dataset `['parent marge bart','parent marge lisa']`, executing the above rule would produce the datum `'sibling bart lisa'`. (In this particular case, executing it could also produce the datum `'sibling bart bart'`, which illustrates an issue that we will discuss with later.)

Of course, the rule system doesn't know what these arbitrary words such as "parent" and "sibling" mean! It doesn't even care that these words occur at the beginning of the statements. Indeed, the rule could also be written as:

```
IF (AND( '(?x) is a parent of (?y)',
         '(?x) is a parent of (?z)' ),
    THEN( '(?y) is a sibling of (?z)' ))
```

Then it will expect its data to look such as `'marge is a parent of lisa'`, and so on. Writing data in more human-readable formats such as this does come at a cost: The assertions and rules become wordy, and rule-matching involves unnecessary matching of symbols such as `'is'` and `'a'`. We will use a mixture of these styles in this lab.

Just remember that the English is for your benefit, not the computer's. The computer's only concern is to pattern-match the data with the rules.

IF / THEN Clauses

In this lab, a rule must always have an IF and a THEN clause. Optionally, a rule can also have a DELETE clause, which specifies some data to delete.

The IF clause (antecedent) is either a single statement or a single AND, OR, or NOT expression:

- AND requires that multiple statements are matched in the dataset
- OR requires that at least one of multiple statements are matched in the dataset
- NOT requires that a statement is *not* matched in the dataset.
- AND, OR, and NOT expressions can be nested within each other. When nested, these expressions form an AND-OR tree (or, perhaps more accurately, an AND-OR-NOT tree). The *leaves* of this tree are string statements, possibly containing variables.

The THEN clause (consequent) contains at least one statement. When the rule **fires**, each statement in the consequent is added to the dataset (unless already present).

The DELETE clause is optional, and contains at least one statement. When the rule **fires**, each statement specified in the DELETE clause is removed from the dataset (if present).

Match and Fire

The terms **match** and **fire** are important. A rule **matches** if its antecedent matches the existing data. A rule that matches can then **fire** if its THEN or DELETE clauses would *change* the data. Otherwise, it fails to fire.

An Iteration of Forward Chaining

The rules and assertions of the system are supplied in a specified order.

During an iteration of forward chaining, the system will check each rule in turn: For each rule, the system searches the data, in order, for assertions that match the requirements of the rule's antecedent. Assertions that appear *earlier* in the data take precedence over assertions that appear *later* in the data.

If an AND or OR expression comprises multiple statements, we always try to match those statements *in the order given*. This policy is particularly important for AND expressions, because if one of the AND expression's statements contains variables, and we match that to a datum, we must apply those same variable bindings to the subsequent statements in the AND expression.

For example, consider the data `['Bill has chocolate','Alex has chocolate','Alex likes eating chocolate']` along with the rule

```
IF (AND( '(?x) has chocolate',
         '(?x) likes eating chocolate' ),
  THEN( '(?x) is happy' ))
```

In an iteration of forward chaining, the system would try to match this rule to the data by doing roughly the following:

1. Check the first part of the antecedent (`'(?x) has chocolate'`) against the first assertion, `'Bill has chocolate'`. This is a match, binding `(?x) = Bill`.

2. Apply the binding `(?x) = Bill` to the second part of the antecedent, yielding `'Bill likes eating chocolate'`.
3. Check `'Bill likes eating chocolate'` against `'Bill has chocolate'`. No possible match.
4. Check `'Bill likes eating chocolate'` against `'Alex has chocolate'`. No possible match.
5. Check `'Bill likes eating chocolate'` against `'Alex likes eating chocolate'`. No possible match.
6. Dead end, so back up.
7. Check the first part of the antecedent (`'(?x) has chocolate'`) against the second assertion, `'Alex has chocolate'`. This is a match, binding `(?x) = Alex`.
8. Apply the binding `(?x) = Alex` to the second part of the antecedent, yielding `'Alex likes eating chocolate'`.
9. Check `'Alex likes eating chocolate'` against `'Bill has chocolate'`. No possible match.
10. Check `'Alex likes eating chocolate'` against `'Alex has chocolate'`. No possible match.
11. Check `'Alex likes eating chocolate'` against `'Alex likes eating chocolate'`. This is a match.
12. Conclude that this rule matches the data with `(?x) = Alex`

Of all the rules that match, the system picks the first such one that can fire, and fires it, adding/deleting assertions as necessary. If no matching rule can fire, forward checking immediately terminates.

A Note about NOTs

If there is a NOT expression in the antecedent, the data is searched to make sure that *no data item* matches the pattern.

For example, the antecedent

```
NOT( 'Mary is a penguin' )
```

matches the data `['Phil is a penguin','Sue is a penguin']`, because `'Mary is a penguin'` is not a member of the data.

However, the antecedent

```
NOT( 'Phil is a penguin' )
```

does **not** match the data `['Phil is a penguin','Sue is a penguin']`, because `'Phil is a penguin'` is a member of the data.

Things get a little bit more complicated when we introduce variables in NOT expressions.

The antecedent

```
AND( '(?x) is a bird',
     NOT( '(?x) is a penguin' ))
```

describes "things that are birds but not penguins." This antecedent matches the data `['Phil is a penguin','Javier is a bird']`, because...

1. The only way to match the first constituent of the AND is with the binding `(?x) = Javier`.
2. Applying the binding to the second constituent of the AND yields `NOT( 'Javier is a penguin' )`.
3. `'Javier is a penguin'` is not in our data, hence, `NOT( 'Javier is a penguin' )` matches.
4. So the entire AND expression successfully matches with `(?x) = Javier`.

But what happens if we *flip* the order of the two constituents? Our new antecedent looks like

```
AND( NOT( '(?x) is a penguin' ),
     '(?x) is a bird')
```

Suppose the data is still `['Phil is a penguin','Sue is a penguin']`. When attempting to match this rule to the data, the matcher first looks at the AND's first constituent, `NOT( '(?x) is a penguin' )`. But what is the value of `(?x)` supposed to be? There are literally an infinite number of bindings for `(?x)` that can make `NOT( '(?x) is a penguin' )` be true. In fact, there are only *two* values of `(?x)` that would *not* yield a match: `Phil` and `Sue`. In short, the matcher goes crazy because it has no positive guesses of what to bind `(?x)` to. We wouldn't expect a student to write down an infinite number of possible bindings on a quiz; neither should we expect our matcher to do that.

So...

A NOT expression should never introduce new variables: Our naive matcher will no know what to do with them, and it will certainly not behave as you expect it to.

Running the system

If you call `from production import forward_chain`, you gain access to a procedure `forward_chain(rules, data, verbose=False)` that will make inferences as described in the previous sections. It's important to note that in our code, `rules` and `data` may either be supplied as lists or tuples; either is acceptable. However, it may be most appropriate to represent `data` as a list because it's mutable, and data may be added or removed.

This method returns the final state of its input data.

Here's an example of using `forward_chain` with a very simple rule system:

```
from production import IF, AND, OR, NOT, THEN, DELETE, forward_chain

theft_rule = IF( 'you have (?x)',
                 THEN( 'i have (?x)' ),
                 DELETE( 'you have (?x)' ))

data = ( 'you have apple',
         'you have orange',
         'you have pear' )
```

```
print(forward_chain([theft_rule], data, verbose=True))
```

We provide the system with a list containing a single rule, called `theft_rule`, which replaces a datum such as `'you have apple'` with `'i have apple'`. Given the three items of data, it will replace each of them in turn.

Here is the output if you ran the code above as a Python script:

```
Rule: IF(you have (?x), THEN('i have (?x)'))
Added assertion: i have apple
Rule: IF(you have (?x), THEN('i have (?x)'))
Deleted assertion: you have apple
Rule: IF(you have (?x), THEN('i have (?x)'))
Added assertion: i have orange
Rule: IF(you have (?x), THEN('i have (?x)'))
Deleted assertion: you have orange
Rule: IF(you have (?x), THEN('i have (?x)'))
Added assertion: i have pear
Rule: IF(you have (?x), THEN('i have (?x)'))
Deleted assertion: you have pear
('i have apple', 'i have orange', 'i have pear')
```

NOTE: The `Rule:`, `Added assertion:`, and `Deleted assertion:` lines come from the verbose printing. The final output is the new list of data after applying the forward chaining procedure.

You can look at a much larger example in the `zookeeper` data in `data.py`, which classifies animals based on their characteristics.

Part 1: Multiple Choice

As you answer the multiple choice questions below, keep in mind:

- that the computer doesn't know English, and anything that reads like English is for the user's benefit only
- that there is a difference between a rule having an antecedent that **matches**, and a rule actually **firing**

Indicate your answers as strings assigned to `ANSWER_i` in `lab4.py` under "Part 1".

Questions 1-2: New assertions

Question 1: In forward chaining, after all the variables in a rule have been bound, which part of the rule may appear as a new assertion in the data?

1. the antecedent

2. the consequent

3. both

4. neither

`ANSWER_1` should be `'1'`, `'2'`, `'3'`, or `'4'`.

Question 2: In backward chaining, after all the variables in a rule have been bound, which part of the rule may appear as a new assertion in the data? (Note, backward chaining is covered in more detail later on in the lab. If you are looking for more background information, see below.)

1. the antecedent

2. the consequent

3. both

4. neither

`ANSWER_2` should be `'1'`, `'2'`, `'3'`, or `'4'`.

Questions 3-5: Hypothetical cats

Consider the following rules about hypothetical cats.

```
rule1 = IF( AND( '(?x) is a hypothetical cat',
                 '(?x) is alive',
                 NOT('(?x) is alive')),
            THEN( '(?x) is a paradox' ) )

rule2 = IF( AND( '(?x) is a hypothetical cat',
                 '(?x) is alive',
                 '(?x) is dead'),
            THEN( "(?x) is Schrodinger's cat" ) )

rule3 = IF( AND( '(?x) is a hypothetical cat',
                 NOT('(?x) is alive'),
                 NOT('(?x) is dead')),
            THEN( '(?x) is amortal' ) )
```

Question 3: Consider the following set of assertions about Kitty.

```
assertions = ( 'Kitty is a hypothetical cat',
               'Kitty is alive',
               'Kitty is dead' )
```

Which rules would match in the first round of forward chaining? Answer with a string of numbers in `ANSWER_3`. (For example, if the assertions match `rule1` and `rule2`, answer `'12'`.) If no rules match, answer `'0'`.

Question 4: Consider the following set of assertions about Nyan.

```
assertions = ( 'Nyan is a hypothetical cat',
               'Nyan is alive',
               'Nyan is not alive' )
```

Which rules would match in the first round of forward chaining? Answer with a string of numbers in `ANSWER_4`. If no rules match, answer `'0'`.

Question 5: Consider the following set of assertions about Garfield.

```
assertions = ( 'Garfield is a hypothetical cat',
               'Garfield likes lasagna' )
```

Which rules would match in the first round of forward chaining? Answer with a string of numbers in `ANSWER_5`. If no rules match, answer `'0'`.

Questions 6-7: Pendergast

In a completely different scenario, suppose we have the following two rules:

```
rule1 = IF( AND( '(?x) has feathers',
                 '(?x) has a beak' ),
            THEN( '(?x) is a bird' ))
rule2 = IF( AND( '(?y) is a bird',
                 '(?y) cannot fly',
                 '(?y) can swim' ),
            THEN( '(?y) is a penguin' ) )
```

and the following list of initial data:

```
( 'Pendergast is a penguin',
  'Pendergast has feathers',
  'Pendergast has a beak',
  'Pendergast cannot fly',
  'Pendergast can swim' )
```

Question 6: After starting the system, which rule fires first? In `ANSWER_6`, answer `'1'` or `'2'`, or `'0'` if neither rule fires.

Question 7: Which rule fires second? In `ANSWER_7`, answer `'1'` or `'2'`, or `'0'` if neither rule fires.

If you're confused about any of the answers, look in `tests.py` for an explanation.

Part 2: Poker Hands

We can use a production system to rank types of poker hands against each other. If we tell it basic things such as `'three-of-a-kind beats two-pair'` and `'two-pair beats pair'`, it would make sense for it to be able to deduce by transitivity that `'three-of-a-kind beats pair'`.

You're given this data about poker hands:

```
poker_data = [ 'two-pair beats pair',
               'three-of-a-kind beats two-pair',
               'straight beats three-of-a-kind',
               'flush beats straight',
               'full-house beats flush',
               'straight-flush beats full-house' ]
```

Write a one-rule system that finds all other combinations of which poker hands beat which, transitively, given some of the rankings already. For example, it should be able to deduce that a three-of-a-kind beats a pair, because a three-of-a-kind beats two-pair and a two-pair beats a pair. The rankings (data) are all provided in the form `'(?x) beats (?y)'`.

Put your one rule in the section "Part 2" of `lab4.py`, and assign it to the variable `transitive_rule`, so that your list of rules is `[transitive_rule]`.

You can test your `transitive_rule` on two additional data sets by uncommenting some or all of the `pprint` (short for pretty print) statements in `lab4.py`:

```
pprint(forward_chain([transitive_rule], abc_data))
pprint(forward_chain([transitive_rule], poker_data))
pprint(forward_chain([transitive_rule], minecraft_data))
```

Note, after uncommenting one of the pretty print statements, you can execute lab4.py directly so that you don't have to run the full tester every time.

Part 3: Family relations

You will be given data that includes two kinds of statements:

- `'person (?x)'`: *x* is a person
- `'parent (?x) (?y)'`: *x* is a parent of *y*

Every person in the data set will be explicitly defined as a person.

Your task is to deduce, wherever you can, the following relations:

- `'sibling (?x) (?y)'`: *x* is the sibling of *y* (*x* and *y* are different people, but share at least one parent)
- `'child (?x) (?y)'`: *x* is the child of *y*
- `'cousin (?x) (?y)'`: *x* and *y* are cousins (a parent of *x* and a parent of *y* are siblings, but *x* and *y* are not siblings)
- `'grandparent (?x) (?y)'`: *x* is the grandparent of *y*

- `'grandchild (?x) (?y)'`: *x* is the grandchild of *y*

Note that for this problem, you are **not** limited to only defining rules that generate one of the five familial relations enumerated above. You are welcome to include rules that inform other relations. You're also welcome to implement additional familial relations such as `great-grandparent` or `nibling` (https://en.wiktionary.org/wiki/nibling), if you feel so inclined.

Keep in mind that some relations are symmetric, so you need to include them both ways. For example, if *a* is a cousin of *b*, then *b* is a cousin of *a*.

First, define all your rules individually -- that is, give them names by assigning them to variables. This will enable you to refer to the rules by name and easily rearrange them if you need to. Then, put them together into a list in order, and call it `family_rules`, so that the rules can be plugged into the forward-chaining system.

We've given you two larger sets of test data -- one for the Simpsons family, and one for a family from Harry Potter -- as well as a couple smaller data sets to help with debugging. To debug what happened in your rules, you can set `verbose=True`.

You will write your solution in `lab4.py` in the section labeled "Part 3". Note that `lab4.py` will automatically define a variable called `harry_potter_family_cousins` which will include all the `'cousin (?x) (?y)'` relations you find in the family from Harry Potter, per your rule set. There should be 14 of them.

IMPORTANT: Make sure you implement all five relations defined above. In this lab, the online tester will be stricter, and may test some relations not tested offline.

Goal trees

For the next problem, we're going to need a representation of goal trees. Specifically, we want to make trees out of AND and OR nodes (expressions), much like the ones that can be in the antecedents of rules. (There won't be any NOT nodes in this problem.) They will be represented as `AND(...)` and `OR(...)` objects.

`AND` and `OR` objects

In our production code, `AND` and `OR` are subclasses of `RuleExpression`, which itself is a subclass of the Python built-in type `list`. Hence, you may iterate, slice, index into, and otherwise do any other standard list-like operation on `AND` and `OR` objects.

The leaves of the goal tree are string assertions. The goal atoms may simply be arbitrary symbols or numbers such as `g1` or `3`.

An **AND node** represents a list of sub-goals that are required to complete a particular goal. If all the branches of an AND node succeed, the AND node succeeds. `AND(g1, g2, g3)` describes a goal that is

completed by completing g1, g2, and g3 in order. Similarly, `AND` can take in an iterable of arguments, so `AND([g1, g2, g3])` is equivalent to `AND(g1, g2, g3)`.

An **OR node** is a list of options for how to complete a goal. If any one of the branches of an OR node succeeds, the OR node succeeds. `OR(g1, g2, g3)` is a goal that you complete by first trying g1, then g2, then g3. Similarly, `OR` can take in an iterable of arguments, so e.g. `OR([g1, g2, g3])` is equivalent to `OR(g1, g2, g3)`.

Unconditional success is represented by an AND node with no requirements: `AND()`.

Unconditional failure is represented by an OR node with no options: `OR()`.

Simplifying Nodes

A problem with goal trees is that you can end up with trees that are described differently but mean exactly the same thing. For example,

```
AND(g1, AND(g2, AND(AND(), g3, g4)))
```

is more reasonably expressed as

```
AND(g1, g2, g3, g4)
```

So, we've provided a function called `simplify` that reduces some of these cases to the same tree. `simplify` won't change the order of any nodes, but it will prune some nodes that are fruitless to check, and otherwise simplify more complicated nodes when possible.

We have provided this code for you. However, you should still understand what it's doing, because you can benefit from its effects. You may want to write code that produces "messy" and unsimplified goal trees, because it's easier, and then simplify them with the `simplify` function.

This is how `simplify` simplifies goal trees:

1. If a node contains another node of the same type, absorb it into the parent node. So `OR(g1, OR(g2, g3), g4)` becomes `OR(g1 g2 g3 g4)`.
2. Any AND node that contains an unconditional failure (`OR()`) has no way to succeed, so replace it with unconditional failure.
3. Any OR node that contains an unconditional success (`AND()`) will always succeed, so replace it with unconditional success.
4. If a node has only one branch, replace it with that branch. `AND(g1)`, `OR(g1)`, and `g1` all represent the same goal.
5. If a node has multiple instances of a variable, replace these with only one instance. `AND(g1, g1, g2)` is the same as `AND(g1, g2)`.

Below are some examples of `simplify` in action:

```
simplify(OR(1, 2, AND()))                                    => AND()
simplify(OR(1, 2, AND(3, AND(4)), AND(5)))                   => OR(1, 2, AND(3, 4), 5)
simplify(AND('g1', AND('g2', AND('g3', AND('g4', AND())))))  => AND('g1', 'g2', 'g3', 'g4')
simplify(AND('g'))                                           => 'g'
simplify(AND('g1', 'g1', 'g2'))                              => AND('g1', 'g2')
```

The Joys of an API

We've provided an abstraction for AND and OR nodes, as well as the function that simplifies them, in `production.py`. Everything you need to know about `production.py` is **described below**, so you shouldn't need to read the source code. There is nothing for you to code in this section, but please make sure to understand this representation, because you are going to be building goal trees in the next section.

Going Backwards

Backward chaining is the opposite of forward chaining: we run a production rule system in reverse. We start with a conclusion (the hypothesis), and then we find which rules, when fired, would yield that hypothesis. Then, we test those rules' antecedents to figure out how we can successfully match them against data we have or other rules' consequents.

The Backward Chaining Process

Here's the general theory behind 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 of the rules' consequents that match are possible options, so you'll take the corresponding antecedents and group them together in an OR node: it is sufficient for any *one* matching rule to fire and yield the hypothesis.
- The hypothesis itself should be added as a (leaf) node to the OR node, because existence of the hypothesis in the data is sufficient to conclude the hypothesis.
- 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.

A few things you should be aware of:

- The branches of the goal tree should be in order: the goal trees for earlier rules should appear before (to the left of) the goal trees for later rules. Intermediate nodes should appear before their expansions.
- The output should be fully simplified (you can use the `simplify` function).

- If two different rules tell you to check the same hypothesis, the goal tree for that hypothesis should be included both times, even though it's largely redundant.

Part 4: Backward Chaining

In this problem, we will do backward chaining by starting from a conclusion, and generating a goal tree of *all* of the statements we may need to test. The leaves of the goal tree will be sentences (strings) such as `'opus swims'`, indicating atomic failure or success based on whether or not 'opus swims' is in our assertions list.

We'll run this backward chainer on the `zookeeper` system of rules, a simple set of production rules for classifying animals, which you will find in `data.py`. As an example, here is the goal tree generated for the hypothesis `'opus is a penguin'`:

```
OR(
  'opus is a penguin',
  AND(
    OR('opus is a bird', 'opus has feathers', AND('opus flies', 'opus lays eggs'))
    'opus does not fly',
    'opus swims',
    'opus has black and white color' ))
```

You will write a procedure, `backchain_to_goal_tree(rules, hypothesis)` (in "Part 4" of `lab4.py`), which outputs the goal tree containing the statements you would need to test to prove the hypothesis. Note that this function is supposed to be a general backchainer, so you should not hard-code anything that is specific to a particular rule set. The backchainer will be tested on rule sets other than `zookeeper_rules`.

The rules you work with will be limited in scope, because general-purpose backward chainers are difficult to write. In particular, for this problem, make the following assumptions:

- All variables that appear in a rule's antecedent also appear in its consequent (so there are no "unknown" variables in the antecedent). In other words, you will not need to do backtracking.
- All assertions are positive: no rules will have DELETE clauses or NOT expressions.
- Rule antecedents never have nested `RuleExpression` nodes. For example, an expression such as `(OR (AND x y) (AND z w))` will never appear within an antecedent, because that contains an AND expression nested under an OR expression.
- Rule consequents always have just a single statement.

Note that an antecedent can be a single hypothesis (a string) or a `RuleExpression`.

Taking advantage of your visual problem-solving apparatus

As a species, humans are very visual learners. If you're having trouble conceptualizing what should be going on in the backward chaining algorithm, we strongly recommend drawing a diagram and working your way down the goal tree by hand.

production.py API

The code in `production.py` has been written by 6.034 staff. It contains all of the infrastructure supporting IF/THEN rules and how they're processed. So that you don't have to trawl through our code trying to figure out how everything works, we use this section to provide you with a quick explanation of which functions you will find useful and how you should interact with them.

`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)`, which 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.

Note: `{}` and `None` are both `False`-like values in Python, so you should make sure to explicitly check if `match`'s return value is `None` (i.e., don't use idioms such as "`if match(a, b):`"). If `match` returns `{}`, that means that the statements match but there are no variables that need to be bound, whereas if `match` returns `None`, that means it's not possible for the statements to match at all!

`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. Note that the expression input to `populate` may either be a string or a more complicated tree.

Examples:

```
populate("(?x) is a (?y)", { x: "John", y: "student" }) => "John is a student"
populate(AND("(?x) is a (?y)", "(?x) loves (?z)"), { x: "John", y: "student" }) => AND("John
is a student", "John loves (?z)")
```

`rule.antecedent()`

Returns the IF part of a rule, which is either a leaf or a `RuleExpression`.

Recall that `RuleExpression` objects act like lists, so you can iterate over them. If you need to know to what class an antecedent belongs, you may find the `isinstance` function to be helpful. For example,

`isinstance(my_antecedent, OR)` returns `True` if `my_antecedent` is an `OR` object, otherwise it returns `False`.

`rule.consequent()`

Returns the THEN part of a rule, which is *always a single statement* for the purposes of this lab. (Note that `rule.consequent()` does *not* return a `THEN` object; it returns the datum enclosed by the `THEN` object.)

Survey

Please answer these questions at the bottom of your lab file:

- `NAME`: What is your name? (string)
- `COLLABORATORS`: Other than 6.034 staff, whom did you work with on this lab? (string, or empty string if you worked alone)
- `HOW_MANY_HOURS_THIS_LAB_TOOK`: Approximately how many hours did you spend on this lab? (number or string)
- `WHAT_I_FOUND_INTERESTING`: Which parts of this lab, if any, did you find interesting? (string)
- `WHAT_I_FOUND_BORING`: Which parts of this lab, if any, did you find boring or tedious? (string)
- (optional) `SUGGESTIONS`: What specific changes would you recommend, if any, to improve this lab for future years? (string)

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)

When you're done, run the online tester to submit your code.

FAQ

Q: I'm getting a somewhat baffling "unhashable type: list" error when working on the family rules part.

A: This probably means your syntax for THENs is incorrect. A THEN with multiple consequents should look like THEN('consequent 1', 'consequent 2', 'consequent 3').