First Order Predicate Calculus

Question 1

Write the following expressions in First Order logic (using ∀, ∃, ∧, ∨, ←). Make sure you use capital letters for variables. Also, you can use the above precedence rules in order not to show all brackets.

Everyone is either a male or a female, but not both.
All female vegetarians are butchers.
Only female vegetarians are butchers.
All butchers are female vegetarians.
No woman likes a butcher who is male.

Strips and Blocks World

Block World
We will define the blocks world as follows.
It has static predicate block(X) which is true if X is a block.
It has static predicate table(X) which is true if X is the table.
It has the primitive predicate on(A,B) if block A is on B where B is a block or B is the table.
It has the primitive predicate clear(A) if block A has no block on top of it and it is not being held by the robot.
The robot can only carry one block at time S. If it’s hand is empty in situation S, then the primitive relation empty will be true.
If the robot is holding block B, holding(B) will be true.
It has the action pickup(Obj,X), which it can execute if Obj is a block and Obj is clear and the robot is not already holding anything and Obj is current on X, where X is the table or a bock.
It has the action putdown(Obj,X), which it can execute if it is holding Obj, and X is a block and clear, or X is the table. (Note that we are allowing at most one block to be directly on top of another block.)

Question 2: Actions

Encode the actions using the Strips formalism (preconditions, delete list, add list). Note that pickup and putdown will each have two definitions, one for the case where the second argument is a table and one for it being a block.

Question 3: Initial World

Encode the world with blocks a, b and c, and table t. Block c is on a and a is on the table, and b is on the table.

Question 4: Plan

Write the sequence of actions that would result in a world in which on(a,b) and on(b,c).
For your sequence of actions, give the sequence of worlds (in terms of all primitive relations). Ensure that each subsequent world was derived by applying the add and delete lists. (This is a good way to check if your delete list and add list is right.)

**Strips and Tower of Hanoi**

In the tower of Hanoi, you have three pegs, pega, pegb and pegc. You also have 4 discs: small, medium, large and grande. A disc can only be on a bigger disk or be on one of the pegs.

**Question 5: Static Relations**

Write the static relations that encode that (a) pega pegb and pegc are pegs; (b) small, medium, large and grande are discs. And, write the 6 predicates that encode bigger(X,Y) if disc X is bigger than disc Y. (You could do this with 3 static facts and two rules, but then we would need our planner to handle derived rules. So let's just explicitly mention all pairs X and Y where X is bigger than Y.)

**Question 6: Primitive Relations in Initial World**

In the initial world, small is on medium, which is on large, which is on grande, which is on pega. Write the primitive relations for the initial world using on(X,Y) and clear(X). on(X,Y) is true if disc X is on Y where Y is either a disc or a peg. clear(X) is true if X is a disc or peg with nothing on it.

**Question 7: Goal**

Let the goal be that small is on medium, which is on large, which is on grande, which is on pega. Write this in Strips.

**Question 8: Actions**

We have a single move operation, called move(X,Y,Z) where we move disc X, which is clear, off of Y, where Y is a disc or a peg and put it on Z, where Z is a disc that is bigger than X or Z is a peg, and where Z is clear. Write the strips action definitions for this action. You should be able to do this with 2 versions of the action.

**Situation Semantics Representation**

For these questions, you will use the Tower of Hanoi problem.

**Question 9**

For representing the initial state of the world in situation semantics, how do you change the facts that define the primitive predicates (on and clear). Do you need to make this change to the static primitives (bigger, peg, and disc)?
Question 10

Give the rules that define ‘poss’ for the move action. Just as with the Strips definition, you might have two rules for it.

Question 11

Give the axioms for the primitive relations (clear and on) in terms of the current action being done and the previous state. You need to specify both the ‘change’ axioms and the frame axioms.

Python: Forward Planner

You will implement a forward chaining planner for Strips. You do not need to worry about derived relations, as the blocks world has been formulated not to need them. Hence, we will not have to do any reasoning inside of a world.

Your forward planner should be breadth first.

The initial world will be on(a,b) on(b,c) on(c,t) and where a, b, and c are blocks and t is a table.

The goal is on(c,b) on(b,a) on(a,t).

You must use the following code. Use the class Action to add the four actions for the domain. I have also provided you with two routines: subset(X,Y) which returns true if X is a subset of Y, and remove(X,Y), which will remove any elements of X from Y. You should use these in filling in the missing code (marked with ‘YOUR CODE HERE’).

from hw4standard import *

class Action:
    all = []
    def __init__(self,head,pre,add,dele):
        self.head = head
        self.pre = pre
        self.add = add
        self.dele = dele
        Action.all.append(self)

        vars = findvariables(head,[])
        vars = findvariables(pre,vars)
        self.vars = vars

    def __str__(self):
        s = "Action %s
        s += " Preconditions:
        for i in self.pre:
            s += " %s" % prettyexpr(i)
        s += "\n"
        s += " Add list:  
        for i in self.add:
            s += " %s" % prettyexpr(i)
        s += "\n"
        s += " Delete list: 
        for i in self.dele:
            s += " %s" % prettyexpr(i)
        s += " Variables are"
for i in self.vars:
    s += " " + i
return s

def subset(sub,set):
    for i in sub:
        if not i in set:
            return False
    return True

def remove(sub,set):
    newset = []
    for i in set:
        if not i in sub:
            newset.append(i)
    return newset

def printWorld(world,indent):
    str = indent
    for p in world:
        if p[0] in Primitives:
            str += " " + prettyexpr(p)
    print(str)

def printPlan(plan,indent):
    str = indent
    for p in plan:
        str += " " + prettyexpr(p)
    print(str)

def prove():
    worldcnt = 0
    # each item on frontier is a tuple of plan + world that results from executing plan
    frontier = [[[],Initial]]

    while frontier:
        topPlan,topWorld = frontier[0]

        print("Current plan:")
        printPlan(topPlan," ")

        print("Current world (primitives):")
        printWorld(topWorld," ")

        # check if the topWorld has goal true in it
        # if so, say that the goal was found and return true

        # YOUR CODE HERE

        # Your code should be able to find the plan after exploring all plans of length 6
        # I put in this stop code in case your code has a bug, so that you won’t keep going forever
        print(""")
        if len(topPlan) > 6:
            print("Could not find plan in 6 steps")
            return False

        neighbors = []
for action in Action.all:

    # We could use a theorem prover to find variable instantiations that make the
    # the preconditions true.
    # Instead, let's enumerate over all variable instantiations

    #----------------------
    # The code between the two sets of dashes iterates all variable instantiations
    # One of the questions asks you to explain how it works

    vars = action.vars
    numvars = len(vars)
    numconstants = len(Constants)
    cnt = int(pow(numconstants,numvars))
    for i in range(cnt):
        subs = {}
        j = i
        for v in vars:
            c = j % numconstants
            subs[v] = Constants[c]
            j = (j - c)//numconstants
        #----------------
        head = substitute(action.head,subs)
        pre = substitute(action.pre,subs)
        add = substitute(action.add,subs)
        dele = substitute(action.dele,subs)

        if subset(pre,topWorld):
            worldcnt += 1
            print("%d: Found applicable action" % worldcnt)
            print(action)

            # Create the new world and the new plan
            # And add them to the neighbors

            # YOUR CODE HERE

            print(" New world:")
            printWorld(newWorld," ")
            print(" New plan ")
            printPlan(newPlan," ")
            print(" ")

            frontier = frontier[1:]+ neighbors

            print("No world with goal true was found")

    Constants = ['a','b','c','t']
    Primitives = ['on','clear','empty','holding']

    # I defined the first two things for you, you need to add the rest
    Initial = [['block','a'],['block','b']]
# I defined the first predicate of the goals, you need to add the rest
Goal = [['on','b','a']]  
## YOUR CODE HERE

# You need to define the 4 actions, using the AddAction procedure.
# You will know when you have defined them correctly, as AddAction will
# print them out formatted correctly


Question 12: Defining Actions

The above code includes the class Action, which you must use for adding defining the actions for the domain. This class has two methods. The first is called __init__ and is used to create a define a new action, and __str__ used for printing out the action. Classes are a convenient way to bundle up multiple variables into a single object. This is how we keep track of the head, preconditions, addlist, and delete list of an action.

Hand in your code for adding the four actions. Also explain how the class keeps track of the actions that have been defined, so that we can iterate over them.

Question 13: Initial World and Goal

Hand in your code for creating the initial world and goal.

Question 14: Prove

Hand in a complete copy of your prove function.

Question 15: Iterating over the variable instantiations

In the code that you were given for the previous question, there is some code to iterate over all of the instantiations of the variables in the preconditions (so that the rest of your planner does not need to work with variables). Explain the intuition behind how the code iterates through all of the variable instantiations. To understand, just run the code between the dashes, and print the value of the variable subs.

Question 16

Your planner probably searched through a lot of partial plans in coming up with its solution. A lot of these partial plans were not very useful. Explain why not.

Question 17

Explain why a depth-first search will not work?

**Prolog: Theorem Prover**

In homework 4, you built a top-down depth-first planner for datalog. In the next few questions, you will build a top-down theorem prover in Prolog. For this assignment, the only special prolog operator you need to use is findall. You cannot use any Prolog command that we did not talk about in class. In fact, you should not even need to use the prolog ‘not’ command. For representing datalog variables, we will use Prolog variables.
This way, we will be able to take advantage of Prolog’s built in unification, and substitution capabilities, and so your code should be very short.

We will start by making a depth-first theorem prover, which will take advantage of Prolog’s natural ability to do depth-first search, and then a breadth-first version in which you will explicitly keep track of the frontier.

You should review the answer for homework 4, as well as the class slides in which we discussed depth-first and breath-first search in Prolog.

**Question 18: Writing a KB in Prolog**

In this question, you will write the knowledge base that your theorem prover will use. We want to write it in a format that is amendable to Prolog.

Use the Prolog predicate ‘kb’ to express any facts and rules you want in your knowledge base. ‘kb’ should take one argument, which will be a prolog list. Facts will just have one item in the list. A rule that has a head and $n$ elements in its body will be a list of length $n + 1$. Since prolog allows atoms to be written directly, we will directly write the atoms in the head and the body.

Consider the following rule:

$$\text{does}(X, Y, Z) \leftarrow \text{likes}(X, Y) \land \text{gets_pleasure}(Y, Z)$$

We will express this in Prolog with:

$$\text{kb}([\text{does}(X, Y, Z), \text{likes}(X, Y), \text{gets_pleasure}(Y, Z)])$$

Now, give the 3 prolog lines of code for expressing the following kb.

$$\text{has_tree}(T, T)$$

$$\text{has_tree}(T, n(N, LT, RT)) \leftarrow \text{has_tree}(T, LT)$$

$$\text{has_tree}(T, n(N, LT, RT)) \leftarrow \text{has_tree}(T, RT)$$

**Question 19: Query**

When doing the top-down proof in Homework 4, you were given a query, and you turned it into an answer clause. To simplify things, we will call the depth first prover on the answer clause. We will represent the answer clause as a prolog list, with the first element the yes predicate followed by the rest of the atoms that need to be proved.

So, if the query was:

$$\text{? does}(\text{mary}, X, Y)$$

We would instead call it will \text{depthFirstProve}([\text{yes}(X, Y), \text{does}(\text{mary}, X, Y)])

Rewrite the following query in the same fashion.

$$\text{? has_tree}(n(X, l(l4), Y), n(n1, n(n2, l(l1), l(l2)), n(n3, l(l3), n(n4, l(l14), l(l15)))))$$

**Question 20: Neighbors**

Write a predicate \text{neighbor}(\text{AnswerClause}, \text{NewAnswerClause}) that when given an answer clause, will find an answer clause that can be derived through the resolution rule, using a fact/rule from the knowledge base whose head matches the 2nd atom in answer clause (as we need to skip over the first atom, which is the yes predicate).
If the predicate is repeated called, it should give you all possible next answer clauses. But a single call to this predicate should just return a single neighboring answer clause.

For example

\[
\text{neighbor([yes(X,Y),does(mary,X,Y)],NewAnswerClause)}
\]

should return

\[
\text{NewAnswerClause=[yes(X,Y),likes(mary,X),gets_pleasure(X,Y)]].}
\]

Hand in the predicate.

**Question 21: Depth First Prover**

Make a predicate called \texttt{depthFirstProve(AnswerClause)}. Its base case is where the answer clause just has one atom in it, in which case it prints the answer clause. You can just use the prolog ‘print’ command, followed by ‘nl’.

Its recursive case is where it uses neighbor to find an answer clause that follows from it, and then executes itself on the new answer clause. You do not need to keep a frontier, as you are using Prolog’s.

Make sure it is able to find the right answer for the ‘has tree’ example.

Hand in your definition of \texttt{depthFirstProve}

**Question 22: Breadth First Prover**

For the breadth-first prover, it will need to keep track of the frontier. To keep things simple, when you initially call it, you will call it with the initial answer clause, but as the only element in a list, which is the frontier.

The base case is where the top element in the frontier just has a single element in it. In the recursive case, it will take the top element off of the frontier, and it will use the findall command to find all neighbors. You will then add the neighbors to the end of the frontier using \texttt{append}. And then recursively call your prover on the new frontier.

Make sure your code works on the ‘has tree’ example.

Hand in a copy of your code.