Tutorial 9

prepared by Anya Tafliovich[1]

July 30, 2003

[1]based on material provided by Diane Horton, Eric Joanis, Alfredo Gabaldon

1 Cut

Consider the following database.

```
male(albert).
male(john).

female(victoria).
female(alice).

parent(john,albert).
parent(victoria,albert).
parent(john,alice).
parent(victoria,alice).
```

Want:

```
sibling(X, Y) : X and Y are siblings.
```

First try:

```
% Post: X is instantiated. NOTE how post-conditions work.
person(X) :- male(X).
person(X) :- female(X).

sibling(X, Y) :- person(X), person(Y), parent(Z, X), parent(Z, Y).
```

Problem:

```
| ?- sibling(X,Y).

X = albert
Y = albert;

X = albert
Y = albert;

X = albert
Y = alice;

X = albert
Y = alice;

X = alice
Y = albert;

X = alice
Y = albert;

X = alice
Y = alice;

X = alice
Y = alice;

no
```

Second try:

```
 sibling(X, Y) :- person(X), person(Y), X\=Y,
                  parent(Z, X), parent(Z, Y).
```

Then:

```
| ?- sibling(X,Y).

X = albert
Y = alice;

X = albert
Y = alice;

X = alice
Y = albert;

X = alice
Y = albert;

no
```

Try using 'cut':

```
 sibling(X,Y) :- person(X), person(Y), \+ X=Y,
                 parent(Z,X), parent(Z,Y), !.
```

Then:

```
| ?- sibling(X,Y).

X = albert
Y = alice;

no
```

What went wrong? What about X=alice, Y=albert ?
Draw a search tree for sibling(X,Y).

```
OK, we can fix this -- a usual trick is defining an extra rule:

 sibling(X,Y) :- person(X), person(Y), \+ X=Y,
                 sibling_check(X,Y).

 % Pre: X and Y are instantiated.
 sibling_check(X,Y) :- parent(Z,X), parent(Z,Y), !.

Then:

| ?- sibling(X,Y).

X = albert
Y = alice;

X = alice
Y = albert;

no


How does this work?
Draw a search tree for sibling(X,Y).
```

2 Cut vs Not

```
replace(oldList, oldElem, newElem, newList) :
  newList is the same as oldList, except all occurrences of
  oldElem are replaced with newElem.

First try:

 replace1([],_,_,[]).

 replace1([Old|TOld],Old,New,[New|TNew]) :-
    replace1(TOld,Old,New,TNew).

 replace1([H|TOld],Old,New,[H|TNew]) :-
    replace1(TOld,Old,New,TNew).

Problem:

replace1 generates both the right answers and wrong answers,
because clause 3 can match on backtracking even when the head
of the oldList is the item to be replaced:

  | ?- replace1([1,2,3,2,1],2,5,X).
  X = [1,5,3,5,1];
  X = [1,5,3,2,1];
  X = [1,2,3,5,1];
  X = [1,2,3,2,1];
  no
```

Second try: use "not":

```
replace2([],_,_,[]).

replace2([Old|TOld],Old,New,[New|TNew]) :-
   replace2(TOld,Old,New,TNew).

replace2([H|TOld],Old,New,[H|TNew]) :-
   H\=Old,  replace2(TOld,Old,New,TNew).
```

Then:

```
| ?- replace2([1,2,3,2,1],2,5,X).
X = [1,5,3,5,1];
no
```

Third try: use "cut":

```
  replace3([],_,_,[]).

  replace3([Old|TOld],Old,New,[New|TNew]) :-
     !, replace3(TOld,Old,New,TNew).

  replace3([H|TOld],Old,New,[H|TNew]) :-
     replace3(TOld,Old,New,TNew).
```

OR:

```
  replace4([],_,_,[]).

  replace4([Old|TOld],Old,New,[New|TNew]) :-
     replace4(TOld,Old,New,TNew), !.

  replace4([H|TOld],Old,New,[H|TNew]) :-
     replace4(TOld,Old,New,TNew).
```

Then:

```
  | ?- replace3([1,2,3,2,1],2,5,X).
  X = [1,5,3,5,1];
  no

  | ?- replace4([1,2,3,2,1],2,5,X).
  X = [1,5,3,5,1];
  no
```

Any call to replace3 can only give one solution because
it either matches clause 1 or clause 2 (not both),
and if it matches clause 2 it cannot backtrack to clause 3.

Hence the above code is equivalent: whether the cut goes before
it or after it doesn't matter -- the important thing is that you
can't succeed on clause 2 and backtrack to clause 3.

BUT Problem:

What if we query with a variable for the first argument?
That is, asking what oldList could a newList correspond to?

```
| ?- replace2(X,1,2,[1]).
no

| ?- replace3(X,1,2,[1]).
X = [1];
no
```

replace2 gives the correct answer -- any 1's in oldList would
have been replaced by 2 in the newList, so [1] cannot be a result.
Why does replace3 not handle this correctly?

For replace2:
The query only matches clause 3, but then fails the first subgoal,
H\=Old. This ensures that a newList element can't be the same as
the element to be replaced.

For replace3:
The query only matches clause 3, but there is no check to ensure that
the element to be replaced can't be in newList.

Here's another example:

```
 | ?- replace2(X,1,2,[2]).
 X = [1];
 X = [2];
 no

 | ?-  replace3(X,1,2,[2]).
 X = [1];
 no
```

replace2 gives the correct answers; replace3 - no!

For replace2:
The query matches both clause 2 and clause 3. When it matches clause 2, the system will first create oldList with the element to be replaced as its first element. When backtracking, it will match clause 3, and create oldList with the first element of the newList as the first element of the oldList. In this case, the element happens to equal the element to be replaced with, but this is not disallowed by the definition.

For replace3:
The query matches clause 2 and creates oldList with the element to be replaced as its first element. Then, because of the cut, it does not match clause 3, so it only finds the first solution.