CS 560: Homework 4

Eric Stevens

October 27, 2018

Question 1:

My initial thought was to add a `n(0)` predicate but then we would have the problem of being able to have expressions like `s(0) == s(n(s(0)))`, resulting in the same number being represented in two different ways. Therefore, it seems that they best way to extend the scheme is to add another constant. We will add the constant `-0`, which will represent a negative zero. The `s(X)` in increment the numerical value of `X` in the direction of the sign of `X`. Examples of the way this would work follows:

```
s('0') = 1
s(s('0')) = 2
s(s(s('0'))) = 3

s('-0') = -1
s(s('-0')) = -2
s(s(s('-0'))) = -3
```

If we ensure that we dont map `-0` to `0` and instead map `s(-0)` to `-1` we will have a number system where there will be exactly one representation for each numerical value.

Question 2:

The language of this question confused me. There seems to be a mixing of Prolog and Datalog in the instructions: "check your definition in Prolog with ‘succ(s(s(s(0))),X)’". It is my understanding that functions are meaningless in Prolog and are treated as constants. Therefore there is no way to parse the `s()` predicates and I believe that they would have to be hard coded. As a result I am turnning in both a Datalog definition and a working simple Prolog line to cover my bases.

Datalog

```
% if X is greater than 0
succ(X,s(X)) :- lt(0,X)

% if X is less than 0
succ(s(X),X) :- lt(X,0)
```

Prolog

Here I am also adding a `predecessor` function to aid in the next question.

```
% simple addition of 1
succ(X,Y) :- Y is X+1.

% predecessor function
pred(X,Y) :- Y is X-1.
```

Question 3:

```
% base: X+0=X
plus(X,0,X).
% decrement Y and put onto Z
plus(X,Y,Z) :-
        succ(X,NewX),
        pred(Y,NewY), % see previous code sample
        plus(NewX,NewY,Z).

% Alternative plus function
alt_plus(X,Y,Z) :- Z is X+Y.
```

Question 4:

```
% Base Case: if tower height is 1 then a list of a single block shou
ld be returned.
multicolortower4(1,[X]) :- block(X).

% Recursive: Builds on the way out.
multicolortower4(Height,[Top|Rest]) :-
        Height > 1,
        block(Top),
        NewHeight is Height-1,
        multicolortower4(NewHeight,Rest).
```

The following is a query and its ouput using the above predicates:

```
?- multicolortower4(9,List).
List = [red1, red1, red1, red1, red1, red1, red1, red1, red1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red2] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red3] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red4] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red5] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red6] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red7] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, gre1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, gre2] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, gre3] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, blu1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, blu2] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, yel1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, bla1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red2, red1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red2, red2] ;
...
...
...
```

Question 5:

```
% Base Case: any block will be different in empty list
differentFromList(Block,[]) :- block(Block).
% recurse through list and ensure Block is not equal to Top
differentFromList(Block,[Top|Rest]) :-
        block(Block),
        not(Block = Top),
        differentFromList(Block,Rest).

% mct5: This function adds Height blocks to list and
% ensures that the blocks that are added are not
% already in the list. This builds the list on the
% way in to minimize the search space.

% Base Case: 0 hight will return Out with List
mct5(0,List,Out) :-
        Out = List.
% Build on way in in at the recursive call
mct5(Height,List,Out) :-
        Height > 0,
        Height < 14, % only 13 blocks
        block(Top),
        differentFromList(Top,List), % distinct
        NewHeight is Height-1,
```

```
        mct5(NewHeight,[Top|List],Out). % build


% multicolortower5(Height,List): This function formats the operation
% in the way described in the assignment. It allows user to ignore
% extra input variable and ensures the desired result, as mct5 can
% have results that are not what is required if used directly.
multicolortower5(Height,List) :-
        mct5(Height,_,List).
```

The following is a query and its ouput using the above predicates:

```
?- multicolortower5(9,List).
List = [gre2, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre3, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu1, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu2, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [yel1, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [bla1, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre1, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre3, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu1, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu2, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [yel1, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [bla1, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre1, gre3, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre2, gre3, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu1, gre3, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu2, gre3, red7, red6, red5, red4, red3, red2, red1] ;
...
...
...
```

Question 6:

```
% differentcolor(): Simply checks to ensure that two blocks
% are not the same color.
differentcolor(X,Y) :-
        block(X),
        block(Y),
        color(X,CX),
        color(Y,CY),
        not(CX=CY).
```

```
% performs way in list building while checking to ensure that
% any two touching blocks do not have the same color.
mct6(1,List,Out) :-
        Out = List.
mct6(Height,[Tl|List],Out) :- % needs top of list to check for color
        Height > 1,
        Height < 14,
        block(Top),
        differentFromList(Top,List),
        differentcolor(Top,Tl), % ensure color difference
        NewHeight is Height-1,
        mct6(NewHeight,[Top|[Tl|List]],Out).

% same purpose as in question 5
multicolortower6(Height,List) :-
        mct6(Height,_,List).
```

The following is a query and its ouput using the above predicates:

```
?- multicolortower6(9,List).
List = [blu2, red4, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [yel1, red4, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [bla1, red4, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [blu2, red5, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [yel1, red5, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [bla1, red5, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [blu2, red6, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [yel1, red6, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [bla1, red6, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [blu2, red7, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [yel1, red7, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [bla1, red7, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [red4, yel1, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [red5, yel1, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [red6, yel1, blu1, red3, gre3, red2, gre2, red1, gre1] ;
...
...
...
```

Question 7:

In [8]:
```
# Question 7

import random
from hw4standard import *

# time seed random variable
from datetime import datetime
random.seed(datetime.now())

def prove(query,kb):

    #**************************************************************#
    answer = query.copy() # Make a copy of query so query is not altered.
    answer.insert(0,findvariables(query,[])) # add variables from query
    answer[0].insert(0,'yes') # add 'yes'
    #**************************************************************#

    print("Initial answer clause is %s \n\n" % prettyclause(answer))

    while len(answer) > 1:
        #give answer clause fresh variables
        answer = freshvariables(answer)
        matches = []
        for r in kb:
            #unify right most atom for scripting ease
            if unify(answer[-1],r[0],{}):
                matches.append(r)


        if matches == []:
            print("No rules match %s" % prettyexpr(answer[-1]))
            return False

        #**************************************************************
        print(prettyclause(answer)) # output formatting

        # Unification
        a = answer[-1] # right-most atom
        b = matches[random.randint(0,len(matches)-1)] # random match
        sub = {} # get subs
        unify(a,b[0],sub)

        # Substitution
        resolution = substitute(b,sub) # sub for match to be used in resolu
        answer = substitute(answer,sub) # sub current answer clause

        # output formatting
        print("Resolve with: ",prettyclause(b))
        print("Substitution: " ,sub, "\n\n")

        # Replace
        answer = answer[:-1] # cut right-most atom
        for a in resolution[1:]: answer.append(a) # add resolution body
        #**************************************************************


    print(prettyclause(answer))
```

```
    return True

def multiprove(query,kb):
    proved = False
    for n in range(100):
        print("-----------------------------------------------------------------
        print("Iteration ", n,"\n")
        if prove(query,kb):
            print("Proved on iteration %d" % n)
            print("-------------------------------------------------------------
            return True
    print("Could not prove")
    print("-----------------------------------------------------------------
    return False
```

In [9]:
```
kb = [[['animal','X'],['dog','X']],
    [['gets_pleasure','X','feeding'],['animal','X']],
    [['likes','X','Y'],['animal','Y'],['owns','X','Y']],
    [['does','X','Y','Z'],['likes','X','Y'],['gets_pleasure','Y','Z']],
    [['dog','fido']],
    [['owns','mary','fido']]]

query=[['does','mary','X','Y']]

multiprove(query,kb)
```

```
----------------------------------------------------------------------------
-----
Iteration  0

Initial answer clause is yes(X,Y) <= does(mary,X,Y)


yes(_546,_547) <= does(mary,_546,_547)
Resolve with:  does(X,Y,Z) <= likes(X,Y) ^ gets_pleasure(Y,Z)
Substitution:  {'X': 'mary', 'Y': '_546', 'Z': '_547'}


yes(_548,_549) <= likes(mary,_548) ^ gets_pleasure(_548,_549)
Resolve with:  gets_pleasure(X,feeding) <= animal(X)
Substitution:  {'X': '_548', '_549': 'feeding'}


yes(_550,feeding) <= likes(mary,_550) ^ animal(_550)
Resolve with:  animal(X) <= dog(X)
Substitution:  {'X': '_550'}


yes(_551,feeding) <= likes(mary,_551) ^ dog(_551)
Resolve with:  dog(fido)
Substitution:  {'_551': 'fido'}


yes(fido,feeding) <= likes(mary,fido)
Resolve with:  likes(X,Y) <= animal(Y) ^ owns(X,Y)
Substitution:  {'X': 'mary', 'Y': 'fido'}


yes(fido,feeding) <= animal(fido) ^ owns(mary,fido)
Resolve with:  owns(mary,fido)
Substitution:  {}


yes(fido,feeding) <= animal(fido)
Resolve with:  animal(X) <= dog(X)
Substitution:  {'X': 'fido'}


yes(fido,feeding) <= dog(fido)
```

```
Resolve with:  dog(fido)
Substitution:  {}


yes(fido,feeding)
Proved on iteration 0
---------------------------------------------------------------------------
-----
```

Out[9]: True

Question 8:

In [58]:
```
kb2 = [
    [['has_tree','T','T']],
    [['has_tree','T',['n','N','LT','RT']],['has_tree','T','LT']],
    [['has_tree','T',['n','N','LT','RT']],['has_tree','T','RT']]
    ]


query2 = [['has_tree',['n','X',['e', 'e4'],'Y'],['n','n1',['n','n2',['e', 'e
        ['n','n4',['e', 'e4'],['e', 'e5']]]]]]

# call
multiprove(query2,kb2)

# selecting submission output of smaller number of iterations.
# somtimes it took 80 iterations to resolve, other times it
# happened on the first iteration.
```

```
------------------------------------------------------------------------
-----
Iteration  0

Initial answer clause is yes(X,Y) <= has_tree(n(X,e(e4),Y),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))


yes(_359096,_359097) <= has_tree(n(_359096,e(e4),_359097),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,RT)
Substitution:  {'T': ['n', '_359096', ['e', 'e4'], '_359097'], 'N': 'n1',
'LT': ['n', 'n2', ['e', 'e1'], ['e', 'e2']], 'RT': ['n', 'n3', ['e', 'e
3'], ['n', 'n4', ['e', 'e4'], ['e', 'e5']]]}


yes(_359098,_359099) <= has_tree(n(_359098,e(e4),_359099),n(n3,e(e3),n(n
4,e(e4),e(e5))))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,LT)
Substitution:  {'T': ['n', '_359098', ['e', 'e4'], '_359099'], 'N': 'n3',
'LT': ['e', 'e3'], 'RT': ['n', 'n4', ['e', 'e4'], ['e', 'e5']]}


No rules match has_tree(n(_359100,e(e4),_359101),e(e3))
------------------------------------------------------------------------
-----
Iteration  1

Initial answer clause is yes(X,Y) <= has_tree(n(X,e(e4),Y),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))


yes(_359102,_359103) <= has_tree(n(_359102,e(e4),_359103),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,LT)
Substitution:  {'T': ['n', '_359102', ['e', 'e4'], '_359103'], 'N': 'n1',
'LT': ['n', 'n2', ['e', 'e1'], ['e', 'e2']], 'RT': ['n', 'n3', ['e', 'e
3'], ['n', 'n4', ['e', 'e4'], ['e', 'e5']]]}
```

```
yes(_359104,_359105) <= has_tree(n(_359104,e(e4),_359105),n(n2,e(e1),e(e
2)))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,RT)
Substitution:  {'T': ['n', '_359104', ['e', 'e4'], '_359105'], 'N': 'n2',
'LT': ['e', 'e1'], 'RT': ['e', 'e2']}


No rules match has_tree(n(_359106,e(e4),_359107),e(e2))
--------------------------------------------------------------------------
-----
Iteration  2

Initial answer clause is yes(X,Y) <= has_tree(n(X,e(e4),Y),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))


yes(_359108,_359109) <= has_tree(n(_359108,e(e4),_359109),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,RT)
Substitution:  {'T': ['n', '_359108', ['e', 'e4'], '_359109'], 'N': 'n1',
'LT': ['n', 'n2', ['e', 'e1'], ['e', 'e2']], 'RT': ['n', 'n3', ['e', 'e
3'], ['n', 'n4', ['e', 'e4'], ['e', 'e5']]]}


yes(_359110,_359111) <= has_tree(n(_359110,e(e4),_359111),n(n3,e(e3),n(n
4,e(e4),e(e5))))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,RT)
Substitution:  {'T': ['n', '_359110', ['e', 'e4'], '_359111'], 'N': 'n3',
'LT': ['e', 'e3'], 'RT': ['n', 'n4', ['e', 'e4'], ['e', 'e5']]}


yes(_359112,_359113) <= has_tree(n(_359112,e(e4),_359113),n(n4,e(e4),e(e
5)))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,LT)
Substitution:  {'T': ['n', '_359112', ['e', 'e4'], '_359113'], 'N': 'n4',
'LT': ['e', 'e4'], 'RT': ['e', 'e5']}


No rules match has_tree(n(_359114,e(e4),_359115),e(e4))
--------------------------------------------------------------------------
-----
Iteration  3

Initial answer clause is yes(X,Y) <= has_tree(n(X,e(e4),Y),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))


yes(_359116,_359117) <= has_tree(n(_359116,e(e4),_359117),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,RT)
Substitution:  {'T': ['n', '_359116', ['e', 'e4'], '_359117'], 'N': 'n1',
'LT': ['n', 'n2', ['e', 'e1'], ['e', 'e2']], 'RT': ['n', 'n3', ['e', 'e
3'], ['n', 'n4', ['e', 'e4'], ['e', 'e5']]]}


yes(_359118,_359119) <= has_tree(n(_359118,e(e4),_359119),n(n3,e(e3),n(n
4,e(e4),e(e5))))
```

```
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,LT)
Substitution:  {'T': ['n', '_359118', ['e', 'e4'], '_359119'], 'N': 'n3',
'LT': ['e', 'e3'], 'RT': ['n', 'n4', ['e', 'e4'], ['e', 'e5']]}


No rules match has_tree(n(_359120,e(e4),_359121),e(e3))
--------------------------------------------------------------------------
-----
Iteration  4

Initial answer clause is yes(X,Y) <= has_tree(n(X,e(e4),Y),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))


yes(_359122,_359123) <= has_tree(n(_359122,e(e4),_359123),n(n1,n(n2,e(e
1),e(e2)),n(n3,e(e3),n(n4,e(e4),e(e5)))))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,RT)
Substitution:  {'T': ['n', '_359122', ['e', 'e4'], '_359123'], 'N': 'n1',
'LT': ['n', 'n2', ['e', 'e1'], ['e', 'e2']], 'RT': ['n', 'n3', ['e', 'e
3'], ['n', 'n4', ['e', 'e4'], ['e', 'e5']]]}


yes(_359124,_359125) <= has_tree(n(_359124,e(e4),_359125),n(n3,e(e3),n(n
4,e(e4),e(e5))))
Resolve with:  has_tree(T,n(N,LT,RT)) <= has_tree(T,RT)
Substitution:  {'T': ['n', '_359124', ['e', 'e4'], '_359125'], 'N': 'n3',
'LT': ['e', 'e3'], 'RT': ['n', 'n4', ['e', 'e4'], ['e', 'e5']]}


yes(_359126,_359127) <= has_tree(n(_359126,e(e4),_359127),n(n4,e(e4),e(e
5)))
Resolve with:  has_tree(T,T)
Substitution:  {'T': ['n', 'n4', ['e', 'e4'], ['e', 'e5']], '_359126': 'n
4', '_359127': ['e', 'e5']}


yes(n4,e(e5))
Proved on iteration 4
--------------------------------------------------------------------------
-----
```

Out[58]: True

Question 9:

This code does not work. Below are two different attempts that I made to solve this problem. The first tries to call a function recursivly and hold the frontier in the recursion. The second tries to maintain the frontier in a list in the the for loop of the program. I have not had success. These functions are both capable of solving the first query as the output demonstrates, but are unable to solve the second query. Something is wrong with the way they move from child back up to parent nodes. They either get stuck in an infinite loop or terminate without completing the proof. I ran out of time to solve the problem.

In [54]:
```
def depth_first(answerclause,kb):

    answer = answerclause.copy()


    # fresh
    answer = freshvariables(answer)


    # get list of matches for right most pred
    matches = []
    for r in kb:
        if unify(answer[-1],r[0],{}):
            matches.append(r)

    if matches == []: return False
    for m in matches:
        print(prettyclause(answer))
        a = answer[-1]
        b = m
        sub = {}
        unify(a,b[0],sub)

        resolution = substitute(b,sub)
        answer = substitute(answer,sub)
        #print("matches: ", matches[0])
        print("Resolve with: ",prettyclause(b))
        print("Substitution: " ,sub, "\n\n")


        answer = answer[:-1]

        if len(resolution) > 0:
            for z in resolution[1:]: answer.append(z)
        else:
            answer.append(resolution)

        if len(answer)==1:

            print(prettyclause(answer))
            return True
        if depth_first(answer,kb): return True

    return False

def dfr(query,kb):
    print("QUERY: ", prettyclause(query),"\n\n")
    # add yes to answer clause
    answer = query.copy()
    answer.insert(0,findvariables(query,[])) # add variables from query
    answer[0].insert(0,'yes') # add 'yes'
    return depth_first(answer,kb)
```

In [55]:
```
dfr(query,kb)
```

```
QUERY:  does(mary,X,Y)


yes(_359084,_359085) <= does(mary,_359084,_359085)
Resolve with:  does(X,Y,Z) <= likes(X,Y) ^ gets_pleasure(Y,Z)
Substitution:  {'X': 'mary', 'Y': '_359084', 'Z': '_359085'}


yes(_359086,_359087) <= likes(mary,_359086) ^ gets_pleasure(_359086,_3590
87)
Resolve with:  gets_pleasure(X,feeding) <= animal(X)
Substitution:  {'X': '_359086', '_359087': 'feeding'}


yes(_359088,feeding) <= likes(mary,_359088) ^ animal(_359088)
Resolve with:  animal(X) <= dog(X)
Substitution:  {'X': '_359088'}


yes(_359089,feeding) <= likes(mary,_359089) ^ dog(_359089)
Resolve with:  dog(fido)
Substitution:  {'_359089': 'fido'}


yes(fido,feeding) <= likes(mary,fido)
Resolve with:  likes(X,Y) <= animal(Y) ^ owns(X,Y)
Substitution:  {'X': 'mary', 'Y': 'fido'}


yes(fido,feeding) <= animal(fido) ^ owns(mary,fido)
Resolve with:  owns(mary,fido)
Substitution:  {}


yes(fido,feeding) <= animal(fido)
Resolve with:  animal(X) <= dog(X)
Substitution:  {'X': 'fido'}


yes(fido,feeding) <= dog(fido)
Resolve with:  dog(fido)
Substitution:  {}


yes(fido,feeding)
```

Out[55]:
```
True
```

In [56]:
```
# Question 7

import random
from hw4standard import *

# time seed random variable
from datetime import datetime
random.seed(datetime.now())

def deep(query,kb,front):

    #****************************************************************#
    answer = query.copy() # Make a copy of query so query is not altered.
    answer.insert(0,findvariables(query,[])) # add variables from query
    answer[0].insert(0,'yes') # add 'yes'
    #****************************************************************#

    print("Initial answer clause is %s \n\n" % prettyclause(answer))

    while len(answer) > 1:
        #give answer clause fresh variables
        answer = freshvariables(answer)
        matches = []
        for r in kb:
            #unify right most atom for scripting ease
            if unify(answer[-1],r[0],{}):
                matches.append(r)
                front.append(r)


        if matches == []:
            if front == []:
                print("No rules match %s" % prettyexpr(answer[0]))
                return False
            else:
                front.pop()
                deep(front[-1],kb,front)
        #****************************************************************
        print(prettyclause(answer)) # output formatting

        # Unification
        a = answer[-1] # right-most atom
        b = front.pop() # random match
        sub = {} # get subs
        unify(a,b[0],sub)

        # Substitution
        resolution = substitute(b,sub) # sub for match to be used in resolu
        answer = substitute(answer,sub) # sub current answer clause

        # output formatting
        print("Resolve with: ",prettyclause(b))
        print("Substitution: " ,sub, "\n\n")

        # Replace
        answer = answer[:-1] # cut right-most atom
        for a in resolution[1:]: answer.append(a) # add resolution body
```

```
        #**************************************************************;

    print(prettyclause(answer))
    return True
```

In [57]:
```
deep(query,kb,[])
```

```
Initial answer clause is yes(X,Y) <= does(mary,X,Y)


yes(_359090,_359091) <= does(mary,_359090,_359091)
Resolve with:  does(X,Y,Z) <= likes(X,Y) ^ gets_pleasure(Y,Z)
Substitution:  {'X': 'mary', 'Y': '_359090', 'Z': '_359091'}


yes(_359092,_359093) <= likes(mary,_359092) ^ gets_pleasure(_359092,_3590
93)
Resolve with:  gets_pleasure(X,feeding) <= animal(X)
Substitution:  {'X': '_359092', '_359093': 'feeding'}


yes(_359094,feeding) <= likes(mary,_359094) ^ animal(_359094)
Resolve with:  animal(X) <= dog(X)
Substitution:  {'X': '_359094'}


yes(_359095,feeding) <= likes(mary,_359095) ^ dog(_359095)
Resolve with:  dog(fido)
Substitution:  {'_359095': 'fido'}


yes(fido,feeding) <= likes(mary,fido)
Resolve with:  likes(X,Y) <= animal(Y) ^ owns(X,Y)
Substitution:  {'X': 'mary', 'Y': 'fido'}


yes(fido,feeding) <= animal(fido) ^ owns(mary,fido)
Resolve with:  owns(mary,fido)
Substitution:  {}


yes(fido,feeding) <= animal(fido)
Resolve with:  animal(X) <= dog(X)
Substitution:  {'X': 'fido'}


yes(fido,feeding) <= dog(fido)
Resolve with:  dog(fido)
Substitution:  {}


yes(fido,feeding)
```

Out[57]: True