Prolog
Programming in Logic

Lecture #5

Ian Lewis, Andrew Rice

Q: is gnd() special in Prolog or is it just a frequently used naming convention?

Q: is gnd() special in Prolog or is it just a frequently used naming convention?

A: swipl has ground(X) which is true if X is a ground term. gnd() is just a compound term. Don't use ground(X) in the exam... also atom(X), var(X) ...

Q: All the methods taught so far (like generate and test) don't seem too efficient computationally. In the exam should we think of more complex logic to do so?

Q: All the methods taught so far (like generate and test) don't seem too efficient computationally. In the exam should we think of more complex logic to do so?

A: If the exam question is interested in efficiency it will say so...past questions have not asked this. You make generate and test more efficient by generating better!

Q: With drawing out the execution traces - it's pretty difficult to understand them if you look back at them. How can we convey it in an exam?

Q: With drawing out the execution traces - it's pretty difficult to understand them if you look back at them. How can we convey it in an exam?

A: I can't remember an exam question where I asked for a search tree to be drawn out. Instead a question might ask for what happens: e.g. what results do you get. We'll see more of the 'search tree' in this lecture.

Today's discussion

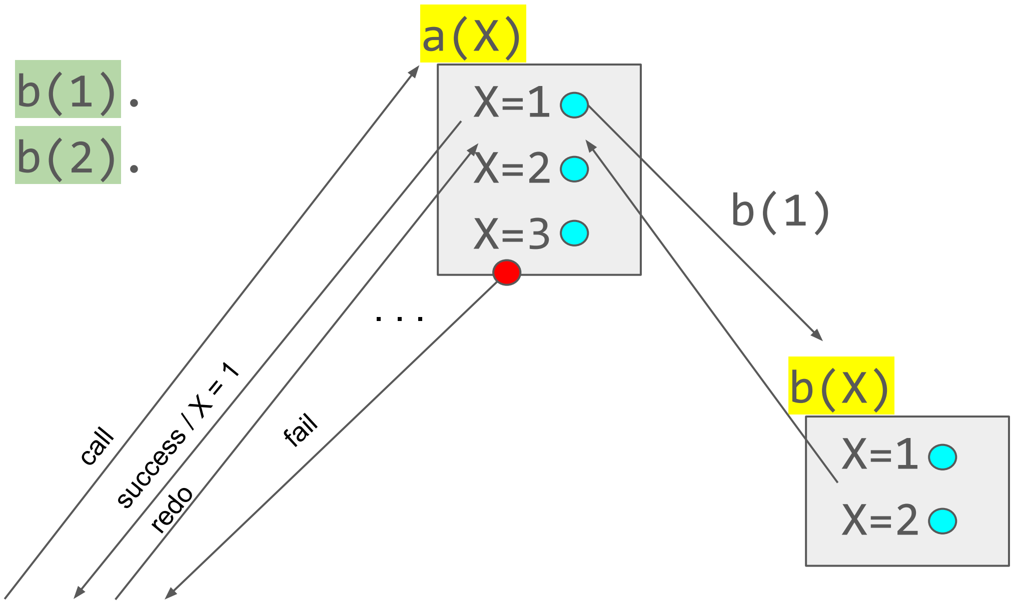
Videos

- Cut
- Negation
- Databases (using the relational calculus for a relational database)

A procedural interpretation of cut(!)

```
a(1) :- b(1).
a(2) :- b(2).
a(3).

b(1).
b(2).
```




```
?- a(X).
```

A procedural interpretation of cut(!)

```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

q :- a(X), b(X), !, c(X).

:- q.
```

A procedural interpretation of cut(!) - without the cut:

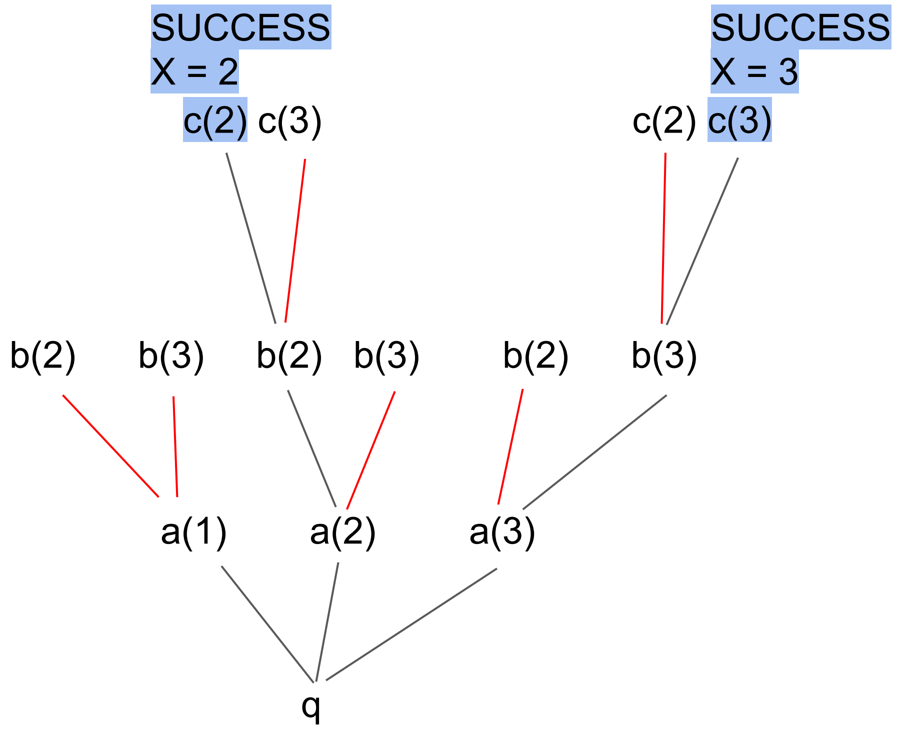
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

q :- a(X), b(X), c(X).

:- q.



A procedural interpretation of cut(!) - without the cut:

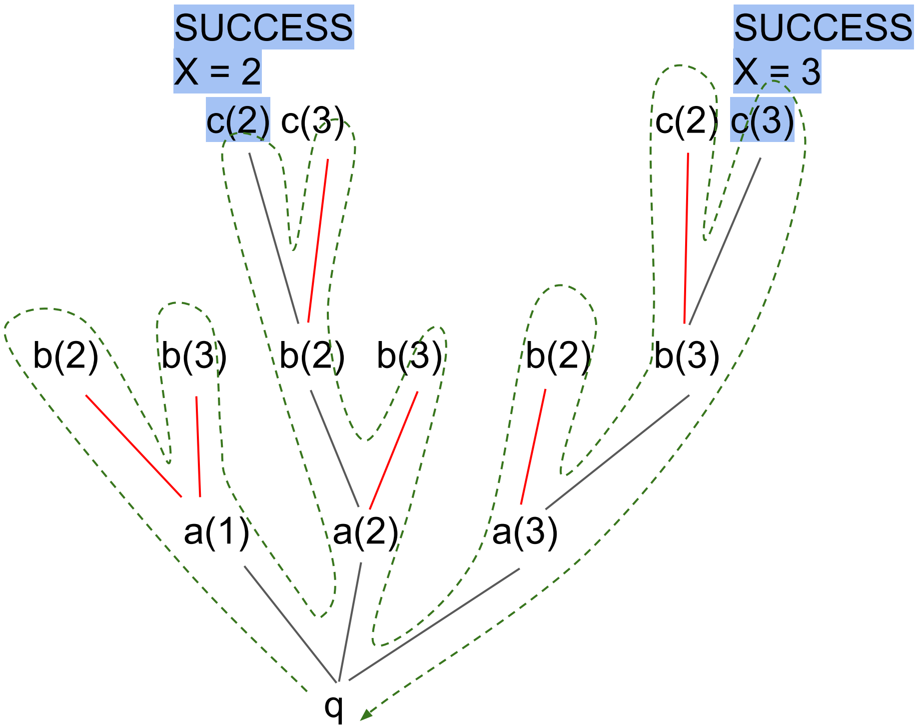
```
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

q :- a(X), b(X), c(X).

:- q.
```



DEPTH-FIRST LEFT-TO-RIGHT SEARCH

A procedural interpretation of cut(!)

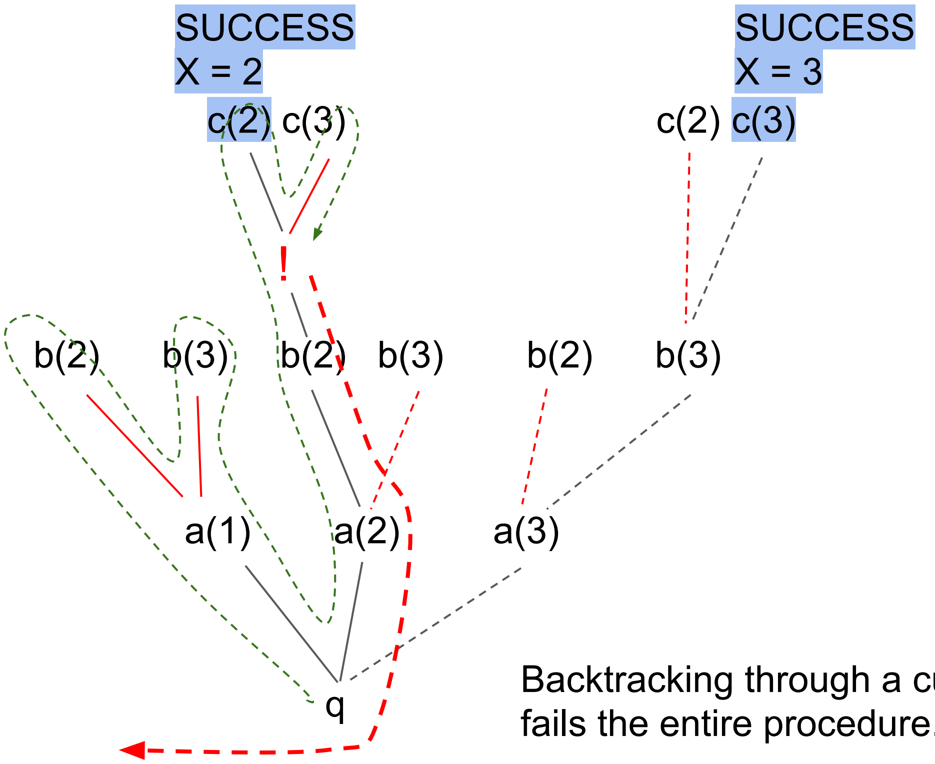
a(1).
a(2).
a(3).

b(2).
b(3).

c(2).
c(3).

q :- a(X), b(X), !, c(X).

:- q.



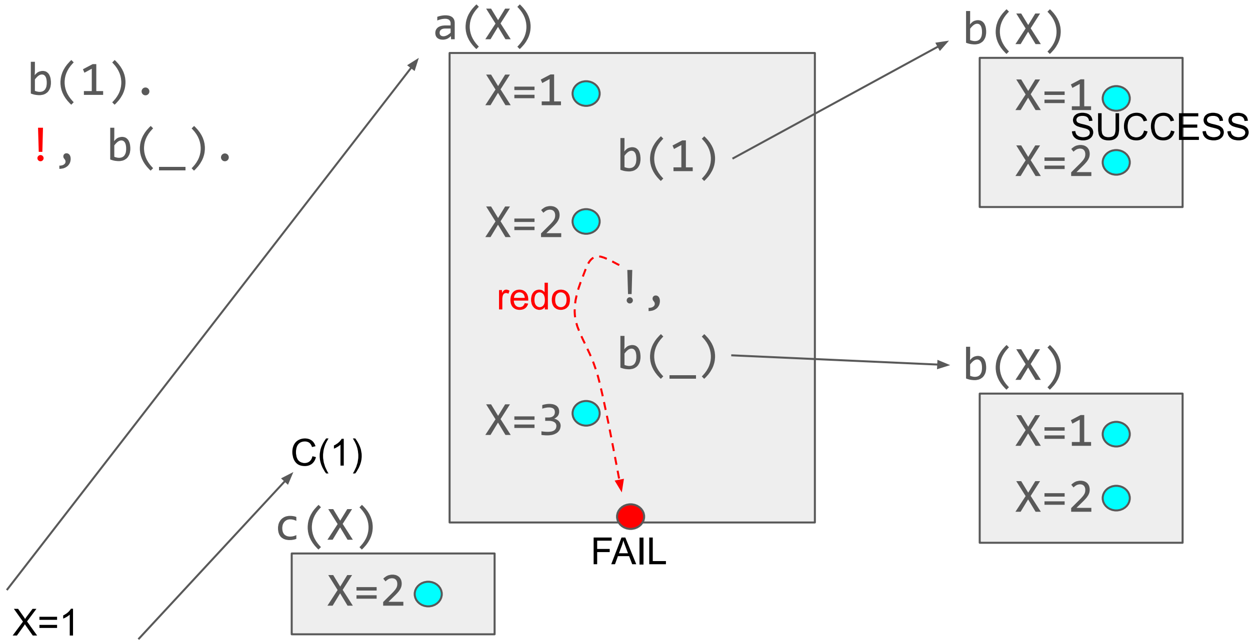
Backtracking through a cut fails the entire procedure.

A procedural interpretation of cut(!)

```
a(1) :- b(1).
a(2) :- !, b(_).
a(3).

b(1).
b(2).

c(2).
```



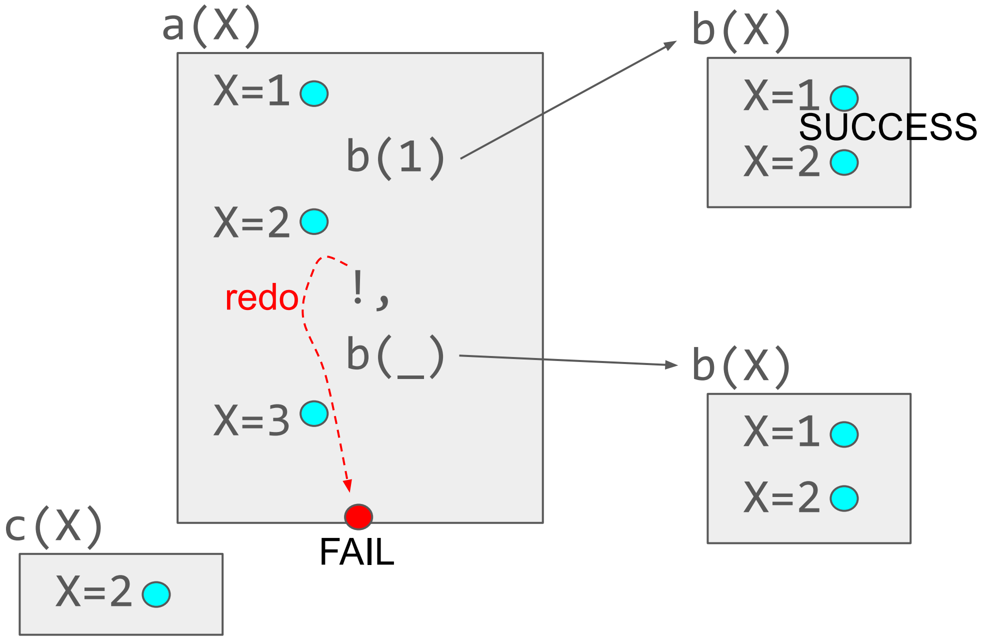

```
?- a(X), c(X).
```

A procedural interpretation of cut(!)

```
a(1) :- b(1).
a(2) :- !, b(_).
a(3).

b(1).
b(2).

c(2).
```




```
?- b(X), a(X), c(X).
```

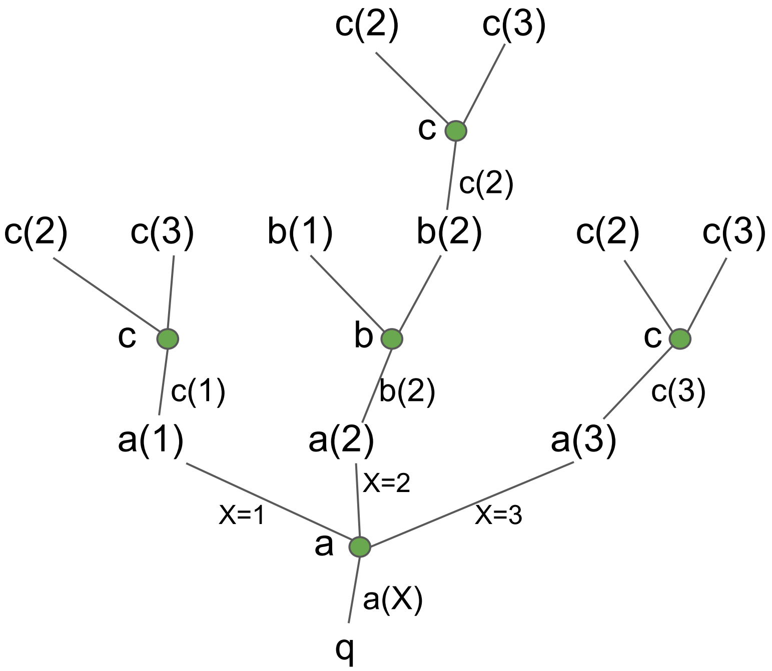
another (very similar) example - without cut

a(1).
a(2) :- b(2).
a(3).

b(1).
b(2).

c(2).
c(3).

q :- a(X), c(X).



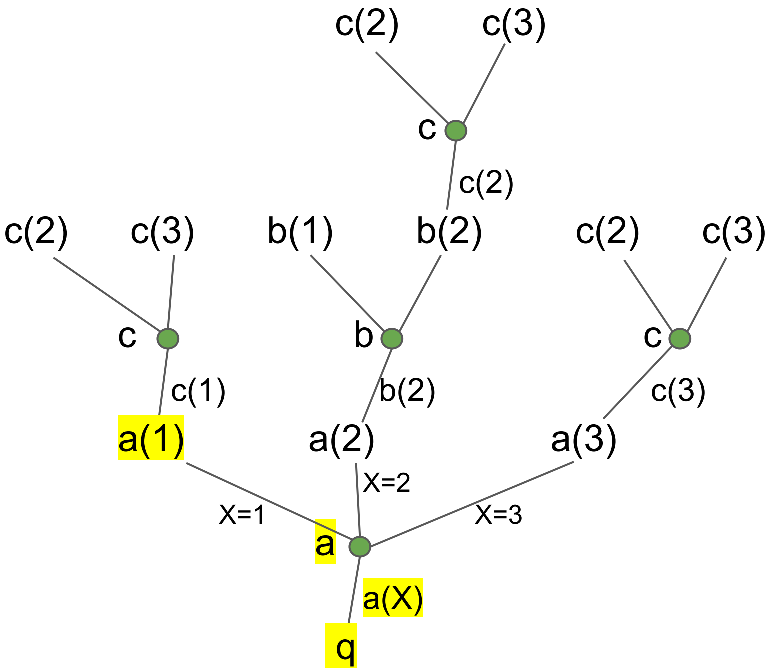
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



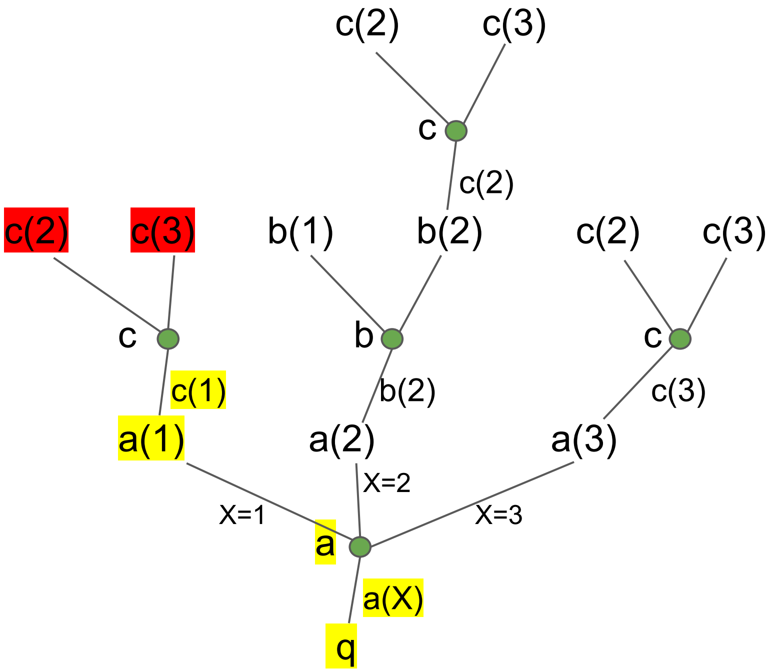
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



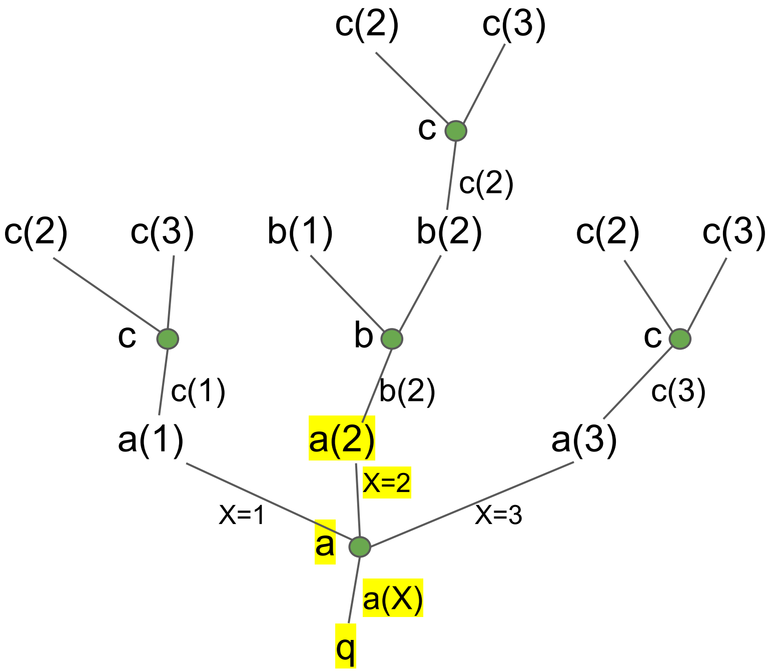
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



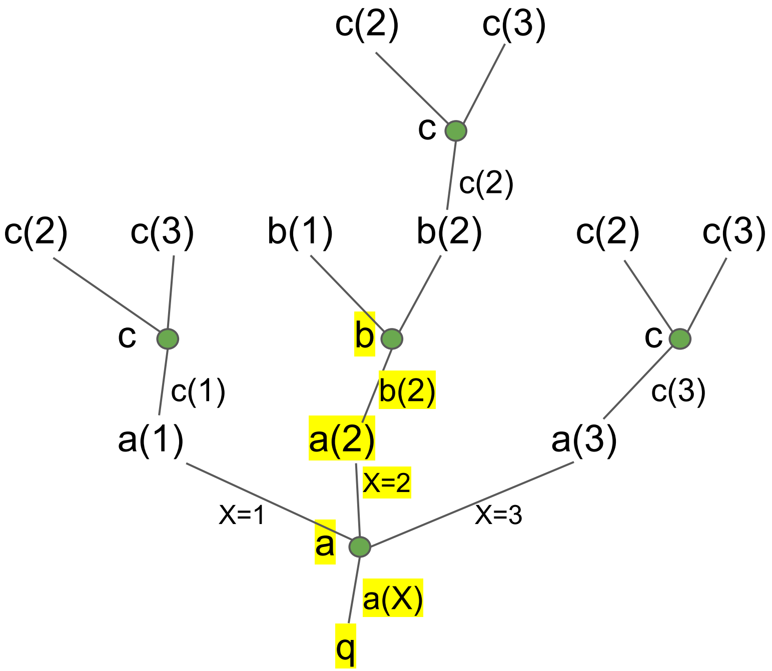
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



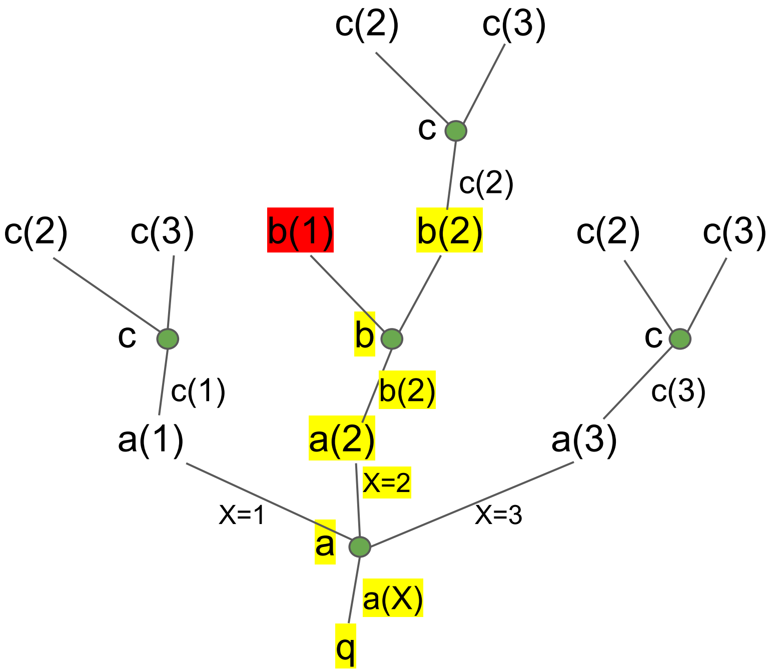
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



another (very similar) example - without cut

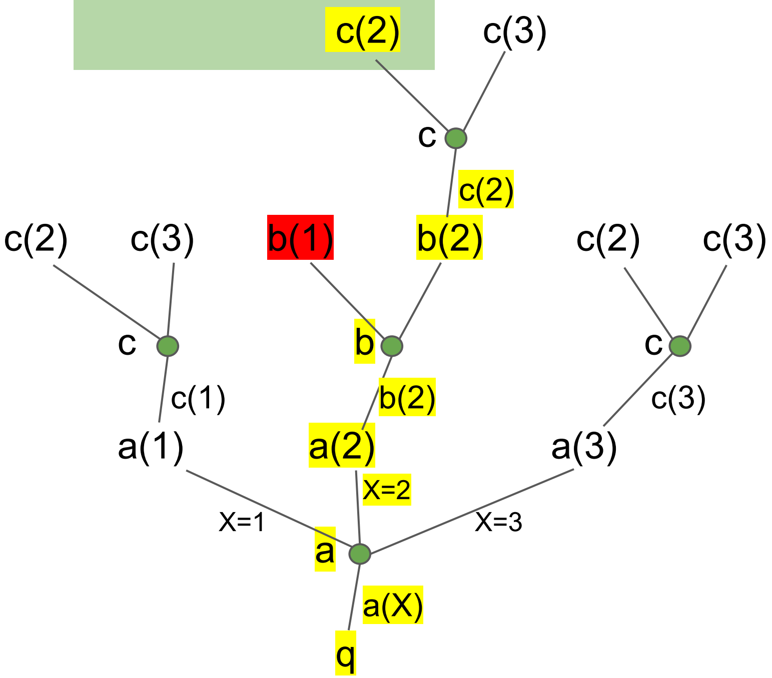
```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```

SOLUTION: X=2



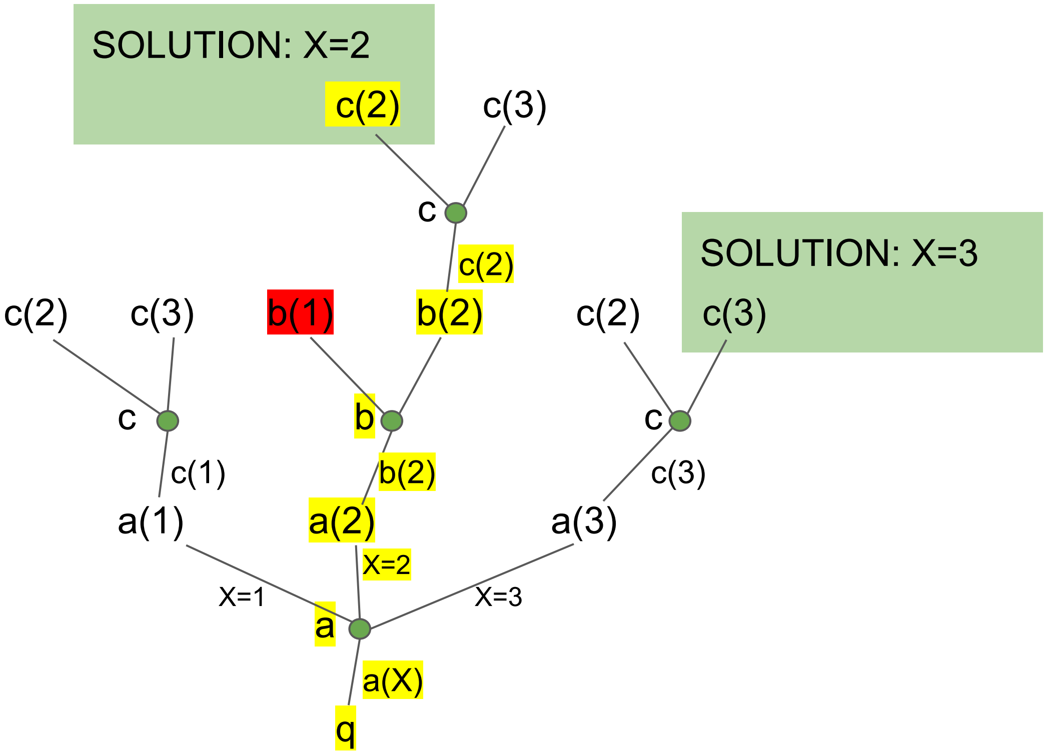
another (very similar) example - without cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```



another (very similar) example - cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```

another (very similar) example - cut

```
a(1).
a(2) :- !, b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```

another (very similar) example - cut

```
a(1).
a(2) :- b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), !, c(X).
```

another (very similar) example - cut

```
a(1).
a(2) :- !, b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- a(X), c(X).
```

the last word on what cut does...

```
a(1).
a(2) :- !, b(2).
a(3).
```

```
b(1).
b(2).
```

```
c(2).
c(3).
```

```
q :- c(X), a(X), c(X).
```

Cut(!) toxicology

```
a(1).
a(2) :- !.
a(3).

b(X) :- a(X).
b(4).
```

```
:- a(X).
X = 1 ;
X = 2
```

```
:- b(X).
X = 1 ;
X = 2 ;
X = 4
```

```
:- a(3).
true
```

Cut(!) toxicology

```
a(1).
a(2) :- !.
a(3).

b(X) :- a(X).
b(4).
```

```
:- a(X).
X = 1 ;
X = 2
```

```
:- b(X).
X = 1 ;
X = 2 ;
X = 4
```

```
:- a(3).
true
```

negation / not

```
q :- ... ,\+ foo(X), ...
```

relational database

```
%    name       age                     name   floor
age(andy,        35).        location(andy,       2).
age(alastair, 45).           location(alastair, 2).
age(ian,         65).        location(ian,        1).
age(jon,         60).
```

```
SELECT name, floor FROM age,location
 WHERE age.name=location.name AND age > 40.
```

```
:- age(Name,Age), location(Name,Floor), Age > 40.
```

List relations - len/2, mem/2

```
% len(+L,-N)
% succeeds if length of list L is N.
len([],0).
len([_|T],N) :- len(T,M), N is M+1.

% mem(?X,?L)
% succeeds if X is in list L.
mem(X,[X|_]).
mem(X,[_|T]) :- mem(X,T).
```

List relations - app/3, reverse/2

```
% app(?L1,?L2,?L3) = APPEND
% succeeds if L1 appended to L2 is L3.
app([],L2,L2).
app([X|T],L2,[X|L3]) :- app(T,L2,L3).

% reverse(+L1,-L2)
% succeeds if list L2 is the reverse of list L1
reverse([],[]).
reverse([X|T], L) :- reverse(T, L1), append(L1,[X],L).
```

List relations - take/3, perm/2

```
% take(+L1,-X,-L2)
% succeeds if list L2 is list L1 minus element X
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

% perm(+L1,-L2)
% succeeds if list L2 is a permutation of list L1
perm([],[]).
perm(List,[H|T]) :- take(List,H,R), perm(R,T).
```

List relations - take/3, perm/2

```
% take(+L1,-X,-L2)
% succeeds if list L2 is list L1 minus element X
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).

% perm(+L1,-L2)
% succeeds if list L2 is a permutation of list L1
perm([],[]).
perm(List,[H|T]) :- take(List,H,R), perm(R,T).
```

List relations - len/2, rev/2 with accumulators

```
% len(+L,-N) succeeds if length of list L is N.
len(L,N) :- len_acc(L,0,N).

% len_acc(+L,+A,-N) succeeds if A is an accumulated length so far,
% L is the remaining list to be counted,  N is the total length of the original list.
len_acc([],A,A).
len_acc([_|T],A, N) :- A1 is A + 1, len_acc(T,A1,N).

% rev(+L1,-L2) succeeds if list L2 is the reverse of list L1
rev(L1, L2) :- rev_acc(L1, [], L2).

% rev_acc(+L1, +ListAcc, -L2) succeeds if ListAcc is the accumulated reversed list so far,
% L1 is the remaining list to be reversed, L2 is the reverse of the original list.
rev_acc([], ListAcc, ListAcc).
rev_acc([H|T], ListAcc, L2) :-  rev_acc(T, [H|ListAcc], L2).
```

List relations - len/2, rev/2 with accumulators

Simple:

```
len(+L,-N)
mem(?X,?L)
app(?L1,?L2,?L3)
reverse(+L1,-L2)
take(+L1,-X,-L2)
perm(+L1,-L2)
```

Accumulator:

```
len(+L,-N)
rev(+L1,-L2)
```

ASSUME member(X,L), append(L1,L2,L3).

Next time

Lecture #6: Videos

- Countdown (more generate-and-test)
- Graph search

Lecture #7: Difference lists

Lecture #8: (Sudoku), Wrap Up.