Lecture 6: Cuts and Negation

- Theory
  - Explain how to control Prolog`s backtracking behaviour with the help of the cut predicate
  - Introduce negation
  - Explain how cut can be packaged into a more structured form, namely negation as failure

The Cut

- Backtracking is a characteristic feature of Prolog
- But backtracking can lead to inefficiency:
  - Prolog can waste time exploring possibilities that lead nowhere
  - It would be nice to have some control
- The cut predicate !/0 offers a way to control backtracking

Example of cut

- The cut is a Prolog predicate, so we can add it to rules in the body:
  - Example:

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

- Cut is a goal that always succeeds
- Commits Prolog to the choices that were made since the parent goal was called

Explaining the cut

- In order to explain the cut, we will
  - Look at a piece of cut-free Prolog code and see what it does in terms of backtracking
  - Add cuts to this Prolog code
  - Examine the same piece of code with added cuts and look how the cuts affect backtracking

Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

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

Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

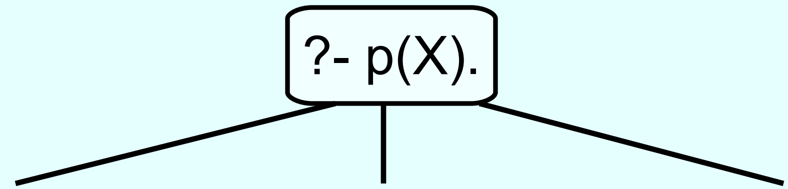
```
?- p(X).
```

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

Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

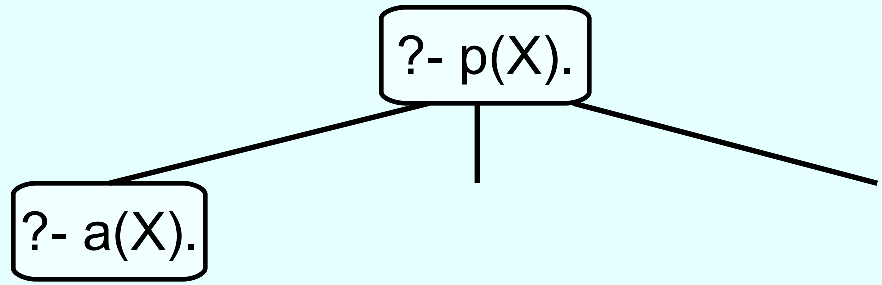
```
?- p(X).
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

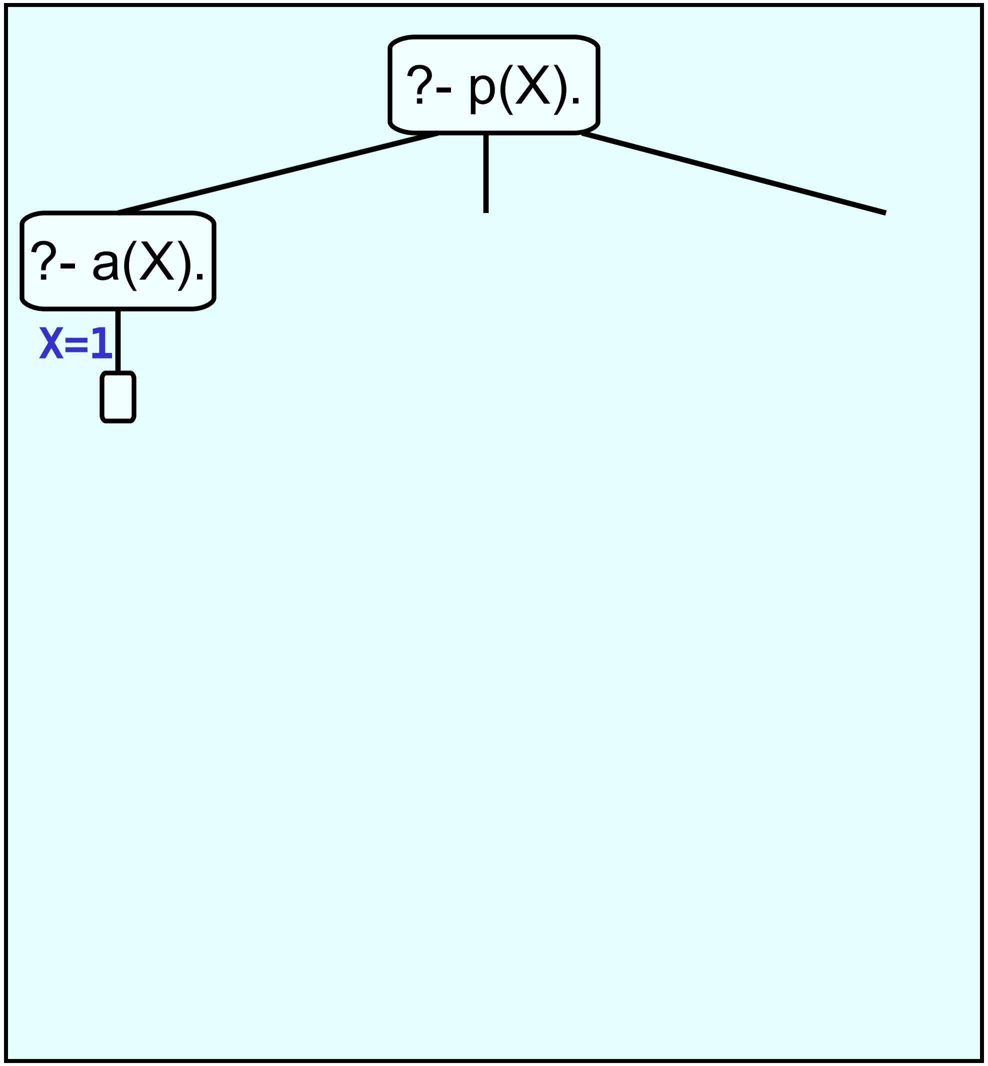
```
?- p(X).
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

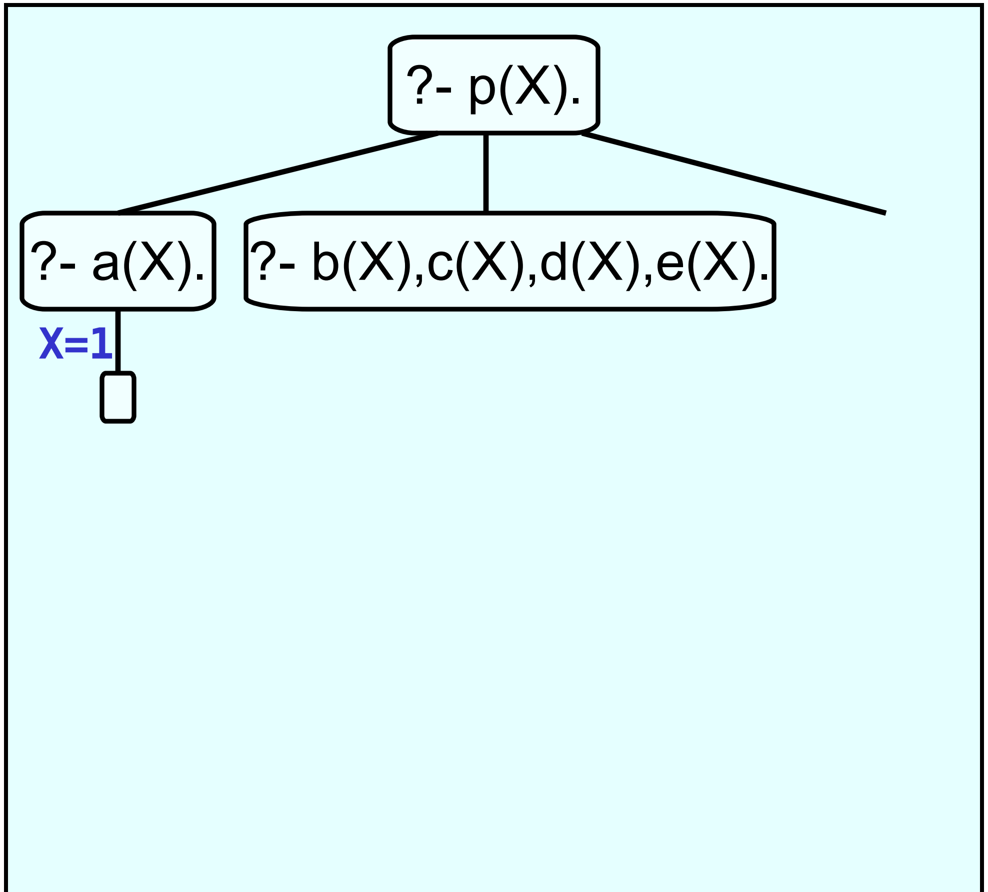
```
?- p(X).
X=1
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

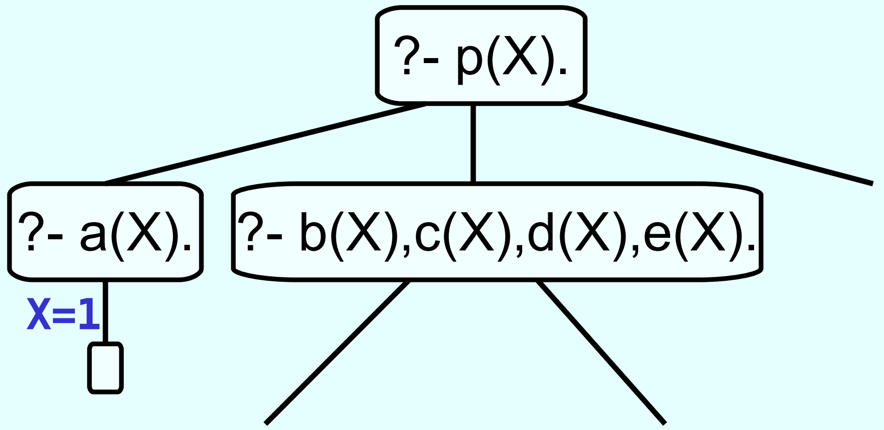
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

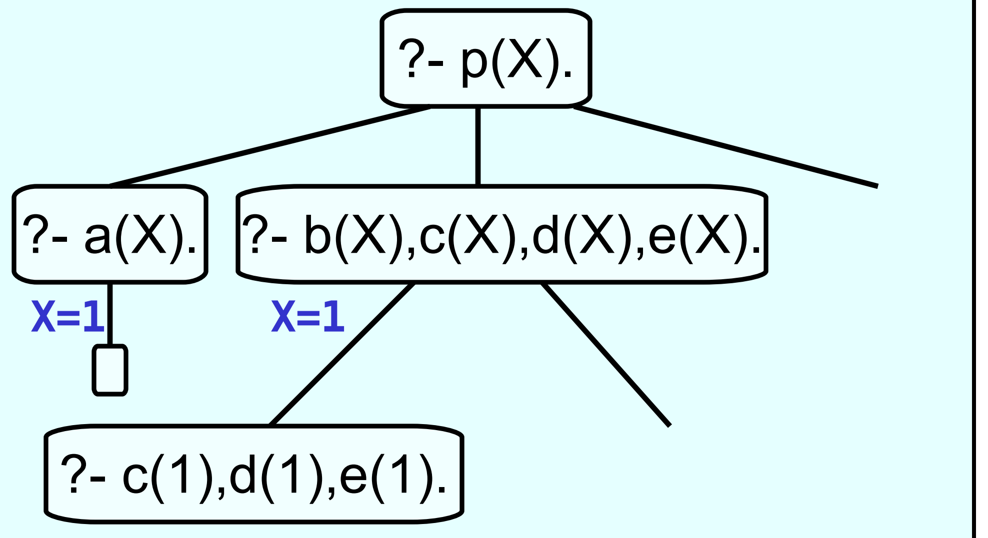
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

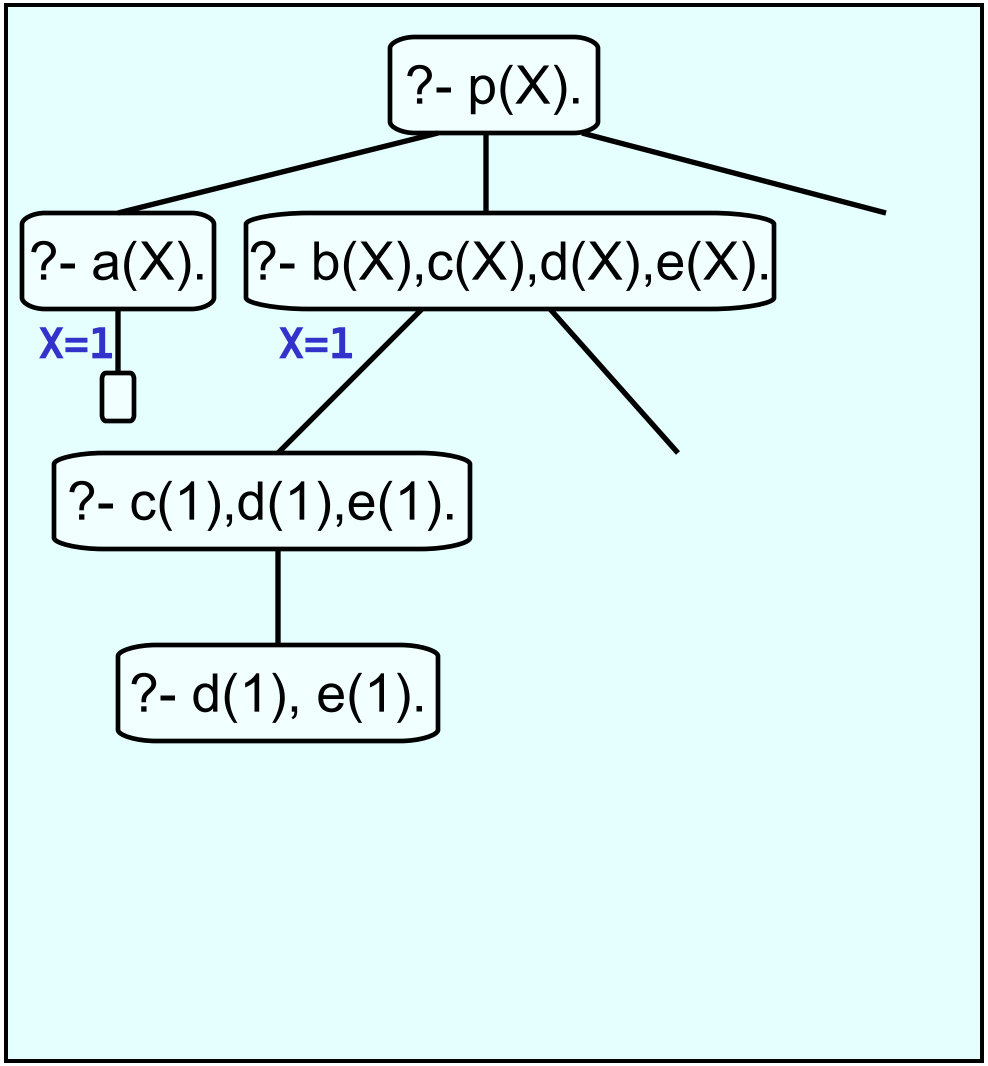
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

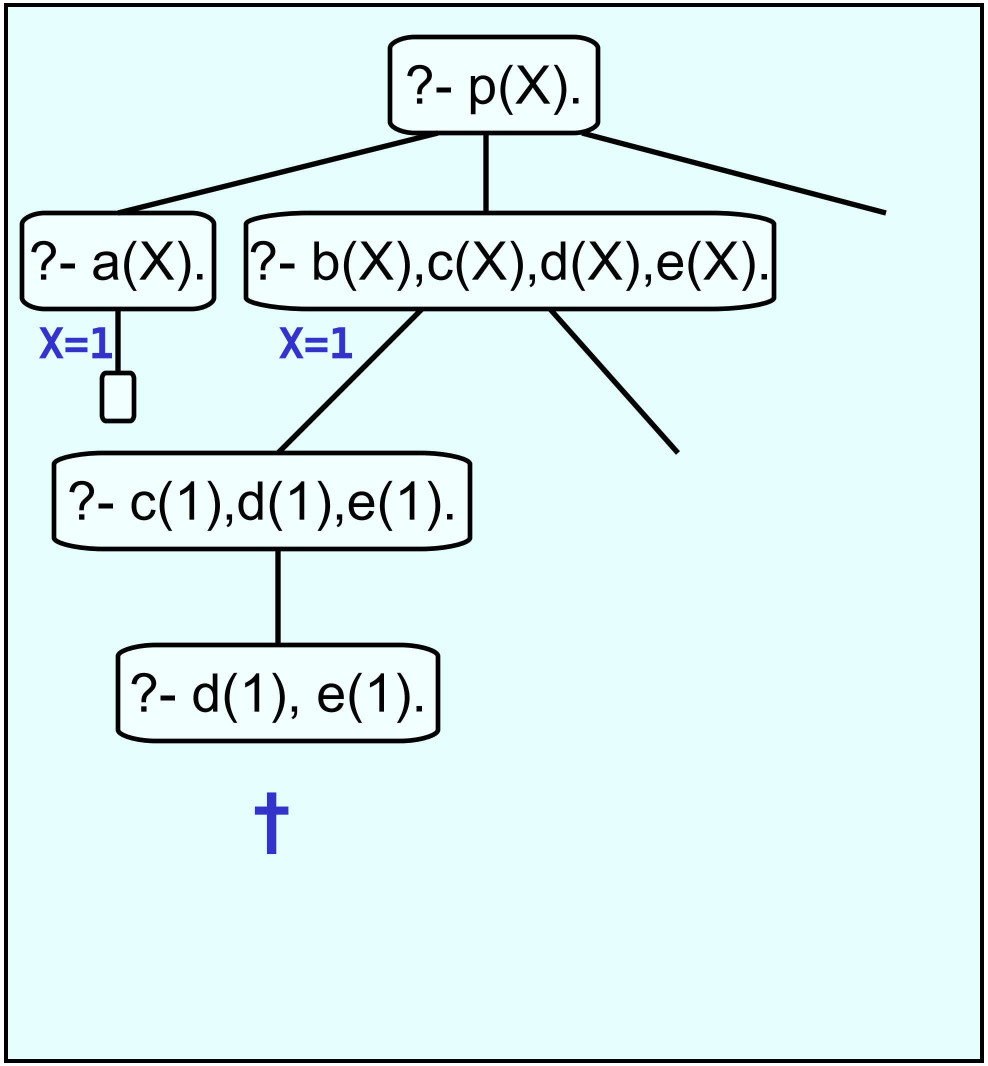
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

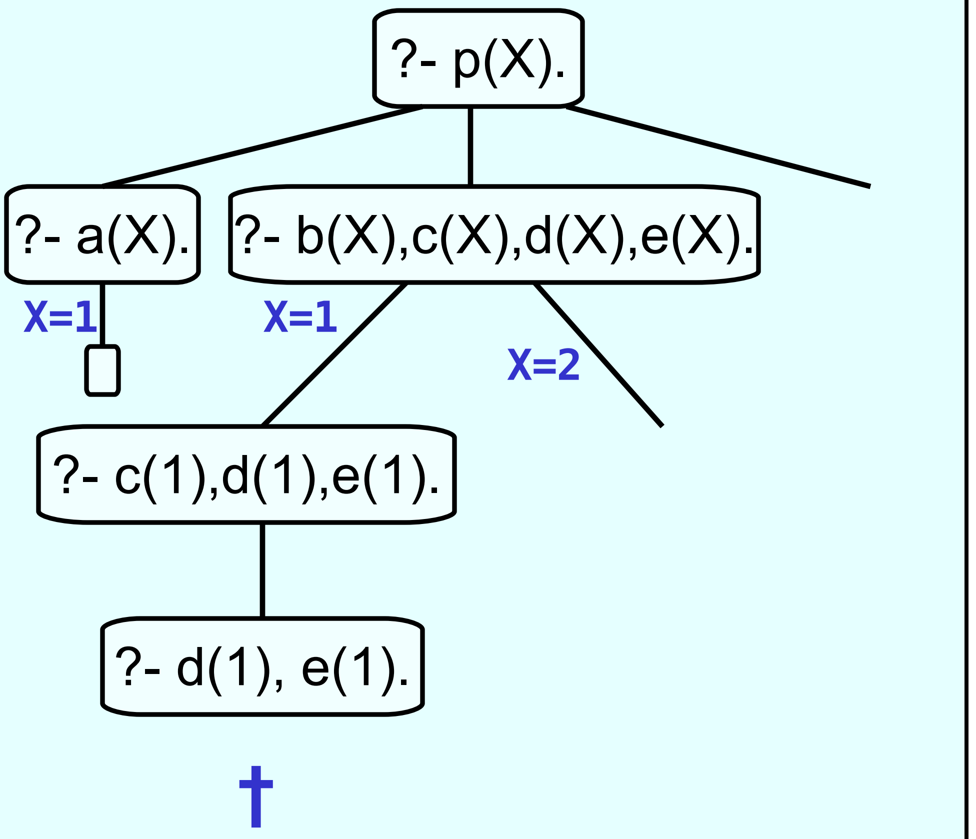
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

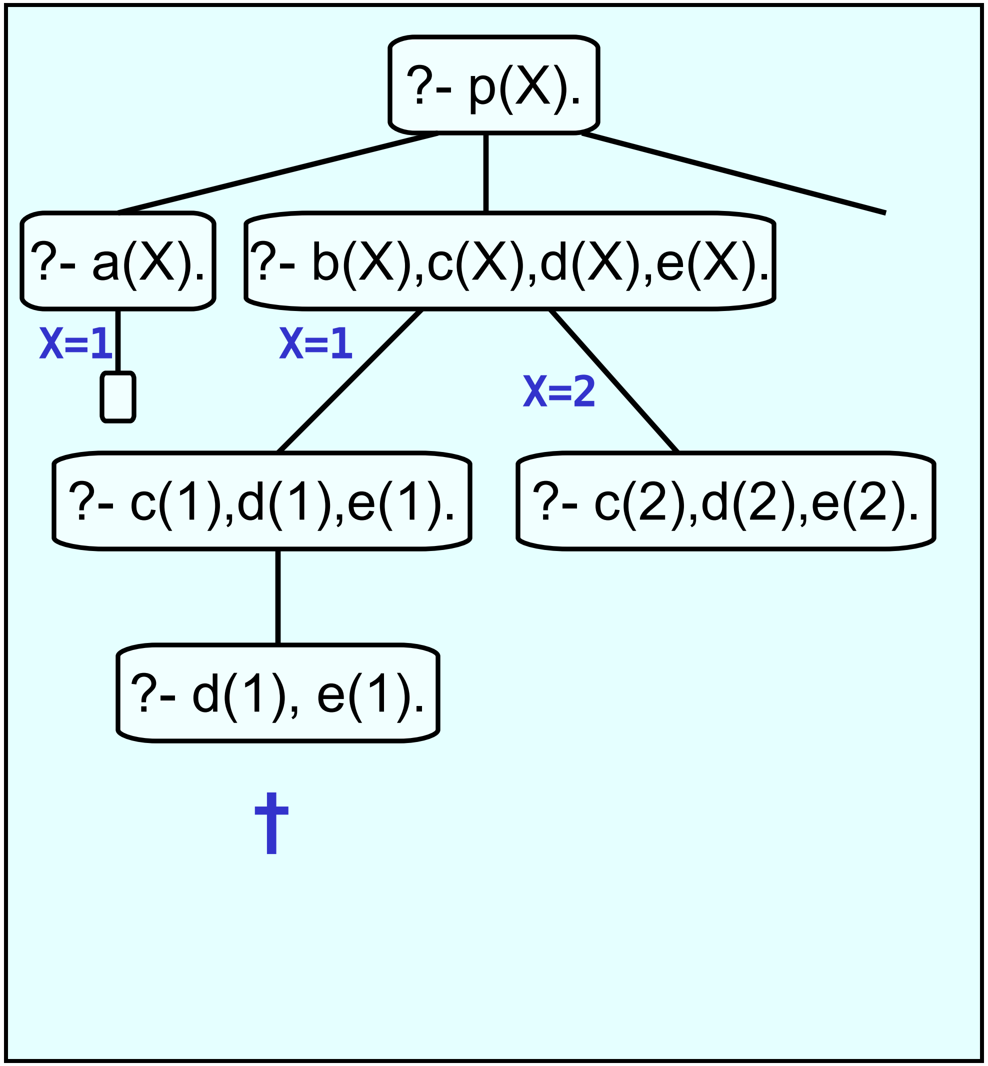
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

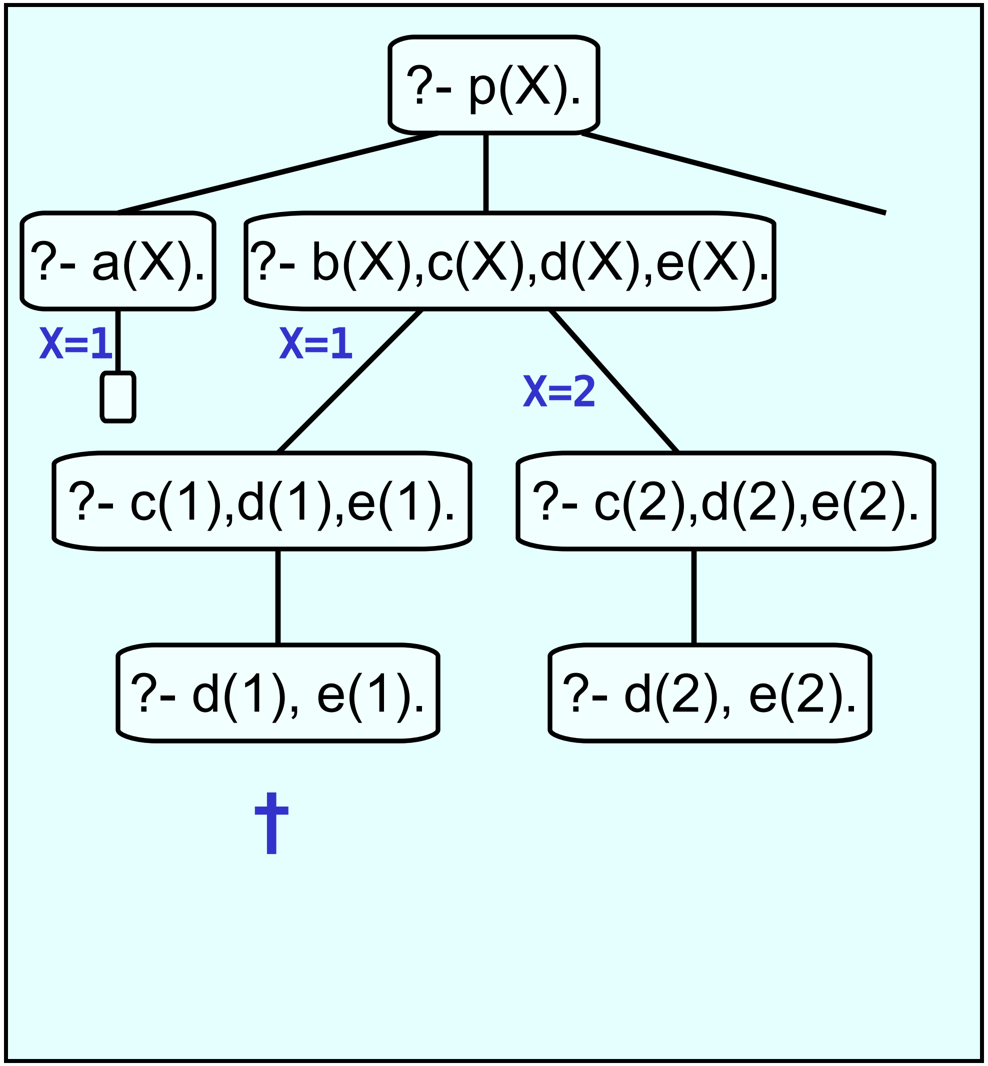
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

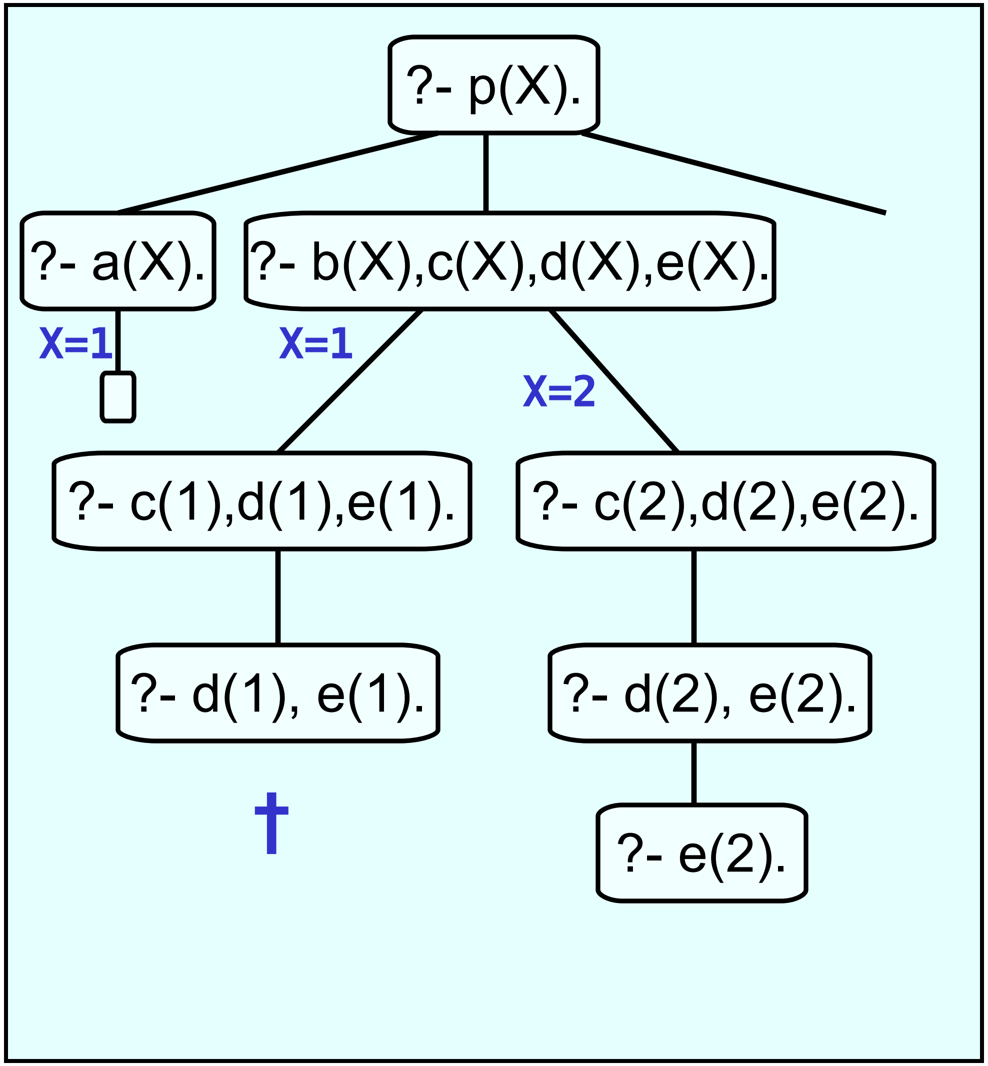
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

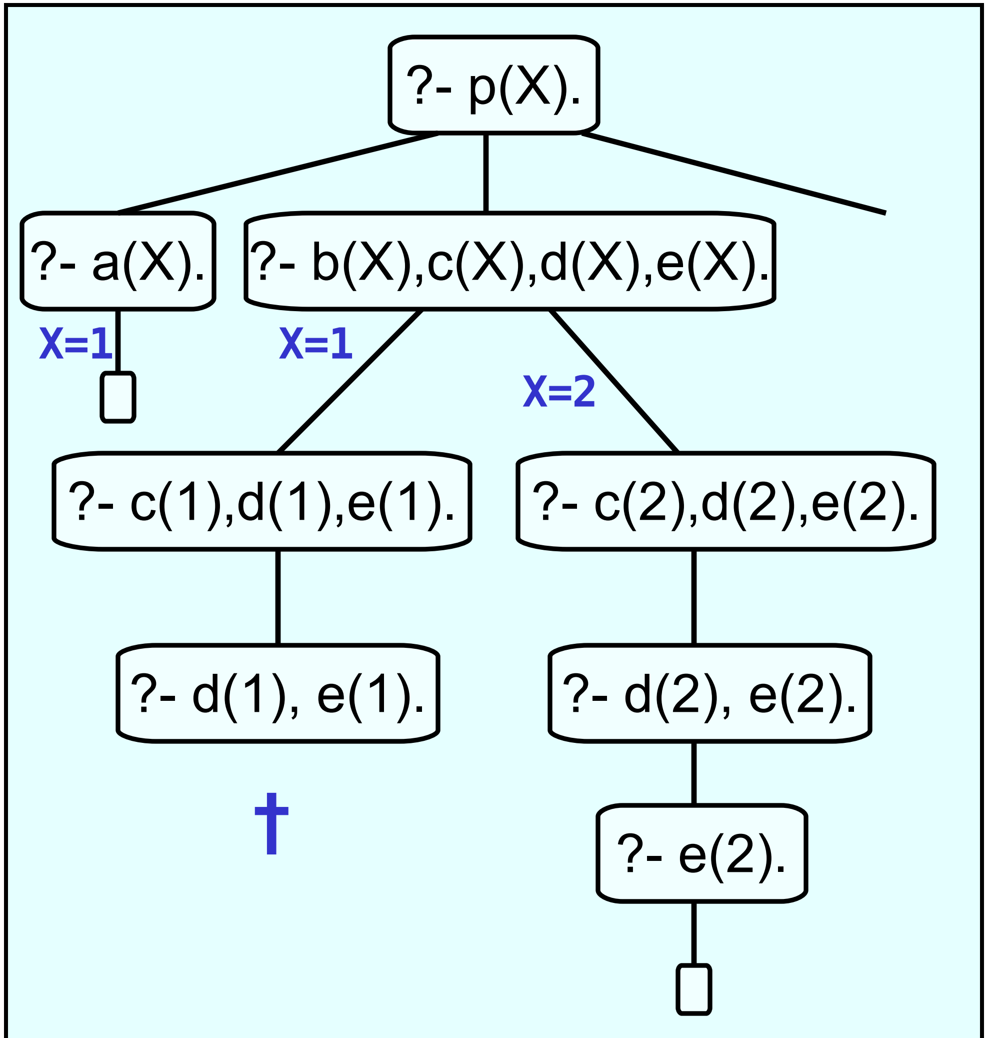
```
?- p(X).
X=1;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

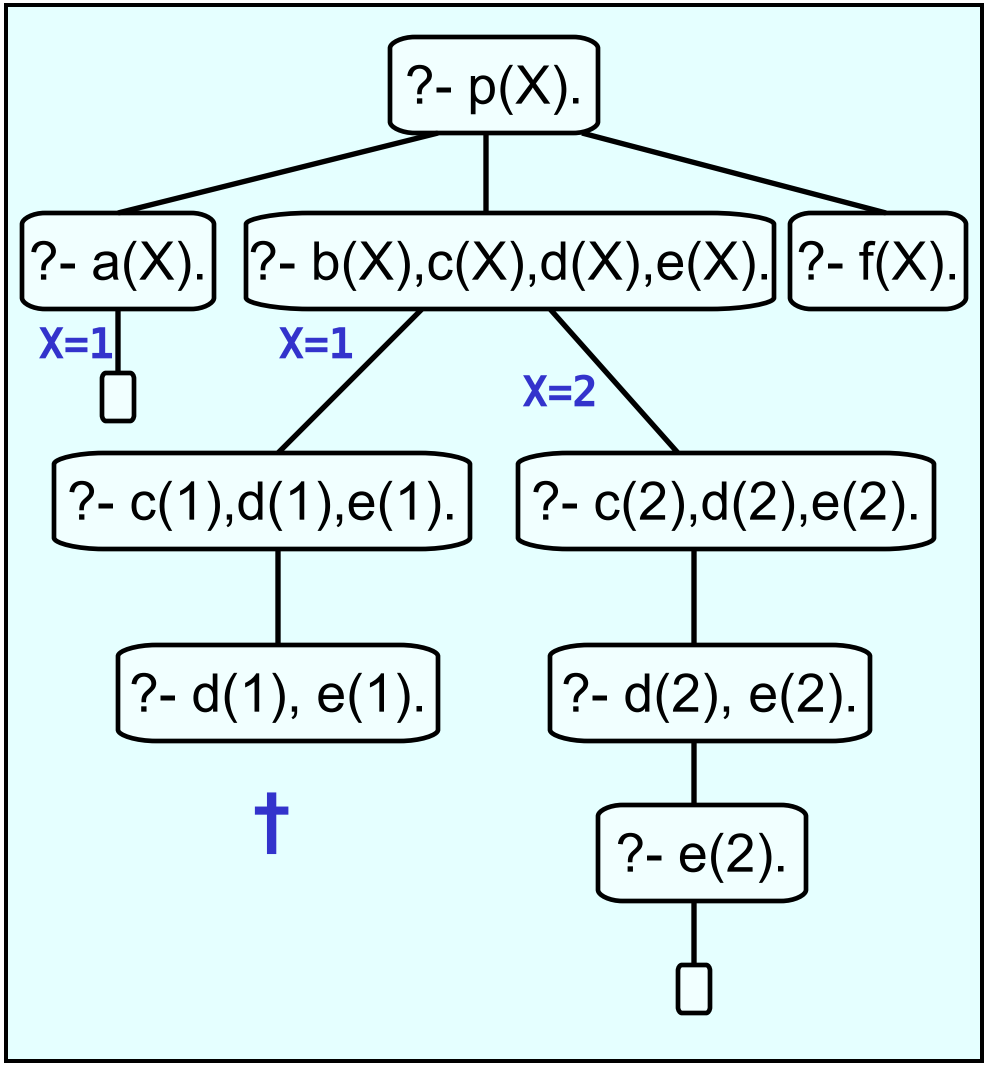
```
?- p(X).
X=1;
X=2
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

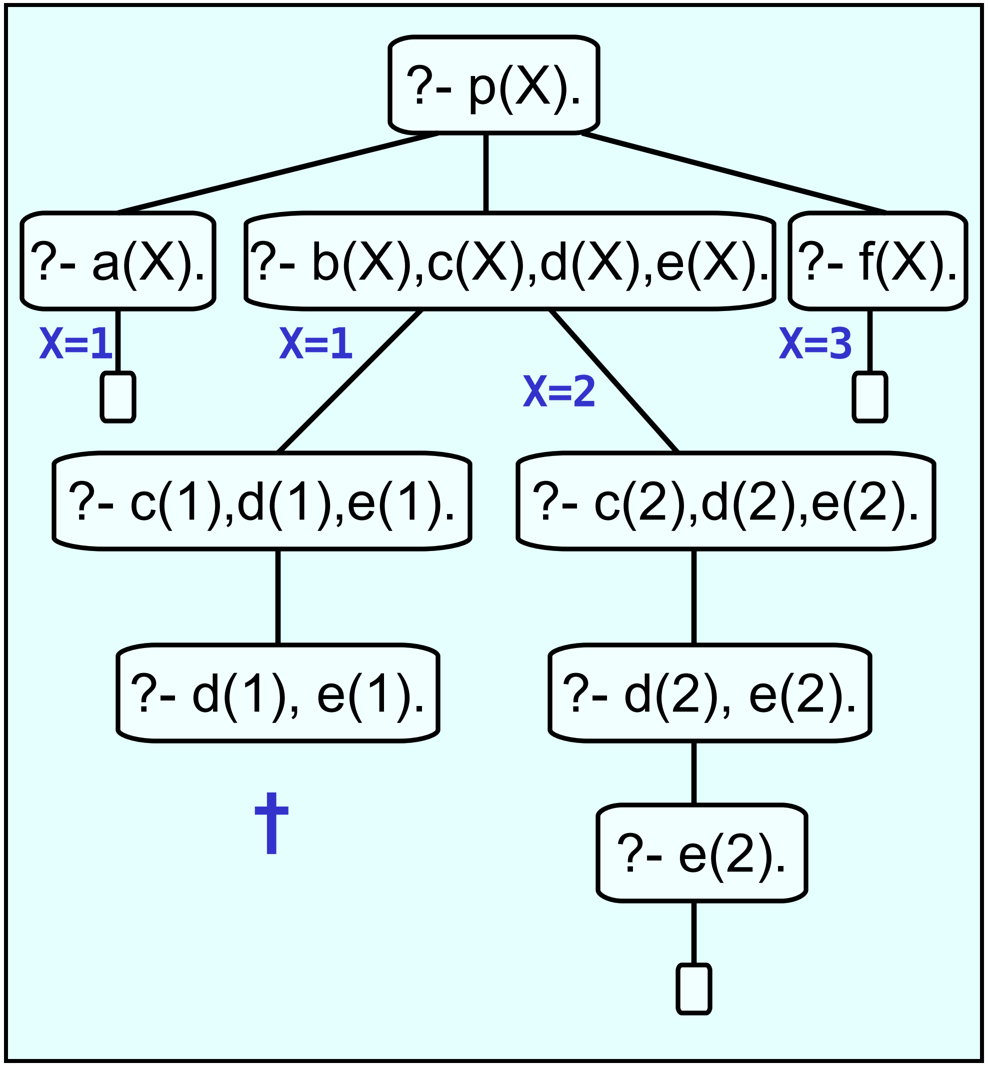
```
?- p(X).
X=1;
X=2;
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

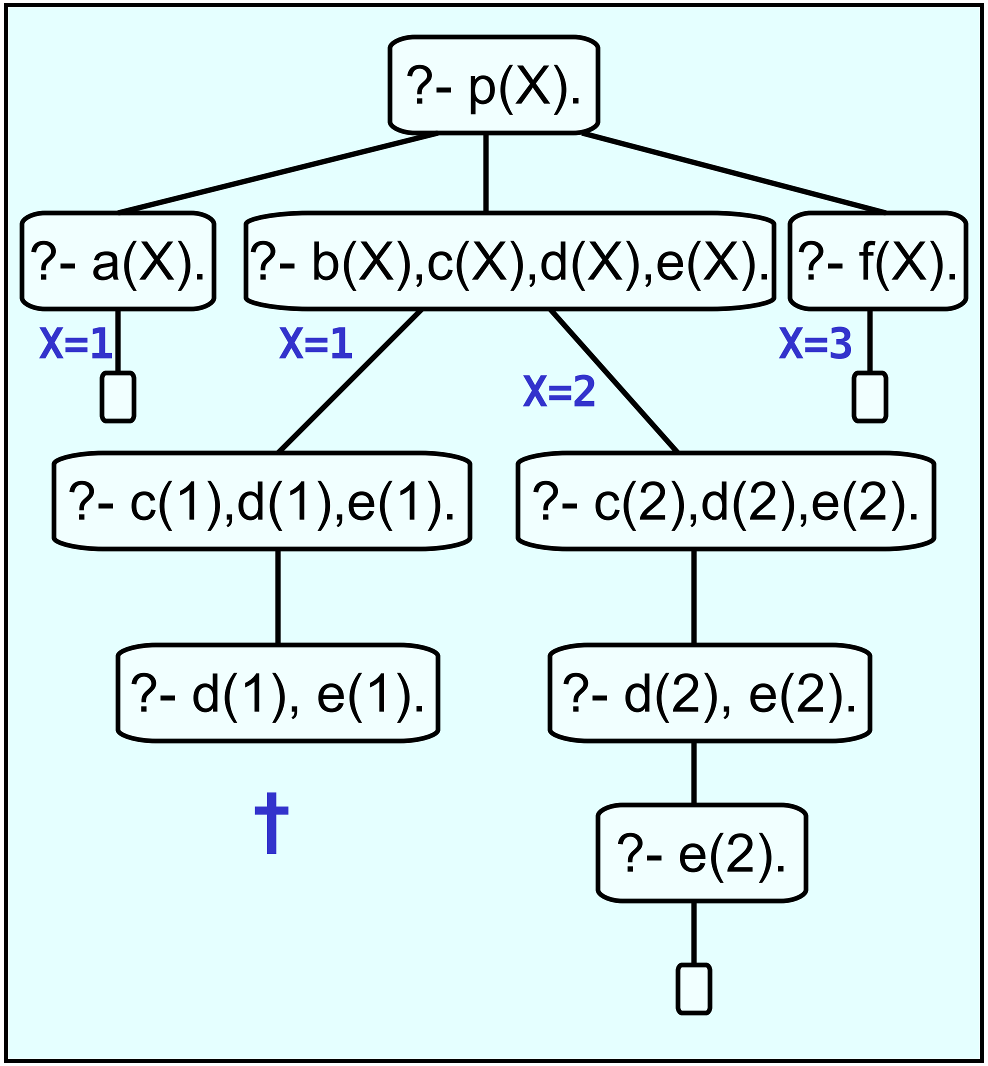
```
?- p(X).
X=1;
X=2;
X=3
```



Example: cut-free code

```
p(X):- a(X).
p(X):- b(X), c(X), d(X), e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

```
?- p(X).
X=1;
X=2;
X=3;
no
```



Adding a cut

- Suppose we insert a cut in the second clause:

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

- If we now pose the same query we will get the following response:

```
?- p(X).
X=1;
no
```

Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

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

Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

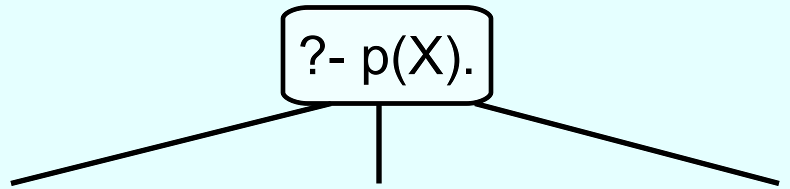
```
?- p(X).
```

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

Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

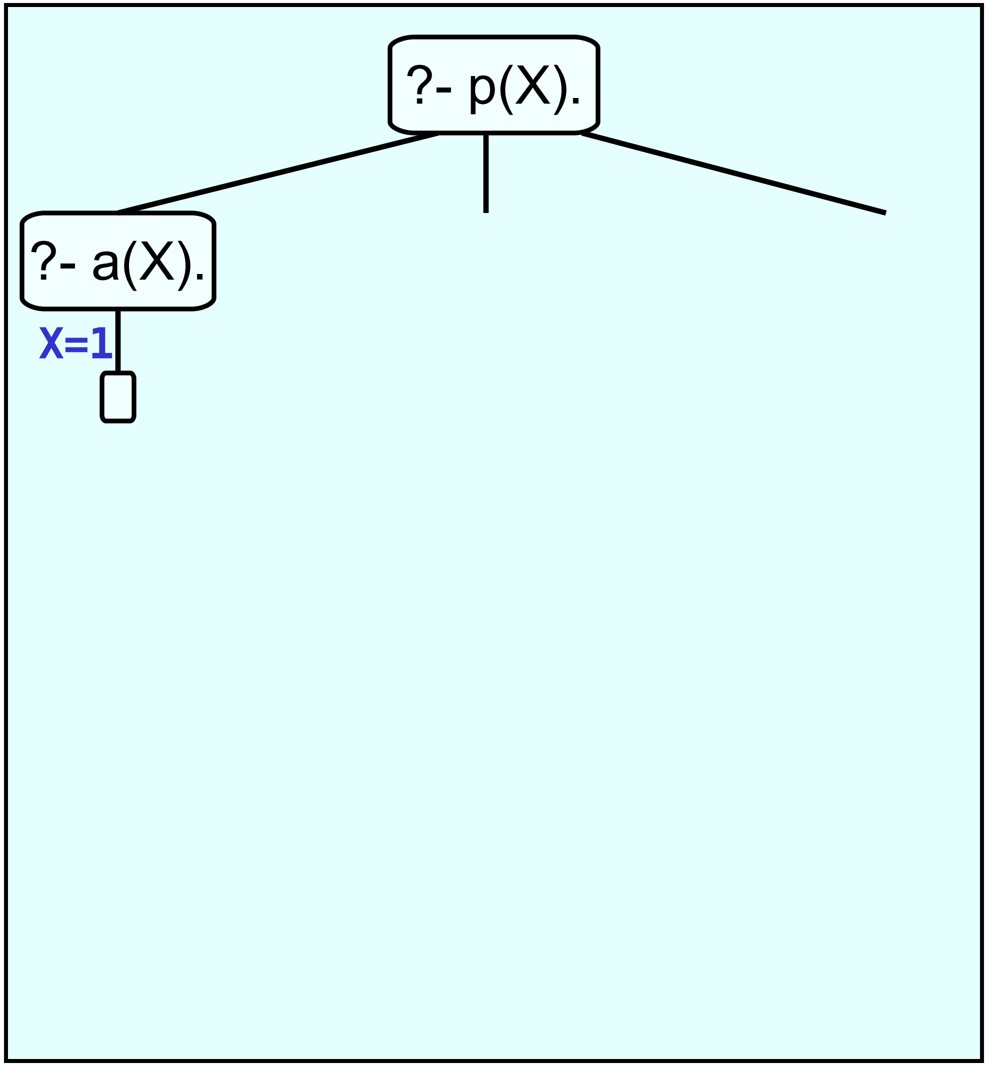
```
?- p(X).
```



Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

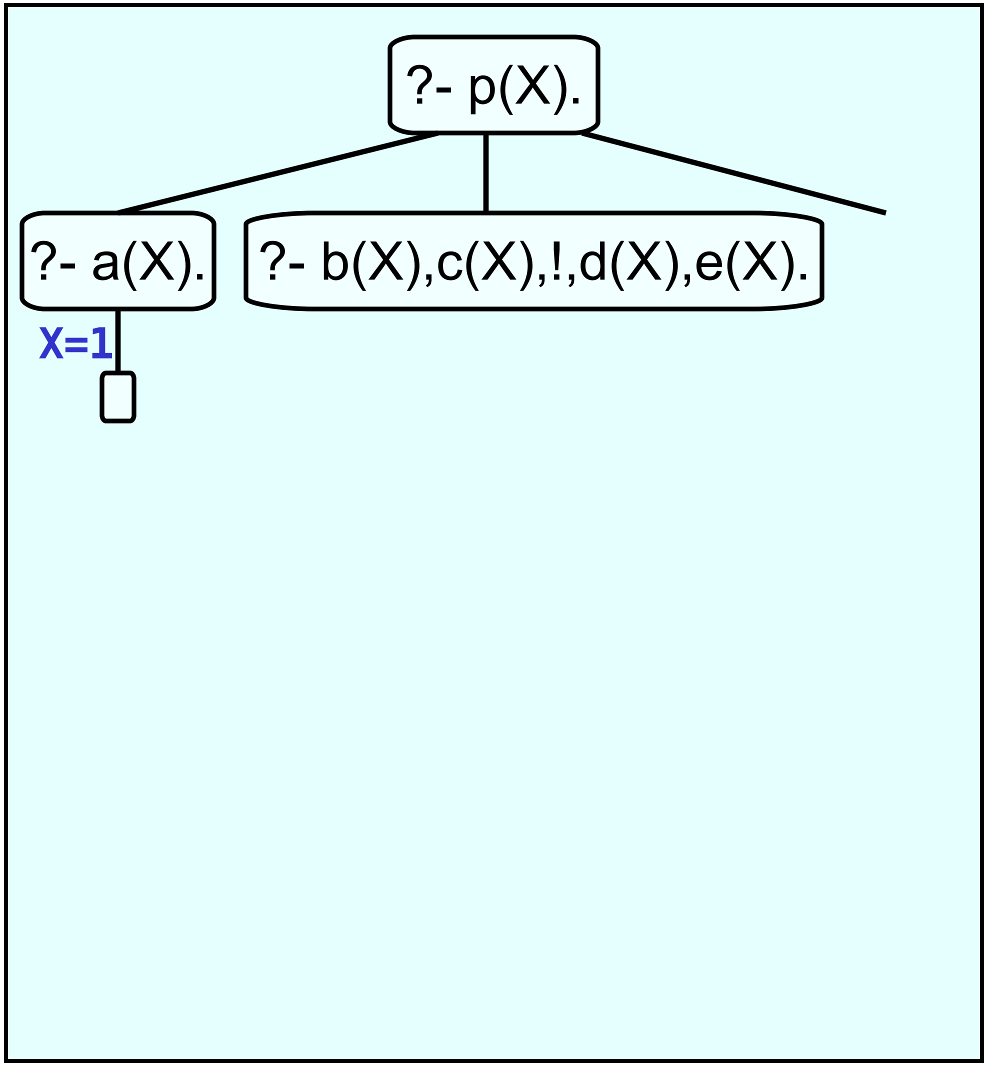
```
?- p(X).
X=1
```



Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

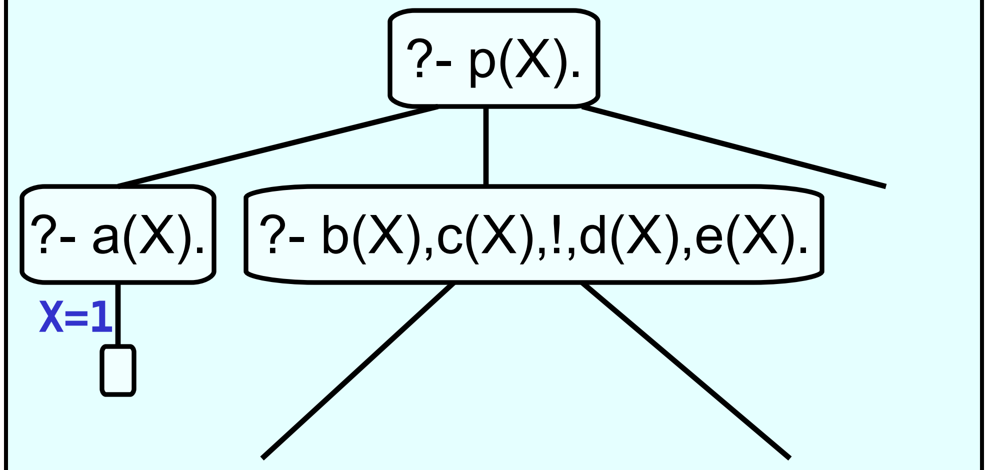
```
?- p(X).
X=1;
```



Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

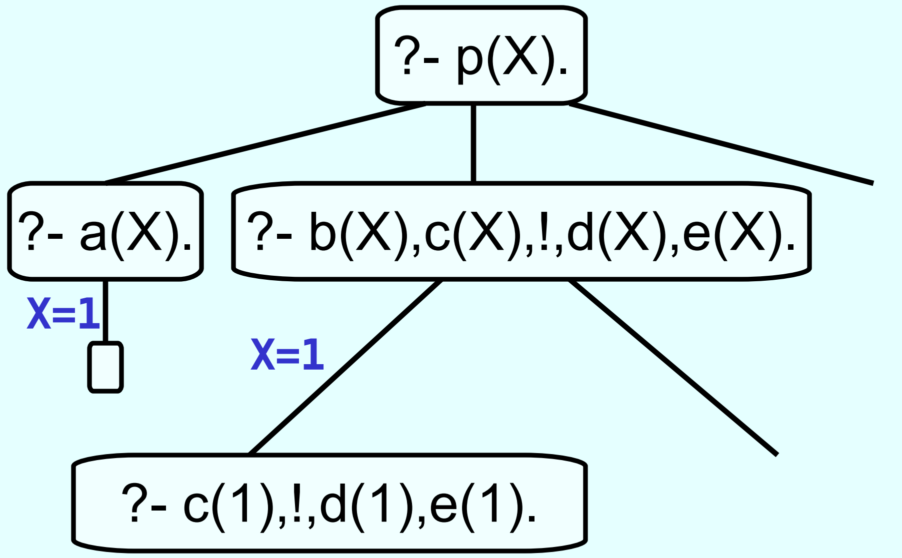
```
?- p(X).
X=1;
```



Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

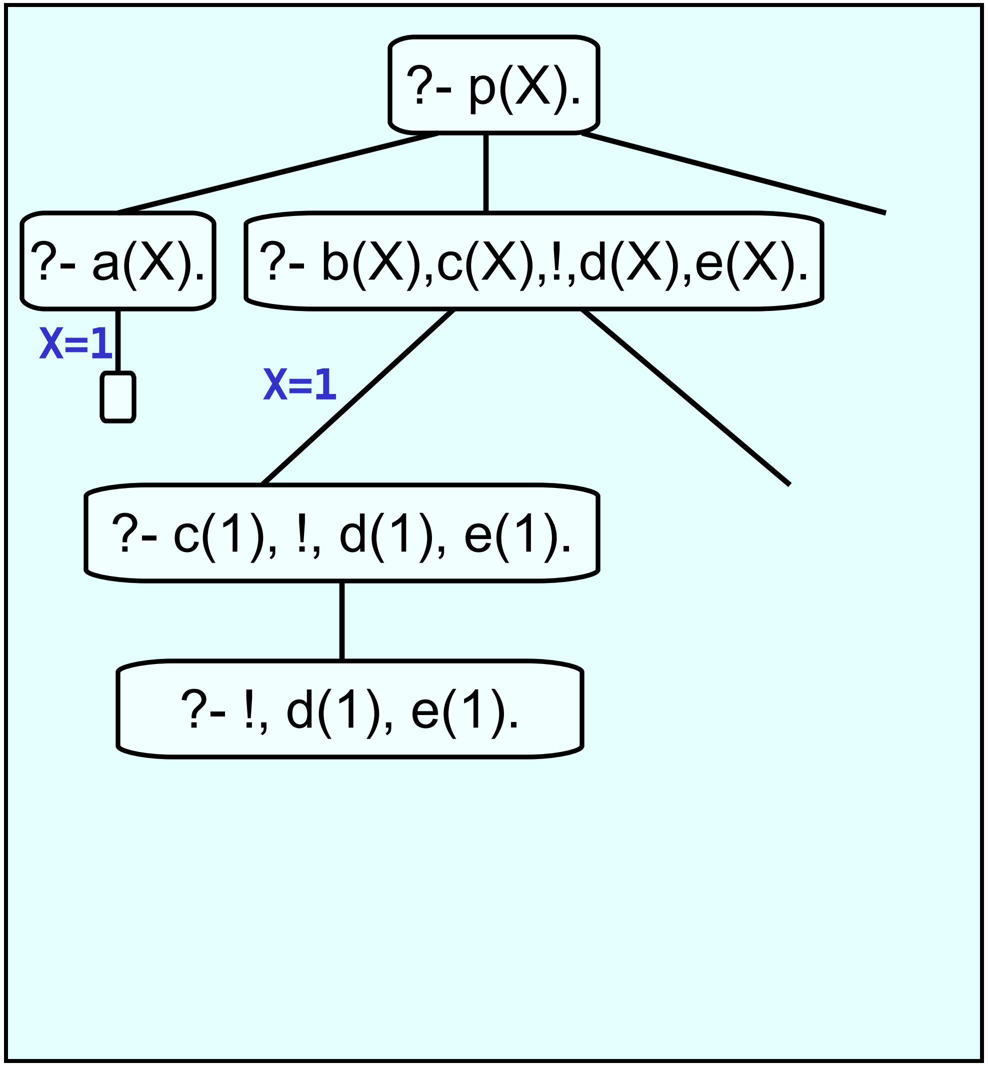
```
?- p(X).
X=1;
```



Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

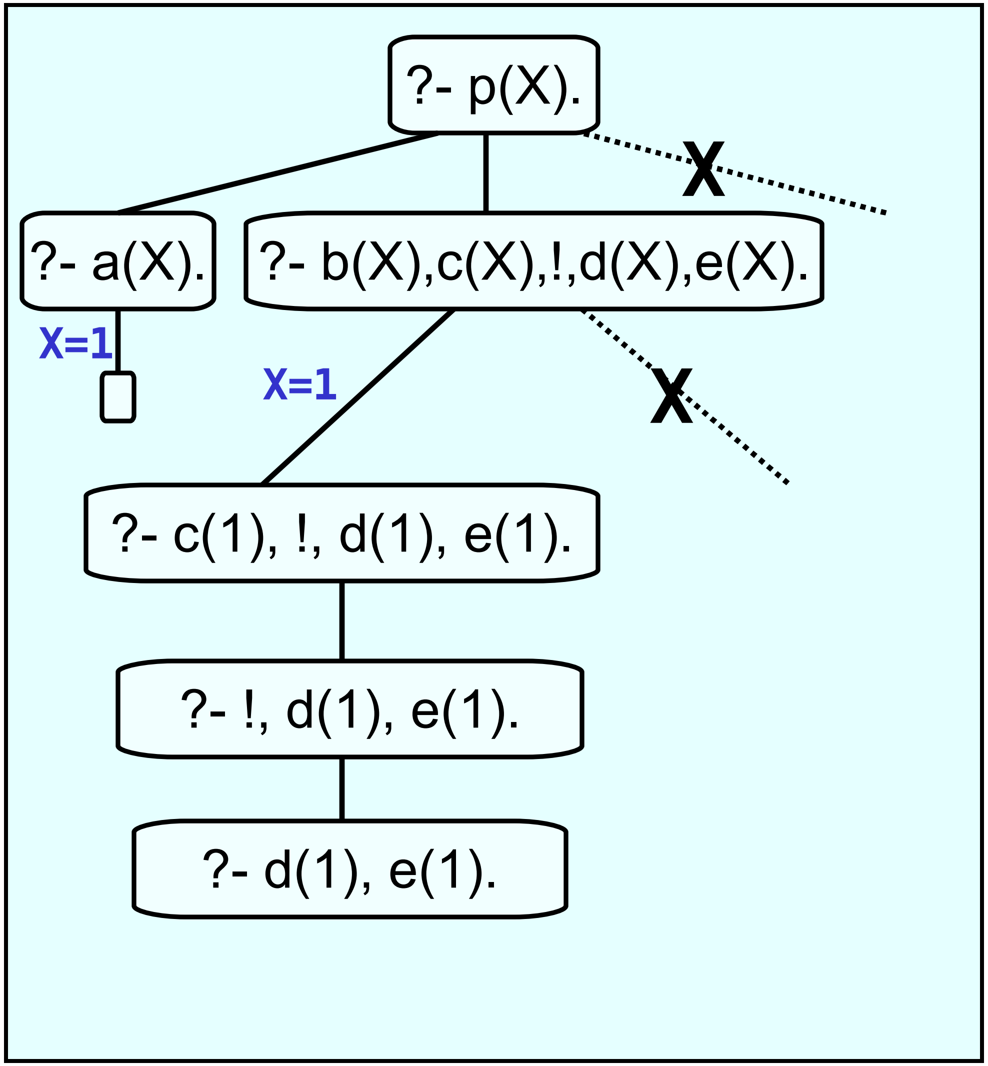
```
?- p(X).
X=1;
```



Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

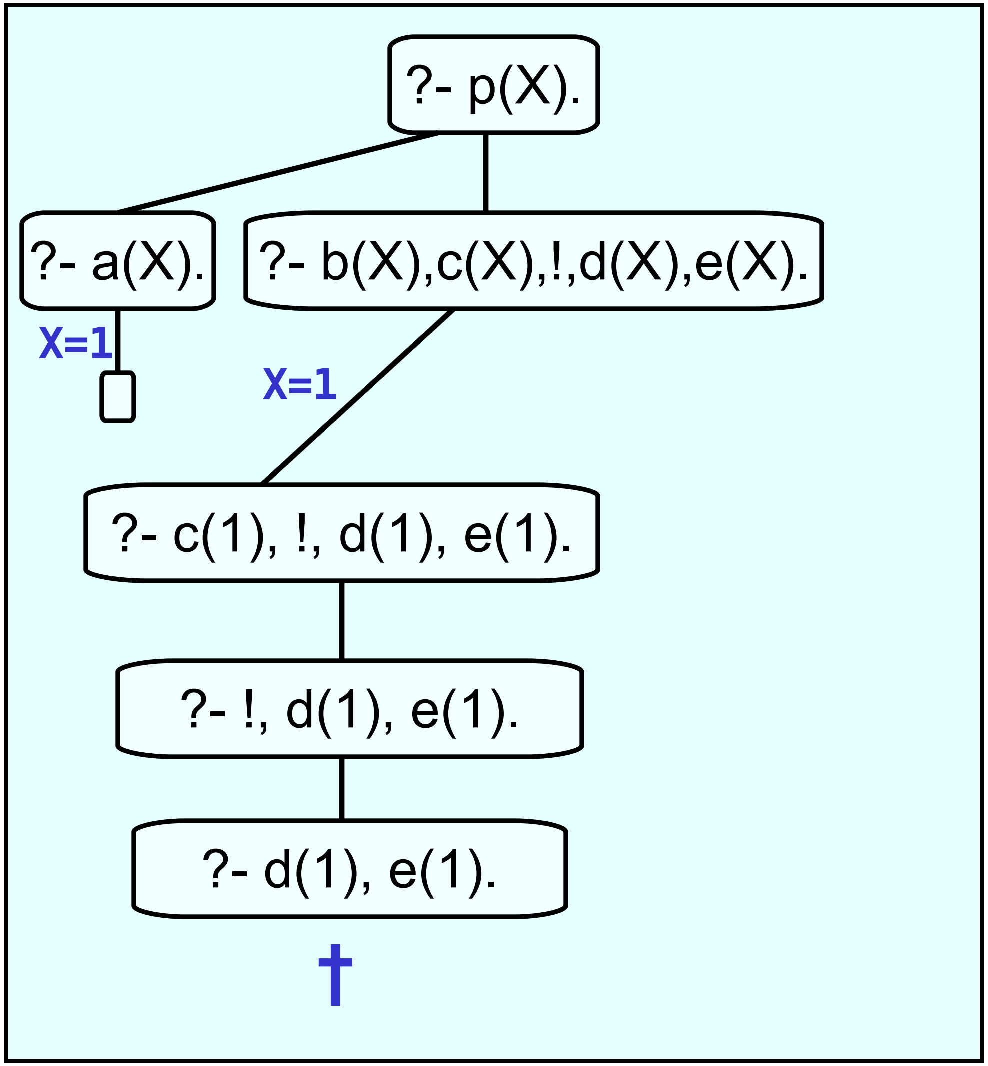
```
?- p(X).
X=1;
```



Example: cut

```
p(X):- a(X).
p(X):- b(X),c(X),!,d(X),e(X).
p(X):- f(X).
a(1).
b(1).   b(2).
c(1).   c(2).
d(2).
e(2).
f(3).
```

```
?- p(X).
X=1;
no
```



What the cut does

- The cut only commits us to choices made since the parent goal was unified with the the left-hand side of the clause containing the cut
- For example, in a rule of the form

  q:- p_1, … , p_n, !, r_1, … , r_n.

  when we reach the cut it commits us:
  - to this particular clause of q
  - to the choices made by p_1, … , p_n
  - NOT to choices made by r_1, … , r_n

Using Cut

- Consider the following predicate max/3 that succeeds if the third argument is the maximum of the first two

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

Using Cut

- Consider the following predicate max/3 that succeeds if the third argument is the maximum of the first two

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

```
?- max(2,3,3).
yes

?- max(7,3,7).
yes
```

Using Cut

- Consider the following predicate max/3 that succeeds if the third argument is the maximum of the first two

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

```
?- max(2,3,2).
no

?- max(2,3,5).
no
```

The max/3 predicate

- What is the problem?
- There is a potential inefficiency
  - Suppose it is called with ?- max(3,4,Y).
  - It will correctly unify Y with 4
  - But when asked for more solutions, it will try to satisfy the second clause. This is completely pointless!

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

max/3 with cut

- With the help of cut this is easy to fix

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

- Note how this works:
  - If the X =< Y succeeds, the cut commits us to this choice, and the second clause of max/3 is not considered
  - If the X =< Y fails, Prolog goes on to the second clause

Green Cuts

- Cuts that do not change the meaning of a predicate are called **<u>green cuts</u>**
- The cut in max/3 is an example of a green cut:
  – the new code gives exactly the same answers as the old version,
  – but it is more efficient

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?
  - okay

```
?- max(200,300,X).
X=300
yes
```

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?
  - okay

```
?- max(400,300,X).
X=400
yes
```

Another max/3 with cut

- Why not remove the body of the second clause? After all, it is redundant.

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

- How good is it?
  - oops….

```
?- max(200,300,200).
yes
```

Revised max/3 with cut

- Unification after crossing the cut

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

- This does work

```
?- max(200,300,200).
no
```

Red Cuts

- Cuts that change the meaning of a predicate are called **red cuts**
- The cut in the revised max/3 is an example of a red cut:
  - If we take out the cut, we don't get an equivalent program
- Programs containing red cuts
  - Are not fully declarative
  - Can be hard to read
  - Can lead to subtle programming mistakes

Another build-in predicate: fail/0

- As the name suggests, this is a goal that will immediately fail when Prolog tries to proof it
- That may not sound too useful
- But remember: when Prolog fails, it tries to backtrack

Vincent and burgers

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

- The cut fail combination allows to code exceptions

Vincent and burgers

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

- The cut fail combination allows to code exceptions

```
?- enjoys(vincent,a).
yes
```

Vincent and burgers

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

- The cut fail combination allows to code exceptions

```
?- enjoys(vincent,b).
no
```

Vincent and burgers

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

- The cut fail combination allows to code exceptions

```
?- enjoys(vincent,c).
yes
```

Vincent and burgers

```
enjoys(vincent,X):- bigKahunaBurger(X), !, fail.
enjoys(vincent,X):- burger(X).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

- The cut fail combination allows to code exceptions

```
?- enjoys(vincent,d).
yes
```

Negation as Failure

- The cut-fail combination seems to be offering us some form of negation
- It is called **negation as failure**, and defined as follows:

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

- Second clause makes sure neg succeeds if Goal was not satisfied in the first clause (i.e. ! was not triggered)

Vincent and burgers revisited

```
enjoys(vincent,X):- burger(X),
                    neg(bigKahunaBurger(X)).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

Vincent and burgers revisited

```
enjoys(vincent,X):- burger(X),
                    neg(bigKahunaBurger(X)).

burger(X):-  bigMac(X).
burger(X):-  bigKahunaBurger(X).
burger(X):-  whopper(X).

bigMac(a).
bigKahunaBurger(b).
bigMac(c).
whopper(d).
```

```
?- enjoys(vincent,X).
X=a
X=c
X=d
```

Another build-in predicate: \+

- Because negation as failure is so often used, there is no need to define it
- In standard Prolog the prefix operator \+ means negation as failure
- So we could define Vincent`s preferences as follows:

```
enjoys(vincent,X):-  burger(X),
                     \+ bigKahunaBurger(X).
```

```
?- enjoys(vincent,X).
X=a
X=c
X=d
```

Negation as failure and logic

- Negation as failure is not logical negation
- Changing the order of the goals in the vincent and burgers program gives a different behaviour:

```
enjoys(vincent,X):- \+ bigKahunaBurger(X),
                       burger(X).
```

```
?- enjoys(vincent,X).
no
```

Exercises(1)

Suppose we have the following database:

- p(1).
- p(2) :- !.
- p(3).
-
- Write all of Prolog's answers to the following queries:
-
- ?- p(X).
- ?- p(X),p(Y).
- ?- p(X),!,p(Y).

Exercises(2)

First, explain what the following program does:

```
class(Number,positive) :- Number > 0.
class(0,zero).
class(Number, negative) :- Number < 0.
```

Second, improve it by adding green cuts.

Exercises(3)

Without using cut, write a predicate split/3 that splits a list of integers into two lists: one containing the positive ones (and zero), the other containing the negative ones. For example:

split([3,4,-5,-1,0,4,-9],P,N)

should return:

P = [3,4,0,4]

N = [-5,-1,-9].

Then improve this program, without changing its meaning, with the help of cut.

Exercises(4)

Define a predicate nu/2 ("not unifiable") which takes two terms as arguments and succeeds if the two terms do not unify. For example:

nu(foo,foo).

no

nu (foo,blob).

yes

nu(foo,X).

no

You should define this predicate in three different ways:

- write it with the help of = and \+
- write it with the help of =, but don't use \+.
- write it using a cut-fail combination. Don't use = and don't use \+.

Exercises(5)

Implement a sudoku-solver (9x9).

Try to make as much use as possible of recursion, i.e. don't just take the straightforward option where every position is assigned a different variable, and all constraints are based on enumerating these variables...

Hint: Represent a board as a list of rows, which are lists of numbers.