WrittenAssignment4 Solution Set

Predicate Logic

1. Assume that you have logical predicates IsParentOf(X,□Y) , which is true of X and Y whenever X is Y's parent, and IsFemale(X) , which is true iff Xis female. Give predicate calculus statements that define IsMotherOf ,IsFatherOf ,IsChildOf ,IsSonOf ,IsDaughterOf ,IsMale in terms of IsParentOf and IsFemale (as well as any other predicates you define).

□ x, y . [IsMotherOf(x,y) <=> (IsParentOf(x,y) □ IsFemale(x))]
□ x . [IsMale(x) <=> □IsFemale(x)]
□ x, y . [IsFatherOf(x,y) <=> (IsParentOf(x,y) □ IsMale(x))]
□ x, y . [IsChildOf(x,y) <=> IsParentOf(y,x)]
□ x, y . [IsDaughterOf(x,y) <=> (IsChildOf(x,y) □ IsFemale(x))]
□ x, y . [IsSonOf(x,y) <=> (IsChildOf(x,y) □ IsMale(x))]

Note that the biconditional (here written <=>) is equivalent to two conditionals: (A => B) □ (B => A) is the same as A <=> B, as is (A □ B)   (□A □ □B). Because these are definitions, they are appropriately biconditional – either side is the same as the other – but you may not need the full biconditional for the proof in the final problem of this section.

2. Using the additional predicate AreMarried(X,Y) , write rules that enforce the symmetry of marriage and the constraints that the spouse of your parent is also your parent and the child of your spouse is also your child.

□ x, y . [AreMarried(x,y) <=> AreMarried(y,x)]
□ x, y, z . [(AreMarried(x,y) □ IsParentOf(x,z)) => IsParentOf(y,z)]

Note that this last is *not* a biconditional; if x and y are married and x is z's parent, then y is z's (possibly step)parent, but it's possible that y is z's parent without being married to any x.

3. Add rules for grandparent/grandchild derived from the above rules. Provide at least the grandfather gendered version; you may include or omit the others as you wish.

□ x, z . [(□ y . [IsParentOf (x,y) □ IsParentOf(y,z)]) => IsGrandparentOf(x,z)]
□ x, y . [(IsGrandparentOf(x,y) □ IsMale(x)) <=> IsGrandfatherOf(x,y)]

An alternative to the last definition would be

□ x, z . [(□ y . [IsFatherOf(x,y) □ IsParentOf(y,z)]) <=> IsGrandfatherOf(x,z)]

4. At this point, for inspiration, you may wish to visit http://www.transload.net/~terrisfunnypages/songs/grandp.html (preferably with audio turned on) or http://www.wwco.com/gean/grandpa/ (you'll have to explicitly visit the .wav file, but the animation is pretty good). You are in effect going to solve problem 12 from page 78 of Luger (the Logic handout). Here's the setup:

Assume the following:

a. AreMarried (I, W)
b. IsMotherOf (W, D)
c. IsFatherOf (F, I)
d. AreMarried (F, D)
e. IsSonOf (S_1, W)
f. IsSonOf (S_2, D)
g. IsMale (I)

Prove that IsGrandfatherOf(□I,□I□).

1.	IsMotherOf(W, D)	Assumption b.
2.	□ x, y . [IsMotherOf(x,y) <=> (IsParentOf(x,y) □ IsFemale(x))]	Definition IsMotherOf
3.	IsMotherOf(W, D) <=> (IsParentOf(W, D) □ IsFemale(W))	Universal instantiation (AKA universal elimination) of #2 (substituting W for x and D for y, both universally quantified variables)
4.	[IsMotherOf(W, D) => (IsParentOf(W, D) □ IsFemale(W))] □ [(IsParentOf(W, D) □ IsFemale(W)) => IsMotherOf(W, D)]	Definition <=> applied to #3
5.	IsMotherOf(W, D) => (IsParentOf(W, D) □ IsFemale(W))	And elimination (left sided) from #4 (from A □ B infer A)
6.	IsParentOf(W, D) □ IsFemale(W)	Modus ponens (AKA implication elimination) from #5 and #1 (from A => B and A infer B)
7.	IsParentOf(W, D)	And elimination (left sided) from #6
8.	AreMarried(I, W)	Assumption a
9.	AreMarried(I, W) □ IsParentOf(W, D)	And introduction from #8 and #7 (from A and B infer A □ B)

10.	□ x, y, z . [(AreMarried(x,y) □ IsParentOf(x,z)) => IsParentOf(y,z)]	(Step)parent rule from part 2.
11.	(AreMarried(I, W) □ IsParentOf(W, D)) => IsParentOf(I, D)	Universal instantiation of #10 (substituting I for x, W for y, and D for Z)
12.	IsParentOf(I, D)	Modus ponens from #11 and #9
13.	IsMale(I)	Assumption g
14.	IsParentOf(I, D) □ IsMale(I)	And introduction from #12 and #13
15.	□ x, y . [IsFatherOf(x,y) <=> (IsParentOf(x,y) □ IsMale(x))]	Definition IsFatherOf
16.	IsFatherOf(I, D) <=> (IsParentOf(I, D) □ IsMale(I))	Universal instantiation of #15 (substituting I for x and D for y)
17.	[IsFatherOf(I, D) => (IsParentOf(I, D) □ IsMale(I))] □ [(IsParentOf(I, D) □ IsMale(I)) => IsFatherOf(I, D)]	Definition biconditional from #16
18.	(IsParentOf(I, D) □ IsMale(I)) => IsFatherOf(I, D)	And elimination (right sided) from #17 (from A □ B infer B)
19.	IsFatherOf(I, D)	Modus ponens from #19 and #14
20.	□ x, y . [IsFatherOf(x,y) <=> (IsParentOf(x,y) □ IsMale(x))]	Definition IsFatherOf (also #15)
21.	IsFatherOf(F, I) <=> (IsParentOf(F, I) □ IsMale(F))	Universal instantiation of #20 (substituting F for x and I for y)
22.	[IsFatherOf(F, I) => (IsParentOf(F, I) □ IsMale(F))] □ [(IsParentOf(F, I) □ IsMale(F)) => IsFatherOf(F, I)]	Definition biconditional from #21
23.	IsFatherOf(F, I) => (IsParentOf(F, I) □ IsMale(F))	And elimination (left sided) from #22
24.	IsFatherOf(F, I)	Assumption c
25.	IsParentOf(F, I) □ IsMale(F)	Modus ponens from #23 and #24
26.	IsParentOf(F, I)	And elimination (left sided) from #25
27.	AreMarried(F, D)	Assumption d
28.	AreMarried(F, D) □ IsParentOf(F, I)	And introduction from #27 and #26
29.	□ x, y, z . [(AreMarried(x,y) □ IsParentOf(x,z)) => IsParentOf(y,z)]	(Step)parent rule from part 2 (also #10)

30.	(AreMarried(F, D) IsParentOf(F, I)) => IsParentOf(D, I)	Universal instantiation of #29 (substituting F for x, D for y, and I for z)
31.	IsParentOf(D, I)	Modus ponens from #30 and #28
32.	IsFatherOf(I, D) IsParentOf(D, I)	And introduction from #19 and #31
33.	y . [IsFatherOf(I, y) IsParentOf(y, I)]	Existential generalization (AKA existential introduction) of #32 (substitution of existentially quantified y for D)
34.	x, y . [(y . [IsFatherOf(x,y) IsParentOf(y,z)]) <=> IsGrandfatherOf(x,z)]	2d definition of IsGrandFatherOf
35.	[(y . [IsFatherOf(I, y) IsParentOf(y, I)]) <=> IsGrandfatherOf(I, I)]	Universal instantiation from #34 (substitution of I for x and of I for y, both universally quantified variables)
36.	[(y . [IsFatherOf(I, y) IsParentOf(y, I)]) => IsGrandfatherOf(I, I)] [IsGrandfatherOf(I, I) => (y . [IsFatherOf(I, y) IsParentOf(y, I)])]	Definition biconditional for #35
37.	(y . [IsFatherOf(I, y) IsParentOf(y, I)]) => IsGrandfatherOf(I, I)	And elimination (left sided) from #36
38.	IsGrandfatherOf(I, I).	Modus ponens from #37 and #33

Prolog

1. The adventure game explanation tells you that Prolog will respond to the following queries in this way:

```
?- location(fox, X).
X = woods
?- connect(yard, X).
X = house
X = woods
```

Verify this, then explain why Prolog gives one answer to the first query and two to the second. (To see this in action, you may have to hit aafter gprolog gives you the first answer to connect(yard,X). This is because gprolog doesn't automatically assume that you want all answers....)

2. You should see that the program's behavior is somewhat different from the transcript at the top of the page. In particular, you can't get anywhere. This is because one precondition for changing your location isn't met: your location needs to be connected to where you're going. Remedy this problem by editing your code and reconsulting it in. Verify that you can move from the house to the duck_pen and back.

3. Now you can get to the duck_pen , but you still can't win. In order to win, you need to make it possible to satisfy you_have(egg). Add rules to make it possible to obtain the egg, but only if you are in the duck_pen . Verify the behavior of these additions and demonstrate that it is now possible to win the game.

4. There are several other additions required to get the code to behave as indicated in the transcript at the beginning of the game description. Extend the game with at least one of these features, *or* add a different feature (and at least one new rule) of your own choosing.

- Add the gate , making it necessary to open the gate in order to get into the duck_pen .
- Add code to allow the ducks to get out of the duck_pen (when the gate is open, if you have created a gate).
- Be creative....I'm sure that you can find something cool to add!

5. Louden 12.26 (with gprolog spelling): Explain the difference in Prolog between the following two definitions of the sibling relationship:

```
sibling1(X,Y) :- X\=Y, parent(Z,X), parent(Z,Y).
sibling2(X,Y) :- parent(Z,X), parent(Z,Y), X\=Y.
```

where X\=Y means Xdoesn't unify with Y.

```
:-dynamic(location/2).
:-dynamic(connected/2).
:-dynamic(is_open/1).

assert(X) :- X, !.               % extra assertion method to prevent redundant assertions
assert(X) :- assertz(X).         % (This is not necessary, but nice.)

location(egg, duck_pen).
location(ducks, duck_pen).
location(fox, woods).
location(you, house).

connected(duck_pen, yard) :- is_open(gate).    % note changes in this predicate:
connected(yard, house).                        % - new name (connect is symmetric)
connected(yard, woods).                        % - intro'd gate in first clause

connect(X,Y) :- connected(X,Y), !.     % Cuts are not strictly necessary here
connect(X,Y) :- connected(Y,X), !.     % but minimize extra backtracking.
                                       % (Only one case can be true.)

gate_accessible( yard ).
gate_accessible( duck_pen ).

open(gate) :- location( you, X ), gate_accessible( X ),
  assert( is_open( gate ) ),
  write( 'The gate is open.' ), nl, !.       % Cut here is not strictly necessary
open(gate) :-                                % but minimizes extraneous can't reach
  write( 'You can\'t reach the gate from here.' ), nl.     % messages

shut(gate) :- location( you, X ), gate_accessible( X ),
  retract( is_open( gate ) ),
  write( 'The gate is shut.' ), nl.
```

```
take( X ) :-
  location( you, L ),
  location( X, L ), !,                % Cut here prevents extraneous can't reach
  retract( location( X, L ) ),        % messages.  Probably shouldn't have two
  assert( location( X, you ) ),       % cuts, though!!
  write( 'You have the ' ), write( X ), nl, !.
take( X ) :-
  write( 'You can\'t reach the '),
  write( X ),
  write( ' from here. '),
  nl.

goto(X) :-
  location(you, L),
  connect(L, X),
  retract( location(you, L) ),
  assert( location(you, X) ),
  write('You are in the '), write(X), nl,!.   % Cut to prevent extraneous backtracking
goto(_) :-
  write('You can\'t get there from here. '), nl.

fox :-
  location(ducks, yard),
  location(you, house),
  write(' The fox has taken a duck. '), nl.
fox.

ducks :-
  location(ducks, X),
  connect(X,Y),
  \+location(ducks,Y),
  random(Number), !,         % Cut here is because ducks only get one shot at
  ducks_do( Number, Y ).     % moving per turn; not extraneous at all!!
ducks.

ducks_do( Number, Where ) :-
  Number > 0.8,
  assert( location(ducks, Where) ),
  write(' The ducks are in the '), write(Where), nl, !.   % If ducks move,
ducks_do( _, _ ) :-                                       % they don't also
  write(' The ducks are staying put for now.'), nl.       % stay put!
```

```
go :- done.
go :-
  write('>> '),
  read(X),
  call(X),
  fox,
  ducks,
  go, !.    % minimize extraneous backtracking; if you get this far, don't back up!

done :-
  location(you, house),
  you_have(egg),
  write(' Thanks for getting the egg. '), nl.

you_have( X ) :- location( X, you ).
```