Programming Paradigms

Lecture 12. Cut and negation as failure in Prolog

Outline

- The Cut
- Using cut for efficiency
- Using cut for fixing overlapping branches
- Negation as failure

Overlapping solutions

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```

Overlapping solutions

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).

?- p(X).
```

Overlapping solutions

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).

?- p(X).
X = 1
```

Overlapping solutions

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).

?- p(X).
X = 1
X = 1
```

Overlapping solutions

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).

?- p(X).
X = 1
X = 1
X = 2
```

The Cut: simple example

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).

?- p(X).
```

The Cut: simple example

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).

?- p(X).
X = 1
```

The Cut: simple example

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1).

?- p(X).
X = 1
```

The cut operator (`!`):

1. Always succeeds.
2. Commits variable substitutions made by pattern matching in the head.
3. Commits variable substitutions made by all prior predicates in the body.

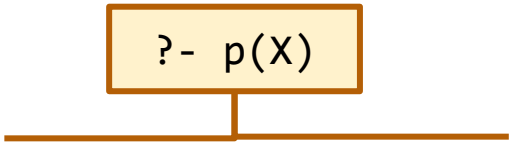
Branching without cut

```
?- p(X)
```

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```

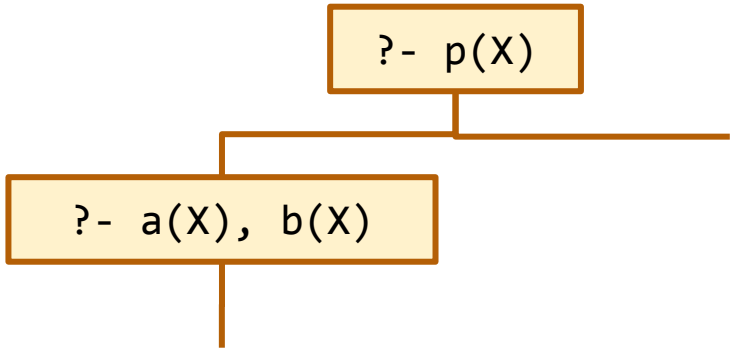
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



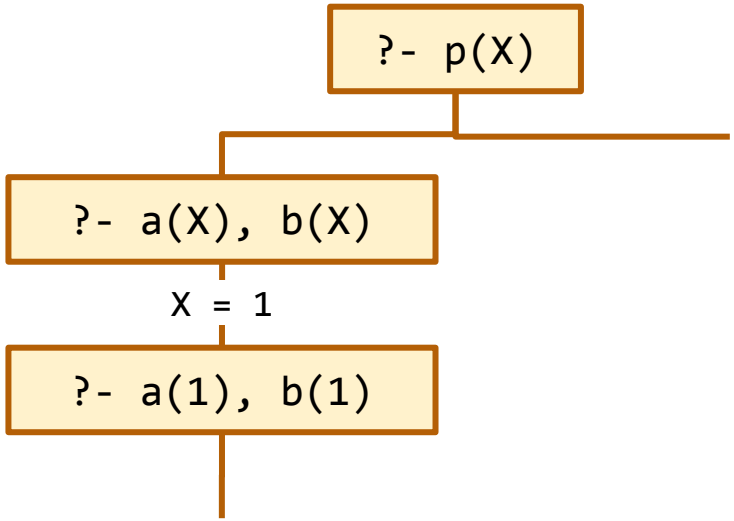
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



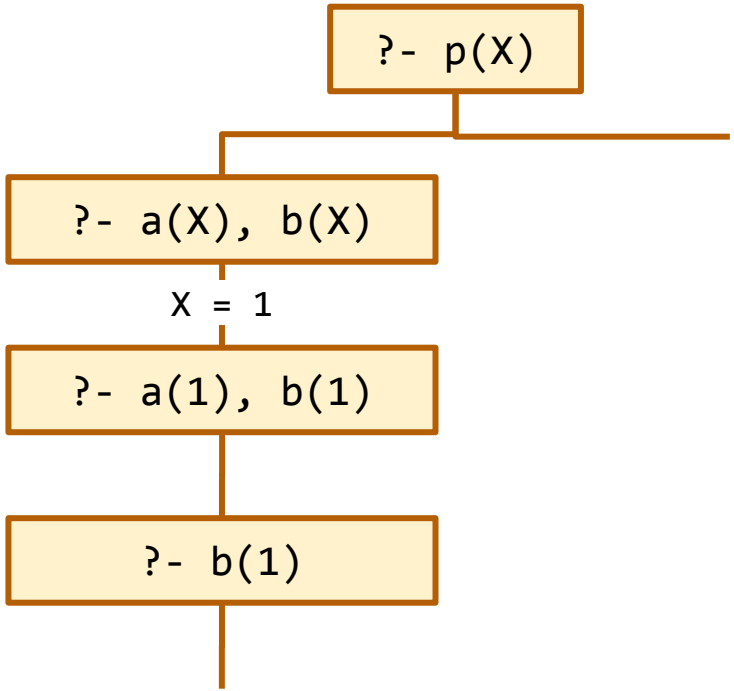
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



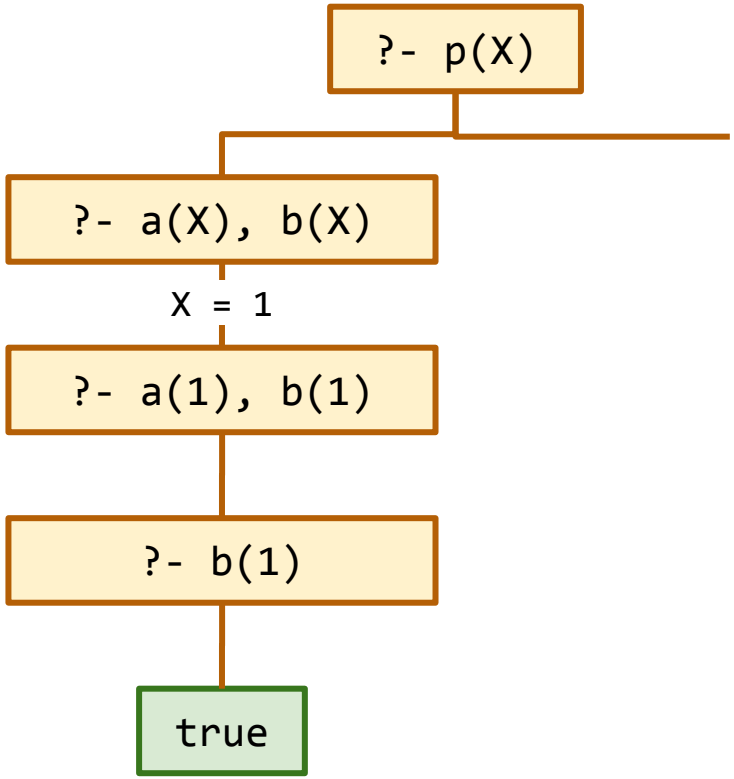
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



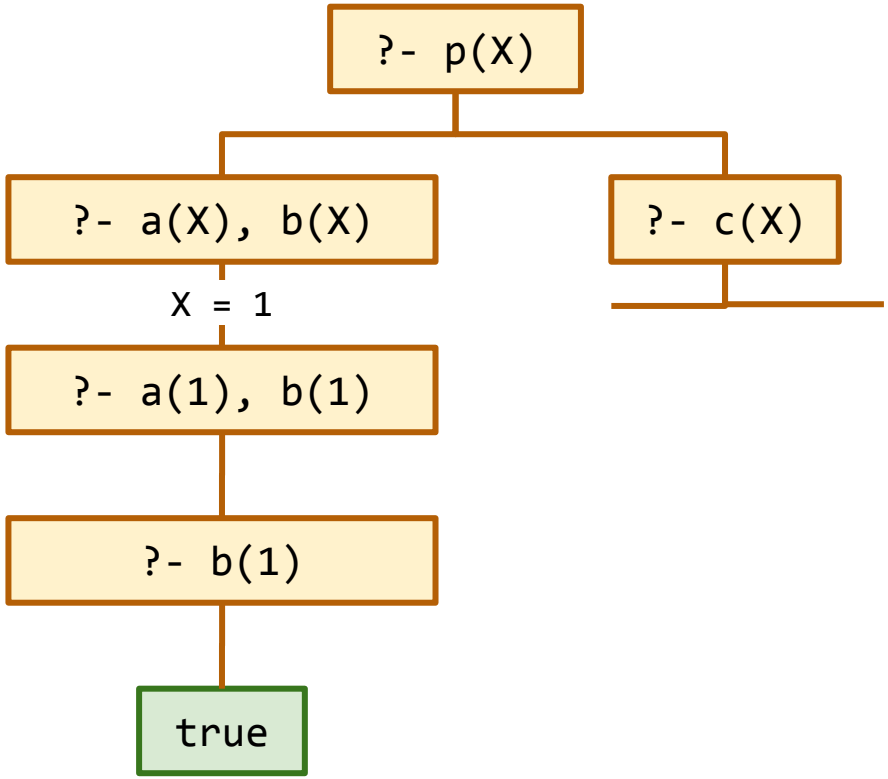
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



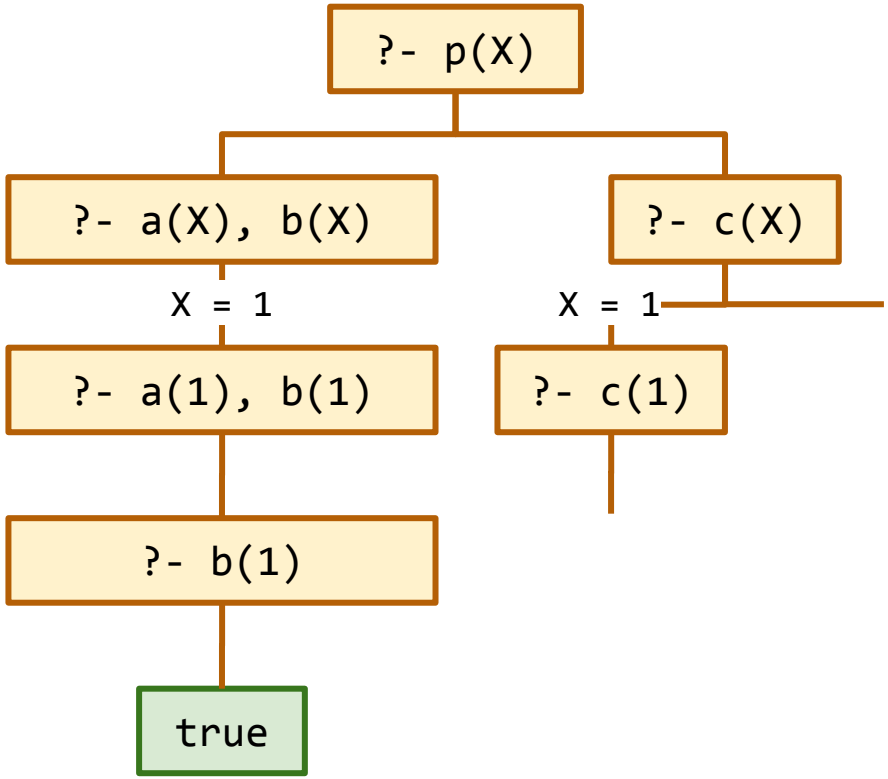
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



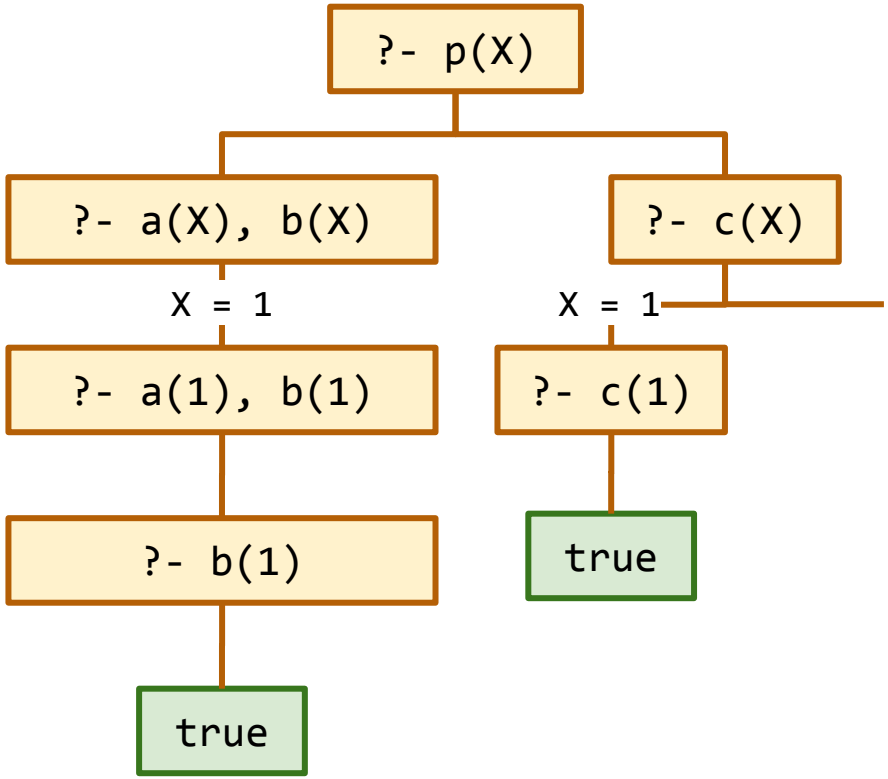
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



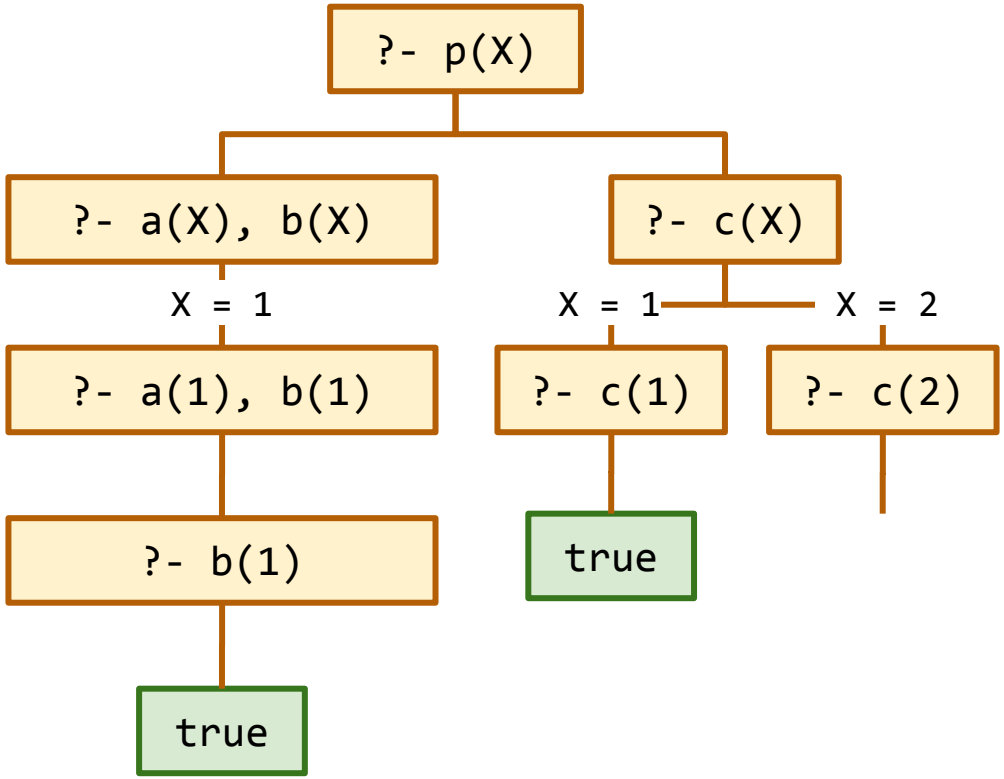
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



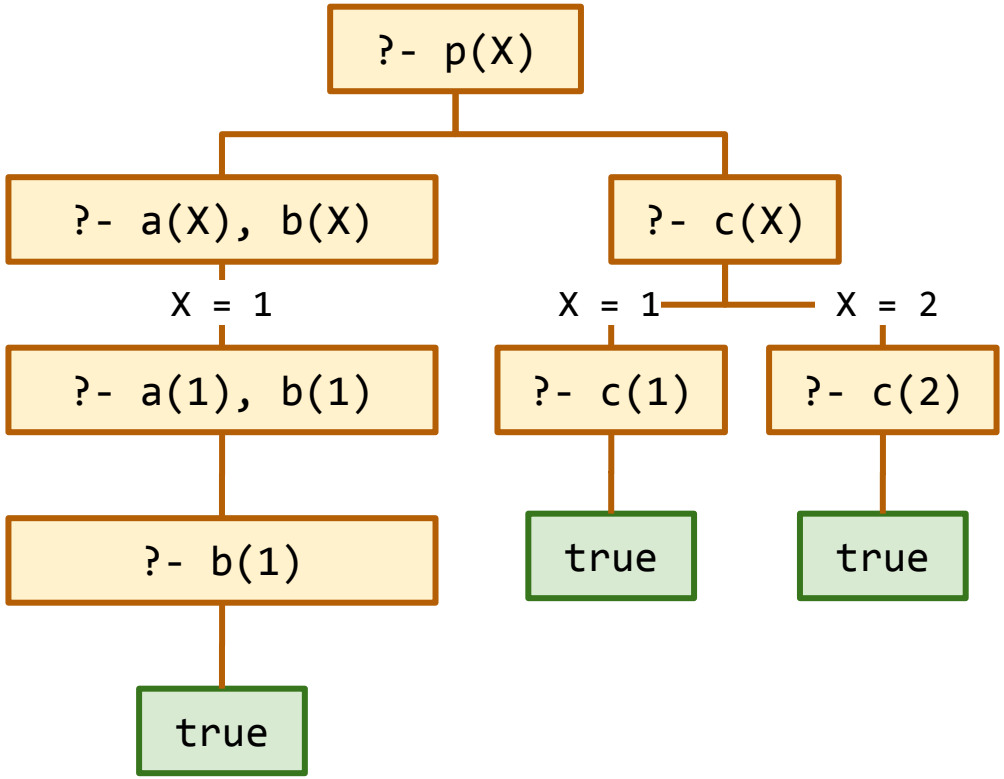
Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



Branching without cut

```
p(X) :- a(X), b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



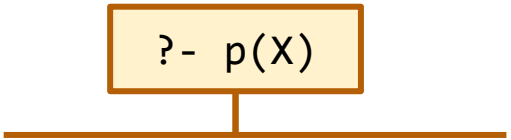
Cutting branches

```
?- p(X)
```

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```

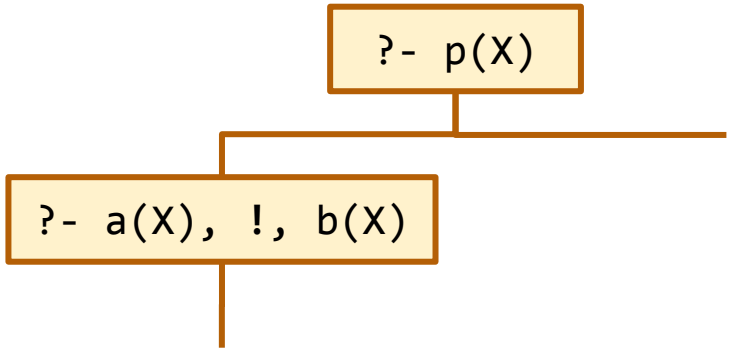
Cutting branches

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



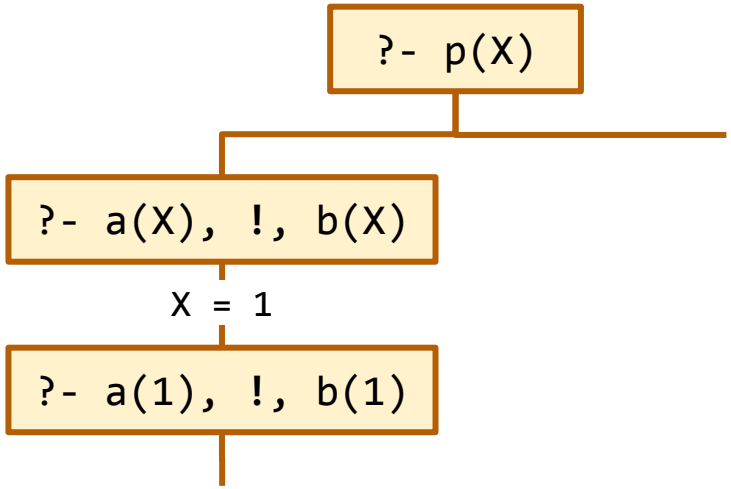
Cutting branches

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



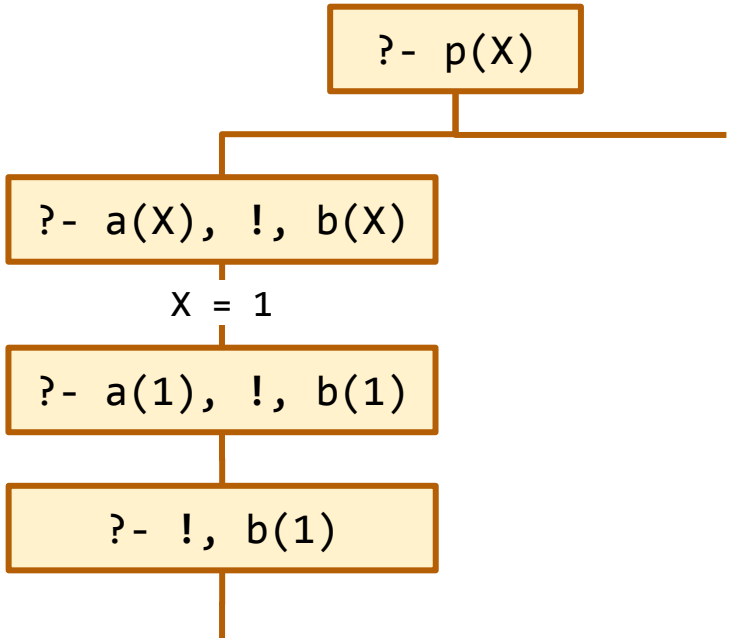
Cutting branches

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



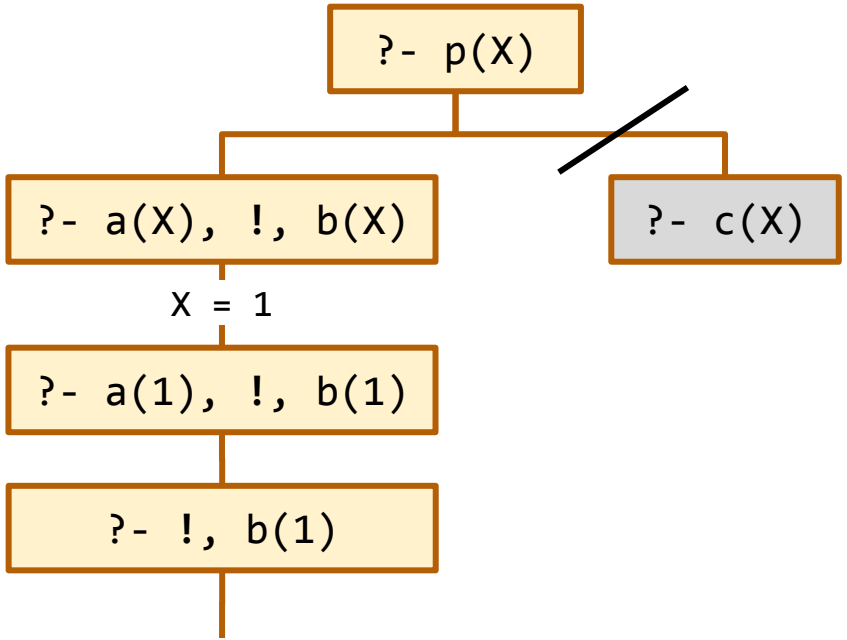
Cutting branches

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



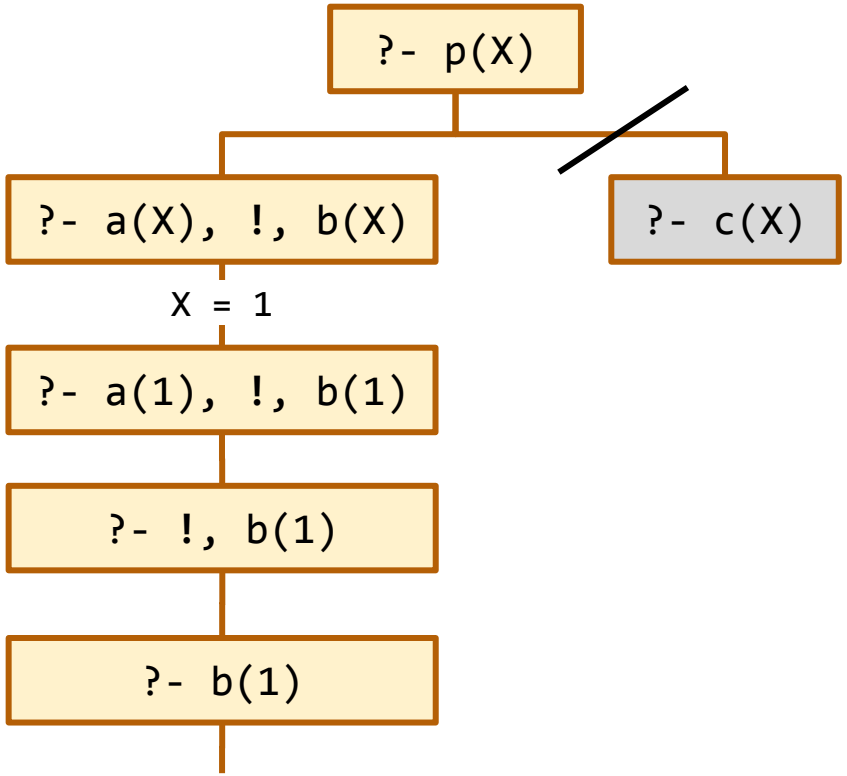
Cutting branches

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



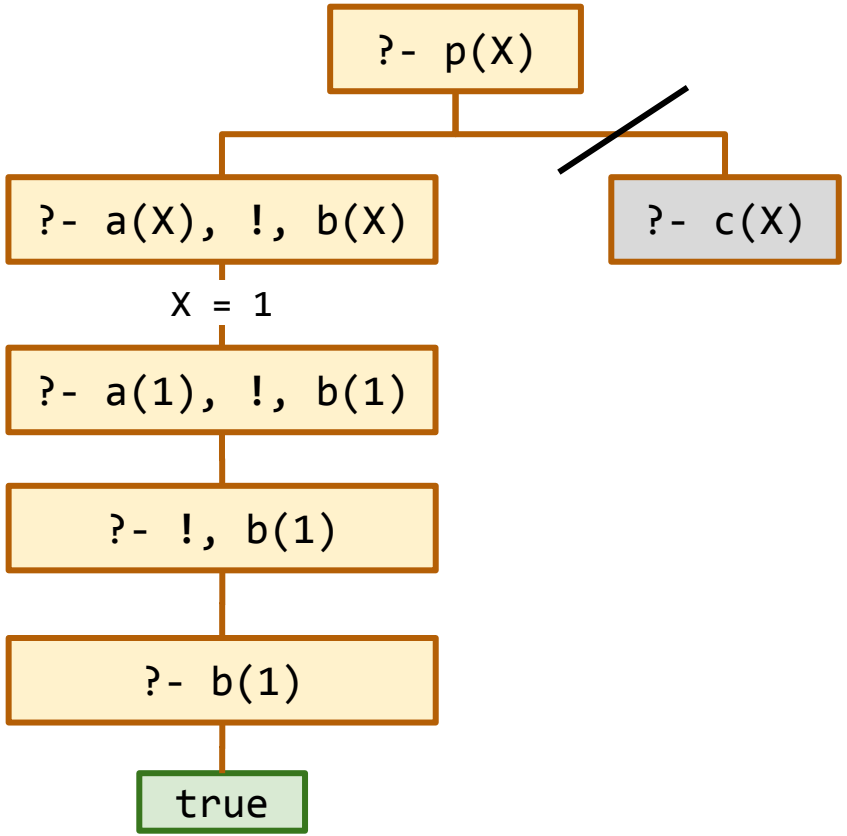
Cutting branches

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



Cutting branches

```
p(X) :- a(X), !, b(X).
p(X) :- c(X).

a(1). b(1). b(2). c(1). c(2).
```



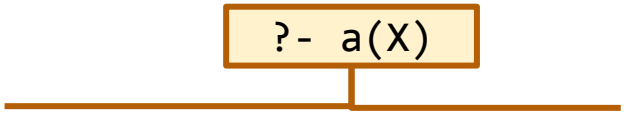
The Cut: general example

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

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```

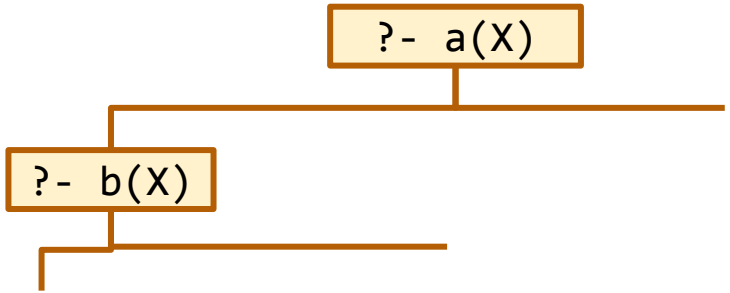
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



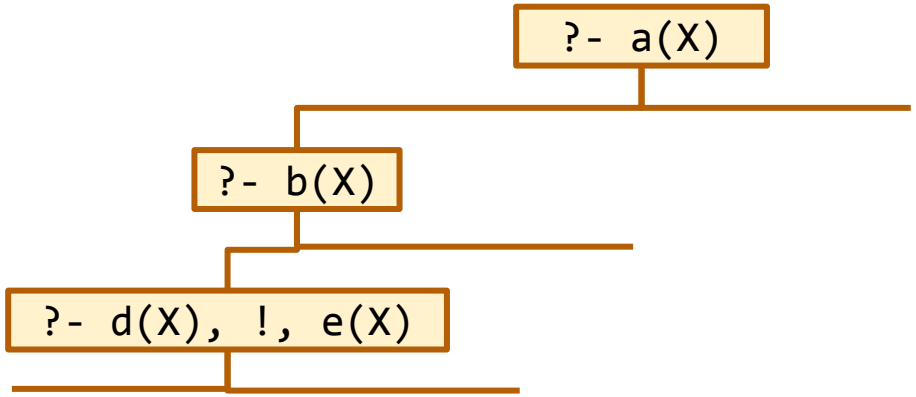
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



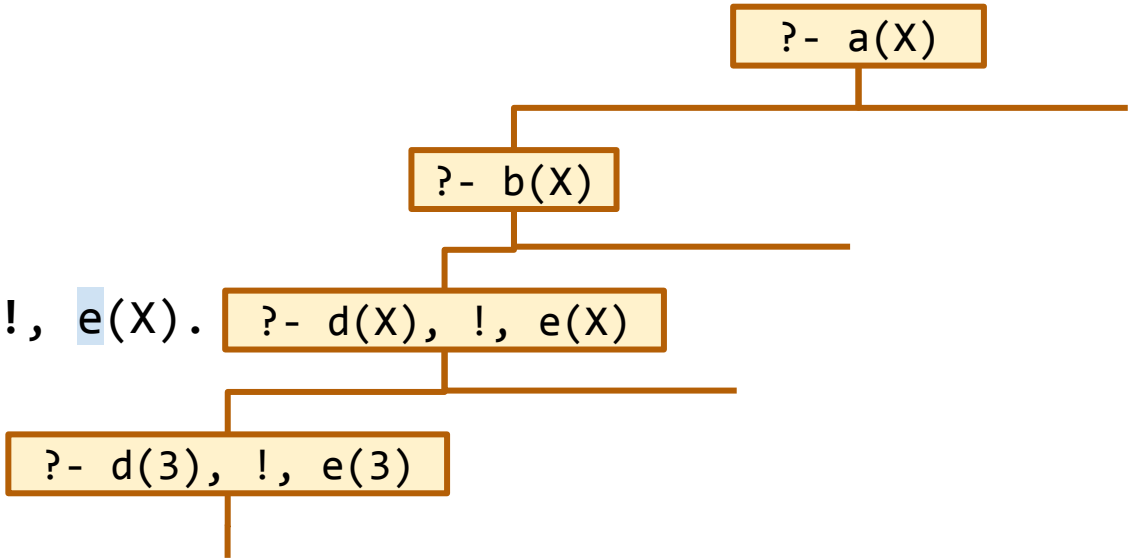
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



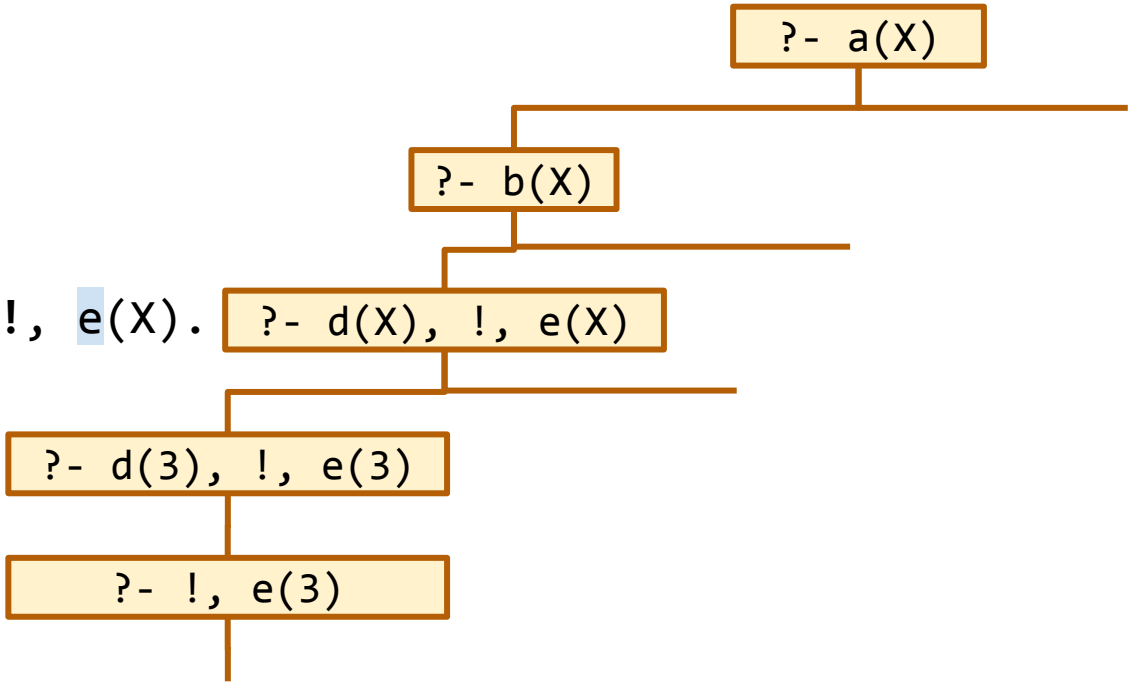
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



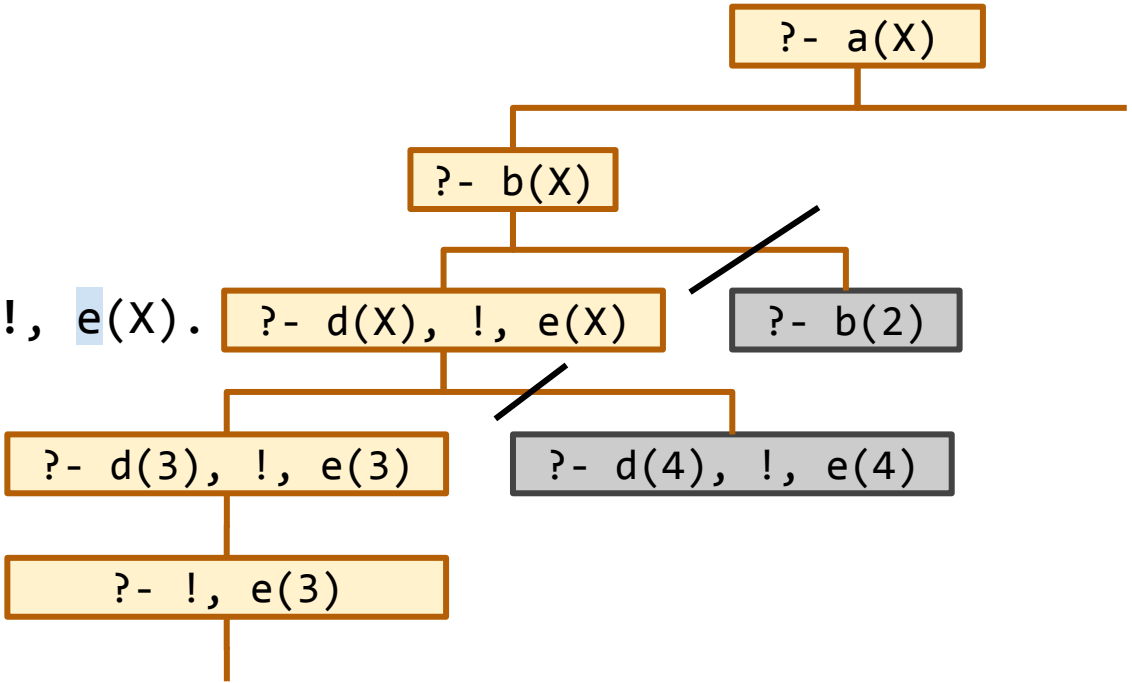
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



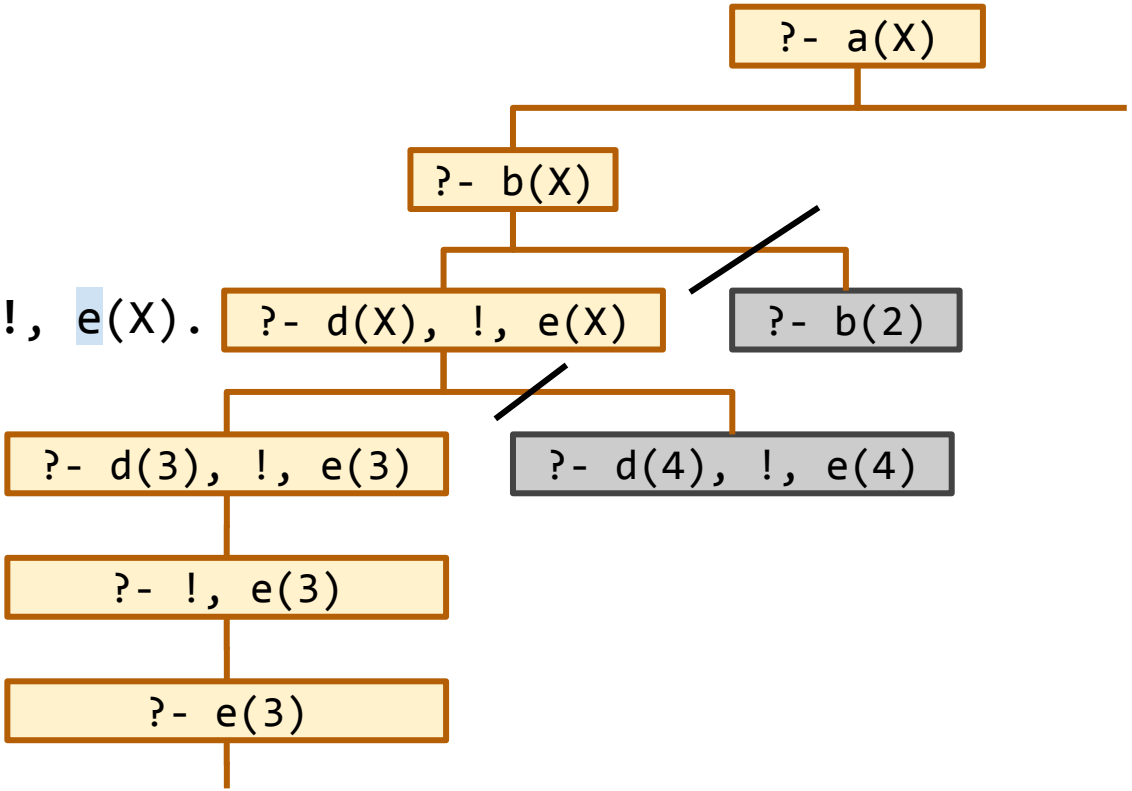
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



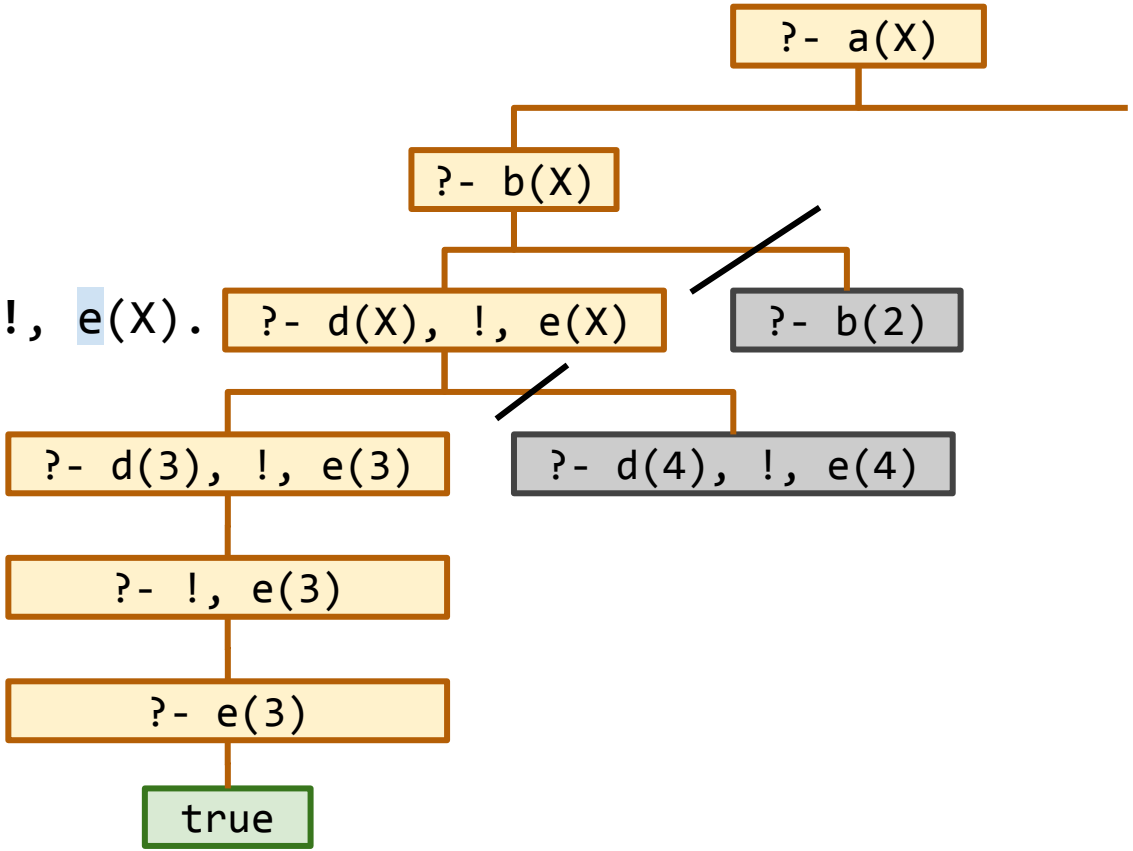
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



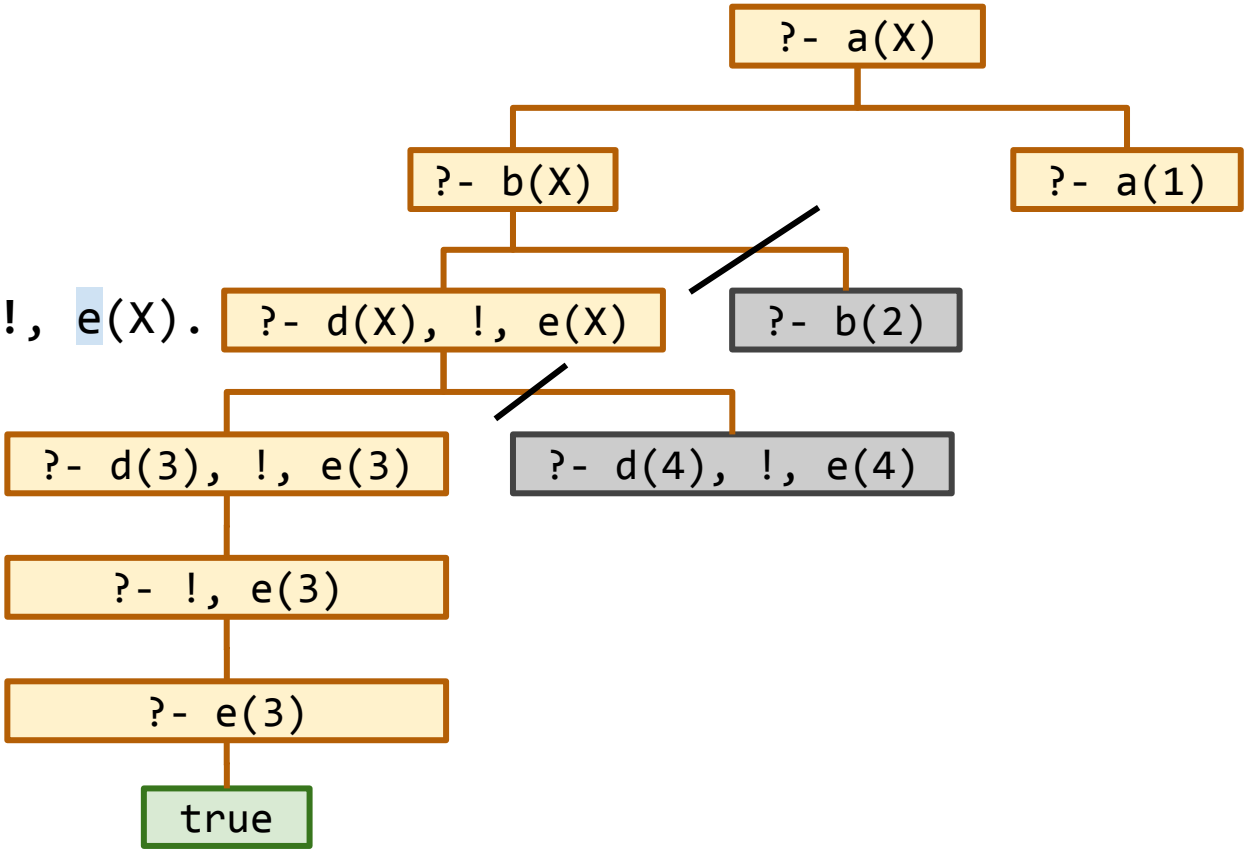
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



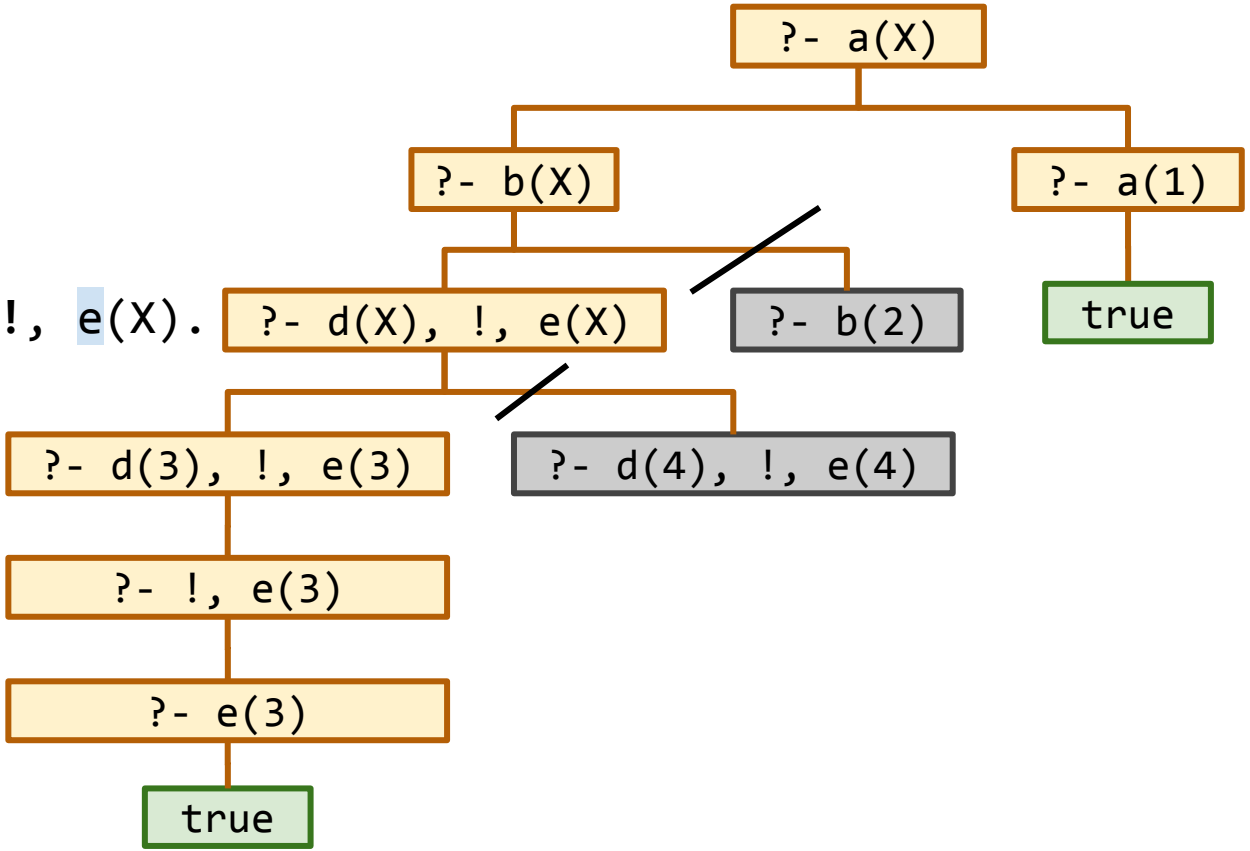
The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



The Cut: general example

```
a(X) :- b(X).
a(1).

b(X) :- d(X), !, e(X).
b(2).

d(3). d(4).
e(3). e(4).
```



Using Prolog's cut for efficiency: max

```
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- X =< Y.
```

What happens when we execute this query?

```
?- max(3, 2, X)
```

Using Prolog's cut for efficiency: max

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y) :- X =< Y.
```

What happens when we execute this query?

```
?- max(3, 2, X)
```

Using Prolog's cut for efficiency: max

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y).
```

What happens when we execute this query?

```
?- max(3, 2, X)
```

No Cut vs Green Cut vs Red Cut

```
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- X =< Y.
```

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y) :- X =< Y.
```

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y).
```

No Cut vs Green Cut vs Red Cut

```
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- X =< Y.
```

No cut — inefficient.

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y) :- X =< Y.
```

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y).
```

No Cut vs Green Cut vs Red Cut

```
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- X =< Y.
```

No cut — inefficient.

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y) :- X =< Y.
```

Green cut. Improves efficiency. Safe to remove (does not change the meaning).

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y).
```

No Cut vs Green Cut vs Red Cut

```
max(X, Y, X) :- X > Y.
max(X, Y, Y) :- X =< Y.
```

No cut — inefficient.

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y) :- X =< Y.
```

Green cut. Improves efficiency. Safe to remove (does not change the meaning).

```
max(X, Y, X) :- X > Y, !.
max(X, Y, Y).
```

Red cut. Improves efficiency. **Unsafe** to remove (will change the meaning).

Using Prolog’s cut to fix overlapping branches

```
interleave([], Y, Y).
interleave(X, [], X).
interleave([HX|TX], [HY|TY], [HX,HY|T])
  :- interleave(TX, TY, T).
```

Using Prolog's cut to fix overlapping branches

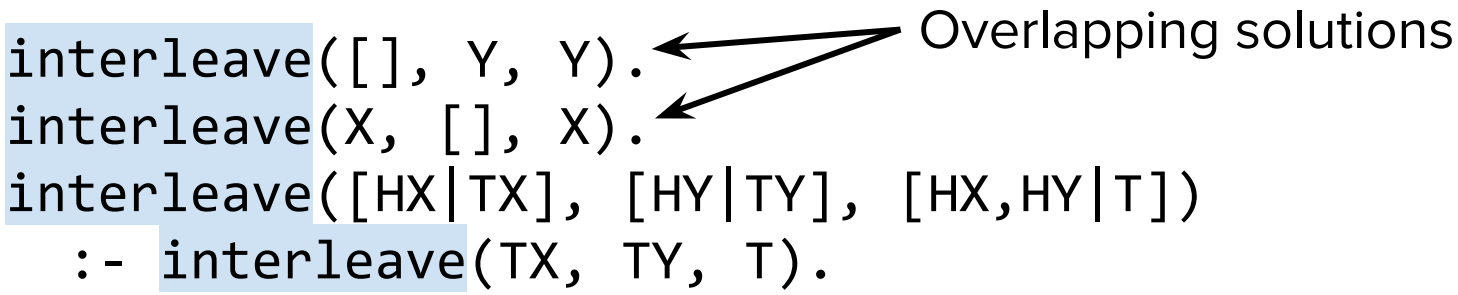
```
interleave([], Y, Y).
interleave(X, [], X).
interleave([HX|TX], [HY|TY], [HX,HY|T])
  :- interleave(TX, TY, T).


?- interleave([1, 2], [a, b], X)
```

Using Prolog's cut to fix overlapping branches

```
interleave([], Y, Y).
interleave(X, [], X).
interleave([HX|TX], [HY|TY], [HX,HY|T])
  :- interleave(TX, TY, T).
```

```
?- interleave([1, 2], [a, b], X)
X = [1, a, 2, b]
X = [1, a, 2, b]
```

Using Prolog's cut to fix overlapping branches




```
interleave([], Y, Y).
interleave(X, [], X).
interleave([HX|TX], [HY|TY], [HX,HY|T])
  :- interleave(TX, TY, T).
```

```
?- interleave([1, 2], [a, b], X)
X = [1, a, 2, b]
X = [1, a, 2, b]
```

Using Prolog’s cut to fix overlapping branches

```
interleave([], Y, Y) :- !.
interleave(X, [], X).
interleave([HX|TX], [HY|TY], [HX,HY|T])
  :- interleave(TX, TY, T).
```

Using Prolog’s cut to fix overlapping branches

```
interleave([], Y, Y) :- !.
interleave(X, [], X).
interleave([HX|TX], [HY|TY], [HX,HY|T])
  :- interleave(TX, TY, T).


?- interleave([1, 2], [a, b], X)
```

Using Prolog's cut to fix overlapping branches

```
interleave([], Y, Y) :- !.
interleave(X, [], X).
interleave([HX|TX], [HY|TY], [HX,HY|T])
  :- interleave(TX, TY, T).
```

```
?- interleave([1, 2], [a, b], X)
X = [1, a, 2, b]
```

Negation using cut-fail combination

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

Negation using cut-fail combination

```
notMember(_, []).
notMember(X, [H|_]) :- H = X, !, fail.
```

Negation using cut-fail combination

```
notMember(_, []).
notMember(X, [H|_]) :- H = X, !, fail.
notMember(X, [_|T]) :- notMember(X, T).
```

Negation using cut-fail combination

```
notMember(_, []).
notMember(X, [H|_]) :- H = X, !, fail.
notMember(X, [_|T]) :- notMember(X, T).

?- notMember(2, [1, 2, 3]).
```

Negation using cut-fail combination

```
notMember(_, []).
notMember(X, [H|_]) :- H = X, !, fail.
notMember(X, [_|T]) :- notMember(X, T).

?- notMember(2, [1, 2, 3]).
false
```

Negation using cut-fail combination

```
notMember(_, []).
notMember(X, [H|_]) :- H = X, !, fail.
notMember(X, [_|T]) :- notMember(X, T).

?- notMember(2, [1, 2, 3]).
false

?- notMember(4, [1, 2, 3]).
```

Negation using cut-fail combination

```
notMember(_, []).
notMember(X, [H|_]) :- H = X, !, fail.
notMember(X, [_|T]) :- notMember(X, T).

?- notMember(2, [1, 2, 3]).
false

?- notMember(4, [1, 2, 3]).
true
```

Negation as failure

```
neg(Goal) :- Goal, !, fail.
neg(_Goal).
```

Negation as failure

```
neg(Goal) :- Goal, !, fail.
neg(_Goal).

notMember(_, []).
notMember(X, [H|T]) :- neg(H = X), notMember(X, T).
```

Negation as failure vs Logical negation

```
neg(Goal) :- Goal, !, fail.
neg(_Goal).

notMember(_, []).
notMember(X, [H|T]) :- neg(H = X), notMember(X, T).

?- notMember(X, [1, 2, 3])
```

Negation as failure vs Logical negation

```
neg(Goal) :- Goal, !, fail.
neg(_Goal).

notMember(_, []).
notMember(X, [H|T]) :- neg(H = X), notMember(X, T).

?- notMember(X, [1, 2, 3])
false
```

Negation as failure vs Logical negation

```
neg(Goal) :- Goal, !, fail.
neg(_Goal).

notMember(_, []).
notMember(X, [H|T]) :- neg(H = X), notMember(X, T).

?- notMember(X, [1, 2, 3])
false

?- notMember(4, [1, X, 3])
```

Negation as failure vs Logical negation

```
neg(Goal) :- Goal, !, fail.
neg(_Goal).

notMember(_, []).
notMember(X, [H|T]) :- neg(H = X), notMember(X, T).

?- notMember(X, [1, 2, 3])
false

?- notMember(4, [1, X, 3])
false
```

What was the most unclear part of the lecture for you?

See Moodle

References

1. Learn Prolog Now!