Reasoning about Equality: Hard

This question is a harder version of the previous one. You might want to attempt this question after you have finished with the rest of the assignment.

\[
\text{ssn(johnsmith)} = 123456789 \\
\text{ssn(johnwsmith)} = 123456789 \\
X = Y \leftarrow \text{ssn(X)} = \text{ssn(Y)}
\]

Question 1: KB

Explain in English what is meant by the third clause in the KB. What should we be able to conclude from this KB.

Question 2: Axioms

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

Question 3: Proof with Axioms

Using the axioms for equality, prove:
\(?\text{johnsmith}=\text{johwsmith}\)

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

Question 4: Paramodulation

Could you have done this proof using paramodulation? Why or why not?

Python

Question 5: Unique Name Assumption

You are to modify a depth-first theorem prover so that it implements the unique name assumption. Let \text{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 \text{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 \text{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 \text{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 `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]
            s = "Proof:"
            for var, val in zip(vars, yesatom[1:]):
                s += " %s=%s" % (var, prettyexpr(val))
            print(s)
            return True

        print("Trying to prove : %s" % prettyclause(answer))
        # give it fresh variables
        answer = freshvariables(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)
            neighbors.append(answercopy)

        frontier = neighbors + frontier[1:]
        print("Added %d neighbors. Frontier now has size %d\n" %
              (len(neighbors), len(frontier)))

    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)`. You can use `[p Top Rest]` notation for lists.

```python
# unification predicate
def unify(X, Y):
    # unification with normal datalog
    return X == Y
```
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]).

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','0'],['p','Block','nil']],[[mct2,['s','X'],'NewTower']],[notequal,'X','0'],[mct2,'X','Tower'],[unify,'Tower',[p,'Top','Rest']],notequal,'TopColor','BlockColor'],[differentfromlist,'Block','Tower'],[color,'Top','TopColor'],[block,'Block'],[color,'Block','BlockColor'],[unify,'NewTower',[p,'Block','Tower']],[[differentfromlist,'X',[p,'Top','Rest']],[notequal,'X','Top'],[differentfromlist,'X','Rest']],[[differentfromlist,'X','nil']],[unify,'X','X'],[block,'red1'],[color,'red1','red'],[block,'red2'],[color,'red2','red'],[block,'red3'],[color,'red3','red'],[block,'red4'],[color,'red4','red'],[block,'red5'],[color,'red5','red'],[block,'red6'],[color,'red6','red'],[block,'red7'],[color,'red7','red'],[block,'gre1'],[color,'gre1','green'],[block,'gre2'],[color,'gre2','green'],[block,'gre3'],[color,'gre3','green'],[block,'blu1'],[color,'blu1','blue'],[block,'blu2'],[color,'blu2','blue'],[block,'yel1'],[color,'yel1','yellow'],[block,'bla1'],[color,'bla1','black']]

Here is a copy of the KB in python list notation (to save you the hassle of converting this yourself).

kb = [[[mct2,['s','0'],['p','Block','nil']],[block,'Block']],[[mct2,['s','X'],'NewTower'],[notequal,'X','0'],[mct2,'X','Tower'],[unify,'Tower',[p,'Top','Rest']],[notequal,'TopColor','BlockColor'],[differentfromlist,'Block','Tower'],[color,'Top','TopColor'],[block,'Block'],[color,'Block','BlockColor'],[unify,'NewTower',[p,'Block','Tower']],[[differentfromlist,'X',[p,'Top','Rest']],[notequal,'X','Top'],[differentfromlist,'X','Rest']],[[differentfromlist,'X','nil']],[unify,'X','X'],[block,'red1'],[color,'red1','red'],[block,'red2'],[color,'red2','red'],[block,'red3'],[color,'red3','red'],[block,'red4'],[color,'red4','red'],[block,'red5'],[color,'red5','red'],[block,'red6'],[color,'red6','red'],[block,'red7'],[color,'red7','red'],[block,'gre1'],[color,'gre1','green'],[block,'gre2'],[color,'gre2','green'],[block,'gre3'],[color,'gre3','green'],[block,'blu1'],[color,'blu1','blue'],[block,'blu2'],[color,'blu2','blue'],[block,'yel1'],[color,'yel1','yellow'],[block,'bla1'],[color,'bla1','black']]

Hand in a copy of your prove and delay procedures.
Prolog

Question 6: Reducing to CNF: Step 1

In the class lecture notes, there is an algorithm for converting an arbitrary expression of disjunctive and negative knowledge expressed with $\neg$, $\land$, $\lor$ and $\leftarrow$ into conjunctive normal form (CFN). The first step is to replace any expressions of $\phi \leftarrow \psi$ to $\phi \lor \neg \psi$.

Implement a prolog predicate called step1 that will do the step 1 of the conversion. The first argument will be an input (completely grounded) and the second argument will return the expression with no implication signs.

Note that the implication might be arbitrarily deep in the expression and that there might be more than one. Also, you should use the cut statement to implement and if-then-else structure. Do not use cut for any other purpose. When using cut for an if-then-else, you will have multiple clauses defining the same predicate. Usually the head of each clause is different as it is specifying a pattern to match against. Hence, you would put a ‘!’ as the first atom to lock prolog’s search into that option. This way, you can include a generic case as the final clause that will implement the ‘else’.

Make sure you only get a single answer for any conversion. Here is my code for testing the routine, and the answers. Note that I am using ‘findall’ in the test code. That is why the answers are in a list. Findall will have Prolog automatically look for alternate proofs, for which none should be found. That is why there should just be a single item in each list.

```
   test1 :-
      findall(X, step1(impliedby(d,e),X), L1),
      write(L1), nl,
      findall(X, step1(impliedby(d,not(e)),X), L2),
      write(L2), nl,
      findall(X, step1(impliedby(d,impliedby(e,f)),X), L3),
      write(L3), nl,
      findall(X, step1(impliedby(impliedby(a,b),impliedby(e,f)),X), L4),
      write(L4), nl,
      findall(X, step1(and(a,(or(b,or(c,(not(impliedby(d,e))))))),X), L5),
      write(L5), nl.
```

```
   [or(d,not(e))]
   [or(d,not(not(e)))]
   [or(d,not(or(e,not(f)))))]
   [or(or(a,not(b)),not(or(e,not(f)))))]
   [and(a,or(b,or(c,not(or(d,not(e))))))]
```