Tutorial 9

Week of March 26, 2007

1 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/2. % plane(X,Y) : there is a direct flight from X to Y
boat/2.  % plane(X,Y) : there is a direct boat trip from X to Y
```

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.
```

2 Negation by failure

Example.

```
p(a).
p(b).
q(c).

?- \+p(X), q(X).
No

?- q(X), \+p(X).
X = c ;
No
```

Another example:

```
bachelor(P) :- male(P), not(married(P)).

male(henry).
male(tom).

married(tom).

?- bachelor(henry).
Yes

?- bachelor(tom).
 No

?- bachelor(Who).
Who= henry ;
No

?- not(married(Who)).
No.
```

The first three responses are correct and as expected. The answer to the fourth query might have been unexpected at first. But consider that the

goal ?-not(married(Who)) fails because for the variable binding Who=tom, married(Who) succeeds, and so the negative goal fails. Thus, negative goals ?-not(g) with variables cannot be expected to produce bindings of the variables for which the goal g fails.

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.

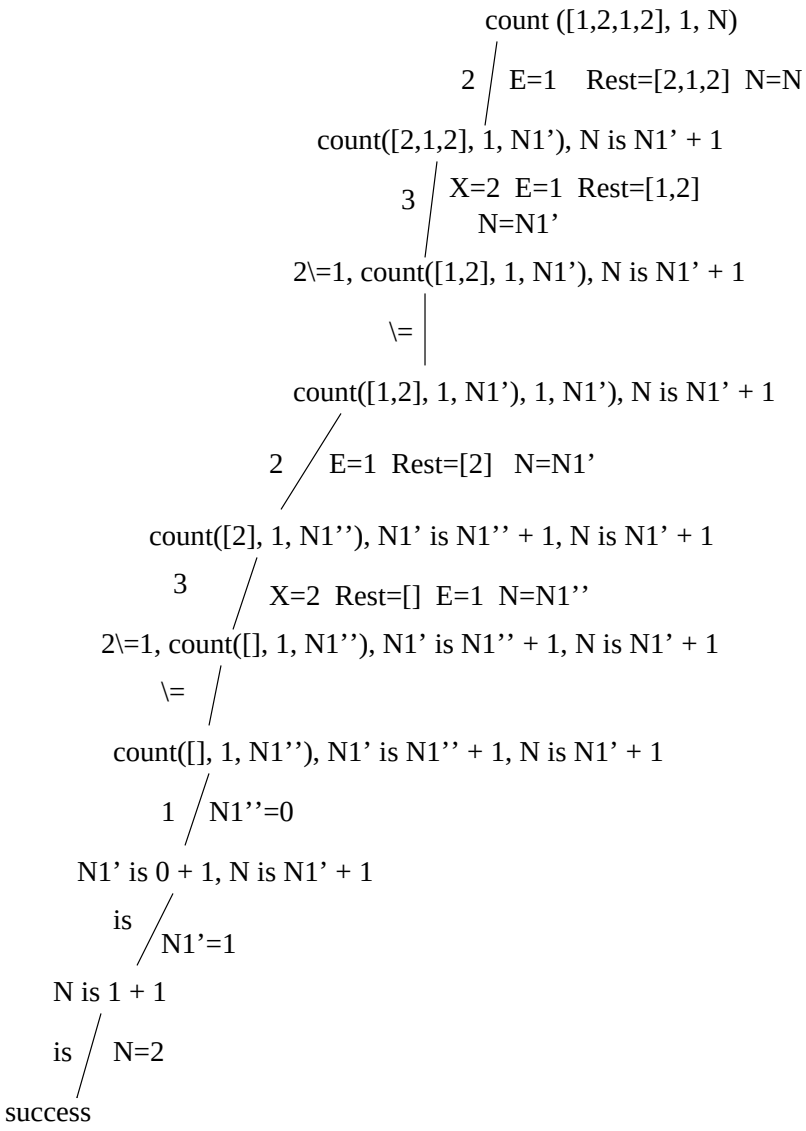
count ([1,2,1,2], 1, N)
2
E=1 Rest=[2,1,2] N=N
count([2,1,2], 1, N1'), N is N1' + 1
3
X=2 E=1 Rest=[1,2]
N=N1'
2\=1, count([1,2], 1, N1'), N is N1' + 1
\=
count([1,2], 1, N1'), 1, N1'), N is N1' + 1
2
E=1 Rest=[2] N=N1'
count([2], 1, N1''), N1' is N1'' + 1, N is N1' + 1
3
X=2 Rest=[] E=1 N=N1''
2\=1, count([], 1, N1''), N1' is N1'' + 1, N is N1' + 1
\=
count([], 1, N1''), N1' is N1'' + 1, N is N1' + 1
1
N1''=0
N1' is 0 + 1, N is N1' + 1
is
N1'=1
N is 1 + 1
is
N=2
success

Figure 1: N = 2

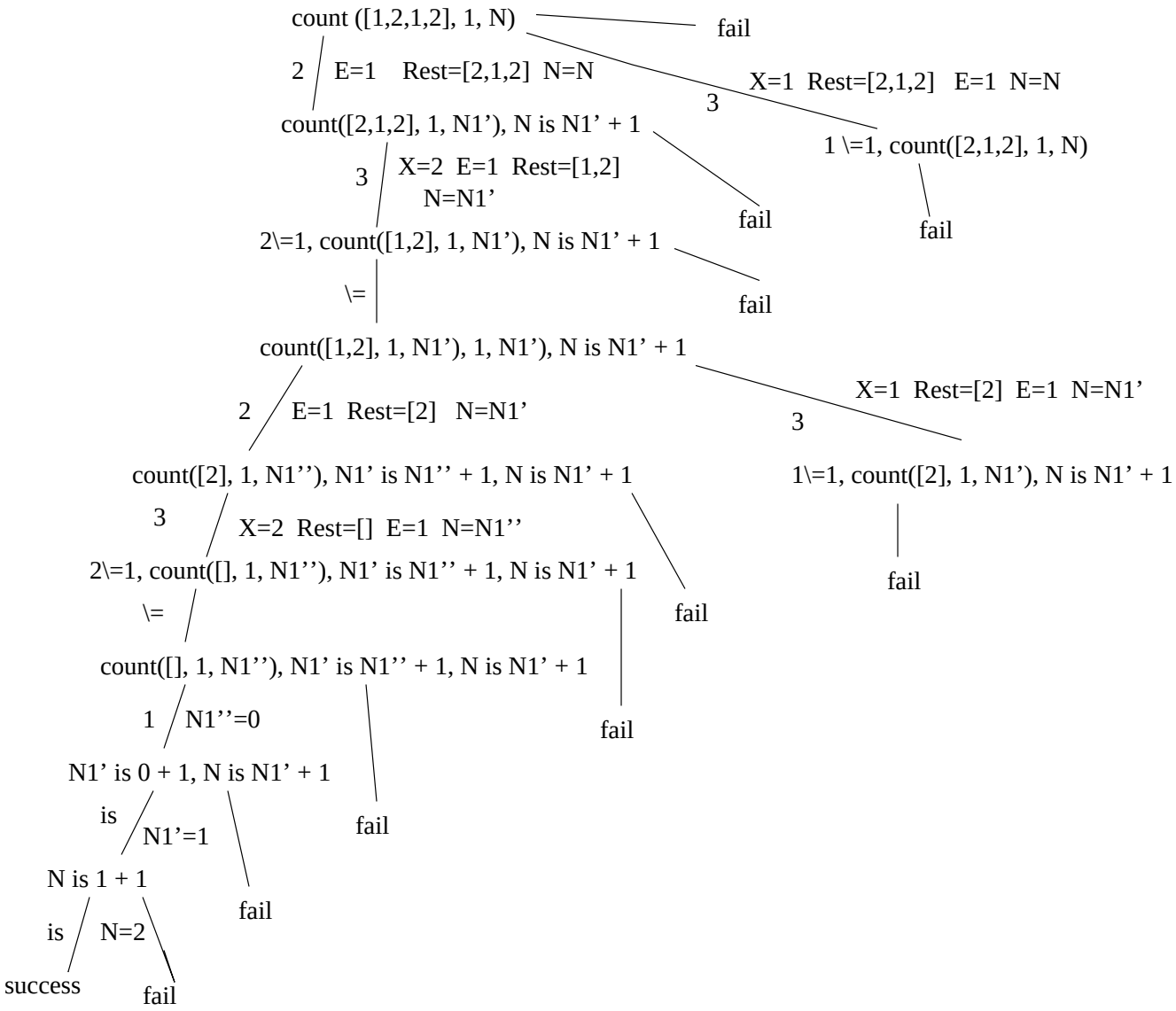
count ([1,2,1,2], 1, N)
fail
2
E=1 Rest=[2,1,2] N=N
3
X=1 Rest=[2,1,2] E=1 N=N
count([2,1,2], 1, N1'), N is N1' + 1
1 \=1, count([2,1,2], 1, N)
3
X=2 E=1 Rest=[1,2]
N=N1'
fail
fail
2\=1, count([1,2], 1, N1'), N is N1' + 1
\=
fail
count([1,2], 1, N1'), 1, N1'), N is N1' + 1
X=1 Rest=[2] E=1 N=N1'
2
E=1 Rest=[2] N=N1'
3
count([2], 1, N1''), N1' is N1'' + 1, N is N1' + 1
1\=1, count([2], 1, N1'), N is N1' + 1
3
X=2 Rest=[] E=1 N=N1''
2\=1, count([], 1, N1''), N1' is N1'' + 1, N is N1' + 1
fail
\=
fail
count([], 1, N1''), N1' is N1'' + 1, N is N1' + 1
1
N1''=0
fail
N1' is 0 + 1, N is N1' + 1
is
N1'=1
fail
N is 1 + 1
is
N=2
success
fail

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.