Tutorial 9

The Week of November 14

Overview

Dear Students,

1. Please be sure to review the SWI Prolog introduction from last week on your own.
2. This tutorial contains some examples that you will go over with TAs in tutorial as well as some extra examples that will be helpful to you in reviewing the material covered in class.

Sheila

1 Prolog programming example - family tree

```
male(tom).
male(peter).
male(doug).
male(david).
female(susan).
parent(doug, susan).
parent(tom, william).
parent(doug, david).
parent(doug, tom).

1) mother(X,Y) :- parent(X,Y), female(X).

2) father(X,Y) :- parent(X,Y), male(X).

3) sibling(X,Y) :- parent(Z, X), parent(Z,Y).

4) grandfather(GP, GC) :- male(GP), parent(GP, X), parent(X, GC).

5) second_cousin(X,Y):- grandchild(X,Z), grandchild(Y,V),
                        sibling(Z,V).
```

2 Prolog programming example - Trip planning

Here we define a new kind of database to deal with "trips", and develop Prolog predicates to compute certain things about the trips.
We start with simply facts such as the following:

```
plane(to, ny).
plane(ny, london).
plane(london, bombay).
plane(london, oslo).
plane(bombay, katmandu).
boat(oslo, stockholm).
boat(stockholm, bombay).
boat(bombay, maldives).
```

We now develop the following predicates:

```
a) cruise(X,Y) -- there is a possible boat journey from X to Y.

  cruise(X,Y) :- boat(X,Y).
  cruise(X,Y) :- boat(X,Z), cruise(Z,Y).


b) trip(X,Y) -- there is a possible journey (using plane or boat)
                from X to Y.

  leg(X,Y) :- plane(X,Y).
  leg(X,Y) :- boat(X,Y).

  trip(X,Y) :- leg(X,Y).
  trip(X,Y) :- leg(X,Z), trip(Z,Y).
```

Note how we use multiple clauses for or'ing subgoals. Note that an advantage of using "leg" is that it makes it easier to extend the knowledge base to have other modes of transport.

```
c) stopover(X,Y,S) -- there is a trip from X to Y with a stop in S.

First, assume that neither X nor Y can equal S.

  stopover(X,Y,S) :- trip(X,S), trip(S,Y).
```

```
Now, assume S could be X or Y (or even both):

  hop(X,X).
  hop(X,Y) :- trip(X,Y).

  stopover(X,Y,S) :- hop(X,S), hop(S,Y).


d) plane_cruise(X,Y) -- there is a trip from X to Y that has at least
                        one plane leg, and at least one boat leg.

  plane_cruise(X,Y) :- plane(X,Z), boat(Z,Y).
  plane_cruise(X,Y) :- boat(X,Z), plane(Z,Y).

  plane_cruise(X,Y) :- leg(X,Z), plane_cruise(Z,Y).
  plane_cruise(X,Y) :- leg(Z,Y), plane_cruise(X,Z).
```

The interesting thing about this solution is to see that to get a "mixed" trip of planes and boats, you will at some point have a plane followed by a boat or vice versa (the base cases). Once you have that, the condition is met, and you can simply add either plane or boat legs on either side to create the full journey.

Think about why we need to have the second rule of plane_cruise be

```
  plane_cruise(X,Y) :- leg(Z,Y), plane_cruise(X,Z).
```

instead of:

```
   plane_cruise(X,Y) :- plane_cruise(X,Z), leg(Z,Y).
```

The latter, while it may be the intuitive way to write it, gives an infinite recursion!!!

```
e) cost(X,Y,C) -- there is a trip from X to Y that costs less than C.
```

We need to add costs to each of the plane and boat predicates, such as plane(to, ny, 300).

```
   leg(X,Y,C) :- plane(X,Y,C).
   leg(X,Y,C) :- boat(X,Y,C).

   trip(X,Y,C) :- leg(X,Y,C).
   trip(X,Y,C) :- leg(X,Z,C1), trip(Z,Y,C2), C is C1 + C2.
```

Now "cost" is simple, because "trip" is doing the addition for us:

```
   cost(X,Y,C) :- trip(X,Y,C_trip), C_trip < C.
```

Note that we have made use of arithmetic in our solution. For now, you should know the following:

1. `is` acts like mathematical equality, with the restriction that everything on the right-hand side of `is` must be instantiated (no variables!)

2. $+$, $-$, $<$, $>$ are all defined in the usual way

3 Lists in Prolog

```
% count(L, E, N) holds iff the list L contains N copies of E
% Pre:   L and E are instantiated

1 count([],_,0).
2 count([E|Rest],E,N) :- count(Rest,E,N1), N is N1 + 1.
3 count([X|Rest],E,N) :- \+(X=E), count(Rest, E, N).
```

Recall that `is` requires that the right-hand side is fully instantiated. Note the use of `_` - the "don't care".
Let's see a Prolog search tree for `count([1,2,1,2], 1, N).`

```
?- count([1,2,1,2], 1, N).

N = 2
```

See Figure 1 for snapshot. We now press ";".

```
?- count([1,2,1,2], 1, N).

N = 2;

No
?-
```

See Figure 2 for snapshot.

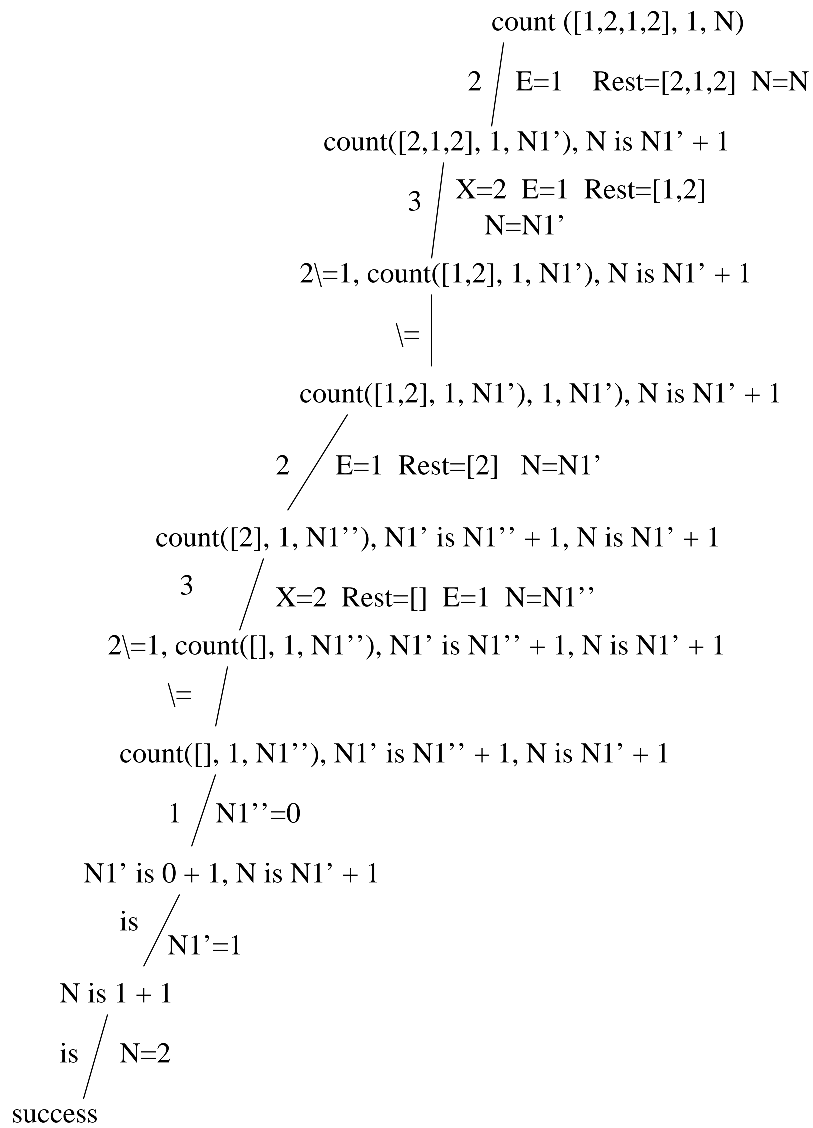

Figure 1: N = 2

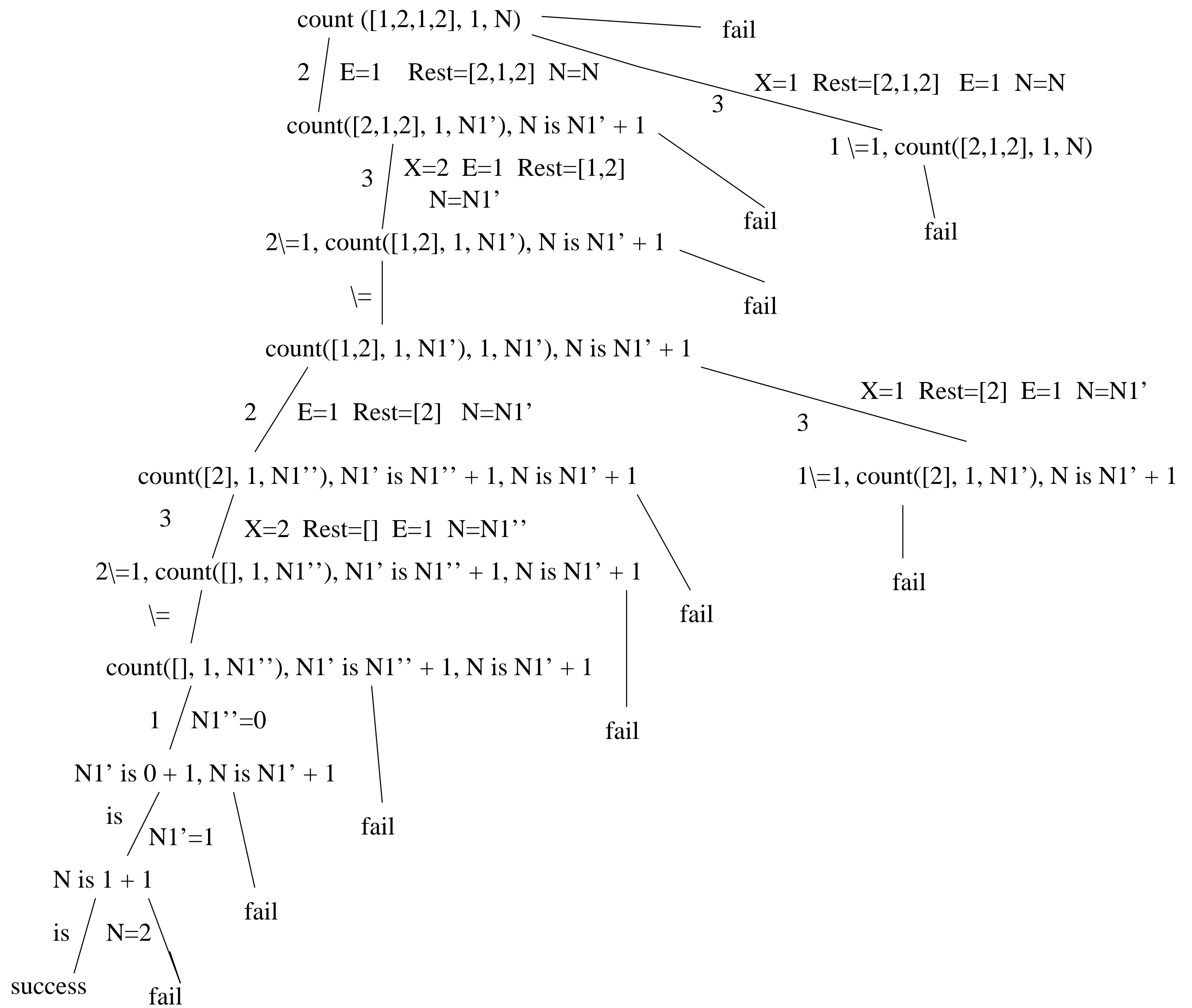

Figure 2: No

```
% delete(E,L1,L2) holds iff list L2 is list L1 with exactly
% one instance of E "deleted"
% Pre: none

delete(E,[E|Rest],Rest).
delete(E,[X|Rest],[X|Rest2]) :- delete(E,Rest,Rest2).
```

Notice: we don’t have X \= E in the recursive case.
That’s because the query

```
 ?- delete(1,[1,2,1],L).
```

should say

```
 L = [2,1];
 L = [1,2];
 no
```

A side-effect of this is that the query

```
 ?- delete(1,[1,1],L).
```

will also say

```
 L = [1];
 L = [1];
 no
```

deleting the second copy of 1, but that is exactly what we wanted.

```
% reverse(L1,L2) holds iff L2 is the list L1 in reverse order
% Pre:   L1 is instantiated.

reverse([],[]).
reverse([A|X],Z) :- reverse(X,Y), append(Y,[A],Z).
```

Notice the use of ’append’ here.
This solution is ‘‘slow’’ - quadratic time.

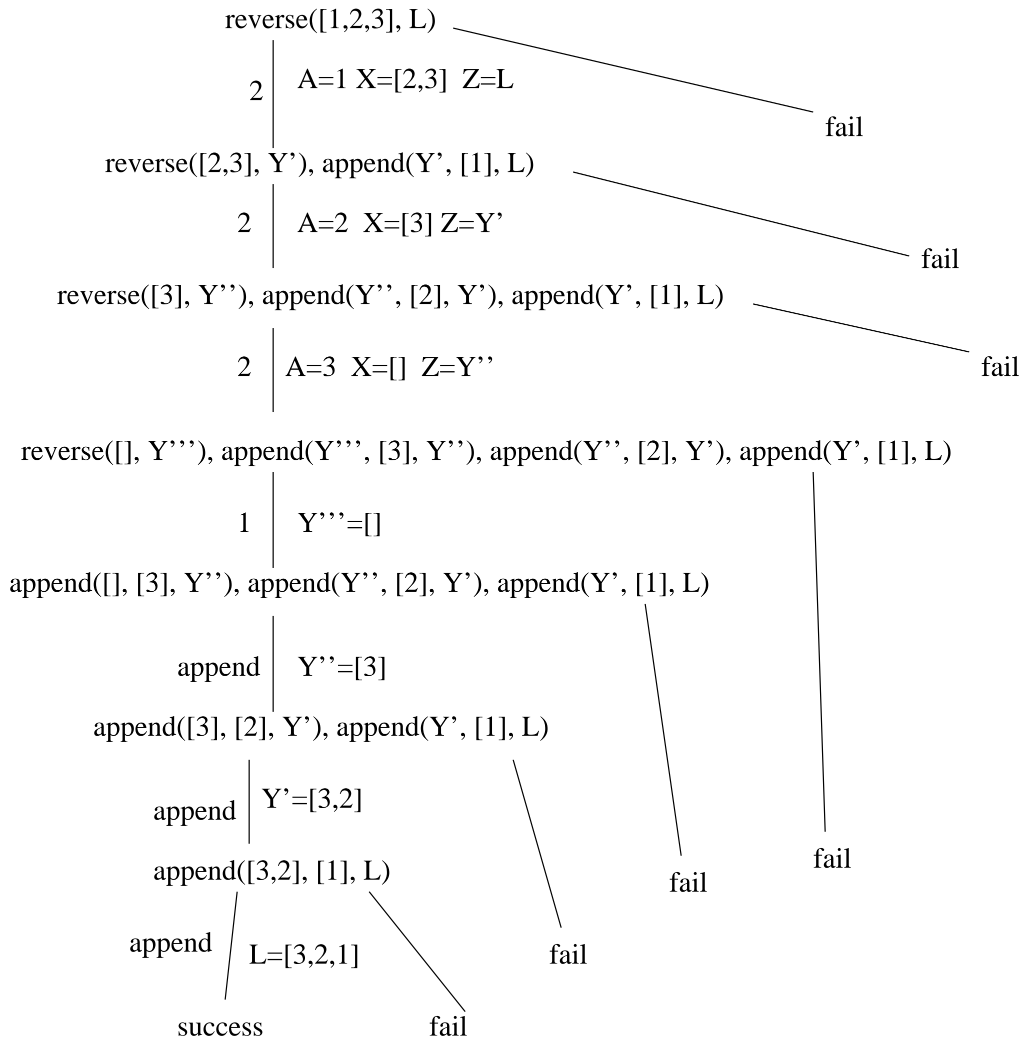

Figure 3: reverse([1,2,3], L)

```
% permutation(L1,L2) holds iss  L2 is some permutation of L1,
% i.e., it has the same elements, but in any order.
% Pre: L1 is instantiated.

permutation([],[]).
permutation(L1,[X|Rest2]) :- delete(X,L1,Rest1),    % defined earlier
                             permutation(Rest1,Rest2).
```

Another Solution:

```
permutation([],[]).
permutation(L1,[X|R2]) :- append(P1,[X|P2],L1),
                          append(P1,P2,R1),
                          permutation(R1,R2).
```

Again, notice how 'append' is used here.

4 Reversing a List

```
% reverse(L1,L2)  L2 is the list L1 in reverse order
% Pre:   L1 is instantiated.

Solution 2:

reverse(X,Z) :- reverse_acc(X,[],Z).

reverse_acc([],Y,Y).
reverse_acc([A|X],Y,Z) :- reverse_acc(X,[A|Y],Z).

Y is called an "accumulator" variable. This solution is "fast"
 -- linear time.

Draw a search tree for reverse([1,2,3],L) to see what's going on.

Now, try reverse(L,[1,2,3]). Begin to draw a search tree, or, better
yet, look at the trace. We get infinite recursion after giving out
the answer.

Thus, we need a precondition:
  the first list is instantiated to a list of known length,
  i.e., the tail is [], and not an uninstantiated variable.

The following implementation works with any input:

reverse(L,R) :- knownlength(L), reverse_acc(L,[],R).
reverse(L,R) :- \+knownlength(L), reverse_acc(R,[],L).

knownlength(L) :- \+var(L), L=[].
knownlength(L) :- \+var(L), L=[_|R], knownlength(R)).

reverse_acc([],Y,Y).
reverse_acc([A|X],Y,Z) :- reverse_acc(X,[A|Y],Z).

Now draw a (part of the) search tree / look at the trace for reverse(L,[1,2,3])
to see how it now works.
```

5 EXTRA EXAMPLES I

Prolog programming example - family tree

```
male(tom).
male(peter).
male(doug).
male(david).
female(susan).
parent(doug, susan).
parent(tom, william).
parent(doug, david).
parent(doug, tom).

1) mother(X,Y) :- parent(X,Y), female(X).

2) father(X,Y) :- parent(X,Y), male(X).

3) sibling(X,Y) :- parent(Z, X), parent(Z,Y).

4) grandfather(GP, GC) :- male(GP), parent(GP, X), parent(X, GC).

5) second_cousin(X,Y):- grandchild(X,Z), grandchild(Y,V),
                        sibling(Z,V).
```

6 EXTRA EXAMPLES II

The Prolog search tree

Need to number the predicates. We also change the database a bit, so that our tree is not too big.

```
1 parent(doug, susan).
2 parent(tom, william).
3 parent(doug, tom).

4 sibling(X,Y) :- parent(Z, X), parent(Z,Y).

?- sibling(X,Y).

X = susan
Y = susan
```

See Figure 4 for a snapshot. Now we type ";".

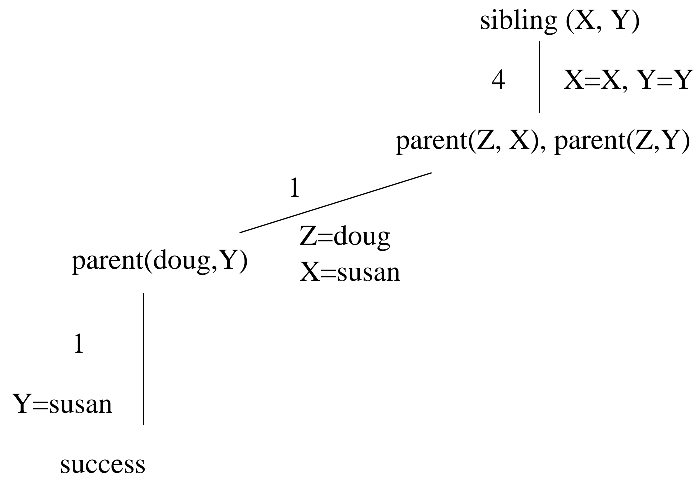

Figure 4: X=susan Y=susan

```
X = susan
Y = tom
```

See Figure 5 for a snapshot. Now we type ";".

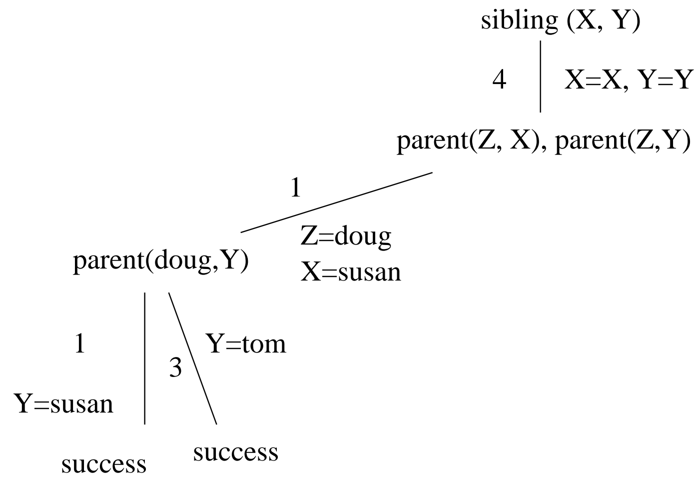

Figure 5: X=susan Y=tom

```
X = william
Y = william
```

See Figure 6 for a snapshot. Now we type ";".

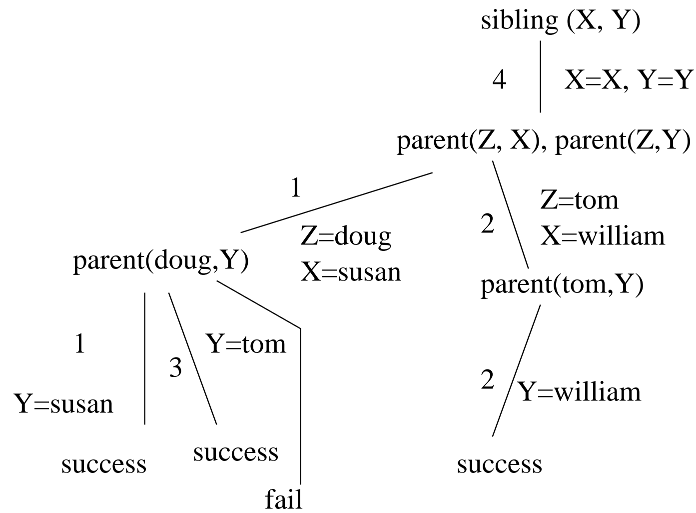

Figure 6: X=william Y=william

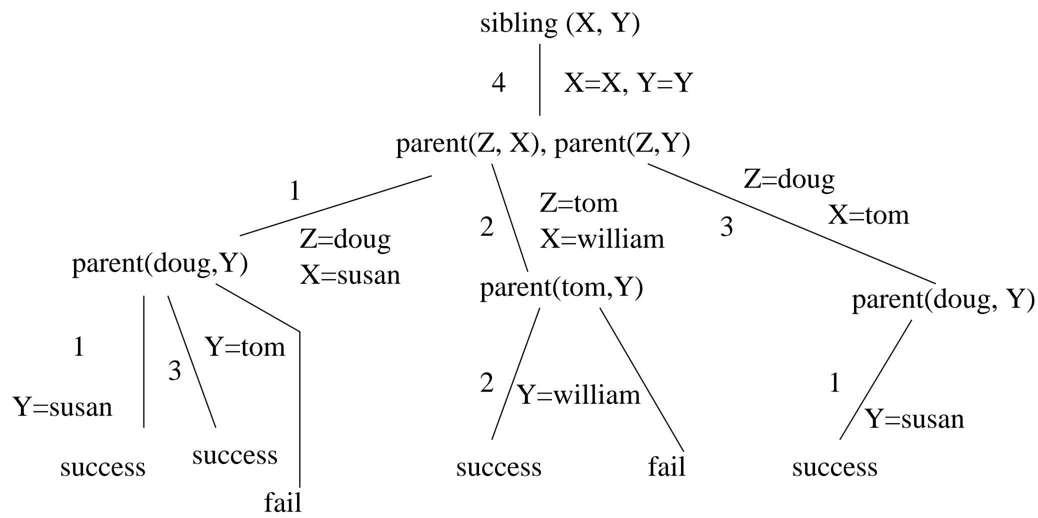

Figure 7: X=tom Y=tom

```
X = tom
Y = susan
```

See Figure 7 for a snapshot. Now we type ";".

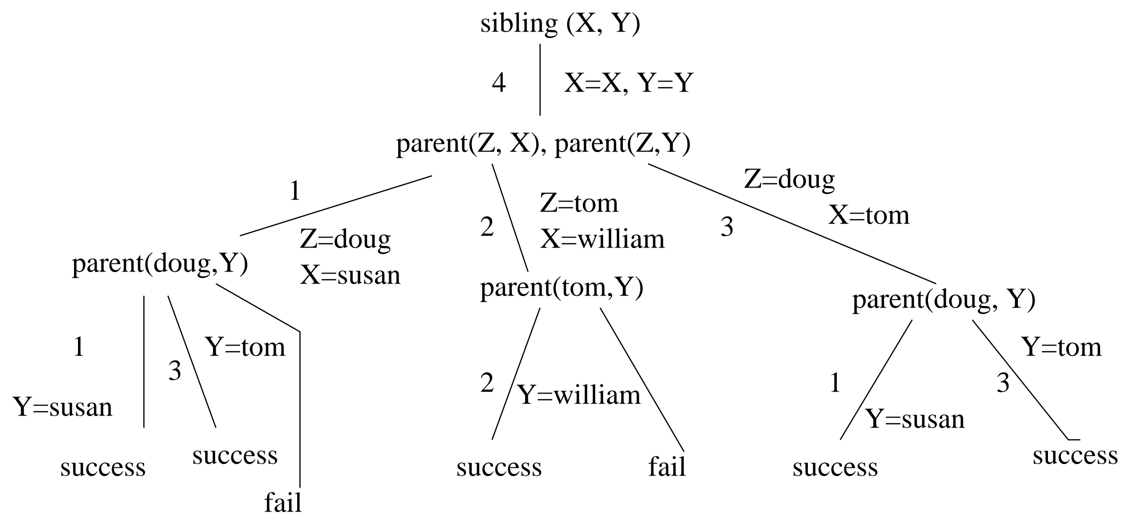

Figure 8: No

```
X = tom
Y = tom
```

See Figure 8 for a snapshot. Now we type ";".

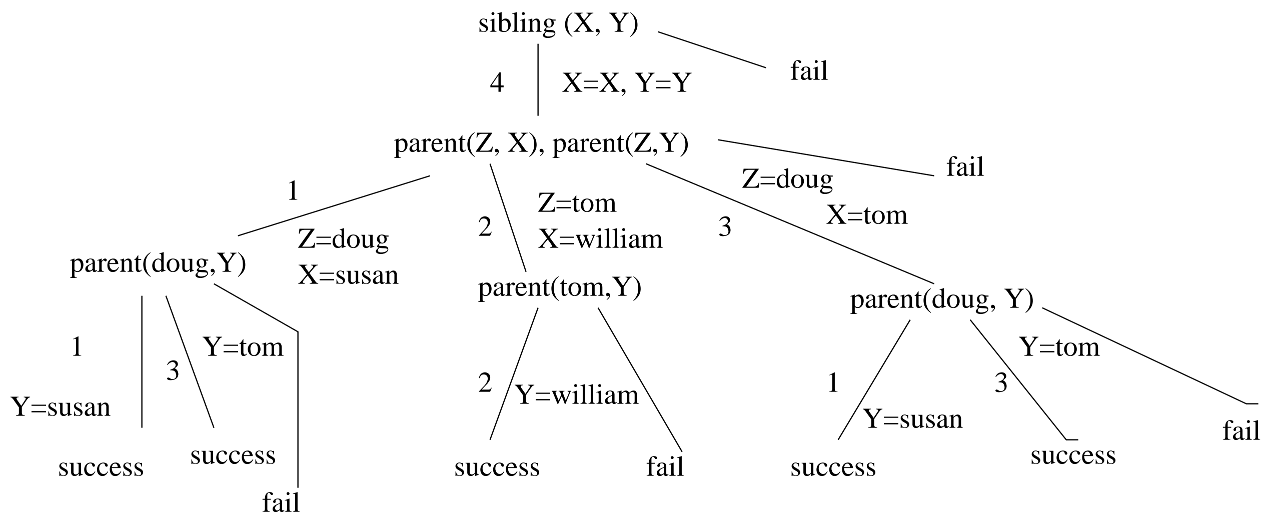

Figure 9: X=susan Y=tom

```
No
?-
```

See Figure 9 for a snapshot. Now we type ";".