6.034 Artificial Intelligence, Fall 2006
Prof. Patrick H. Winston

Problem Set 1

This problem set is due Wednesday, September 20. If you have questions about it, ask the TA email list. Your response will probably come from a TA.

To work on this problem set, you will need to get the code, much like you did for PS0.

> The code can be downloaded from the assignments section.

Most of your answers belong in the main file `ps1.scm`. However, the more involved coding problems in section 2 have their own separate files.

You may want the ability to use the list operations `filter`, `foldl`, and `foldr`. You'll be able to use these if you load them, by including this line at the top of your file:

```
(require (lib "list.ss"))
```

Also remember that you can get at the functions in another .scm file with the (load) procedure:

```
(load "production.scm")
```

See Using DrScheme for more.

1. Forward chaining

1.1. 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, which work just like they did in ps0. 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 a Scheme-like 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, where they look like Scheme functions. 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.

1.1.1. 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.
- A rule is an expression that can have an IF part, a THEN part, and a DELETE part. Any of these parts may be omitted.
- The IF part (the 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 with in each other and can contain ordinary patterns, but they can't appear in the middle of an ordinary pattern.
- 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, 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))
```

We're going to define the terms *trigger* and *fire*. The word *trigger*, unfortunately, has been used inconsistently in the materials for this course over the years, and we'll try to avoid it from now on.

- If the IF part matches, the rule **triggers**, meaning that it is checked to see if it will have an effect. The expression in the THEN part (with variables substituted appropriately) will be added if it's not there already, and the (substituted) expression in the DELETE part will be deleted if it is there. (So in this definition of "trigger", a rule *can* trigger without being able to change the data. You may have learned that it can't. Say that it can for this problem set.)
- If either the THEN or DELETE makes a change, the rule **fires** and the data are updated accordingly. Once a rule fires, the system will start again from the first rule. This lets earlier rules take precedence over later ones.

1.1.2. Running the system

If you `(load "production.scm")`, you get a procedure `(run-production-rules rules data)` that will make inferences as described above. It returns the final state of its data.

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

```
(define theft-rule
  '(IF (you have (? x))
       THEN (i have (? x))
       DELETE (you have (? x))))

(define data
  '((you have apple)
    (you have orange)
    (you have pear)))

(run-production-rules (list theft-rule) data)
```

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.

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

If you're getting confusing results from a rule system, you may want it to display on the screen every rule that fires. You can do this by changing `(define DEBUG #f)` to `(define DEBUG #t)` at the top of production.scm.

1.2. Multiple choice

Bear the following in mind as you answer the multiple choice questions in ps1.scm:

- 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**

1.3. Rule systems

1.3.1. 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 `(list transitive-rule)`.

1.3.2. 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 will 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)

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 generate a datum like `(same-identity (? x) (? x))` for every person, and make sure that potential siblings don't have `same-identity`. The order of the rules will matter, of course.

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.

`ps1.scm` 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.

2. Backward chaining and goal trees

You'll be building on the existing code in this section. Here are some useful helper functions:

- In `match.scm`:
  - `(variable? exp)` - says whether *exp* is a variable expression, such as (? x).
  - `(atom? exp)` - says whether *exp* is an atomic value (that is, not a pair).
  - `(add-to-end elt lst)` - returns a list with *elt* appended onto the end.
  - `(substitute 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. This is also called *instantiating* the expression.
  - `(match pattern datum)` - Remember this from PS0? This attempts to assign values to variables so that *pattern* and *datum* are the same. If it succeeds, it returns those variable values as a list of *bindings*, and if it fails, it returns `#f`.
- In `production.scm`:
  - `(rule-antecedent rule)`: returns the IF part of a rule.

    - `(rule-consequent rule)`: returns the THEN part of a rule.
- In `backchain.scm`:
    - `(has-variables? exp)`: tests if exp has any variable expressions in it.

Here's another function which is built in, but which it will be useful to remember you can use:

- `(member item lst)`: Test whether `lst` contains `item`. Specifically, it looks for `item` in `lst` by comparing it to each element using `equal?`. If it finds an equal item, it returns a true value (which happens to be the list from that point onward), and if it doesn't, it returns `#f`.

2.1. 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 lists that begin with the tag 'AND or 'OR. Untagged lists, or atomic values that aren't lists, 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)`. You can refer to this node with the variable `*succeed*`.

Unconditional failure is represented by an OR node with no options: `(OR)`. You can refer to this node with the variable `*fail*`.

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)`.

We want to be able to reduce 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. Our goal is to get a tree that meets these requirements:

- Levels of the tree alternate between AND and OR nodes; that is, no AND node contains another AND node, and no OR node contains another OR node.

- Every AND or OR node has at least two branches, unless there are no branches in the entire tree (which would happen if the tree represents unconditional success or unconditional failure).
    - Note that it is also possible for there to be no AND or OR nodes, if the tree gets reduced to a single leaf.

You can accomplish this with the following transformations:

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.

Your task is to write a function, `(simplify goal)`, that takes in an arbitrary AND-OR goal tree, and uses the four transformations above to put it in a canonical form that meets the requirements, without changing the order of the nodes that remain. We've provided an abstraction for AND and OR nodes in `goaltree.scm`.

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
```

2.2. 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 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.scm`. As an example, here is the goal tree generated for the hypothesis `(opus is a penguin)`:

```
(AND
```

```
(OR (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 statements to test as a goal tree.

You should only output a statement as a test if there is no way to deduce it from the rules. For example, the goal tree for `(opus is a bird)` is not simply `(opus is a bird)` -- that would be cheap. You can't include `(opus is a bird)` in the goal tree, because you can break down the process of finding out whether Opus is a bird into smaller questions, like whether Opus has feathers.

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. If you're given a hypothesis with variables in it, you should return *fail*, also known as (OR). The given procedure `has-variables?` will be useful.
- All assertions are positive: no rules will have DELETE parts or NOT clauses.

2.2.1. 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.
- The output should be simplified as in the previous problem (you can use the `simplify` function you wrote). 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.

3. Survey

Please answer these questions at the bottom of your `ps1.scm` 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?

(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 tester and submit your .scm files to your `6.034-psets/ps1` directory on Athena.

4. Problems?

If you find what you think is an error in the problem set, tell the TA email list about it.

We've gotten a lot of people saying that `tester-utils.scm` is complaining that the `sort` procedure doesn't exist. If this happens to you:

- Are you sure you're running the latest version (352) of DrScheme? Some people have solved the problem by installing the latest version.
- If that doesn't work for whatever reason, you can change `sort` to `quicksort`, and then it should work.

4.1. Clarification about the word "trigger"

Unfortunately, the word "trigger" has been used inconsistently in materials for this course over the years. We'll try to avoid it from here on. On this problem set, a rule "triggers" if the rule system checks its antecedent and finds that it matches the data.