CSE341 Spring '06 Due Wednesday, April 26
Assignment 3

This assignment continues with the Boolean expression example started in Homework #2, this time for knowledge representation and planning in artificial intelligence.

As I write this, I'm at home. I'm hungry. Omelettes are crossing my mind. Omelettes require eggs. I don't have eggs. To buy eggs, I need to be where eggs are sold. QFC[1] sells eggs... How can I represent this complex and seemingly amorphous collection of wisdom in an orderly way so that a computer could do something with it? Boolean logic to the rescue! The Boolean expressions above contain only Boolean variables and constants. To reason about the world, we introduce *predicates*, which are Boolean-valued functions of (typically) non-Boolean variables. For example, suppose I have a predicate `At` over a *domain* of locations including `Home`, `QFC` and `Ace` (a hardware store). In my current state `At(Home)` is true, but `At(QFC)` and `At(Ace)` are false. Similarly, I might represent knowledge about availability of goods using another predicate `Sells`, whose domain is (locations * products). For example, `Sells(QFC,Eggs)` and `Sells(Ace,Hammers)`.

In general, the world is described by a *state*, which is a list of true predicates. (All others are implictly false.) E.g., a state might be `[At(home), Sells(QFC, Eggs), Sells(Ace, Hammers)]`.

Returning to that which is uppermost in my mind, can I make an omelette? Unfortunately not, because I'm at home and the eggs are at QFC, and I can't buy something unless I'm at a place that sells it. I can codify this principle by saying the *action* `BUY(place, item)` has a *precondition* that must be fulfilled before it can happen, namely `At(place) and Sells(place, item)`.

I'm getting hungrier all the time, but we're closer to having some homework: we have a Boolean expression, but the basic terms in it are predicates, rather than simple Boolean variables like `x` and `y` that we had before. We need one more concept. The predicate `At(QFC)`, involving the location constant `QFC` is called a *ground predicate*, and in our application we'll always know whether each ground predicate is true or false. (In particular, the *state* contains exactly the true ground predicates.) On the other hand, we can't evaluate "`At(place)`", involving the location *variable* `place` without knowing a *binding* of `place` to an actual location. We call `QFC` a *constant atom* and `place` a *variable atom*. We'll represent the difference more explicitly in our ML implementation, but in this write-up, we'll use the convention that constant atoms start with an upper-case letter and variable atoms are lower-case: `QFC`, `Eggs` and `Ace` are constant atoms; `place` and `item` are variable atoms.

So, can I buy eggs? More generally, what can I buy right now? This is equivalent to asking to find all possible bindings of `x` to locations and `y` to items such that the precondition of `BUY(x,y)` is satisfied by my current state. So that's our goal—given a `predicate expr` (as opposed to `string expr` as before) representing a precondition like the one for `BUY(x,y)`, and a state, find all bindings of variables that satisfy the expression.

For example, take the following state.

```
start  = [ At(Home), Sells(QFC, Eggs), Sells(QFC, Bread), Sells(Ace, Hammers) ]
```

Then the set of all assignments to `x` and `y` satisfying the precondition `At(x) and Sells(x,y)` is empty, but if I am in the following state

```
start  = [ At(QFC), Sells(QFC, Eggs), Sells(QFC, Bread), Sells(Ace, Hammers) ]
```

[1] the University of Washington makes no endorsement of companies, goods, services or products mentioned on this page.

then the set of satisfying bindings for the precondition is

```
[[(x, QFC), (y, Eggs)], [(x, QFC), (y, Bread)]]
```

As usual, we'll break the problem into a number of steps. One point to bear in mind is that the particular constants and predicates like `Home`, `QFC` and `Sells` used in the discussion above are just examples; your code should work for any input of this sort. (And if this all seems a little daunting, rest assured that the solution is significantly shorter than this write-up!)

In ML, we will express predicates as

```
datatype atom = AtomConst of string | AtomVar of string
type predicate = { pred : string, vals : atom list }
```

A state is then a `predicate list` (where the predicates better be over constant atoms), and a precondition is a `predicate expr`, like the following.

```
val state = [ {pred="At", vals=[AtomConst "home"]},
              {pred="Sells",vals=[AtomConst "QFC", AtomConst "Eggs"]} ];
val buy_precond = And({pred="At", vals=[AtomVar "place"]},
                      {pred="Sells", vals=[AtomVar "place",AtomVar "item"]});
```

In the problem descriptions below, we'll shorten statements like the above to `state=[At(home)]` and `buy_precond = And(At(place),Sells(place,item))`. See **Predicate Shorthand for ML** below to see how to make the shorthand actually work in ML.

For the functions below, you may use all of the functions from Homework #2 (either your solution, or ours, which we will release soon after HW#2 is due).

1. The first thing to do is change variable atoms to constants. Write a Curried function `bindatom` that takes a pair of strings `(x,c)` and an atom `a`, and returns `AtomConst c` if `a` is `AtomVar x`, and otherwise returns `a` unchanged.

2. Write a function `bindpred (x,c) pred` that binds all atoms in `pred` according to the variable `x` and constant `c`. You *must* use `map` and `bindatom`. Note: see how writing `bindatom` as a Curried function makes things easier?

3. Write a function `filter` (from scratch) that takes a function `pred` and a list $a_1, a_2, \ldots$, and returns the list of all a_i for which `pred` a_i is `true`. Using `filter`, write functions `getconsts` and `getvars` that take `atom lists` and return a list of all constant or variable atoms.

4. Let `p` be a predicate. We say that a predicate `q` *matches* `p` if it has the same name, and the corresponding atoms are either both the same constant, or the atom in `p` is variable.

   Write a function `satisfying_matches` that takes a predicate `p` and a state, and produces a list of all value lists from predicates in the state matching `p`. For example,

   ```
   satisfying_matches (Sells(x,y)) [Sells(QFC,Eggs),Sells(Ace,Nails),
                                    At(Home),Sells(QFC,Milk)]
     --> [["QFC","Eggs"],["Ace","Nails"],["QFC","Milk"]]
   satisfying_matches (Sells(QFC,y)) [Sells(QFC,Eggs),Sells(Ace,Nails),
                                      At(Home),Sells(QFC,Milk)]
     --> [["QFC","Eggs"],["QFC","Milk"]]
   ```

5. Write a function `get_binding` that given two lists of atoms of the same length, returns the implied binding (a list of string pairs). E.g., pairing `[x,y]` with `[QFC,Eggs]` should produce the list `[("x","QFC"),("y","Eggs")]`; that is, `x` is bound to `QFC` and `y` is bound to `Eggs`. Similarly, pairing `[QFC,y]` with `[QFC,Eggs]` gives `[("y","Eggs")]`. Define an exception `BadGetBinding` and raise it if the lists are not the same length or otherwise not matching (you can assume `get_binding` will only be run on the atom lists of satisfying matches).

6. What's the point of having a binding if we don't apply it? Write a function `apply_binding` that takes a binding as returned from `get_binding` and a list of predicates, and returns a list of predicates with the binding applied. For example:

```
apply_binding [("x","QFC"),("y","Eggs")]
              [Sells(x,y),Have(y),Have(z),Have(Nails)]
  --> [Sells(QFC,Eggs),Have(Eggs),Have(z),Have(Nails)]
```

   Use `map` and `foldr`; your implementation may *not* be recursive. Hint: a single binding gets mapped across all predicates; the list of bindings is folded into the predicate list.

7. Write a one-line function `bindings_for_one_pred` that, given a predicate p and a state s, returns a list of the bindings for each satisfying match to p in s. Use `map`; your function cannot be recursive. Hint: the Curried implementation of `get_binding` makes this elegant.

8. In a comment, explain the difference—and there should be a difference!—between running `bindings_for_one_pred (At(Home))` with the state `[At(Home)]` and the state `[At(Away)]` (here both `Home` and `Away` are constant atoms).

9. The previous function gives a list of all the bindings for a state that satisfy a single predicate. We now want the bindings that satisfy an entire list of predicates. Write a function `bindings_for_preds` that takes a predicate list and a state and does just that. Use `bindings_for_one_pred`, `map` and `@`. Here are a few examples.

```
bindings_for_preds [At(x),Sells(x,y)] [At(QFC),Sells(QFC,Eggs)]
  --> [[(x,QFC),(y,Eggs)]]
bindings_for_preds [At(x),Sells(x,y)] [At(QFC),Sells(QFC,Eggs),
                                       Sells(QFC,Milk)]
  --> [[(x,QFC),(y,Eggs)],[(x,QFC),(y,Milk)]]
bindings_for_preds [At(x),Sells(x,y)] [At(QFC),Sells(QFC,Eggs)
                                       At(Ace),Sells(Ace,Nails)]
  --> [[(x,QFC),(y,Eggs)],[(x,Ace),(y,Nails)]]
```

10. Suppose we had a precondition for buying things that looked like `(not Broke) and At(place) and Sells(place,item)`. Write this precondition as a `predicate expr` and bind it to `p1` (i.e., `val p1 = ...`). In a comment, give the output of running `satisfying_assignments` on `p1` and explain what that tells us about a state satisfing `p1`.

11. This function puts it all together. Write a function `satisfying_bindings` that takes a state and a precondition, and returns a list of all bindings (if any) for which the state satisfies the precondition. Hint: hopefully after answering the last question you see that for a state to satisfy a predicate, it must find a binding that's good for some assignment returned by `satisfying_assignments`. This can be done by concatenating (via `foldl`?) the results of doing the following to each satisfying assignment `asgn` of the precondition. Take each

binding that satisfies all the true predicates in `asgn`, apply that binding to the false predicates in `asgn`, and keep (via `filter`?) the bindings if the state doesn't include any of those false bindings (via `isect`?). Phew! Think about this carefully before implementing it.

It's essential for this assignment that you really understand what's going on. There's not that much code to write—about 100 lines—but most require careful thought. *Start early!*

Type Bindings

Your solution should generate the following bindings (or their synonyms).

```
datatype 'a expr = And of 'a expr * 'a expr
                 | Or  of 'a expr * 'a expr
                 | Not of 'a expr
                 | Var of 'a
                 | Const of bool
datatype atom = AtomConst of string | AtomVar of string
type predicate = {pred:string, vals:atom list}
val bindatom = fn : string * string -> atom -> atom
val bindpred = fn : string * string -> predicate -> predicate
val filter = fn : ('a -> bool) -> 'a list -> 'a list
val getconsts = fn : atom list -> atom list
val getvars = fn : atom list -> atom list
val satisfying_matches = fn : predicate -> predicate list -> atom list list
exception BadGetBinding
val get_binding = fn : atom list -> atom list -> (string * string) list
val apply_binding = fn
  : (string * string) list -> predicate list -> predicate list
val bindings_for_one_pred = fn
  : predicate -> predicate list -> (string * string) list list
val bindings_for_preds = fn
  : predicate list -> predicate list -> (string * string) list list
val satisfying_bindings = fn
  : predicate list -> predicate expr -> (string * string) list list
```

Predicate Shorthand for ML

The shorthand for predicates I've been using is handy; it saves typing a lot of brackets, atom constructors, etc., and reduces typos. Here's a way to use similar notation in ML.

```
val x = AtomVar "x"; val y = AtomVar "y";
val QFC = AtomConst "QFC"; val Ace = AtomConst "Ace";
val Eggs = AtomConst "Eggs"; val Nails = AtomConst "Nails";
fun At(x) = {pred="At", vals=[x]};
fun Sells(x,y) = {pred="Sells",vals=[x,y]};
```

Now you can type things like `bindings_for_preds [At(x),Sells(x,y)] [At(QFC), Sells(QFC, Eggs)]` directly into ML. Use this for testing! Make up your own predicates! I did...

Extra Credit

Here are some ideas for extra credit. If you have another idea, talk to us. Remember the course policy on extra credit: if you're only concerned about your GPA, it's not worth your time. We've rated these extra credit problems where more stars means a harder problem.

1. (⋆⋆) Write an expression parser that takes a string like `"x and (y or true)"` and returns the corresponding `string expr`. If you're ambitious, use the `IO` structure to interact with Read-Eval-Print loop (see Project 1 from CSE341 in winter 2004).

2. (⋆) Let's formalize *actions*, which describe how to change state. An action is defined over a list of variables, and is a precondition with an *effect*. An effect is a list of *allow* and *deny* predicates. For example, the action `BUY(x,y)` has the precondition `At(x) and Sells(x,y)` and effect {`allow=Have(y),deny=nil`}. An action `GO(x,y)` might have precondition `At(x)` and effect {`allow=At(y),deny=At(x)`}, so that `GO(Home,QFC)` requires that `At(Home)` be true, and changes the state so that `At(Home)` is not present and `At(QFC)` is.

   For this extra-credit, write a datatype for actions, a function that tests if an action can be applied to a state, and a function that applies the allow and deny lists of an action to a state.

3. (⋆⋆) A problem is defined by a list of actions, a starting state and a goal state, and produces a sequence of actions that goes from the start to the goal. In our omelette example, the actions could be

```
GO(x,y)          = precond At(x)
                   effect allow=[At(y)], deny=[At(x)]
BUY(item,place)  = precond At(place) and Sells(place,item)
                   effect allow=[Have(item)], deny=[]
MAKE_OMELETTE    = precond At(Home) and Have(Eggs)
                   effect allow=[Have(Omelette)], deny=[]
```

   The start and goal states would then be

```
start  = At(Home), Sells(QFC,Eggs), Sells(QFC,Bread), Sells(Ace, Hammers)
goal   = Have(Omelette)
```

   A *plan* is a series of actions beginning from the start state that results in a state containing the goal state. In our example, a plan might be `GO(Home,QFC), BUY(QFC,Eggs), GO(QFC,Home), MAKE_OMELETTE`.

   Write a planner in ML. One idea is to use breadth-first search, generating all states reachable by applying valid actions from the start state and searching until the goal is found. A better solution would use iterative-deepening depth-first search to avoid the exponential-space problem of breadth-first search. The key to this problem is finding the right representation of a tree in ML. Don't try to avoid searching the same states multiple times, that will needlessly complicate your solution.