Reasoning about Equality

Consider the following KB. Although it is using the ‘=’, this is for true equality, not explicit unification.

eveningstar = venus
morningstar = venus
planet(venus)

Question 1: Axioms

List all of the equality axioms that should be added to the KB in order to reason about equality.

Question 2: Proof with Axioms

Using the axioms for equality, prove:
?planet(morningstar)

Do this as a top-down proof. For each derivation, show what rule from the KB you used and the substitution.

Warning: Do not use paramodulation. Review the lecture slides for an example of doing a proof with the axioms.

Question 3: Paramodulation

For paramodulation, what are the rewrite rules that the proof procedure can use, in addition to the resolution rule.

Question 4: Proof with Paramodulation

Prove planet(morningstar) using paramodulation.

Python

Question 5: Unique Name Assumption

You are to modify a depth-first theorem prover so that it implements the unique name assumption. Let notequal X Y be a predicate that is true if X and Y have different names, and hence, under the unique name assumption, are different objects. You need to be careful in how you evaluate notequal in your theorem prover. As discussed in the lecture notes and page 239 of the textbook, here is how you should evaluate it.

• If X and Y cannot be unified, succeed.
• If X and Y are identical (including having the same variables in the same positions), fail.
• Otherwise, you need to delay evaluating this.

Your code should be able to work even if there are multiple notequal that must be delayed.
Modify the code below (a working version of the depth-first theorem prover from homework 4). Make a procedure called `delay` that takes one argument, an atom. This should return true if the atom needs to be delayed. After you assign fresh variables to the answer clause and execute ‘set neighbors {}’, call `delay` on each conjunct in the answer clause until you find one that does not need to be delayed. If it is `notequal`, evaluate it and update neighbors appropriately; otherwise, check for the rules in the knowledge base to apply to the atom. Do not be concerned if there is some duplication in your code between the `delay` procedure and your code that evaluates `notequal`. We are more concerned with correctness than efficiency.

Use the following code to start with, which is the solution to the depth-first theorem prover from Homework 4.

```python
def prove(query, kb):
    vars = findvariables(query, [])
    print("Vars in query are %s" % vars)
    answer = [['yes'] + vars] + query
    print("Initial answer clause is %s" % prettyclause(answer))

    # the initial frontier is a list whose only element is the initial
    # answer clause
    frontier = [answer]

    while frontier:
        # give answer clause fresh variables
        answer = frontier[0]
        if len(answer) == 1:
            yesatom = answer[0]
            str = "Proof:"
            for var, val in zip(vars, yesatom[1:]):
                str += " %s=%s" % (var, prettyexpr(val))
            print(str)
            return True

        print("Trying to prove : %s" % prettyclause(answer))
        # give it fresh variables
        answer = freshvariables(answer)
        print(" after fresh vars : %s" % prettyclause(answer))

        neighbors = []
        for rule in kb:
            subs = {}
            if not unify(rule[0], answer[1], subs):
                continue

            print(" using rule %s" % prettyclause(rule))
            print(" proven by %s with %s" % (rule[0], subs))
            # create answer clause
            answercopy = answer[0:1] + rule[1:] + answer[2:]
            # apply substitution
            answercopy = substitute(answercopy, subs)
            print(" result: " + prettyclause(answercopy))
            neighbors.append(answercopy)

        frontier = neighbors + frontier[1:]
    return False
```

Make sure that your prover works on the following version of the multicolumn tower. Note that the atoms in the body of the clause have been specifically arranged so that delaying will need to be done. Also note
that a **unify** predicate is defined. This is just doing normal datalog unification, and is defined below as simply **unify**(*X*, *X*). For those of you doing this in Python (the easier solution), you can use **[p Top Rest]** notation for lists.

```prolog
mct2(s(0), [Block]) :-
  block(Block).

mct2(s(X), NewTower) :-
  notequal(X, 0),
  mct2(X, Tower),
  unify(Tower, [Top|_]),
  notequal(TopColor, BlockColor),
  differentfromlist(Block, Tower),
  color(Top, TopColor),
  block(Block),
  color(Block, BlockColor),
  unify(NewTower, [Block|Tower]).
```

```prolog
differentfromlist(X, [Top|Rest]) :-
  notequal(X, Top),
  differentfromlist(X, Rest).
differentfromlist(X, []).
unify(X, X).
```

Make sure that you can do the following proof:

```
prove(['mct2', ['s', ['s', ['s', ['s', '0']]]], 'List'])
```

Hand in a copy of your knowledge base.

Hand in a copy of your prove and delay procedures.

---

**Prolog**

**Question 6: Cliques**

Consider the following prolog knowledge base.

```prolog
node(1). node(2). node(3). node(4). node(5). node(6).
node(7). node(8). node(9). node(10). node(11). node(12).
node(13). node(14). node(15).
c(1,2). c(1,3). c(1,5). c(1,7). c(1,8). c(1,11).
c(2,4). c(2,8). c(2,12). c(3,5). c(3,7). c(3,11).
c(4,6). c(5,7). c(5,11). c(6,8). c(7,11). c(8,11).
c(9,12). c(10,12). c(13,14). c(13,15). c(14,15).
```

This knowledge base is a graph, with each node represented with the predicate `node`, and each bi-directional arc represented with the predicate `c`.

Make a datalog program for finding a *k*-clique in the graph. A *k*-clique is a set of *k* distinct nodes that are all connected to each other. You should make a recursive definition. You must define all predicates that you use, and cannot use any prolog capabilities that are outside of datalog.

We will be assuming the unique name assume, so you can use `not(X=Y)`, if you are sure that the X and Y will be instantiated when that atom is evaluated by Prolog.

Use Prolog’s lists `[Top|Rest]` and Prolog’s numbers, and use the built-in ‘is’ and ‘>’. Again, you need to make sure that the appropriate parameters are bound.
You should make sure that your code gives you appropriate alternatives when you press the ‘;’ after it gives an answer. Your code should not go into an endless loop, or return incorrect solutions.

To avoid your code supplying many different variations of the same solution, impose an additional constraint that cliques need to be grown in ascending order of the nodes. In other words, when adding a new node to an existing clique, make sure the new node is greater in value than the nodes in the existing clique (this is why I used numbers to represent the nodes).

Hand in a copy of your Prolog code.

**Question 7: Prolog Part II**

Now, we will focus on efficiency. We will count the number of times that Prolog gets the name of a block. We will use the predicate `cnt` to hold the current number of times that `node` has been called. Use `countclique` to call your clique predicate, which will reset `cnt` to 0. Use the following `getnode` predicate instead of directly calling `node`.

```prolog
countclique(N,Result) :-
  retractall(cnt(_)),
  assert(cnt(0)),
  clique(N,Result).

countclique(_,_) :-
  cnt(N),
  print(N).

getnode(Node) :-
  node(Node),
  retract(cnt(Cnt)),
  Cnt1 is Cnt + 1,
  assert(cnt(Cnt1)).
```

Look up on the web what `retract` and `assert` do. Explain how they are affected when Prolog backtracks. Explain how they can keep track of the actual number of times that node is called, regardless of whether backtracks.

Also explain what the second case of `countclique` is doing.

**Question 8: Prolog Part III**

Modify your code so that you use the above two predicates (`countclique` and `getnode`). My code is able to fully search for all 1-cliques with executing `getnode` 15 times, 2-cliques with 240 times, 3-cliques with 585 times (and found 13 different ones), 4-cliques with 780 times (and found 5 different ones), and 5-cliques with 855 times (and found 1 one).

Your code should give a similar amount. If your code is reporting greater numbers, try to determine what it might be doing that is inefficient to improve your code.

Hand in a your code, and report how how many times `getnode` was called, and how many solutions you found, for finding 1, 2, 3, 4, and 5-cliques.