Lab 1

To work on this problem set, you will need to get the code, much like you did for Lab 0.

Most of your answers belong in the main file `lab1.py`. However, the more involved coding problems in section 2 have their own separate files.

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.

Those who think recursively may also benefit from the Python function `reduce` (the equivalent of Scheme's "fold-left" or "accumulate"), documented in the Python library reference.

Forward chaining

Explanation

This section is an explanation of the system you'll be working with. There aren't any problems to solve. Read it carefully anyway.

This problem set will make use of a *production rule system*. The system is given a list of rules and a list of data. The rules look for certain things in the data -- these things are the *antecedents* of the rules -- and usually produce a new piece of data, called the *consequent*. Rules can also delete existing data.

Importantly, rules can contain variables. This allows a rule to match more than one possible datum. The consequent can contain variables that were bound in the antecedent.

A rule is an expression that contains certain keywords, like IF, THEN, AND, OR, and NOT. An example of a rule looks like this:

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

This could be taken to mean:

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

Given data that look like `'parent marge bart'` and `'parent marge lisa'`, then, it will produce further data like `'sibling bart lisa'`. (It will also produce `'sibling bart bart'`, which is something that will need to be dealt with.)

Of course, the rule system doesn't know what these arbitrary words "parent" and "sibling" mean! It doesn't even care that they're at the beginning of the expression. The rule could also be written like this:

```
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 like `'marge is a parent of lisa'`. This gets wordy and involves some unnecessary matching of symbols like `'is'` and `'a'`, and it doesn't help anything for this problem, but we'll write some later rule systems in this English-like way for clarity.

Just remember that the English is for you to understand, not the computer.

Rule expressions

Here's a more complete description of how the system works.

The rules are given in a specified order, and the system will check each rule in turn: for each rule, it will go through all the data searching for matches to that rule's antecedent, before moving on to the next rule.

A rule is an expression that can have an IF antecedent and a THEN consequent. Both of these parts are required. Optionally, a rule can also have a DELETE clause, which specifies some data to delete.

The IF antecedent can contain AND, OR, and NOT expressions. AND requires that multiple statements are matched in the dataset, OR requires that one of multiple statements are matched in the dataset, and NOT requires that a statement is *not* matched in the dataset. AND, OR, and NOT expressions can be nested within each other. When nested like this, these expressions form an AND/OR tree (or really an AND/OR/NOT tree). At the bottom of this tree are strings, possibly with variables in them.

The data are searched for items that match the requirements of the antecedent. Data items that appear earlier in the data take precedence. Each pattern in an AND clause will match the data in order, so that later ones have the variables of the earlier ones.

If there is a NOT clause in the antecedent, the data are searched to make sure that *no* items in the data match the pattern. A NOT clause should not introduce new variables - the matcher won't know what to do with them. Generally, NOT clauses will appear inside an AND clause, and earlier parts of the AND clause will introduce the variables. For example, this clause will match objects that are asserted to be birds, but are not asserted to be penguins:

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

The other way around won't work:

```
AND( NOT( '(?x) is a penguin' ),  # don't do this!
     '(?x) is a bird' )
```

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

Only one rule can fire at a time. When a rule successfully fires, the system changes the data appropriately, and then starts again from the first rule. This lets earlier rules take precedence over later ones. (In other systems, the precedence order of rules can be defined differently.)

Running the system

If you `from production import forward_chain`, you get a procedure `forward_chain(rules, data, verbose=False)` that will make inferences as described above. It returns the final state of its input data.

Here's an example of using it 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 like `'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 copy-and-pasted the code above and ran it as a python script:

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

NOTE: The `Rule:` and `Added:` lines come from the verbose printing. The final output is the set of assertions after applying the forward chaining procedure.

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

Multiple choice

Bear the following in mind as you answer the multiple choice questions in lab1.py:

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

Rule systems

Poker hands

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

Write a one-rule system that ranks poker hands (or anything else, really) transitively, given some of the rankings already. The rankings will all be provided in the form `'(?x) beats (?y)'`.

Call the one rule you write `transitive-rule`, so that your list of rules is `[ transitive-rule ]`.

Just for this problem, it is okay if your transitive rule adds `'X beats X'`, even though in real-life transitivity may not always imply reflexivity.

Family relations

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

- `'male x'`: *x* is male
- `'female x'`: *x* is female
- `'parent x y'`: *x* is a parent of *y*

Every person in the data set will be defined to be either male or female.

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

- `'brother x y'`: *x* is the brother of *y* (sharing at least one parent)
- `'sister x y'`: *x* is the sister of *y* (sharing at least one parent)
- `'mother x y'`: *x* is the mother of *y*
- `'father x y'`: *x* is the father of *y*
- `'son x y'`: *x* is the son of *y*
- `'daughter x y'`: *x* is the daughter of *y*
- `'cousin x y'`: *x* and *y* are cousins (a parent of *x* and a parent of *y* are siblings)
- `'grandparent x y'`: *x* is the grandparent of *y*
- `'grandchild x y'`: *x* is the grandchild of *y*

You will probably run into the problem that the system wants to conclude that everyone is his or her own sibling. To avoid this, you will probably want to write a rule that adds `'same-identity (?x) (?x)'` for every person, and make sure that potential siblings don't have `same-identity`. (Hint: You can assume that every person will be mentioned in a clause stating his gender (either male or female)). The order of the rules will matter, of course. Note that it's fine to include statements that are not any of the specified relations (such as `same-identity` or `sibling`).

Some relationships are symmetrical, and you need to include them both ways. For example, if *a* is a cousin of *b*, then *b* is a cousin of *a*.

As the answer to this problem, you should provide a list called `family-rules` that contains the rules you wrote in order, so it can be plugged into the rule system. We've given you two sets of test data: one for the Simpsons family, and one for the Black family from Harry Potter.

`lab1.py` will automatically define `black_family_cousins` to include all the `'cousin x y'` relationships you find in the Black family. There should be 14 of them.

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

Backward chaining and goal trees

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, much like the ones that can be in the antecedents of rules. (There won't be any NOT nodes.) They will be represented as AND() and OR() objects. Note that both 'AND' and 'OR' inherit from the built-in Python type 'list', so you can treat them just like lists.

Strings will be the leaves of the goal tree. For this problem, the leaf goals will simply be arbitrary symbols or numbers like `g1` or `3`.

An **AND node** represents a list of subgoals 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.

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.

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

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 you a function that reduces some of these cases to the same tree. We won't change the order of any nodes, but we will prune some nodes that it is fruitless to check.

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

This is how we simplify 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)`.

We've provided an abstraction for AND and OR nodes, and a function that simplifies them, in `production.py`. There is nothing for you to code in this section, but please make sure to understand this representation, because you're going to be building goal trees in the next section. Some examples:

```
 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')
```

Backward chaining

Backward chaining is running a production rule system in reverse. You start with a conclusion, and then you see what statements would lead to it, and test to see if those statements are true.

In this problem, we will do backward chaining by starting from a conclusion, and generating a goal tree of *all* the statements we may need to test. The leaves of the goal tree will be statements like `'opus swims'`, meaning that at that point we would need to find out whether we know that Opus swims or not.

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 `zookeeper.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)`, which outputs the goal tree.

The rules you work with will be limited in scope, because general-purpose backward chainers are difficult to write. In particular:

- You will never have to test a hypothesis with unknown variables. All variables that appear in the antecedent will also appear in the consequent.
- All assertions are positive: no rules will have DELETE parts or NOT clauses.
- Antecedents are not nested. Something like `(OR (AND x y) (AND z w))` will not appear in the antecedent parts of rules.

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

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.

Other requirements:

- 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 simplified as in the previous problem (you can use the `simplify` function). This way, you can create the goal trees using an unnecessary number of OR nodes, and they will be conglomerated together nicely in the end.
- 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 seems a bit redundant.

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.

Survey

Please answer these questions at the bottom of your `lab1.py` file:

- How many hours did this problem set take?
- Which parts of this problem set, if any, did you find interesting?
- Which parts of this problem set, if any, did you find boring or tedious?

6.034 Artificial Intelligence

Fall 2010